PowerMAX OS Real-Time Guide

concurrent

0890466-100
January 2006

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- May 1995	000	PowerUX Release 2.1
Previous Release -- June 2001	070	PowerMAX OS Release 5.1
Previous Release -- August 2003	080	PowerMAX OS Release 6.0
Previous Release -- November 2004	090	PowerMAX OS Release 6.2
Current Release -- January 2006	100	PowerMAX OS Release 6.3

Preface

Scope of Manual

This manual provides an introduction to the real-time features of PowerMAX OS and describes techniques for improving response time and increasing determinism. It contains documentation for interfaces that are used primarily by real-time applications.

Structure of Manual

This guide consists of the following 14 chapters, 5 appendixes, and index:

- Chapter 1:
  - Overviews the real-time features of PowerMAX OS.
  - Introduces this guide.
- Chapter 2 treats:
  - Achieving real-time response.
  - Increasing process dispatch latency.
- Chapter 3 describes techniques to create a deterministic environment to run application.
- Chapter 4 explains the procedures for using the **`ktrace`** utility.
- Chapter 5 discusses:
  - Real-time interprocess communication.
  - Procedures for using the POSIX® message-passing facilities.
- Chapter 6 describes PowerMAX OS interprocess synchronization tools.
- Chapter 7 describes some of the facilities that can be used for timing.
- Chapter 8 explains:
  - PowerMAX OS support for user-level interrupt routines.
  - Using user-level interrupt routines.
- Chapter 9 describes the Virtual Interrupt System and how to use it.
- Chapter 10 describes:
  - The 60 Hz clock interrupt and its interrupt service routine.
  - Disabling the 60 Hz clock interrupt on one or more CPUs.

- Chapter 11 explains:
  - Performing direct disk I/O
  - Virtual partitions.
  - POSIX synchronized I/O interfaces.
- Chapter 12 describes the PowerMAX OS asynchronous I/O facilities and explains how to use them.
- Chapter 13 explains how to use the following PowerMAX OS facilities:
  - Real-time clocks.
  - Edge-triggered interrupts.
  - High-speed data enhanced devices.
  - DR11W emulator.
  - 1553 Advanced Bus Interface.
  - High-performance serial controller.
- Chapter 14 describes STREAMS Network Buffers and how to use them.
- Appendix A contains an example C program that illustrates use of the POSIX message queue facilities.
- Appendix B contains an example C program that illustrates use of the interprocess synchronization tools.
- Appendix C contains an example C program that demonstrates use of the user-level interrupt routine facility by a user program that executes a user-level interrupt process and interrupt-handling routine.
- Appendix D contains an example C program that demonstrates use of the user-level interrupt routine facility by a multithreaded program that creates two interrupt connections to two separate real-time clocks.
- Appendix E contains example C programs that have been developed to illustrate use of the high–speed data enhanced device, HSDE.
- The index contains an alphabetical reference to key terms and concepts and numbers of pages where they occur in the text.

Syntax Notation

The following notation is used throughout this manual:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *itali*cs.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and system manual page references also appear in **`list bold`** type.

`list`	Operating system and program output such as prompts and messages and listings of files and programs appear in `list` type.
`[ ]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such options or arguments

Related Publications

The following are related publications:

0830045	*HN6800 Console Reference Manual*
0830046	*HN6800 Architecture Manual*
0830047	*HN6200 Console Reference Manual*
0830048	*HN6200 Architecture Manual*
0830050	*Power Hawk Series 600 Console Reference Manual*
0830059	*Power Hawk Series 700 Console Reference Manual*
0830060	*Power Hawk Series 900 Console Reference Manual*
0890288	*HAPSE Reference Manual*
0890398	*NightTrace Manual*
0890423	*PowerMAX OS Programming Guide*
0890425	*Device Driver Programming*
0890429	*System Administration Volume 1*
0890430	*System Administration Volume 2*
0890459	*Compilation Systems Volume 1 (Tools)*
0890493	*Data Monitoring Reference Manual*
0891080	*Power Hawk Series 600 Diskless Systems Administrator's Guide*
0891081	*Power Hawk Series 600 Closely-Coupled Programming Guide*
0891086	*Power Hawk Series 700 Diskless Systems Administrator's Guide*
0891087	*Power Hawk Series 700 Closely-Coupled Programming Guide*
0891090	*Power Hawk Series 900 Diskless Systems Administrator's Guide*

Contents

1 Introduction

1 Introduction

This chapter describes the focus of this guide and provides an overview of the real-time features of PowerMAX OS.

Focus of Guide

This manual provides an introduction to the real-time features of PowerMAX OS and describes techniques for improving response time and increasing determinism. It contains documentation for interfaces that are used primarily by real-time applications. These interfaces include those for interprocess synchronization tools, timing facilities, user-level interrupt routines, synchronized I/O, and asynchronous I/O.

It is intended that this guide be used in conjunction with the *PowerMAX OS Programming Guide*. The *PowerMAX OS Programming Guide* contains documentation for interfaces that are used generally by both real-time and secure applications (for example, process management facilities, POSIX scheduling interfaces, signal management facilities, memory management facilities, and Threads Library facilities).

Real-Time Features of PowerMAX OS

The Real-Time features of PowerMAX OS are as follows:

- Support for the POSIX real-time extension
- Shielded processors
- Exclusive binding of processes and processors
- Static priority scheduling
- Memory resident processes
- Memory mapping and data sharing
- Real–time process synchronization tools
- Message queues
- Asynchronous input/output
- Direct asynchronous I/O to disk partitions
- Synchronized I/O

- User–level interrupt routines
- User–level device drivers
- High–resolution time-out facilities
- Real–time signal behavior
- Watch-dog timer function.

All of these features are described in the sections that follow.

POSIX Real-Time Extension

PowerMAX OS supports the POSIX real-time extension as set forth in ISO/IEC 9945-1. This extension includes the following functional areas:

- Semaphores
- Process memory locking
- Memory mapped files
- Shared memory
- Priority scheduling
- Real-time signal extension
- Timers
- Interprocess communication
- Synchronized input and output
- Asynchronous input and output.

Shielded Processors

PowerMAX OS provides you with the capability of shielding selected processors from the unpredictable processing associated with interrupts and system daemons. By allowing you to bind critical, high–priority tasks to particular CPUs and to direct most interrupts and system daemons to other CPUs, it provides you with a means of obtaining the best process dispatch latency possible on a particular processor in a multiprocessor system. Chapter 2 presents a model for shielding processors and describes the techniques that you can use for improving response time. Chapter 3 describes techniques that you can use for increasing determinism.

Exclusive Binding

PowerMAX OS provides you with the concepts of exclusive-use and general-purpose processors, and of exclusively bound and unbound processes. An exclusive-use processor executes only processes which are exclusively bound to it, plus any system daemons biased to that processor. All processes which are not exclusively bound are unbound. The unbound processes run on only the general purpose (nonexclusive) processors mentioned in their biases. Any exclusive-use processor mentioned in the bias of an unbound process is silently avoided.

A processor dynamically becomes exclusive-use when it acquires its first exclusively bound process. A processor automatically reverts back to being general purpose when the last exclusively bound process leaves it. On reverting back, any unbound process which has that processor in its bias will silently start using it again.

Although exclusive binding can be done with any processor, it is an especially convenient way of moving off at one time all ordinary processes from a shielded processor at the moment a real-time application starts up, and letting them back on at the moment the final real-time application using the shielded processor shuts down. In this way the system dynamically achieves an efficient utilization of the available processors.

Static Priority Scheduling

PowerMAX OS accommodates static priority scheduling—that is, processes scheduled under certain System V scheduler classes or POSIX scheduling policies do not have their priorities changed by the operating system in response to their run–time behavior. The resulting benefits are reduced kernel overhead and increased user control.

Process scheduling and management facilities are fully described in the *PowerMAX OS Programming Guide*.

Memory Resident Processes

Paging and swapping often add an unpredictable amount of system overhead time to application programs. To eliminate performance losses due to paging and swapping, PowerMAX OS allows you to make certain portions of a process's virtual address space resident. The **`mlockall(3C)`**, **`munlockall(3C)`**, **`mlock(3C)`**, and **`munlock(3C)`** library routines allow you to lock all or a portion of a process's virtual address space in physical memory. The **`userdma(2)`** system call allows you to lock an application's I/0 buffer in physical memory. Locked regions are immune to paging or swapping. In contrast to traditional UNIX® operating systems, pages that are not resident at the time of the call are immediately faulted into memory and locked. Use of each of these facilities is explained in detail in the *PowerMAX OS Programming Guide*.

Memory Mapping and Data Sharing

PowerMAX OS supports shared memory and memory-mapping facilities that are based on IEEE Standard 1003.1b-1993 and UNIX System V. These facilities allow processes to share data through the use of memory objects. Memory objects are named regions of storage that can be mapped to the address space of one or more processes to allow them to share the associated memory. Processes can access the data in a memory object directly by mapping portions of their address spaces onto the objects. The term *memory object* includes POSIX shared memory objects, regular files, and some devices, but it does not include all file system objects (terminals and network devices, for example). One of the advantages of accessing data in a memory object through a mapping to the object is that it is generally more efficient than accessing the data through use of the **`read(2)`** and **`write(2)`** system calls. The reason is that the data do not have to be copied between the kernel and the application. The POSIX shared memory facilities and the memory-mapping facilities are fully described in the *PowerMAX OS Programming Guide.*

PowerMAX OS also supports the System V IPC shared memory mechanism, which allows processes to communicate by sharing portions of their virtual address space. Use of this mechanism is supported by a set of system calls and utilities. You can use the System V shared memory mechanism and facilities to map a user's virtual address space onto a particular range of physical memory addresses. The System V shared memory mechanism and supporting facilities are described in the *PowerMAX OS Programming Guide.* An explanation of the procedures for mapping physical memory is included.

Real–Time Process Synchronization

PowerMAX OS provides a variety of tools that cooperating processes can use to synchronize access to shared resources.

Counting semaphore interfaces that are based on IEEE Standard 1003.1b-1993 provide a mechanism that allows multiple processes to synchronize their access to the same set of resources. A counting semaphore has associated with it a value. As long as the semaphore value is positive, resources are available for use, and one of the resources is allocated to the next process that tries to acquire it. When the semaphore value is zero or negative, none of the resources are available; a process trying to acquire a resource must wait until one becomes available. Procedures for using POSIX counting semaphores and the related library routines are explained in Chapter 6.

A set of real–time process synchronization tools developed by Concurrent provides the most efficient means for synchronizing processes. This set includes tools for controlling a process's vulnerability to rescheduling, serializing processes' access to critical sections with busy–wait mutual exclusion mechanisms, and coordinating client–server interaction among processes. From these tools, a mechanism for providing sleepy–wait mutual exclusion with bounded priority inversion can be constructed. Descriptions of the tools and explanations of the procedures for using them are provided in Chapter 6.

Message Queues

PowerMAX OS supports message–passing facilities that are based on IEEE Standard 1003.1b-1993. These facilities provide a means of passing arbitrary amounts of data between cooperating processes. They allow processes to communicate through message queues, which are accessed by using names that are visible to all processes in the system. Messages can be assigned priorities that range from zero to 31. Procedures for using the message–passing facilities are explained in Chapter 5.

Asynchronous Input/Output

Being able to perform I/O operations asynchronously means that you can set up for an I/O operation and return without blocking on I/O completion. PowerMAX OS accommodates asynchronous I/O with a group of library routines that are based on IEEE Standard 1003.1b-1993. These interfaces allow a process to perform asynchronous read and write operations, initiate multiple asynchronous I/O operations with a single call, wait for completion of an asynchronous I/O operation, cancel a pending asynchronous I/O operation, and perform asynchronous file synchronization. In addition to asynchronous disk I/O, asynchronous network output through network buffers is also supported.

Asynchronous I/O is fully described in Chapter 11. Its use with the DR11W emulator is explained in Chapter 12.

Direct Asynchronous I/O to Disk Partitions

PowerMAX OS also supports asynchronous I/O to raw disk partitions through use of the same POSIX interfaces that are described in the preceding section. This capability allows a user process to transfer data directly between its I/O buffers and disk, thereby bypassing intermediate operating system buffering. Requirements and procedures for performing asynchronous I/O to raw disk partitions are explained in Chapter 11.

Synchronized I/O

PowerMAX OS supports the synchronized I/O facilities that are based on IEEE Standard 1003.1b-1993.   POSIX synchronized I/O provides the means for ensuring the integrity of an application’s data and files. A synchronized output operation ensures that data that are written to an output device are actually recorded by the device. A synchronized input operation ensures that data read from a device are an image of the data that currently reside on disk. The synchronized I/O interfaces are fully described in Chapter 10.

User–Level Interrupt Routines

PowerMAX OS provides the support necessary to allow a user–level process to connect a routine to an interrupt vector that corresponds to the interrupt generated by a selected device and to enable the connection. When a process enables the connection to an interrupt vector, it blocks in the kernel; it no longer executes at normal program level. It executes only at interrupt level—executing the user–level interrupt handling routine when the connected interrupt becomes active. Operating system support for user–level interrupt routines includes the **iconnect(2)** and **ienable(2)** system calls and the **uistat(1)** utility. Related operating system support includes the capability to raise and lower the interrupt priority level (IPL) from a user–level process, the capability to control edge–triggered interrupts from a user–level process, and the capability to dynamically allocate interrupt vectors from the kernel interrupt vector table. An overview of user–level interrupt routines and the ways in which they may be used is provided in Chapter 8. Procedures for using user–level interrupt routines and the supporting system calls, library routines, and utilities are explained.

User–Level Device Drivers

PowerMAX OS provides support for user–level device drivers. A user–level device driver consists of a library of routines that allows a user application program to perform I/O and control operations for a particular device directly from user level without entering the kernel. User–level device drivers benefit both system responsiveness and program responsiveness. The ways in which they do so are explained in Chapter 2. The techniques for writing a user–level device driver are explained in *Device Driver Programming*. Information on the user-level device driver for the DR11W emulator and the 1553 Advanced Bus Interface (ABI) is provided in Chapter 12.

High–Resolution Timeout Facilities

A finer resolution can be obtained on certain types of timeout requests by configuring a real–time clock to be used for triggering events in the high–resolution callout queue. If you configure a real–time clock for this purpose, you can obtain a resolution of one microsecond instead of 1/60 of a second, which is the resolution provided by the 60 Hz system–wide clock. By default, a real–time clock is configured into the system for use with the high–resolution callout queue. Entries can be placed in the high–resolution callout queue by using the **client_block(2)** and **server_block(2)** system calls and the **nanosleep(3C)** and **timer_settime(3C)** library routines. Procedures for using these facilities are explained in Chapter 6 and Chapter 7, respectively.

Real–Time Signal Behavior

Real–time signal behavior that is specified by IEEE Standard 1003.1b-1993 includes specification of a range of real–time signal numbers, support for queuing of multiple occurrences of a particular signal, and support for specification of an application–defined value when a signal is generated to allow for differentiation among multiple occurrences of signals of the same type. The POSIX signal-management facilities include the **sigtimedwait(2)**, **sigwaitinfo(2)**, and **sigqueue(2)** system calls, which allow a process to wait for receipt of a signal and queue a signal and an application–defined value to a process. The POSIX real–time signal behaviors and signal management facilities are described in detail in the *PowerMAX OS Programming Guide*.

Watch-Dog Timer Function

With the PowerMAXION an application program can use the fifth real time clock on the first processor board (board in slot 1) as a watch-dog timer. When programmed as a watch-dog timer this real time clock's time-out generates an exception to the PPC604 processor on the first board. The application can connect a user level interrupt routine to this exception. The operating system passes program control to the user's service routine at interrupt level. Refer to Chapter 13 for more information on the watch-dog timer function of the real-time clock.

2
Improving Response Time

2
Improving Response Time

This chapter treats some of the issues involved in achieving real–time response. One aspect of response time that is important to a real–time application is the amount of time that is required to start responding to a real–world event. The time required to respond to real–world events is affected by the system metric *process dispatch latency*. This chapter provides an overview of process dispatch latency and explains how it is affected by raising a processor's interrupt priority level (IPL) and receiving interrupts. The chapter presents a model for obtaining the best process dispatch latency that is possible on a particular processor in a multiprocessor system. It describes techniques that you can use to obtain the best process dispatch latency.

Process Dispatch Latency

The term *process dispatch latency* denotes the time that elapses from the occurrence of an external event, which is signified by an interrupt, until the process that is waiting for that external event executes its first instruction in user mode. Process dispatch latency comprises the time that it takes for the following sequence of events to occur:

1. The interrupt controller notices the interrupt and generates the interrupt exception to the CPU.
2. The interrupt routine is executed, and the process that is waiting for the interrupt (target process) is wakened.
3. The currently executing process is suspended, and a context switch is performed so that the target process can run.
4. The target process must exit from the kernel, where it was blocked waiting for the interrupt.
5. The target process runs in user mode.

This sequence of events represents the ideal case for process dispatch latency; it is illustrated by Figure 2-1.

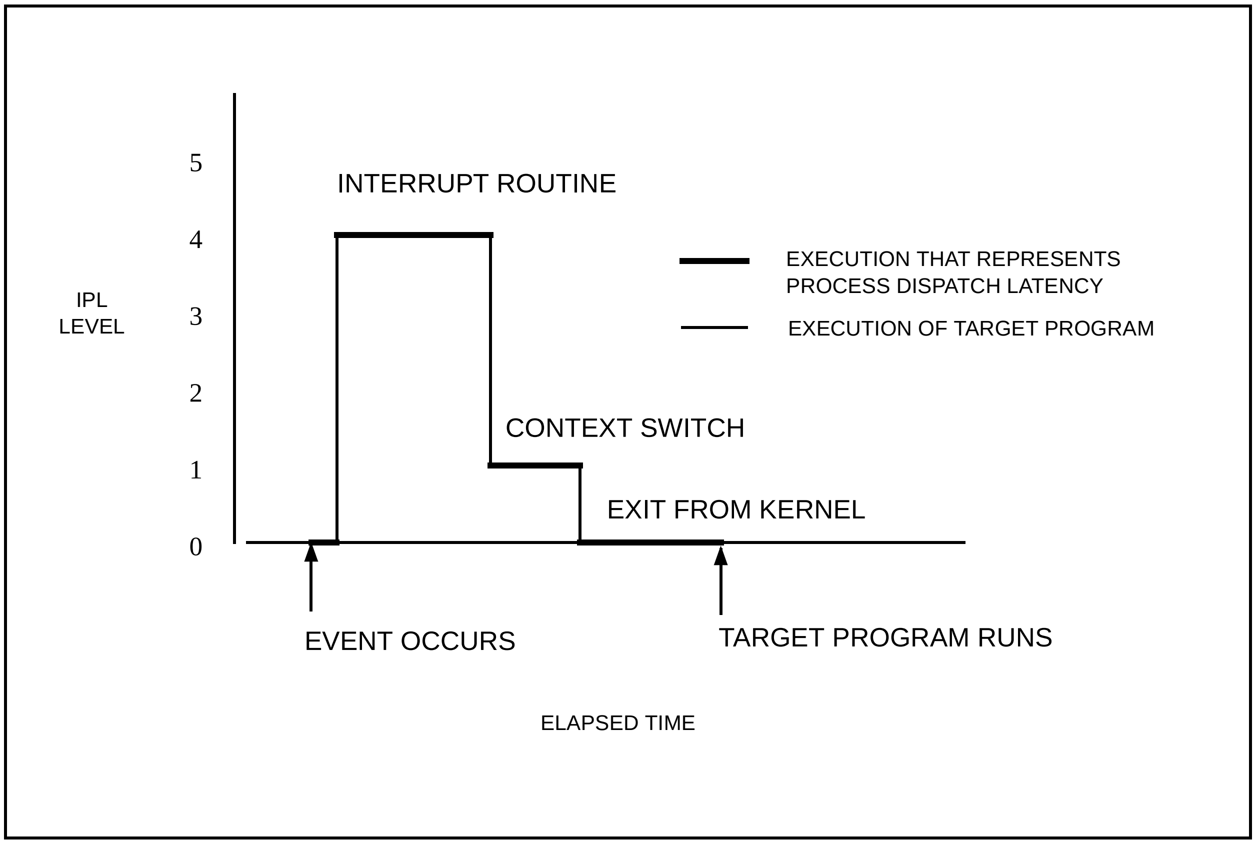


Figure 2-1. Ideal Process Dispatch Latency

The process dispatch latency is a very important metric for event–driven real–time applications because it represents the speed with which the application can respond to an external event. Most developers of real–time applications are interested in the worst case process dispatch latency; the reason is that their applications must meet certain timing constraints.

Process dispatch latency is affected by raising a processor's IPL and by receiving interrupts. The effect of raising IPL is described in "Effect of IPL." The effect of receiving interrupts is described in "Effect of Interrupts."

Effect of IPL

Each interrupt on the system is assigned an interrupt priority level. Interrupts at or below a certain level can be deferred by temporarily raising a processor's interrupt priority level (IPL). If the processor's IPL is greater than or equal to the IPL of an interrupt that occurs, the interrupt is blocked until the processor's IPL is lowered. The kernel can raise the IPL of the CPU to protect critical sections of code. By raising the IPL to PL4, for example, the kernel ensures that (H)VME devices that interrupt at or below level 4 are prevented from interrupting the CPU. The kernel raises the IPL for short periods of time to protect critical sections of code in which it is not desirable to receive a particular interrupt or be pre-empted.

The IPL is specific to a particular CPU. Setting the IPL on one CPU does not affect the IPL on another CPU. Interrupts at or below the value to which the IPL is set on one CPU may continue to be received on other CPUs.

The IPL may be raised by the operating system when a kernel daemon runs or when the application makes a system call. It can also be raised by an application program that uses the **`spl`** library routines and macro, which make it possible to modify the IPL from user level (for information on these routines, refer to the **`spl_request(3X)`** system manual page and Chapter 8 of this guide).

Raising IPL affects worst case process dispatch latency because the system cannot immediately receive an interrupt that interrupts at or below the current IPL. If the interrupt that is being held out is one that signifies an event for which a task is waiting, the process dispatch latency for that task is extended by the length of time that the interrupt remains blocked. Figure 2-2 shows the extent to which process dispatch latency is affected by raising IPL.

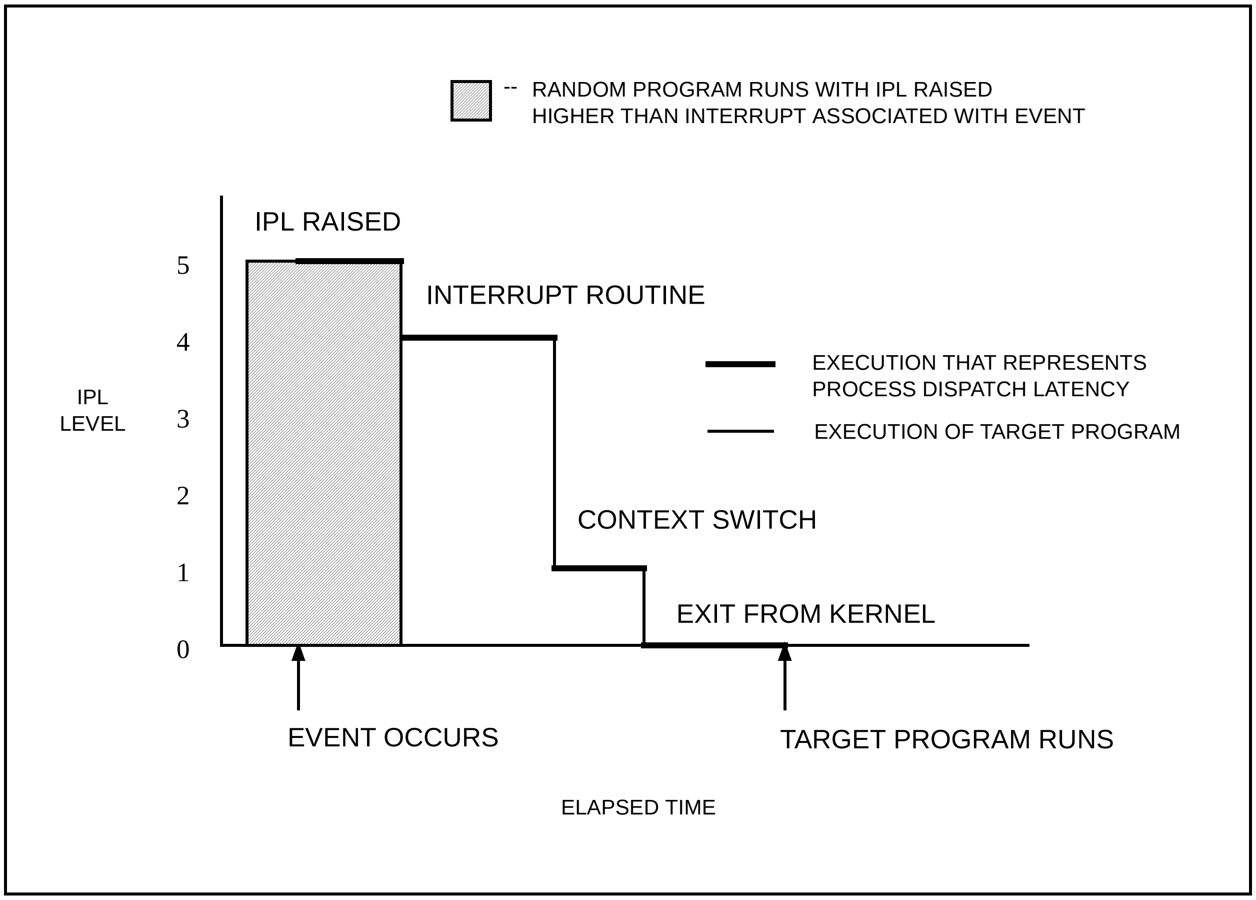

-- RANDOM PROGRAM RUNS WITH IPL RAISED HIGHER THAN INTERRUPT ASSOCIATED WITH EVENT
IPL RAISED
INTERRUPT ROUTINE
EXECUTION THAT REPRESENTS PROCESS DISPATCH LATENCY
EXECUTION OF TARGET PROGRAM
IPL LEVEL
5
4
3
2
1
0
CONTEXT SWITCH
EXIT FROM KERNEL
EVENT OCCURS
TARGET PROGRAM RUNS
ELAPSED TIME

Figure 2-2. Effect of Raised IPL on Process Dispatch Latency

The level to which IPL is raised does not affect the length of time by which process dispatch latency will be extended. The reason is that the context switch to the target process cannot be performed until IPL has been lowered to zero. Figure 2-3 illustrates what happens when IPL is raised to a level that is not as high as the level assigned to the interrupt for which the target process is waiting.

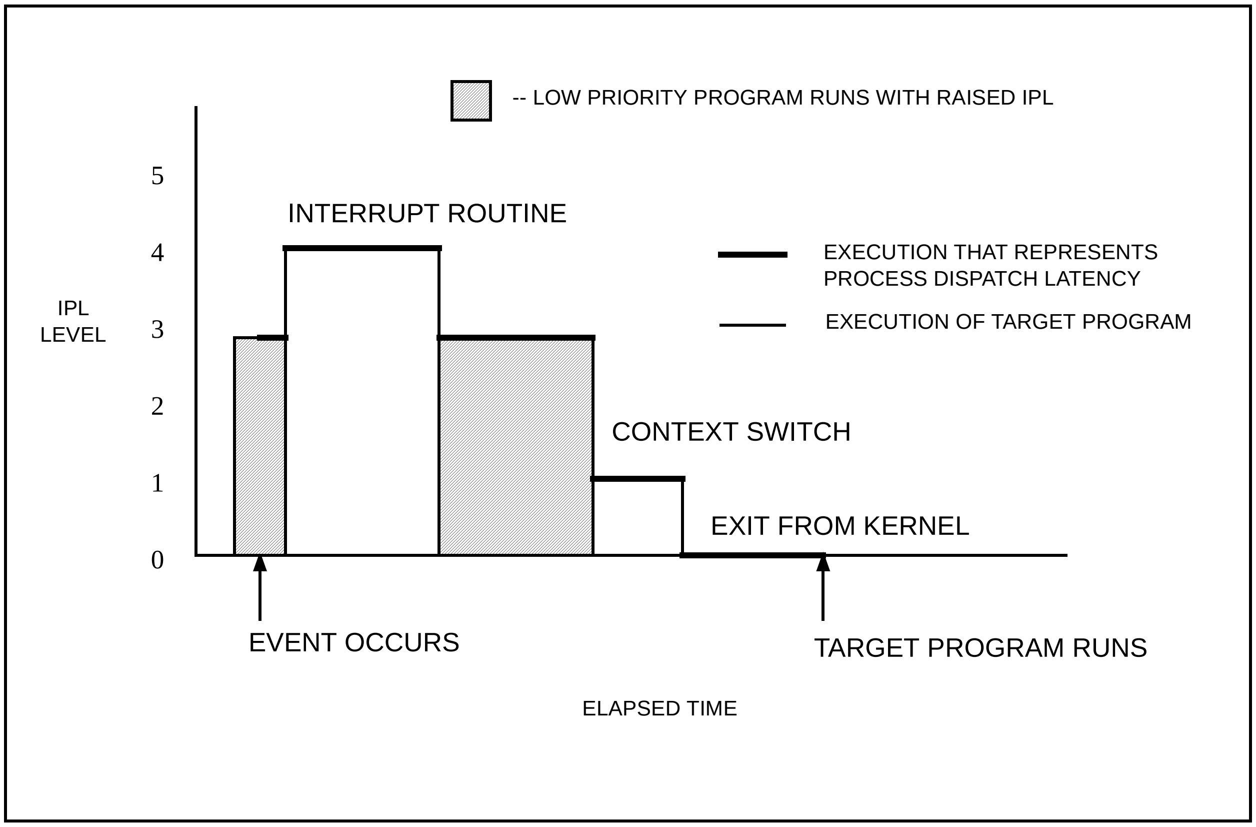

-- LOW PRIORITY PROGRAM RUNS WITH RAISED IPL
IPL LEVEL
5
4
3
2
1
0
INTERRUPT ROUTINE
EXECUTION THAT REPRESENTS PROCESS DISPATCH LATENCY
EXECUTION OF TARGET PROGRAM
CONTEXT SWITCH
EXIT FROM KERNEL
EVENT OCCURS
TARGET PROGRAM RUNS
ELAPSED TIME

Figure 2-3. Effect of Low IPL on Process Dispatch Latency

In the situation depicted in Figure 2-3, the interrupt routine executes as quickly as in the ideal case for process dispatch latency. When the interrupt routine completes its processing, however, the section of code that was executing with raised IPL resumes execution and then lowers the IPL. A context switch cannot be performed until IPL is lowered to zero. Raising the IPL on a processor causes the worst case process dispatch latency to be extended by the length of time that interrupts are held out by the raised IPL.

Effect of Interrupts

Receiving an interrupt affects process dispatch latency in the same way that raising the IPL does because receipt of an interrupt causes the IPL to be raised (the hardware raises the processor's IPL to that of the interrupt being handled). The shaded blocks in Figures 2–2 and 2–3, for example, can as easily represent the execution of interrupt routines as the execution of programs with raised IPL.

When multiple interrupts are assigned to the same CPU, the determinism of process dispatch latency is affected to an even greater extent. The reason is that a CPU can receive multiple interrupts at the same time. When more than one interrupt is active at the same time, interrupts are stacked. When the CPU receives an interrupt from an external device, the hardware raises the IPL for the duration of the related interrupt routine. If a second interrupt occurs whose interrupt priority level is at or below the IPL, it is blocked. When servicing of the first interrupt has been completed, the second interrupt becomes active and is serviced. If a second interrupt whose interrupt priority level is above the current IPL occurs, it is immediately serviced. After servicing of the second interrupt has been completed, processing of the first interrupt continues. In both cases, user processes are prevented from running until all of the pending interrupts have been serviced. When multiple interrupts are assigned to a particular processor, process dispatch latency is less predictable on that processor because of the way in which the interrupts can be stacked. Figure 2-4 illustrates how multiple interrupts affect process dispatch latency.

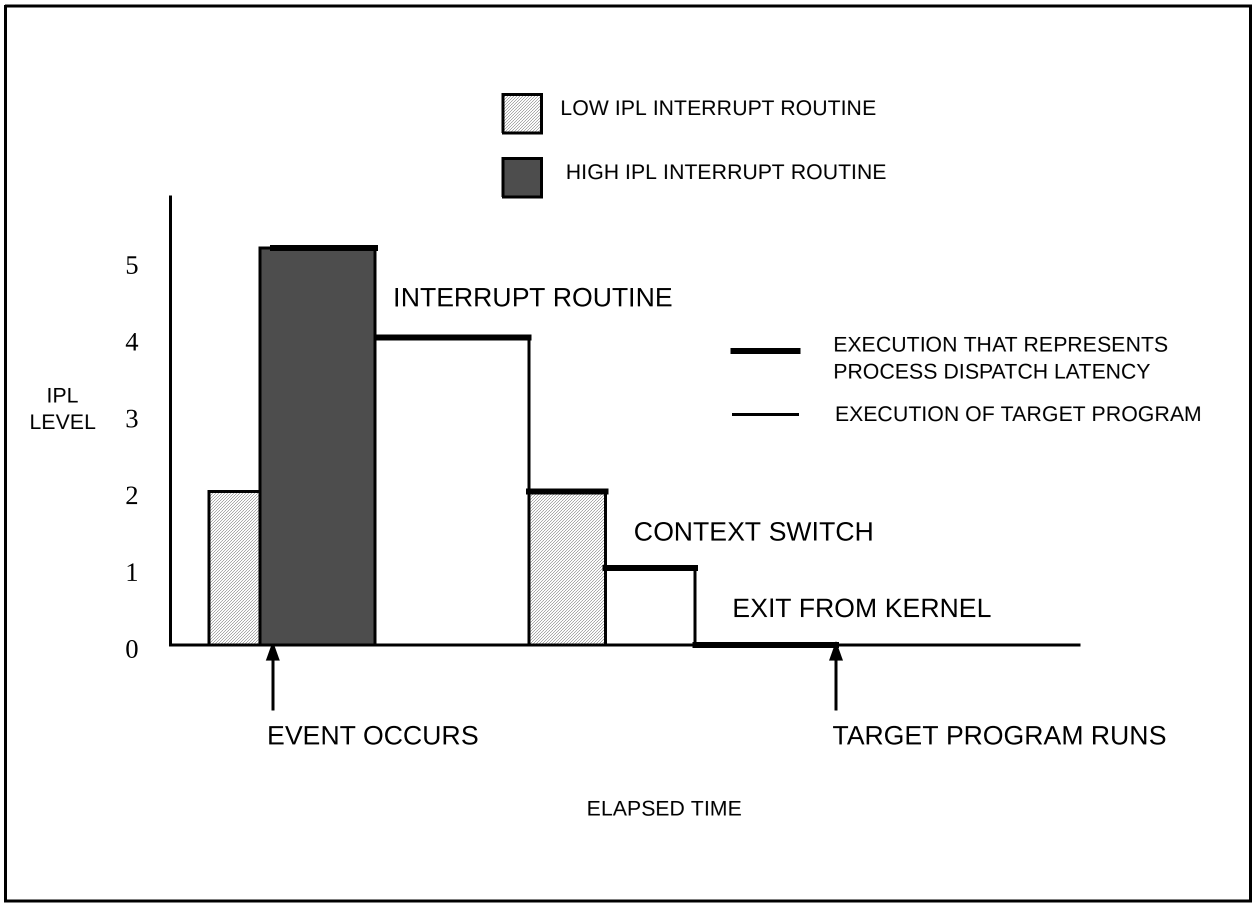


Figure 2-4. Effect of Interrupts on Process Dispatch Latency

The Shielded Processor Model

The *shielded processor model* represents an approach to obtaining the best process dispatch latency that is possible on a particular processor in a multiprocessor system. This model does not apply to uniprocessor systems.

In the shielded processor model, tasks and interrupts are assigned to CPUs in such a way as to guarantee a high grade of service to certain important, real–time functions. In particular, a critical, high–priority task is bound to one or more CPUs, and most interrupts and all system daemons are bound to *other* CPUs. The CPUs responsible for running the high–

priority tasks are *shielded* from the unpredictable processing associated with interrupts and system daemons and are, therefore, called *shielded processors*. Some examples of the types of tasks that should be run on shielded processors are as follows:

- Tasks that require the fastest response time
- Tasks that must be run at very high frequencies
- Tasks that have no tolerance for being interrupted by the operating system

Interrupts have higher priorities than tasks. Thus, shielded processors should receive only those interrupts that signal events that are important to the tasks they run. Other interrupts should be directed to other CPUs.

Generally, any CPU other than the boot processor may serve as a shielded processor. The boot processor is the CPU that initially booted the system. It assumes certain responsibilities that are not shared with other CPUs. Its **`hardclock()`** interrupt routine, for example, keeps track of the time of day, ages entries in the callout queue, and so on. As a result, the amount of time required to service the periodic 60 Hz clock interrupt is greater on the boot processor than on other processors. For this reason, you should consider using the boot processor, and others if necessary, to run the system daemons and most of the interrupt load.

The shielded processor model does not imply a master–slave configuration. All CPUs in the system can share equally in servicing an application’s workload. However, you are guaranteed much better process dispatch latency on the shielded processors when the unpredictable processing associated with interrupts and system daemons can be confined to other CPUs. The shielded processor model does not imply that the shielded processors must be idling most of the time. Background tasks are not a serious impediment to fast and predictable process dispatch latency. You are not required to dedicate a processor to a single task to obtain good process dispatch latency.

It is important to note that it is not essential that you use the shielded processor model for all real–time applications. Some applications may benefit from a more balanced sharing of the workload.

Improving Process Dispatch Latency

You can improve process dispatch latency by using the techniques that are described in this section. The techniques that are essential to the shielded processor model are presented first. Procedures for assigning processes and interrupts to CPUs are explained in”Assigning Processes to CPUs” and “Assigning Interrupts to CPUs,” respectively. Procedures for controlling the processors on which the 60 Hz clock interrupt is serviced are described in “Hardclock Interrupt Handling.” Procedures for assigning certain types of daemons to CPUs are explained in “Assigning Daemons to CPUs.” The extent to which you can improve process dispatch latency by using user–level interrupt routines and user–level device drivers is described in “User–Level Interrupt Routines” and “User–Level Device Drivers,” respectively. The way in which threads scheduling affects response time is explained in “Threads and Response Time.”

Assigning Processes to CPUs

By default, an LWP can execute on any processor in the system. Every LWP has a bit mask, or CPU bias, that determines the processor or processors on which it can be scheduled. An LWP inherits its CPU bias from its creator during a **`fork(2)`** or an **`_lwp_create(2)`** but may thereafter change it.

You can set the CPU bias for one or more LWPs by specifying the **CPU_SETBIAS** command on a call to **`cpu_bias(2)`**, the **MPA_PRC_SETBIAS** command on a call to **`mpadvise(3C)`**, or the **`-b`** *bias* option to the **`run(1)`** and **`rerun(1)`** commands.

To set the CPU bias with **`mpadvise`**, **`cpu_bias`**, and **`rerun`**, the following conditions must be met:

- The real or effective user ID of the calling process must match the real or saved user ID of the LWP for which the bias is being set, or the calling process must have the `P_OWNER` privilege.
- To add a CPU to an LWP's CPU bias, the calling process must have the `P_CPUBIAS` privilege.

  Note that the `P_CPUBIAS` privilege is also required to use the **`run`** command for this purpose.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

> The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

You can change the CPU assignment for one or more LWPs by specifying the **CPU_SETRUN** command on a call to **`cpu_bias(2)`**, the **MPA_PRC_SETRUN** command on a call to **`mpadvise(3C)`**, or the **`-c`** *cpu_id* option to the **`rerun(1)`** command.

To change an LWP's CPU assignment, the following conditions must be met:

- The real or effective user ID of the calling process must match the real or saved user ID of the LWP for which the CPU assignment is being changed, or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

> The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

A CPU bias can be assigned to the **`init(1m)`** process. All general processes are a descendant from **`init(1m)`**. As a result, most general processes would have the same CPU bias as or a subset of **`init(1m)`**. Only privileged processes (as described above) are able to increase their CPU bias. Assigning a restricted CPU bias to **`init(1m)`**, tends to restrict all general processes to the same subset of CPUs as **`init(1m)`**. The exception is selected processes that have the appropriate privilege. The tunable `ALLBIAS` is used to specify an initial CPU bias for **`init(1m)`**.

For additional information on use of the **`cpu_bias(2)`** system call, the **`mpadvise(3C)`** library routine, and the **`run(1)`** and **`rerun(1)`** commands for these purposes, refer to the *PowerMAX OS Programming Guide* and the corresponding system manual pages.

Assigning Process to CPUs Via Exclusive Binding

An alternative way of assigning LWPs to CPUs is to exclusively bind them. A CPU which has LWPs exclusively bound to it will run only those LWPs plus any system daemons assigned to it. LWPs which are not exclusively bound will avoid the CPU, even if the CPU is mentioned in their biases. We call these "merely biased" LWPs *unbound* LWPs.

An exclusive binding may be done with the **`SET_CPUXBIAS`** command of **`cpu_bias(2)`**, the **`MPA_PRC_SETXBIAS`** command of **`mpadvise(3c)`**, or from the shell with **`pexbind(1)`** or with the **`-x`** option of **`run(1)`** and **`rerun(1)`**.

To turn an exclusively bound LWP back into an unbound LWP, simply bias it back to the desired set of CPUs, as discussed in the previous section. The bias must contain at least one general purpose (i.e. non-exclusive) cpu.

An exclusive binding is inherited by a child upon creation, and across **`exec(2)`**.

A LWP may be exclusively bound to only one CPU at a time.

An exclusive-use CPU can still run those system daemons which have that CPU in their biases. Use the techniques discussed in the Shielded Processor section to prevent most of these daemons from running on shielded processors.

Exclusive binding is subject to all the privilege requirements of simple biasing, discussed in the previous section, with the following addition: the `P_CPUBIAS` privilege is always required to make a LWP exclusively bound, or to release such a binding.

For additional information see the man pages on the commands and library services mentioned above.

Assigning Interrupts to CPUs

You can lessen the effect of interrupts on the determinism of process dispatch latency for a particular processor if you have the capability of directing interrupts to the other CPUs. Process dispatch latency is important to a high–priority process that will block waiting for an external event that is delivered via an interrupt. This type of process should be bound to the CPU where the interrupt associated with the external event will be received. Other interrupts should be directed away from this CPU.

Statically Configuring Interrupt Assignments

On PowerMAX OS systems, you have the capability of assigning external interrupts to specific CPUs. Each external interrupt is connected to a particular interrupt source (**intsrc**) line, and these **intsrc** lines are connected as inputs into the interrupt controller.

The interrupt controller hardware is configured by the operating system to direct each incoming **intsrc** line to a specific CPU.

The operating system controls the interrupt priority associated with each **intsrc** line, but the user can control which CPU receives interrupts coming in from a particular **intsrc** line by using the **`idtune(1M)`** or **`config(1M)`** command to modify the value of the corresponding kernel tunable that controls this intsrc line CPU assignment.

The **`idtune(1M)`** or **`config(1M)`** utilities should always be used to modify any kernel tunable. However, the user may obtain information about all the kernel tunable interrupt source assignments by directly reading the **`/etc/conf/mtune.d/bspall`** tunable file. This file contains the kernel tunable names and their descriptions, of all the kernel tunables that are related to CPU interrupt source assignments. The user should take care not to modify this file, since it also contains the default values of various kernel tunables.

Once the user has modified one or more of these tunables with **`idtune(1M)`** or **`config(1M)`**, they should then rebuild their kernel and reboot the system in order for these new CPU assignments to take affect.

NOTE

There is a quick and easy kernel configuration tunable that may suit many real-time environments. When the tunable ALLINTSONCPU0 is enabled, then all configurable interrupts that are set to use the round-robin algorithm (which is the default kernel tunable setting for interrupt sources) will be directed to the boot CPU (CPU0).

Note that even when the ALLINTSONCPU0 tunable is enabled, it is still possible to explicitly assign specific interrupts to CPUs other than CPU0, with all other interrupts being assigned to CPU0.

On multi-CPU board systems, there may be a small number of architectural-specific interrupts that must be assigned to a CPU on a specific CPU board. In this case, these interrupts will be directed to the first CPU on the board.

The effect of this tunable may be too general for some systems. If this is the case, this tunable should not be used; instead the individual tunables for each interrupt source line should be used.

Dynamically Modifying Interrupt Assignments

On Power Hawk Series 700 and Series 900 systems you may also change CPU interrupt source assignments dynamically, without the need for statically reconfiguring, rebuilding and rebooting the kernel. Dynamically changing CPU interrupt source assignments is usually accomplished via the **`intconfig(1M)`** utility; however, the user may also change these assignments directly through the use of the **`syscx(2)`** system service. Both of these methods are described below.

Using intconfig(1M)

Dynamic CPU interrupt assignments may be accomplished through the use of the **`intconfig(1M)`** utility. However, note that any CPU interrupt re-assignments that are made with **`intconfig(1M)`** remain in effect only until the next system reboot or until a different assignment is made with **`intconfig(1M)`**. During the next system reboot, the CPU interrupt assignments will be made by using the usual static kernel tunable interrupt configuration values.

The **`intconfig(1M)`** utility is intended as an aid for more quickly and easily determining the optimal CPU interrupt assignment configuration for a user's application by repetitively running the application, changing CPU assignments, running the application again and observing any differences, and so forth. Once a user has determined the optimal CPU interrupt source assignments for their application, they may either reconfigure and rebuild their kernel, or they may invoke **`intconfig(1M)`** in a system startup script so that the desired explicit dynamic CPU interrupt assignments are made during every system boot.

In addition to changing the interrupt CPU assignments, the **`intconfig(1M)`** utility may also be used to query or disable the CPU assignment of any configurable interrupt source in the system.

See the **`intconfig(1M)`** system manual page for more information on these additional features.

When **`intconfig`** is invoked with no options, a table is output to stdout that contains an entry for each configurable interrupt source in the system. Each table entry contains the following information:

- The index value of this entry. This index may be used when specifying a specific interrupt source index to query or modify.
- The corresponding kernel tunable name of this interrupt source.
- A description of the interrupt source.
- The current CPU assignment of this interrupt source.
- An indication of whether or not this interrupt source may be disabled.

A sample of the output from invoking **`intconfig`** with no options is shown in Table 2-1.

Table 2-1. Sample Output of `intconfig` with no Options

Index	Tunable Name	Description	Current CPU Assignment	May Disable?
0	RPIC_RTC_0C0	rtc 0c0	1	No
1	RPIC_SERIAL_A	serial port A	0	Yes
2	RPIC_SERIAL_B	serial port B	1	Yes
3	RPIC_PMC_INTA	PMC S1 IntA/S2 IntC	1	Yes
4	RPIC_PMC_INTB	PMC S1 IntB/S2 IntD	0	Yes

Table 2-1. Sample Output of `intconfig` with no Options (Cont.)

Index	Tunable Name	Description	Current CPU Assignment	May Disable?
5	RPIC_PMC_INTC	PMC S1 IntC/S2 IntA	1	Yes
6	RPIC_PMC_INTD	PMC S1 IntD/S2 IntB	0	Yes
7	RPIC_INTA_VME6	PCI IntA/VME level 6	1	Yes
8	RPIC_INTB_VME5	PCI IntB/VME level 5	0	Yes
9	RPIC_INTC_VME4	PCI IntC/VME level 4	1	Yes
10	RPIC_INTC_VME3	PCI IntD/VME lev 321	0	Yes
11	RPIC_VME7_DMAC	VME level 7/DMAC	1	Yes
12	RPIC_VME_SWIACK	VME Software Int Ack	0	No
13	RPIC_GT_ETH	GT64260 Ethernet	0	Yes
14	RPIC_GT_ERR	GT64260 errors	1	No
15	RPIC_VME_ERR	VME buserr/powerfail	0	No
16	RPIC_RTC_0C1_2	rtc 0c1/0c2	1	No
17	RPIC_RTC_0C3_4	rtc 0c3/0c4	0	No
18	RPIC_RTC_0C5_6	rtc 0c5/0c6	1	No
19	RPIC_RTC_0C7_8	rtc 0c7/0c8	0	No

NOTE

The **intsrc** index values do not change across system reboot or kernel rebuilds. However, the index value for a given interrupt source tunable MAY change between installations of PowerMAX OS patches or future releases.

The user may dynamically assign a single interrupt source to a specific CPU through the use of the '**`-a`**' assign option. For example, to assign the real-time clock **`/dev/rrtc/0c0`** to CPU1 (based on the example values in Table 2-1), use the following options to **`intconfig`**:

```
intconfig -a 1 -i 0
```

where '**`-a 1`**' indicates CPU1, and '**`-i 0`**' indicates **intsrc** index 0)

You may also assign ALL configurable interrupts to a specific CPU by using the **`-A`** option. For example:

```
intconfig -A 0
```

will assign all configurable interrupts to CPU0.

It is also possible to 'shield' specific CPU(s) from all configurable interrupt sources with the **`-S`** option. When this option is used, then any configurable interrupt sources that are currently assigned to the specified CPU(s) are re-assigned to the other CPUs in the system.

Unlike other **`intconfig`** options, this option uses a hexadecimal CPUID bitmask instead of a CPU ID value, so that more than one shielded CPU may be specified. However, note that at least one CPU must remain unshielded; therefore, on systems with only 2 CPUs, this CPUID mask may contain only 1 bit.

The following example will shield CPUs 2 and 3 on a system that has 4 CPUs:

```
intconfig -S 0xc
```

And the following example will shield CPU 1 on a system that has 2 CPUs:

```
intconfig -S 0x2
```

When there are two or more CPUs in the system that are not specified in CPUID mask, then the interrupt sources that are moved off of the CPU(s) in CPUID mask are assigned to the other non-shielded CPUs in a round-robin fashion.

For information, refer to the **`intconfig(1M)`** system manual page.

Using syscx(2) GET_PIN_CPU and SET_PIN_CPU

While the **`intconfig(1M)`** utility should meet the requirements of most users, it is also possible for users to query and modify interrupt source CPU assignments directly, through the **`syscx(2)`** system service interface.

The GET_PIN_CPU command may be used to query/get a specific interrupt source CPU assignment, and the SET_PIN_CPU command may be used to set the interrupt source CPU assignment of a specific interrupt source.

For example, to query the current CPU assignment of the interrupt source index 2 (this would be serial port B if we use Table 2-1):

```
int retval, cpu_id;

retval = syscx(GET_PIN_CPU, 2, &cpu_id);
```

where the CPU ID of the cpu that is currently assigned to this interrupt source will be returned in the location “cpu_id” if a successful status value of 0 is returned in “`retval`”.

Similarly, to change the CPU assignment for interrupt source 2, the following SET_PIN_CPU call would be used:

```
retval = syscx(SET_PIN_CPU, 2, 1);
```

where the above call will assign interrupt source index 2 to CPU1, if a value of 0 (success) is returned in “`retval`”.

See the GET_PIN_CPU and SET_PIN_CPU commands information in the **`syscx(2)`** system manual page for more details on how to use these interfaces.

As was previously mentioned, the **intsrc** index values used on these **`syscx(2)`** commands will not change across system reboots or kernel rebuilds. However, the index value for any given interrupt source tunable MAY change between new PowerMAX OS patches or future releases.

CPU interrupt source assignment changes that are made with the **`syscx(2)`** system service remain in effect only until the next system reboot. These interrupt CPU assignments will be re-evaluated during the next system boot using the standard kernel static configuration method.

Querying the Interrupt Configuration

You can determine the CPU to which a particular interrupt is directed by using the **`intstat(1M)`** utility. This utility gathers information about all interrupts associated with configured devices and software interrupts associated with the operating system. For each, the intstat utility provides the following information: (1) the name of the device or interrupt service routine, (2) the interrupt vector number, (3) the interrupt priority level, (4) the assigned I/O bus (where applicable), (5) the (H)VME interrupt level (where applicable), and (6) the CPU to which the interrupt is directed.

By default, the intstat utility sorts the output information according to interrupt vector number and displays it on the terminal screen. Options that you can specify to the utility make it possible for you to develop your own input file; obtain information for all types of interrupts; have the output information sorted by interrupt priority level, (H)VME interrupt level, or CPU number; and supply the name of the running kernel.

The format for executing the **`intstat`** utility is as follows:

`intstat` [**`-l`**] [**`-a`**] [**`-x`**] [**`-p`** *or* **`-v`** *or* **`-c`**] [**`-n`** *kernel_name*] [**`-d`** *dump_file*] \
[**`-t`** *input_file*]

Options are described in Table 2-2.

Table 2-2. Options to the intstat Utility

Option	Description
`-l`	list the interrupt levels for each I/O bus on the system and the CPU to which interrupts at each level are directed
`-a`	Lists for all interrupt service routines defined in the system— both those associated with device interrupts and those associated with software interrupts: - Associated devices or routines - Interrupt vector numbers Interrupt priority levels - Assigned I/O bus (where applicable) - (H)VME interrupt levels (where applicable) - CPUs to which the interrupts are directed
`-x`	list the interrupt vector number and the interrupt priority level as hexadecimal numbers
`-p`	sort the output list according to interrupt priority level
`-v`	sort the output list according to (H)VME interrupt level

Table 2-2. Options to the intstat Utility (Cont.)

Option	Description
-c	sort the output list according to CPU number
-n *kernel_name*	use the kernel specified by *kernel_name* The default kernel is **/stand/unix**. If the kernel running on your system is different from the default, specify this option.
-d *dump_file*	dump the list of interrupt routines being compared with the symbol table to the file specified by *dump_file* You can edit the list obtained by specifying this option to include the names of interrupt routines associated with devices that have been added to your system and are not supported by the **intstat** utility. By using the **-t** option, you can then use the edited file as input to **intstat**.
-t *input_file*	use the file specified by *input_file* to obtain the list of interrupt routines to be compared with the symbol table

For additional information, refer to the **intstat(1M)** system manual page.

Hardclock Interrupt Handling

The system-wide 60 Hz clock interrupts every CPU in the system 60 times per second. Receiving this interrupt on a shielded processor affects worst case process dispatch latency (see “Effect of Interrupts” for an explanation of the way in which interrupts affect process dispatch latency). The 60 Hz clock interrupt routine is one of the most frequently executed interrupt routines in the system. The frequency with which it is executed does not generally coincide with the frequency with which the cyclically-scheduled processes of a real-time application run; the 60 Hz clock interrupt is, therefore, a source of random jitter to the application.

The boot processor serves as the system timekeeper; it is responsible for maintaining system time and the callout queue. The callout queue is a list of kernel functions that must be invoked when a certain amount of time expires. Because of these timekeeping responsibilities, hardclock, the operating system mechanism that services the 60 Hz clock interrupt, performs system-wide timing functions on the boot processor that it does not perform on the other processors in the system. You can configure your system so that some of these system-wide functions are performed by a system daemon instead of an interrupt routine. The availability of this option is particularly helpful on a uniprocessor system, where the shielded processor model does not apply, because it allows you to reduce the amount of work that is performed at interrupt level.

You can configure a daemon to control system-wide timing functions on the boot CPU by setting the value of the system tunable parameter TODCINTRDAEMON. For a complete description of the system-wide timing functions performed by hardclock and the procedures for using a system daemon to perform some of these functions, refer to Chapter 9 of this guide.

On each CPU in the system, hardclock also performs a number of functions for the currently running process. You can improve worst case process dispatch latency on PowerMAX OS systems by disabling these local timing functions on a shielded processor. You may use the **`mpadvise(3C)`** library routine, the **`hardclock(1m)`** command or the tunable `CLOCKINTR` to control the processors on which the local timing functions are performed. For a complete description of the local timing functions performed by hardclock and the procedures for using the system call and the utility to disable these functions on selected CPUs, refer to Chapter 9 of this guide.

CAUTION

Disabling servicing of the 60 Hz clock interrupt affects certain aspects of process scheduling. It also affects the manner in which a number of system calls, routines, commands, and utilities function. Refer to Chapter 9 of this guide for details.

Using Interrupt Daemons

The length of time that is spent in an interrupt routine directly affects the worst case process dispatch latency. The reason is that the operating system almost always executes an interrupt routine before any user-level activity on a particular processor. The only exceptions are user-level interrupt routines and user code that is executed with the IPL raised above the level of the incoming interrupt. For some applications, the work performed in an interrupt routine is actually of lower priority than the highest priority tasks on the system. An interrupt daemon provides you with a means of ensuring that a user process has higher priority than the processing performed by an interrupt routine.

PowerMAX OS provides you with the option of using a kernel daemon to handle processing of interrupts from certain controllers. Although this capability is provided on both uniprocessor and multiprocessor systems, it is intended primarily for use on uniprocessor systems. The reason is that on uniprocessor systems, the shielded processor model does not apply; that is, interrupts and tasks cannot be directed to other CPUs. To improve process dispatch latency on uniprocessor systems, you must attempt to minimize the length of time that is spent in interrupt routines. Interrupt daemons afford you this capability.

On multiprocessor systems, interrupt daemons provide you with a means of removing the interrupt workload from a processor that is not shielded. Interrupt daemons are not commonly used for this purpose because of the resultant degradation in I/O throughput; generally system administrators are willing to sacrifice response time on one of the processors for high I/O throughput on the others, however.

You configure an interrupt daemon into the kernel by setting the value of the associated tunable parameter. Table 2-3 presents the controllers for which interrupt daemons can be configured on PowerMAX OS systems:

Table 2-3. Controllers with Associated Interrupt Daemons

Description	System	Tunable Parameter
IDE adapter	PowerStack	`IDEINTRDAEMON`
STREAMS scheduling	All	`STRSCHED_DAEMON_MSK`
Hardclock interrupt	All	`TODCINTRDAEMON`
MPEG decoder (cld)	PCI or PMC bus systems	`CLDINTRDAEMON`
AMD ethernet (amd)	PCI or PMC bus systems	`AMDINTRDAEMON`
GT64260 ethernet (gte)	Power Hawk Model 910/920 systems	`GTEINTRDAEMON`
DEC ethernet (dec)	Power Hawk Series 600 or PMC bus systems	`DECINTRDAEMON`
MV64460 ethernet (mve)	PowerHawk Model 940 systems	`MVEINTRDAEMON`
PMC FDDI (ip)	PMC bus systems	`IPINTRDAEMON`
Symbios ethernet (sym)	Power Hawk Series 700	`SYMINTRDAEMON`
Fibre channel (fibre)	PCI or PMC bus systems	`FIB_INTR_DAEMON`
RAMiX ethernet (rmxf)	PMC bus systems	`RMXFINTRDAEMON`
NCR SCSI (ncr)	Power Hawk systems	`NCRINTRDAEMON`
Zoran MPEG Decoder (zld)	PCI bus systems	`ZLDINTRDAEMON`
Symbios DMA (sym_dma)	Power Hawk Series 700 Client CCS systems	`SYM_DMA_INTR_DAEMON`
Condor ethernet (cnd)	VME bus systems	`CNDINTRDAEMON`
VME Peregrine FDDI (pg)	VME bus systems	`PGINTRDAEMON`
VME DMA (dmac)	PCI-to-VME64 bridge (Universe) systems	`DMAC_DAEMON`
High Speed Data (hsde)	VME bus systems	`HSDEINTRDAEMON`
HSA/VIA SCSI (via)	VME bus systems	`VIAINTRDAEMON`
Systech Serial (hps)	VME bus systems	`HPSINTRDAEMON`
Async Comm MUX (mvcs)	VME bus systems	`MVCSINTRDAEMON`
High Speed Data (vhsd)	VME bus systems	`VHSDINTRDAEMON`

To ensure that interrupts from one of these controllers are handled by a system daemon, set the value of the associated system tunable parameter to **`1`**. If you wish to ensure that all interrupt daemons are enabled, you may set the value of the `ALLINTRDAEMONS` system tunable parameter to 2.

You can use the **`config(1M)`** utility to (1) determine whether the values of these parameters have been modified for your system, (2) change the values of these parameters, and (3) rebuild the kernel. Note that you must be the root user to change the value of a tunable parameter and rebuild the kernel. After rebuilding the kernel, you must then reboot your system.

Assigning Daemons to CPUs

You can improve worst case process dispatch latency on a given CPU by ensuring that certain daemons run on other CPUs. You may wish to ensure that these daemons run on the boot CPU. The boot CPU has the least amount of determinism because hardclock, the interrupt mechanism for the system–wide 60 Hz clock interrupt, performs system–wide timekeeping functions only on the boot CPU.

Because daemons may raise the IPL on the processor on which they run, it is important that you assign them to processors that are not shielded. As explained in “Effect of IPL,” raising the IPL on a particular processor directly affects worst case process dispatch latency for that processor.

The only way that you can affect the CPU bias of a system daemon is to change the value of the system tunable parameter associated with that daemon’s CPU bias. You can examine and modify the values of system tunable parameters associated with system daemons by using the **`config(1M)`** utility. For an explanation of the procedures for using this utility, refer to the “Configuring and Building the Kernel” chapter of *System Administration Volume 2*. Note that after changing a tunable parameter, you must rebuild the kernel and then reboot your system. If you wish to change the CPU bias of all of the system daemons, you may use the **`daemon_tune(1M)`** command or the tunable `ALLDAEMONSBIAS`. In this case also, you must then rebuild the kernel and reboot your system.

Controlling STREAMS Scheduling

PowerMAX OS uses the STREAMS facility to transfer all networking and terminal data to and from the external devices that are connected to the system. STREAMS scheduling transfers queued and in-bound data between the modules of a STREAM by the execution of STREAMS service procedures. By default, STREAMS scheduling is performed in an interrupt routine on all processors in the system. Performing STREAMS scheduling at interrupt level provides optimal performance for the STREAMS facility, but it may not be desirable for real-time systems. Because interrupts are always at a higher priority than program-level activity, STREAMS scheduling can interfere with the activity of time-critical real-time tasks.

Selecting CPUs and Local Daemons

PowerMAX OS provides the flexibility to trade STREAMS performance for determinism for high-priority real-time tasks. The system administrator can (1) select the processors on which STREAMS scheduling is executed and (2) determine whether or not STREAMS scheduling is performed in an interrupt routine.

Note

The STREAMS scheduler will not usually execute STREAMS service procedures on CPUs that are currently exclusively bound. The one exception to this rule is discussed in "the Initialization of the per-STREAM CPU Bias Mask" on page 2-19.

Control over STREAMS scheduling is provided in the form of system tunable parameters. These parameters are identified as follows:

`STRSCHED_DISABLE_MSK`	allows STREAMS scheduling to be disabled on selected processors
`ALLINTRDAEMONS`	Configures all interrupt daemons. In addition, STREAMS scheduling is performed by the local system daemon on all processors. (i.e. as if `STRSCHED_DAEMON_MSK` was configured for all processors.)
`STRSCHED_DAEMON_MSK`	allows STREAMS scheduling to be performed by the **`local`** system daemon on selected processors

In each case, the value of the tunable parameter is a bit mask in which bits 0 through 7 correspond to processors 0 through 7. The default value for each parameter is zero. Processors are selected by changing the value of the parameter to a hexadecimal number that sets the bits corresponding to the desired processors in the mask. System tunable parameters may be changed by the system administrator. You can use the **`config(1M)`** utility to obtain more detailed information on the STREAMS scheduling parameters and constraints on their use.

When determining how to handle STREAMS scheduling on the processors in your system, the following should be considered:

- STREAMS scheduling should not be performed at interrupt level on a shielded processor.

  The interrupt-level activity associated with STREAMS scheduling occurs at random times. It will interfere with the execution of a real-time task running on a shielded processor.

- STREAMS scheduling may be performed by the **`local`** daemon on a shielded processor.

  The **`local`** daemon is scheduled under the system scheduler class; as a result, it does not interfere with the execution of processes scheduled under the fixed-priority class.

- If the throughput of STREAMS devices is important in your system, at least one processor should be configured to execute STREAMS scheduling at interrupt level.

  The number of processors that should be configured to perform STREAMS scheduling at interrupt level depends upon the I/O load and throughput requirements of your system. The optimum configuration can be determined only through experimentation.

Note that the processors that perform STREAMS scheduling at interrupt level are always used first when a STREAMS queue is ready for processing.

Per-STREAM CPU Biasing of Service Procedures

On systems with more than one CPU, in addition to being able to select the set of CPUs that execute all STREAMS service procedures with the `STRSCHED_DISABLE_MSK`, some situations may require a higher level of control over which CPUs execute each STREAM's own service procedures.

When the `STRSCHED_SERVBIAS` tunable is set to 1, then per-STREAM CPU biasing of service procedures is enabled in the kernel. When this feature is enabled, then each STREAM in the system has its own CPU bias mask, where this mask defines the set of CPUs that may execute this STREAM's service procedures.

The following scenarios show some of the possible reasons for making use of the per-STREAM CPU biasing of service procedures feature:

- A system is configured with two Ethernet devices. While both of these Ethernet interfaces are used for TCP/IP traffic, one Ethernet device is used for real-time traffic, and the other Ethernet device is used for background non-real-time TCP/IP traffic.

  In this situation, it would be desirable to split up the servicing of networking STREAMS service procedures between CPUs based on real-time or non-real-time traffic. That is, the real-time CPU(s) would execute just the STREAMS service procedures that are related to the real-time networking traffic.

- A system may have just one Ethernet device. A real-time CPU may make use of a direct Data Link Provider Interface (DLPI) application for sending and receiving data through this Ethernet device, while the non-real-time CPU(s) may make use of this same Ethernet device for background TCP/IP traffic. In this case, it might be desirable to have the STREAMS service procedures of the DLPI application executed on the real-time CPU, and the other background TCP/IP service procedures executed on non-real-time CPUs.

While these two examples are networking-based examples, the per-STREAM CPU biasing of service procedure executions may be used to provide control for all types of STREAMS, including STREAMS contained within STREAMS multiplexor (MUX) modules, as well as customer-written STREAMS modules, drivers and STREAM stack configurations.

Initialization of the per-STREAM CPU Bias Mask

If `STRSCHED_SERVBIAS` is enabled, then when a new STREAM is opened, that STREAM's service procedure CPU bias mask will be set to the user-visible CPU bias mask of the calling LWP that is opening that STREAM.

Bits that represent engines that are in the `STRSCHED_DISABLE_MSK` are removed from the CPU bias mask.

All service procedures within that STREAM will be executed only by one of the CPUs within that STREAM's CPU bias mask.

Generally speaking, exclusively bound CPUs will be dynamically removed from consideration by the STREAMS scheduler each time a CPU is selected to execute a STREAM's service procedures. The one exception to this rule (only when `STRSCHED_SERVBIAS` is enabled) is that if the LWP making the **`open(2)`** call is currently exclusively bound to a CPU. In this special case, only that exclusively bound CPU will execute this STREAM's service procedures.

Note that when the resulting CPU bias mask for a STREAM's service procedures contains all the CPUs in the system that are not in the `STRSCHED_DISABLE_MSK`, then the STREAMS scheduler method defaults back to the normal STREAMS scheduler method; that is, for this particular STREAM, it will be as if the `STRSCHED_SERVBIAS` tunable was not enabled.

Changing the per-STREAM CPU Bias Mask

Two **`ioctl(2)`** commands, `I_GETSERVBIAS` and `I_SETSERVBIAS`, are available for getting and setting a STREAM's CPU bias mask, respectively. The `I_SETSERVBIAS` command may be used by a process to change the CPU bias masks of any STREAM that it currently has **`open(2)`**. However, this **`ioctl(2)`** command may not be used to change the CPU bias mask of STREAMS currently opened by other processes. See the **`streamio(7)`** man page for more details on the `I_GETSERVBIAS` and `I_SETSERVBIAS` **`ioctl(2)`** command.

Note that when the user-visible CPU bias of a process or a LWP within that process is modified, the CPU bias masks associated with all the STREAMS currently opened by that process will NOT change. In this case, the `I_SETSERVBIAS` **`ioctl(2)`** command may be used to change the CPU bias masks of the process's open STREAMS, if this is required.

Setting Kernel Tunables

To enable the per-STREAM CPU biasing of service procedures, the kernel tunable `STRSCHED_SERVBIAS` should be set to 1 with the **`config(1M)`** utility program in order to enable per-STREAM service procedure CPU biasing. When this tunable is set to 0 (the default), then per-STREAM CPU biasing is disabled, and the normal system-wide STREAMS scheduler CPU selection method is used.

When `STRSCHED_SERVBIAS` is enabled, then the `STRSCHED_DISABLE_MSK` and `STRSCHED_DAEMON_MSK` tunables still function in the same way as before. However, the following tunables behave differently when the `STRSCHED_SERVBIAS` tunable is enabled:

`STRSCHED_GLOBAL_MSK`

This tunable is used by the kernel only when the `STRSCHED_SERVBIAS` tunable is also enabled. This tunable contains a bit mask of those engines that are permitted to process STREAMS service procedures of STREAMS that have a CPU bias mask containing all engines in the system (minus any `STRSCHED_DISABLE_MSK` CPUs). CPUs not in the `STRSCHED_GLOBAL_MSK` will process only STREAMS service procedures of STREAMS that are biased to a subset of CPUs in the system. By default, `STRSCHED_GLOBAL_MSK` is set to `0xffff` so that all engines may process system-wide biased STREAMS service procedures.

However, in real-time system configurations, system administrators may wish to remove from this mask any engines that are shielded for execution of only time critical applications.

`STRNSCHED`

When `STRSCHED_SERVBIAS` is enabled, this tunable still defines the maximum number of STREAMS service procedures that a CPU may execute for STREAMS with system-wide CPU bias masks, in any single invocation by the STREAMS scheduler.

When the `STRSCHED_BACKOFF` tunable is non-zero, then this tunable also applies to service procedures of STREAMS with biased CPU masks (CPU masks with less than all CPUs in the system).

However, when the `STRSCHED_BACKOFF` tunable is zero, then a CPU will continue to execute STREAMS service procedures of STREAMS with biased CPU masks until no more biased STREAMS service procedures remain to be processed, within a single invocation of the STREAMS scheduler.

`STRSCHED_BACKOFF`

When this tunable is non-zero, and a CPU has executed at least `STRNSCHED STREAMS` service procedures, but there are still service procedures of CPU biased (not system-wide CPU bias masks) STREAMS that need processing, then this tunable defines in HZ ticks, the amount of time that a CPU should stop processing CPU biased (not system-wide biased) service procedures and return to other activities before returning back to execute additional STREAMS service procedures.

The default value for this tunable is 0. In this case, a CPU will continue to execute service procedures of STREAMS that have a CPU bias mask with less than all the CPUs in the system, until no more of these types of service procedures remain queued for execution by that CPU.

This tunable will be ignored if the `STRSCHED_SERVBIAS` tunable is not also enabled.

Biasing init(1)

If the **`init(1M)`** program's CPU bias is modified during system initialization with a **`rerun(1)`** command in the **`/etc/rc2.d/S03rerun`** script, then all STREAMS that are created by **`init(1M)`** at system initialization time will have the CPU bias associated with init's user-visible CPU bias mask. This approach can be particularly useful on systems where it is desirable to place most networking daemons and networking-related STREAMS service procedure executions on a subset of CPUs.

Alternatively, or in addition to this approach, the STRSCHED_GLOBAL_MSK can be set to a subset of all the CPUs in the system so that any remaining system-wide CPU biased STREAMS service procedures will execute only on a subset of CPUs. This can be useful for handling any STREAMS that have already been created at the point that **init(1M)**'s CPU bias is modified with the **rerun(1)** command.

Bound STREAMS

Some STREAMS modules and drivers that are not multi-processor safe, or that must be executed on a particular single CPU, are 'bound' to a single CPU (see **devflag(D1)** and **Master(4)**). Bound STREAMS may only be executed on one CPU, including the execution of all of that STREAM's service procedures. Even when the STRSCHED_SERVBIAS tunable is enabled, the service procedures of bound STREAMS may execute only on one statically selected engine; the CPU bias mask for these STREAMS may not be modified. Further, the engine that a 'bound' a STREAM executes upon is not based upon the calling LWP's user-visible CPU bias mask.

Biasing with strmuxbias(1M)

Although STREAMS that are created and linked onto a multiplexor (MUX) module contain the STREAMS service procedure CPU bias mask based on the LWP that created these STREAMS, sometimes some additional control may be desirable. (See the STREAMS Modules and Drivers Manual for more information about multiplexors and their use.)

In general, applications that open and create multiplexing STREAMS stack configurations often build these configurations within the scope of one process. Rather than require that these applications and scripts be written to take per-STREAM CPU biasing masks into account when building these STREAMS configurations, the **strmuxbias(1M)** utility is provided instead, for changing the STREAM CPU bias masks of these lower level STREAMS after they have already been constructed.

Note that using **strmuxbias(1M)** to change lower level MUX STREAMS is optional; per-STREAM CPU biasing of service procedures will function properly without making any CPU bias mask adjustments with **strmuxbias(1M)**.

When multiplexors are connected to several lower level STREAMS, the use of these lower level STREAMS is usually shared among the upper level STREAMS. Therefore, system administrators may want to set different CPU biases for each of these lower level STREAMS after these STREAMS have already been created and linked onto a multiplexor module.

One example would be the Internet multiplexor, mip, which may have several lower level STREAMS, with several sets of lower level streams existing for each different Ethernet networking device. For example, some of these lower level STREAMS might connect to a Condor device, and some of the other lower level STREAMS might connect to a FDDI device. Some system administrators of real-time systems may wish to separate the CPUs that execute these two different sets of STREAMS service procedures.

For more information on using **strmuxbias(1M)**, see the **strmuxbias(1M)** man page.

Example for Using **strmuxbias(1M)**

Since one of the more likely uses of **`strmuxbias(1M)`** would be to change the CPU bias masks of TCP/IP STREAMS that lie below the Internet Multiplexor module (mip), these STREAMS are discussed briefly here as an example of one way to make use of the **`strmuxbias(1M)`** utility.

The following lines show a sample of TCP/IP related lines of output from a "**`strmuxbias -g`**" invocation on a system with two Condor Ethernet devices and four CPUs:

```
0xf  /dev/cnd01  108/1  mip arpm cnd
0xf  /dev/cnd01  108/2  mip arpm ipm cnd
0xf  /dev/cnd00  107/1  mip arpm cnd
0xf  /dev/cnd00  107/2  mip arpm ipm cnd
0xf  /dev/tcp     55/8  mip tcpm mip
```

The first two lines are for IP traffic through the first Condor Ethernet device, and the next two lines after that are for IP traffic through the second Condor Ethernet device. The arpm modules handles Address Resolution Protocol (ARP) processing, and the ipm module handles the Internet Protocol (IP) processing.

The last line is for the STREAM that handles the Transmission Control Protocol (TCP) processing for all TCP/IP traffic.

As an example, if the first Condor Ethernet device is handling real-time networking traffic, and it is desirable for only the last two CPUs in the system to process this real-time networking data, then the system administrator might want change the first two lines so that the CPU bias masks of these STREAMS contain only the last two CPUs in the system. Further, the system administrator might want to place the rest of the non-real-time TCP/IP related STREAMS service procedure processing on the first two (presumably non-real-time) CPUs.

Thus, the following lines might represent the contents of a CPU bias mask input file that would be used on a "**`strmuxbias -s`**" invocation to set the CPU bias masks of these lower level TCP/IP STREAMS:

```
0xc  /dev/cnd01  108/1  mip arpm cnd
0xc  /dev/cnd01  108/2  mip arpm ipm cnd
0x3  /dev/cnd00  107/1  mip arpm cnd
0x3  /dev/cnd00  107/2  mip arpm ipm cnd
0x3  /dev/tcp     55/8  mip tcpm mip
```

User–Level Interrupt Routines

The operating system provides the support necessary for a user–level process to connect a routine to an interrupt vector that corresponds to the interrupt generated by a selected device and to enable the connection. You can significantly improve process dispatch latency for an individual program by using the user–level interrupt routine facility. Connecting a user–level interrupt routine to an interrupt vector allows you to avoid several of the events that make up process dispatch latency when the kernel catches the interrupt. If you use a user–level interrupt routine, it is not necessary to execute a kernel interrupt routine or wake the process that is waiting for the event signified by the interrupt. In addition, process dispatch latency is not affected by interrupts or raised IPL unless the interrupt or the raised IPL has a higher interrupt priority level than the connected interrupt.

Figure 2-5 illustrates the extent to which process dispatch latency is affected by use of user–level interrupt routines.

An overview of user–level interrupt routines and a detailed explanation of the procedures for using them are provided in Chapter 8 of this guide.

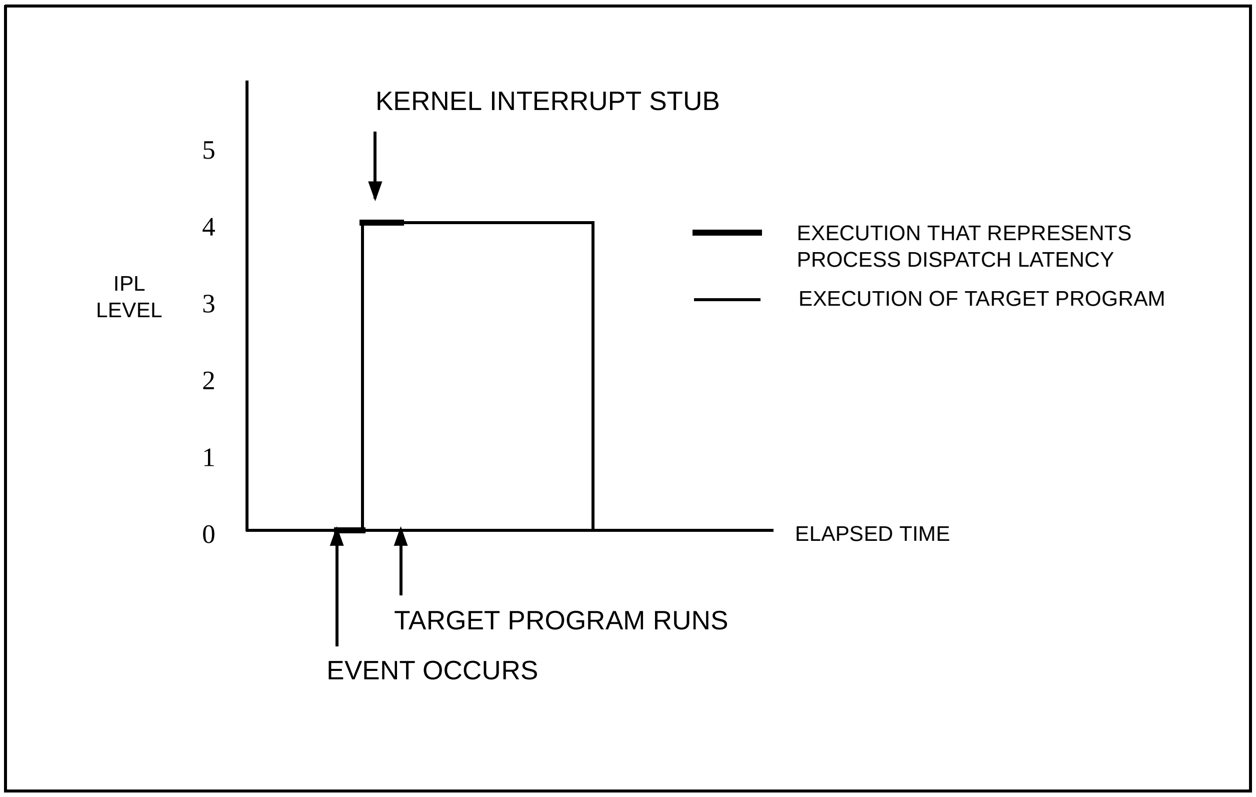


Figure 2-5. Effect of User–Level Interrupts on Process Dispatch Latency

User–Level Device Drivers

The operating system provides support for user–level device drivers. A user–level device driver consists of a set of library routines that allows a user program to perform I/O and control operations for a particular device directly from user level without entering the kernel. Direct access to the device is achieved by mapping the (H)VME addresses associated with the device's hardware registers onto the user's virtual address space.

A user–level device driver benefits worst case process dispatch latency because it gives you access to the driver's source file and allows you to directly control the length of critical sections. As a result, you can directly control the length of the critical sections that are protected by raising the IPL (see “Effect of IPL” for information on the way in which the length of a critical section that is protected by raising IPL affects worst case process dispatch latency). You can also directly control the length of a user–level device driver's interrupt routine (see “Effect of Interrupts” for information on the way in which the length of the interrupt routine affects process dispatch latency). Because most user–level device drivers are simple in design, their critical sections and interrupt routines are much shorter than those of a kernel device driver.

A user–level device driver also provides an alternative, low–overhead means of performing I/O operations. Without a user–level device driver, you must use system calls to perform I/O operations—a procedure that is costly in terms of overhead. A system call involves crossing the user–kernel boundary and several layers of kernel routines before finally calling the device driver routine associated with the particular system call. After the kernel device driver performs the work required for completion of the requested I/O operation, the same path must be traced back through the various layers to exit the kernel. A user–level device driver provides a means of performing I/O operations without having to enter and exit the kernel.

Other advantages of a user–level device driver include the following: (1) it does not require that the kernel be modified or rebooted, and (2) it may permit I/O operations to be performed directly to a user process's virtual address space.

An overview of user–level device drivers and the advantages and disadvantages of using them is provided in *Device Driver Programming*. This manual describes the components that make up a user–level driver and the issues that are involved in developing one. It also provides an introduction to the operating system support for user–level device drivers.

A user–level device driver that enables you to access a DR11W emulator directly from user space is available (the **dr11w** emulator is a high–speed 16–bit interface between a Series 6000 system and a device that uses the Digital Equipment Corporation DR11W interface.) By performing I/O through the DR11W user–level driver, you can achieve a significant reduction in the overhead associated with issuing an I/O request. An overview of the DR11W emulator is provided in Chapter 12 of this guide. Procedures for using the related user–level device driver are included.

Threads and Response Time

The threads library allows an application to be divided into multiple execution streams that execute concurrently. This programming model can produce significant performance gains on a multiprocessor system because true concurrency can be achieved. If you use this programming model for real-time applications, it is important to understand how threads are scheduled by the threads library, how the threads-library level scheduler and the system scheduler interact, and how bound-thread scheduling differs from multiplexed-thread scheduling (see the "Thread Scheduling" section of the chapter entitled "Programming with the Threads Library" in the *PowerMAX OS Programming Guide*).

In multi-threaded real-time applications, it is recommended that you use only bound threads to perform time-critical tasks. Use of bound threads is recommended for the following reasons:

- When a bound thread is created, it is permanently attached to the LWP that is created with it. A bound thread that is ready to run executes when the system scheduler schedules its associated LWP for execution.

  A runnable multiplexed thread is temporarily assigned to an LWP by the threads library scheduler when an LWP becomes available. After the thread is assigned to an LWP, the LWP is scheduled for execution by the kernel, The additional scheduling overhead of the threads library causes the process dispatch latency for a multiplexed thread to be worse than that for a bound thread.

- A bound thread can be assigned a real-time scheduling policy and priority by using the **thr_setscheduler(3thread)** routine. The threads library scheduler assigns the specified policy and priority to the LWP to which the bound thread is attached. This allows direct control of the underlying LWP's priority.

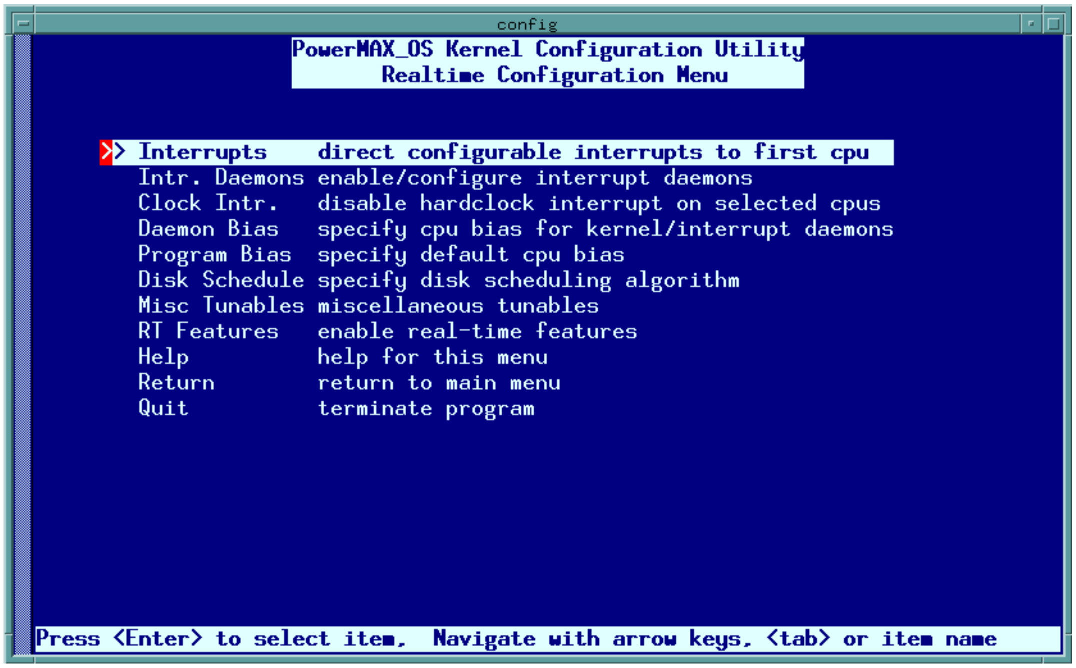
  A multiplexed thread can be assigned a scheduling priority by using **thr_setscheduler**, but that priority is used only by the threads library scheduler to assign multiplexed threads to LWPs. A higher priority thread is assigned to an LWP before a lower priority thread; the priority of the LWP to which the thread is assigned is not taken into consideration. A multiplexed thread runs under the scheduling policy and priority of the LWP to which it is assigned. With multiplexed threads, the application has no direct control of the underlying LWP's scheduling priority.

- The kernel's real-time scheduling policies are not available to multiplexed threads through the **thr_setscheduler** interface, which is the only scheduling control interface that should be used to set the scheduling policy of multiplexed threads.

Real-Time System Configuration Using Config

The kernel configuration utility, **config(1M)**, can be used to tune a system for use in a real-time environment. The easiest way to do this is by selecting the "Real-Time" sub-menu from the main-menu in **config(1M)**. This sub-menu provides a convenient way to examine and modify the configuration features listed earlier in this chapter.

The real-time sub-menu will appear as follows:



```
                              config
           PowerMAX_OS Kernel Configuration Utility
               Realtime Configuration Menu


    > Interrupts     direct configurable interrupts to first cpu
      Intr. Daemons  enable/configure interrupt daemons
      Clock Intr.    disable hardclock interrupt on selected cpus
      Daemon Bias    specify cpu bias for kernel/interrupt daemons
      Program Bias   specify default cpu bias
      Disk Schedule  specify disk scheduling algorithm
      Misc Tunables  miscellaneous tunables
      RT Features    enable real-time features
      Help           help for this menu
      Return         return to main menu
      Quit           terminate program

Press <Enter> to select item.  Navigate with arrow keys, <tab> or item name
```

Screen 2-1. Realtime Configuration Menu

The items on this sub-menu may be used to do the following:

- direct interrupts to first CPU

  In this mode, all fully configurable interrupts are directed to the boot processor (i.e. CPU0). Depending on the architecture, some interrupts may only be directed to a specific CPU board, in which case, the interrupt will be bound to the first CPU on that board.

  Note that this configuration may be too rigid for some systems. In this case, this mode should not be selected. Instead, the tunable for each individual interrupt should be configured appropriately. Refer to “Assigning Interrupts to CPUs.”

- enable/configure interrupt daemons

  Used to configure interrupt daemons by setting the tunable `ALLINTRDAEMONS`. Interrupt daemons can be fully enabled, fully disabled, or selectively enabled. Refer to “Using Interrupt Daemons.”

- disable hardclock interrupts on selected cpus

  Can be used to selectively disable hardclock interrupts on individual processors. Refer to “Hardclock Interrupt Handling.”

- specify CPUs that are eligible to run kernel daemons

  Used to specify a CPU mask of processors that can run interrupt and kernel daemons.

  Note that this configuration may be too rigid for some systems. In this case, this mode should not be selected. Instead, the tunable for each individual daemon should be configured appropriately. Refer to “Assigning Daemons to CPUs.”

- specify CPUs that are eligible to run general programs

  Used to specify a CPU mask of processors that can run general programs. Note that this CPU bias is assigned to the **`init(1m)`** program and will be inherited by all children and sub-children processes. Any user with the appropriate privilege may be able to decrease or increase the scope of the CPU bias. Refer to “Assigning Processes to CPUs.”

- specify disk scheduling algorithm

  Used to configure the disk scheduling algorithm that will be used. Scheduling can be priority-based (which may be ideal in real-time environments). In addition FIFO or CSCAN queueing can be specified.

- specify miscellaneous tunables

  Can examine and modify tunables `STRSCHED_DISABLE_MSK` and `STRSCHED_DAEMON_MSK`. Refer to “Controlling STREAMS Scheduling.”

  Can examine and modify tunable `KMA_GBACK_DISABLE` which is used to control the set of cpus that will run the `KMA` giveback daemon.

- enable real-time features

  Can examine and modify tunable `HIGHRESTIMING` which is used to configure high-resolution (nano-second) timing.

  Can enable/disable the kernel modules for frequency based scheduling (FBS) or user-level interrupts.

3
Increasing Determinism

3
Increasing Determinism

This chapter describes techniques that you can use to create a deterministic environment for execution of an application program. It examines the advantages and disadvantages of using those techniques.

Overview of Determinism

Determinism refers to a computer system's ability to execute a particular code path (a set of instructions that is executed in sequence) in a fixed amount of time. The extent to which the execution time for the code path varies from one instance to another indicates the degree of determinism in the system.

Determinism applies not only to the amount of time that is required to execute a time-critical portion of a user's application but also to the amount of time that is required to execute system code in the kernel. The determinism of the process dispatch latency, for example, depends upon the code path that must be executed to handle an interrupt, wake the target process, perform a context switch, and allow the target process to exit from the kernel. (Chapter 2 defines the term *process dispatch latency* and presents a model for obtaining the best process dispatch latency that is possible on a particular processor in a multiprocessor system.)

A variety of issues affects determinism. These issues relate to architecture and interprocessor interrupts. "Architectural Issues" describes the architectural issues. "Interprocessor Interrupts" describes the issues related to interprocessor interrupts.

Architectural Issues

In a shared memory, multiprocessor system, in which several processes are executing simultaneously on multiple processors, it is impossible to achieve absolute determinism because the individual processors share system resources. Contention for shared system resources during the application's execution is unpredictable. When a process running on one processor attempts to access a shared resource that is being used by another process running on a different processor, it must wait until the other process releases the resource. The number of times that a conflict will occur in accessing a shared resource is unpredictable. As a result, the process's execution time is indeterminate.

The key system resources that processors share are the system, or frontplane, bus and the local bus. The system bus provides access to global memory from all of the system's processor boards. The local bus provides access to the system bus from a processor board and provides access to the local memory pool on that board. Located on a processor board and contending for access to the buses are one or more CPUs, the cache controllers, and such I/O daughter boards as the ISE (Integral SCSI/Ethernet). The system bus is the resource for which contention is highest and is, therefore, the chief source of indeterminism in a system.

Reducing Contention for the System Bus

You can reduce contention for the system bus in four ways:

- Using the shielded processor model
- Confining background tasks to a single processor or processor board and restricting their access to the system bus
- Using direct I/O
- Using user-level device drivers.

The shielded processor model is fully described in Chapter 2. The approach of the model is to assign high–priority functions to one or more CPUs and to restrict the interrupts directed to those CPUs. Only the interrupts that signal events important to the high–priority functions are directed to those CPUs; other interrupts and all system daemons are assigned to the other CPUs. The processor(s) on which the high–priority functions are running are shielded from the unpredictable processing associated with most interrupts and system daemons. Another aspect of the model is to place as much of the application program as possible in local memory. By using local memory, the processes executing on the board associated with that local memory pool are shielded from activity on other processor boards.

The shielded processor model provides two safeguards against indeterminism:

- Unnecessary interrupts are directed away from the shielded processor(s). Interrupts are a major source of indeterminism because they preempt the currently executing process, regardless of its priority.
- High–priority functions assigned to the shielded processor(s) use local memory as much as possible. Accessing local memory is faster than accessing global memory and does not add to contention for the system bus.

Only programs with very limited functionality can be completely shielded from the indeterminism that is caused by contention for the system bus. First, kernel data and sometimes kernel text are stored in global memory. Each time a process makes a system call, it must access global memory and the system bus. Shielding is temporarily lost. Second, certain types of shared data are stored in global memory. Many real-time applications, for example, require large shared memory regions for communication among the processes that cooperate to perform the real-time task. If the processes execute on separate processor boards, the shared data must be stored in global memory in order to be accessible to all of the cooperating processes.

Because most tasks will require some access to global memory, you cannot sufficiently reduce contention for the system bus by simply using the shielded processor model. You must also attempt to confine the activity of background tasks to a single processor or processor board so that they do not contribute to the contention for the system bus. Background tasks include compilations, functions performed by kernel daemons, and application tasks that do not require deterministic execution (for example, logging of input data). Such tasks typically make numerous system calls, which require access to global memory and the system bus. If you assign the background tasks to one or more processors on a single processor board, you can restrict their access to the system bus by taking advantage of the following features of the PowerMAX OS virtual memory subsystem:

- The ability to ensure that kernel text is replicated in the local memory pool on a particular processor board by using a system tunable parameter

  This is an advantage when background tasks are making many system calls because accesses to kernel instructions will be local memory accesses.

- The ability to influence the kernel's page placement decisions by selecting the hard-local NUMA (non-uniform memory access) policy for different portions of a background task's address space

  When the local memory pool is full, the hard-local policy causes a process to block until the pages become available. This feature allows you to confine the pages used by background tasks to local memory. Consequently, background tasks do not compete for the system bus and cause contention with the real-time tasks on other processors.

  The hard-local policy cannot guarantee that file pages will be placed in a local memory pool. File pages will <u>not</u> be placed in a local memory pool, for example, if they are writable and are being accessed from different processor boards. They will be placed in the global memory pool instead.

  For an overview of primary memory and an explanation of NUMA policies and the procedures for selecting them, refer to the *PowerMAX OS Programming Guide*.

To ensure that kernel text is replicated in the local memory pool on the processor board on which the background tasks are executing, set the value of the corresponding `KTEXTLOCAL`*n* system tunable parameter to `1` (*n* denotes a number ranging from **1** to **4**, where **1** represents the first local memory pool, **2** the second, and so on, in order according to processor board slot number). You can examine and modify the values of system tunable parameters by using the **`config(1M)`** utility. For an explanation of the procedures for using this utility, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*. After changing a tunable parameter, you must rebuild the kernel and then reboot your system.

To select the hard-local NUMA policy for the background tasks' pages, use one of the following methods:

1. Invoke the **`memdefaults(2)`** system call, specify the MDF_SETNUMA command, and specify a flag that sets <u>all</u> of the following bits:

MDF_TEXT_HARDLOCAL	select the hard-local NUMA policy for text
MDF_PRDATA_HARDLOCAL	select the hard-local NUMA policy for private data
MDF_SHDATA_HARDLOCAL	select the hard-local NUMA policy for shared data
MDF_UBLOCK_HARDLOCAL	select the hard-local NUMA policy for the U-block

2. Invoke the `run(1)` or `rerun(1)` command from the shell and specify the `-m` *NUMA* option.

   NUMA specifies one or more keywords that select the NUMA policies for parts of the process's address space. Specify the keyword **hard** or all of the following keywords:

text_hard	select the hard-local NUMA policy for text
prdata_hard	select the hard-local NUMA policy for private data
shdata_hard	select the hard-local NUMA policy for shared data
ublock_hard	select the hard-local NUMA policy for the U-block

For additional information on use of the `memdefaults(2)` system call and the `run` and `rerun` commands, refer to the corresponding system manual pages.

Confining background tasks to a single processor or processor board not only reduces contention for the system bus but also causes access to the system bus from the shielded processor(s) to be deterministic.

The Integral SCSI Ethernet (ISE) interface provides an inexpensive disk and Ethernet controller. When the ISE is used for direct I/O to raw disk partitions and the data are transferred to or from the local memory pool on the processor board on which the ISE is mounted, there is no access to global memory or the system bus. This technique provides a means of performing large disk transfers without affecting the determinism of any of the processors on other processor boards. When an application contains one or more processes that perform large disk transfers, using direct I/O from an ISE to local memory on the same processor board provides an I/O mechanism that reduces contention on the system bus. Note that the process performing disk I/O must be bound to a processor on the same processor board as the ISE and the buffers that are used for I/O must be in local memory.

Interprocessor Interrupts

The operating system uses a special set of interrupt lines—one per CPU—to build a simple interprocessor communication facility. With this facility, one CPU can cause another CPU to perform an action in its behalf. Interprocessor interrupts (IPIs) are among the highest priority interrupts in the system. Although they are usually short in duration, they can be a source of jitter for some real-time processes.

The extent to which interprocessor interrupts are used is less in PowerMAX OS than it has been in previous operating system releases. It is no longer necessary for the kernel to perform interprocessor interrupt operations to maintain cache coherency. The operating system uses interprocessor interrupts only when a process invokes the POSIX `clock_settime(3C)` library routine to modify CLOCK_REALTIME. In this case, the interval timer must be modified—an operation that can be performed only on the boot processor. If `clock_settime` is called on a different processor, the operating system must issue an interprocessor interrupt to the boot processor to perform the operation.

Procedures for Increasing Determinism

You can increase determinism of a program's execution by locking a process's pages in memory and by using local memory. These techniques are described in "Locking Pages in Memory" and "Using Local and Global Memory." When certain programs require very deterministic response times, you should use static priorities and assign the most favorable priorities to the tasks that require the most deterministic response. This technique is explained in "Setting the Program Priority." Certain system calls and library routines that block the calling process contain timeouts to control the length of time that the process is blocked. You can obtain a high resolution on the timeout by using the facilities that are described in "Using High–Resolution Timeout Facilities." When an application uses multiple tasks that require deterministic communication through use of shared memory regions, you should use the **`server`** system calls. This technique is explained in "Waking Another Process."

Locking Pages in Memory

You can avoid the overhead that is associated with paging and swapping by using the **`mlockall(3C)`**, **`munlockall(3C)`**, **`mlock(3C)`**, and **`munlock(3C)`** library routines and the **`userdma(2)`** system call.

The **`mlockall(3C)`**, **`munlockall(3C)`**, **`mlock(3C)`**, and **`munlock(3C)`** library routines allow you to lock and unlock all or a portion of a process's virtual address space in physical memory. These interfaces are based on UNIX System V Release 4 and IEEE Standard 1003.1b-1993

The **`userdma`** system call allows you to use an I/O controller's DMA (Direct Memory Access) capabilities directly from user mode. It prepares an I/O buffer located in a user process's virtual address space for DMA transfers. It locks the I/O buffer in physical memory, returns the buffer's physical location, and ensures that the mapping of virtual to physical addresses does not change. If you indicate whether the I/O buffer is to be used for writing to or reading from the device, **`userdma`** sets the cache modes of the pages containing the buffer to keep memory and cache coherent.

With each of these calls, pages that are not resident at the time of the call are faulted into memory and locked. To use the **`mlockall(3C)`**, **`munlockall(3C)`**, **`mlock(3C)`**, and **`munlock(3C)`** library routines and the **`userdma`** system call, you must have the P_PLOCK privilege (for additional information on privileges, refer to the **`intro(2)`** system manual page and the *PowerMAX OS Programming Guide*.

Procedures for using these library routines and system calls are fully explained in the *PowerMAX OS Programming Guide*; reference information is provided in the corresponding system manual pages.

Using Local and Global Memory

Local memory is available on all Model 6800 systems. It is designed to improve the performance of multiprocessor systems by reducing system bus traffic and contention.

Local memory is a pool of memory that is physically located on a processor board. Each processor has a data path to its local memory pool that does not require use of the system bus. Note that local memory is actually shared among all processors on each processor board.

Global memory, or memory that is shared by all processors in the system, is located on a separate board and is available to all processors via the system bus.

Local memory has a number of advantages that benefit program responsiveness, but it also has some disadvantages. The advantages of local memory are as follows:

- Local memory provides faster memory access times than global memory.
- No system bus traffic is generated when accessing local memory. Because the system bus is a resource that is shared by all processors, use of local memory improves determinism for the system as a whole.

The disadvantages of local memory are as follows:

- I/O operations to and from local memory can slow memory accesses from the processors located on the same board.
- Accesses from a processor on one processor board to local memory on a different processor board are very slow and can affect the performance of the entire system.

In general, the more portions of a process's address space that you place in local memory, the more deterministic that the execution of that process will be. Because the amount of local memory in a system is limited, it may not be possible to place all portions of a particular process's address space in local memory. The *PowerMAX OS Programming Guide* provides an overview of the hardware and software features of primary memory and the NUMA (non-uniform memory access) policies that govern the type of memory in which different portions of a process's address space may be placed. It includes guidelines for you to use in determining the policies that are appropriate for your application. It is recommended that you read the "Primary Memory" section of the "Memory Management" chapter of that manual and that you refer to the **`memory(7)`** system manual page as background for the material that is presented here.

The paragraphs that follow highlight special concerns that should be taken into consideration when determining the appropriate NUMA policy for various portions of a real-time application's address space.

If shared, writable pages created via the **`mmap(2)`** system call are placed in local memory and those pages are subsequently accessed from a remote processor board, the pages will migrate to global memory. All accesses to a page while it is migrating will stall until the migration is complete. Note that such migrations occur even if the page has been locked in memory. **`Mmap`** allows a process to override the default NUMA policy for writable pages so that they are placed in global memory. Shared, writable pages that are placed in global memory never migrate. If these migrations are undesirable, shared, writable pages should be placed in local memory only when all accesses to the pages are performed by processes that are running on the processor board on which the local memory pool is located.

Shared, writable pages in System V IPC shared memory segments are handled differently: they never migrate because of accesses from remote processor boards. In general, access from a remote processor board to a shared memory segment that is bound to a local memory pool is not permitted—the **`shmat(2)`** system call will fail with an EACCES error. It is possible to obtain access to a remote shared memory segment by setting the **SHM_FLMEM** bit in the *shmflg* argument on the call to **`shmat`**; however, even in this case, the pages will not migrate to global memory. Instead, the calling process will access the pages remotely. Because of the properties of remote memory accesses, use of this feature is not generally recommended.

In addition, pages mapped by using the **`usermap(2)`** system call will never migrate because of accesses by the calling process. **`Usermap`** is intended to be a debugging or monitoring tool; consequently, it attempts to provide access to the target process's pages without affecting the process itself.

Note that use of the any-pool policy is strongly discouraged because it allows remote memory accesses. Remote memory accesses are especially detrimental to system determinism because they must use the paths to two local memories as well as the system bus. Contention results from attempts to gain control of each of the buses. In addition, when remote accesses occur, a single word rather than an entire cache line is transferred on each access. Consequently, every time a word of remote memory is accessed, contention for each of the buses arises.

For details about the procedures for selecting NUMA policies for the different parts of a process's address space, refer to the **`memory(7)`** system manual page and the *PowerMAX OS Programming Guide*.

Setting the Program Priority

The PowerMAX OS kernel accommodates static priority scheduling—that is, processes scheduled under certain POSIX scheduling policies or System V scheduler classes do not have their priorities changed by the operating system in response to their run–time behavior.

Processes that are scheduled under one of the POSIX real–time scheduling policies always have static priorities. (The real–time scheduling policies are SCHED_RR and SCHED_FIFO; they are explained in the *PowerMAX OS Programming Guide*.) To change a process's scheduling priority, you may use the **`sched_setscheduler(3C)`** and the **`sched_setparam(3C)`** library routines. Note that to use these routines to change the priority of a process to a higher (more favorable) value, you must have the P_RTIME privilege (for complete information on privilege requirements for using these routines, refer to the corresponding system manual pages and the *PowerMAX OS Programming Guide*).

Processes that are scheduled under the System V fixed-priority or fixed class also have static priorities. The fixed-priority class is the highest priority scheduling class in the system; it is used only for the most important real-time tasks in an application. The fixed class is used for tasks that require a fixed priority that is not influenced by such factors as CPU usage but that are not considered important enough to be scheduled in the fixed-priority class. To assign a process to the fixed class or the fixed-priority class, you must use the **`priocntl(2)`** system call (for information on scheduler classes and use of this system call, refer to the corresponding system manual page and the *PowerMAX OS Programming Guide*).

The highest priority process running on a particular processor will have the best process dispatch latency. If a process is not assigned a higher priority than other processes running on a processor, its process dispatch latency will be affected by the time that the higher priority processes spend running. As a result, if you have more than one process that requires good process dispatch latency, it is recommended that you distribute those processes among several processors. Refer to Chapter 2 for an explanation of the procedures for assigning processes to particular processors.

Process scheduling is fully described in the *PowerMAX OS Programming Guide*. Procedures for using the **`sched_setscheduler`** and **`sched_setparam`** library routines to change a process's priority are also explained.

Using High–Resolution Timeout Facilities

You can obtain a finer resolution on certain types of blocking requests by configuring a real–time clock to be used for triggering events in the high–resolution callout queue. By default, a real–time clock is configured into the system for this purpose. Consequently, you can obtain a resolution of one microsecond instead of 1/60 of a second, which is the resolution provided by the system–wide 60 Hz clock.

The real–time clock is set up to generate an interrupt when the time of the next entry in the high–resolution callout queue expires. When the interrupt occurs, an associated interrupt service routine arranges for execution of the specified routine and removal of the entry from the callout queue. The interfaces that use the high-resolution callout queue are the **`server_block(2)`** and **`client_block(2)`** system calls and the **`nanosleep(3C)`** and the **`timer_settime(3C)`** library routines.

Entries are placed in the high–resolution callout queue as a result of blocking requests made by using the **`server_block(2)`** and **`client_block(2)`** system calls. These calls are briefly described as follows:

`client_block`	block the calling thread (a client) and pass its priority to another thread (a server)
`server_block`	block the calling thread only if no wake–up request has occurred since the last return from **`server_block`**

Each of these calls allows you to supply a *timeout* argument that specifies the maximum length of time that the calling thread will be blocked. If a real–time clock has been configured for use with the high–resolution callout queue, you will obtain a resolution of one microsecond on the timeout request; otherwise, you will obtain a resolution of 1/60 of a second. Procedures for using the **`client_block`** and **`server_block`** system calls are fully explained in Chapter 6.

Entries are also placed in the high–resolution callout queue as a result of blocking requests made by using the **`nanosleep(3C)`** library routine. The interface to this routine is based on IEEE Standard 1003.1b-1993. The **`nanosleep`** routine causes execution of the calling process to be suspended until (1) a specified period of time elapses or (2) a signal is received and the associated action is to execute a signal–handling routine or terminate the process. If a real–time clock has been configured for use with the high–resolution callout queue, you will obtain a resolution of one microsecond on the blocking request; otherwise, the call will fail. Procedures for using the **`nanosleep`** library routine are fully explained in Chapter 6 of this guide.

An entry is also placed in the high–resolution callout queue as a result of a call to the **`timer_settime(3C)`** library routine. This routine is based on IEEE Standard 1003.1b-1993. It allows a process to set the time at which a POSIX timer will expire. If a real-time clock is not configured for the high-resolution callout queue, the **`timer_settime`** routine will fail. An explanation of POSIX clocks and timers and the related clock and timer library interfaces is provided in Chapter 7 of this guide.

Waking Another Process

In multiprocess applications, you often need to wake a process to perform a particular task. One measure of the system's responsiveness is the speed with which one process can wake another process. The fastest method that you can use to perform this switch to another task is to use the **`server_block(2)`** system call to block the current process and the **`server_wake1(2)`** system call to wake the blocked process. These calls are briefly described as follows:

server_block	block the calling thread only if no wake–up request has occurred since the last return from **`server_block`**
server_wake1	wake a single server that is blocked in the **`server_block`** system call; if the specified server is not blocked in this call, the wake–up request is applied to the server's next call to **`server_block`**

A process may use the **`server_wake1`** system call to wake a higher priority process that is blocked waiting for a specific event to occur. A process may use the **`server_block`** system call to block itself in order to allow a lower priority process to run.

Procedures for using the server system calls are fully explained in Chapter 6 of this guide.

4
Using the ktrace Utility

4
Using the ktrace Utility

This chapter provides an overview of the **ktrace** utility and explains the procedures for using it.

Overview of ktrace

When you are executing a real–time application, it is important to know what the operating system overhead is. The **ktrace(1)** program allows you to determine the amount of time that is devoted to processing within the operating system—time that is spent servicing system calls (including I/O system calls to devices) and handling interrupts and exceptions. The **ktrace** program provides a summary of all of the system activity that occurs while it is being used to log system events.

To be able to use the **ktrace** program, you must have the NightTrace^{TM} product, and you must ensure that the kernel trace module (**trace**) is configured into the kernel. You can use the **config(1M)** utility to (1) determine whether or not the **trace** module is enabled in your kernel, (2) enable or disable the **trace** module, and (3) rebuild the kernel. Note that you must be the root user to enable a module and rebuild the kernel. After rebuilding the kernel, you must then reboot your system. For an explanation of the procedures for using **config(1M)**, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*.

If your application requires very fast real–time response, you are advised not to have the kernel trace module configured when you are building a kernel for production purposes. When the kernel trace module is configured into the kernel, overhead associated with checks for the enabling of trace points is added to context switches, interrupt entry and exit, and system calls (including I/O system calls to devices).

Configuring a Kernel for Kernel Tracing

Kernel tracing is optional. To configure a kernel with kernel tracing enabled, you must modify the **trace** file in the **/etc/conf/sdevice.d** directory. Set the **configure** field in the **trace** file to **Y**, and build a kernel with the **idbuild(1M)** utility. (Refer to the **idbuild(1M)** system manual page for specific details.) To disable kernel tracing, set the **configure** field in the **trace** file to **N**, and build a new kernel with the **idbuild(1M)** utility.

Even if trace points are compiled into the kernel, a trace point will not log data until it has been enabled via an **octl** call. (See **/usr/include/sys/ktrace.h** for details about this **octl** call.) By default, **ktrace** enables the following kernel trace points:

- Interrupt entry/exit
- Exception entry/exit
- Context switch
- System call entry
- I/O service call entry
- Shared interrupt handler dispatch
- Naming of a process

The default trace points are required to get meaningful performance data. You must have kernel source to compile trace points other than the defaults into a kernel.

The most common way to run **ktrace** is to collect the raw trace data in an output file, then use **ktrace** to analyze the raw data and produce its summary.

Procedures for Using ktrace

It is recommended that you run the **ktrace** program once to log data about system events and then run it again to analyze the data and obtain a summary of the statistics. Using **ktrace** to analyze **ktrace** data consumes CPU time. You should avoid using it to perform data gathering and analysis at the same time for the following reasons:

- The CPU time used for **ktrace** analysis will affect the workload being measured.
- The **ktrace** program is more likely to lose **ktrace** records because those records will be read from the kernel at a slower rate.

To use the **ktrace** program, perform the following steps:

1. Run **ktrace** in the background to gather raw data from a running kernel.

   An example of the format that you can use to run **ktrace** for this purpose is as follows:

   ktrace -output *raw_data_file* **&**

   The **-output** option allows you to specify the name of a file to which you wish raw trace data to be written. When you are ready to obtain an analysis of the data, you can use this file as input to **ktrace**. Note that when you specify the **-output** option, analysis of the trace data is not performed.

   It is recommended that you use a file system that is local to the system being traced rather than an NFS file system.

2. While **ktrace** is running in the background, run the workload that you wish to monitor.

3. Terminate the **ktrace** program by sending it the SIGINTR signal.

4. Run **`ktrace`** again to analyze the raw data gathered in Step 1.

   An example of the format that you can use to run **ktrace** for this purpose is as follows:

   `ktrace -input` *raw_data_file* **`>`** *summary_data_file*

   The **`-input`** option allows you to specify the name of a file that contains the raw trace data to be used as input. You may specify the name of a file to which you wish summary data output to be redirected. The **`ktrace`** program has many other options. These options are fully described in the **`ktrace(1)`** system manual page. Among the other options that you may find useful are the following:

`-cpu` *cpu_id*	display trace point data only for the logical CPU identifier specified by *cpu_id* The default is all CPUs.
`-process` *pid*	display trace point data only for the process whose process identification number (PID) is specified by *pid* The default is all PIDs.
`-wall`	calculate all times in terms of wall time When you use the **`-wall`** option, the times reported for system calls (including I/O calls) and other exceptions include (1) the time that other processes are running if the current process blocks in the kernel and (2) the time spent handling interrupts that preempt execution of the current process. The times reported for interrupts include the time spent handling higher priority interrupts that preempt execution of the current interrupt.
`-priority` *priority*	run **`ktrace`** at the real-time priority specified by *priority* The **`ktrace`** program always runs under the fixed-priority scheduler class. You can determine the range of acceptable priority values for this class by invoking the **`run(1)`** command without specifying any options or by invoking the **`priocntl(1)`** command and specifying the **`-l`** option. The default is the maximum real-time priority.
`-clock` *source*	Specifies the clock to use as the source of event time stamps. If this option is given on the command line, then *source* must be given. If this option is not specified on the

command line, then the system's interval timer (Night Hawk Series 6000, PowerMaxion, TurboHawk) or the Time Base Register (Power Hawk Series 600/700/900, PowerStack II and III) will be used to obtain trace event time stamps.

Valid *source* names are:

default	The default system clock
rcim_tick	The RCIM synchronized tick clock

NOTE: If you specify *rcim_tick* for the *source* and the system on which you are tracing does not have an RCIM installed or configured or if the RCIM synchronized tick clock on the system on which you are tracing is stopped, **`ktrace`** will exit with an error.

The summary that is produced when you use the **`ktrace`** program to analyze raw **`ktrace`** data includes the following types of information. Note that times are reported in microseconds unless otherwise indicated; *elapsed time* is measured from the time that **`ktrace`** started running.

- A configuration summary that provides details about the configuration of the system at the time that **`ktrace`** was run to gather the data
- A system call summary that lists the system calls that were made; the number of times that each call was invoked; the minimum, average, and maximum amounts of time that were spent executing a particular call; and the elapsed time in seconds at which the calls taking the minimum and maximum amounts of time occurred
- An exception summary that lists the types of exceptions that occurred; the number of times that each type occurred; the minimum, average, and maximum amounts of time that were required for handling a particular type of exception; and the elapsed time in seconds at which the exceptions requiring the minimum and maximum amounts of time occurred
- An interrupt summary that lists the types of interrupts that occurred; the CPUs on which they were received; the minimum, average, and maximum amounts of time that were required for handling a particular type of interrupt; and the elapsed time in seconds at which the interrupts requiring the minimum and maximum amounts of time occurred
- A summary of the total time spent handling each interrupt and exception and the total time spent handling all interrupts and exceptions
- An I/O system call summary that lists the system calls that were made to devices; the number of times that they were invoked; the minimum, average, and maximum amounts of time that were spent executing a particular call; and the elapsed time in seconds at which the calls requiring the minimum and maximum amounts of time occurred

Examples of the output for each type of summary are provided in the *NightTrace Manual*. You are encouraged to use the **`ntfilter`** and **`ntrace`** tools, which are included in the NightTrace product, for graphical displays, searches, and summaries of kernel and user trace events.

It is important to note that the maximum times that the **`ktrace`** program reports are greater than the times that accrue in a kernel configured without the kernel trace module. Refer to the *NightTrace Manual* for additional information.

5
Real-Time Interprocess Communication

5
Real-Time Interprocess Communication

Interprocess communication refers to transferring data between processes. *Interprocess synchronization* refers to coordinating the execution of processes. This chapter describes PowerMAX OS support for real-time interprocess communication. Chapter 6 describes PowerMAX OS support for interprocess synchronization.

PowerMAX OS real-time interprocess communication support includes POSIX message-passing facilities that are based on IEEE Standard 1003.1b-1993. The POSIX message passing facilities provide a means of passing arbitrary amounts of data between cooperating processes. Use of the POSIX message-passing facilities is supported by a group of interfaces that is presented in "Using the Library Routines."

PowerMAX OS also provides the System V Interprocess Communication (IPC) package, which includes the following mechanisms: messages, semaphores, and shared memory. Each of these mechanisms is supported by a set of system calls. Refer to the *PowerMAX OS Programming Guide* for a detailed explanation of the procedures for using these mechanisms and the related system calls.

POSIX message queues are the recommended message-passing mechanism. The performance of System V message queues is an order of magnitude slower than the POSIX message queues.

POSIX message queues can also be used to communicate between processes that are running on separate boards in a closely-coupled system.

Understanding Message Queue Concepts

An application may consist of multiple cooperating processes, possibly running on separate processors. These processes may use system-wide POSIX message queues to efficiently communicate and coordinate their activities.

The primary use of POSIX message queues is for passing data between <u>processes</u>. In contrast, there is little need for functions that pass data between cooperating <u>threads</u> in the same process because threads within the same process already share the entire address space. However, nothing prevents an application from using message queues to pass data between threads in one or more processes.

"Understanding Basic Concepts" presents basic concepts related to the use of POSIX message queues. "Understanding Advanced Concepts" presents advanced concepts.

Understanding Basic Concepts

A message queue consists of message slots. To optimize message sending and receiving, all message slots within one message queue are the same size. A message slot can hold one message. The message size may differ from the message slot size, but it must not exceed the message slot size. Messages are not padded or null-terminated; message length is determined by byte count. Message queues may differ by their message slot sizes and the maximum number of messages they hold. Figure 5-1 illustrates some of these facts.

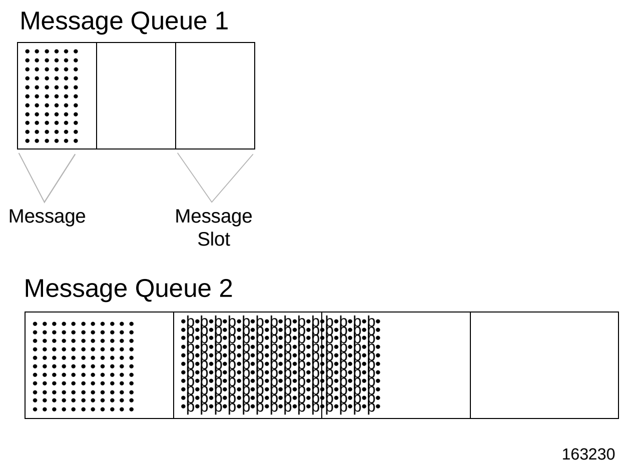


Figure 5-1. Example of Two Message Queues and Their Messages

POSIX message queue library routines allow a process to:

- Create, open, query, close, and destroy a message queue
- Send messages to and receive messages from a message queue
- Associate a priority with a message to be sent
- Request asynchronous notification via a user-specified signal when a message arrives at a specific empty message queue

Processes communicate with message queues via message queue descriptors. A child process created by a **`fork(2)`** system call inherits all of the parent process's open message queue descriptors. The **`exec(2)`** and **`exit(2)`** system calls close all open message queue descriptors.

When one thread within a process opens a message queue, all threads within that process can use the message queue if they have access to the message queue descriptor.

A process attempting to send messages to or receive messages from a message queue may have to wait. Waiting is also known as being blocked.

Two different types of priority play a role in message sending and receiving: message priority and process-scheduling priority. Every message has a message priority. The oldest, highest-priority message is received first by a process.

Every process has a scheduling priority. Assume that multiple processes are blocked to <u>send</u> a message to a full message queue. When a message slot becomes free in that message queue, the system wakes the highest-priority process that has been blocked the longest; this process sends the next message. Assume that multiple processes are blocked to <u>receive</u> a message from an empty message queue. When a message arrives at that message queue, the system wakes the highest-priority process that has been blocked the longest; this process receives the message.

Understanding Advanced Concepts

Message queues are implemented at user level rather than in the kernel to provide for the most efficient means of passing data between processes. The storage space for a message queue is in a memory-mapped file. The following paragraphs describe the effects of this implementation on processes and system resource usage.

User-level spin locks synchronize access to the message queue, protecting message queue structures. While a spin lock is locked, most signals are blocked to prevent the application from aborting. However, certain signals cannot be blocked. Assume that an application uses message queues and has a lock. If a signal aborts this application, the message queue becomes unusable by any process; all processes attempting to access the message queue hang attempting to gain access to the lock. For successful accesses to be possible again, a process must destroy the message queue via **`mq_unlink(3)`** and re-create the message queue via **`mq_open(3)`**. For more information on **`mq_unlink`** and **`mq_open`**, see "Using the mq_unlink Routine" and "Using the mq_open Routine", respectively. For more information on signals, see the *PowerMAX OS Programming Guide*.

To prevent a process from spinning forever while attempting to gain access to a message queue, the message queue routines spin for a while, then block for a short period of time. To allow a process to block for periods of time less than 1/60 of a second (one 60Hz clock tick), you must ensure that a real-time clock is configured for use with the high-resolution callout queue. For more information, see the section on using high-resolution timeout facilities in Chapter 3 of this guide.

The message queue implementation uses a rescheduling variable to protect the critical sections inside the message queue interfaces. If you do not have the access required to initialize a rescheduling variable, then none will be used. In this case, the functionality of the message queue interfaces does not change; however, it is possible to be preempted by higher priority processes from within critical sections. Performance of the message queue operations may be degraded as a result. (You must have the P_RTIME privilege to initialize a rescheduling variable. Use of rescheduling variables is explained in Chapter 6 of this guide.)

The sections that follow describe the POSIX message queue interfaces and provide additional information about these interfaces. "Understanding the Message Queue Attribute Structure" describes the message queue attribute structure. "Using the Library Routines" presents the POSIX message queue library routines. Appendix A provides a sample program that uses many of the POSIX message queue library routines.

Performance Issue

The message queue implementation uses a rescheduling variable to protect the critical sections inside the message queue interfaces. If you do not have the privilege required to initialize a rescheduling variable, then none will be used. In this case, the functionality of the message queue interfaces does not change; however, it is possible to be preempted by higher priority processes from within critical sections. If such a preemption occurs, another LWP or process that is attempting an operation on the message queue will block until the preempted LWP or process is able to run again and complete the critical section. Performance of the message queue operations may be substantially degraded as a result.

The privilege that is required to initialize a rescheduling variable is P_RTIME. In time-critical applications, it is important to ensure that this privilege is granted to all of the processes that access a particular message queue.

Note that the page where a rescheduling variable is located will be locked down in memory. This locked page may prevent the changing of a process's or LWP's CPU bias, if the new CPU bias value requires migrating to a different CPU board on a Night Hawk system.

The procedures and limitations for using rescheduling variables are fully explained in Chapter 6, “Rescheduling Control” on page 6-18.

Remote Message Queues

A closely-coupled system is composed of multiple single board computers (SBC) that share a VMEbus. For more information on closely-coupled systems see the Power Hawk Series 600 or 700 *Diskless Systems Administrator’s Guide*. Several families of interfaces are supported for communicating between processes running on separate SBCs in a closely-coupled system. One of those interface families is remote message queues. Refer to the Power Hawk Series 600 or 700 *Closely-Coupled Programming Guide* for more information on the interfaces available in a closely-coupled configuration.

Remote message queues can be used to pass data between processes that are executing on separate SBCs in a closely-coupled configuration. When it is opened, a remote message queue is defined to be resident on one of the SBCs in the closely-coupled configuration. Processes on other SBCs in the closely-coupled configuration can operate on the remote message queue using standard Posix message queue operations, which result in RPC-like messages being passed to the SBC where the remote message queue is resident. It is important to note that the data passed through a remote message queue is stored in buffers on the SBC where the message queue has been defined to be resident. Therefore the SBC where a message queue is resident should be chosen to be the SBC that is most active in sending or receiving data.

The full functionality of the Posix message queue interfaces is available when using a remote message queue.

When programming with remote message queues, the only significant difference in the Posix message queue interfaces is in the specification of the message queue name when creating and removing a remote message queue. For remote message queues, the name of a remote message queue reflects the name of the SBC where the message queue resides. The host name of the SBC where the remote message queue is defined to be resident is prepended to the name of the message queue. Refer to the section on "Using the mq_open Routine" for more information.

The remote message queue implementation utilizes a server process that runs on the SBC where a remote message queue resides. A unique connection is established between a process that opens a remote message queue and this server process. The server process responds to RPC requests to perform operations on the remote message queue. The server process is named **`sbc_msgd(3)`**. **`Sbc_msgd(3)`** will start up by default on the server SBC in a closely-coupled configuration. On client SBCs, **`sbc_msgd(3)`** does not start up by default, so it must be configured to start up by enabling the CCS_IPC **`vmebootconfig(1M)`** subsystem. Refer to the **`sbc_msgd(3)`** and **`vmebootconfig(1M)`** manual pages for more information on this subject.

Understanding Message Queue Library Routines

The POSIX library routines that support message queues depend on a message queue attribute structure. "Understanding the Message Queue Attribute Structure" describes this structure. "Using the Library Routines" presents the library routines.

All applications that call message queue library routines must link in the **`thread`** library. You may link this library either statically or dynamically. (For information about static and dynamic linking, see the "Link Editor and Linking" chapter in *Compilation Systems Volume 1 (Tools)*.) The following example shows the typical command-line format:

`cc [`*options***`] -D_REENTRANT`** *file* **`-lthread`**

Understanding the Message Queue Attribute Structure

The message-queue attribute structure **`mq_attr`** holds status and attribute information about a specific message queue. When a process creates a message queue, it automatically creates and initializes this structure. Every attempt to send messages to or receive messages from this message queue updates the information in this structure. Processes can query the values in this structure.

You supply a pointer to an **`mq_attr`** structure when you invoke **`mq_getattr(3)`** and optionally when you invoke **`mq_open(3)`**. "Using the mq_getattr Routine" and "Using the mq_open Routine", respectively, describe these POSIX message queue library routines.

The **`mq_attr`** structure is defined in **`<mqueue.h>`** as follows:

```
struct mq_attr {
    long mq_flags;
    long mq_maxmsg;
    long mq_msgsize;
    long mq_curmsgs;
};
```

The fields in the structure are described as follows.

mq_flags	a flag that indicates whether or not the operations associated with this message queue are in nonblocking mode
mq_maxmsg	the maximum number of messages this message queue can hold
mq_msgsize	the maximum size in bytes of a message in this message queue. This is also the message slot size for this message queue.
mq_curmsgs	the number of messages currently in this message queue

Using the Library Routines

The POSIX library routines that support message queues are briefly described as follows:

`mq_open`	create and open a new message queue or open an existing message queue
`mq_close`	close an open message queue
`mq_unlink`	remove a message queue and any messages in it
`mq_send`	write a message to an open message queue
`mq_receive`	read the oldest, highest-priority message from an open message queue
`mq_notify`	register for notification of the arrival of a message at an empty message queue such that when a message arrives, the calling process is sent a user-specified signal
`mq_setattr`	set the attributes associated with a message queue
`mq_getattr`	obtain status and attribute information about an open message queue
`mq_remote_timeout`	specify the timeout value for a remote message queue descriptor

Procedures for using each of the routines are presented in the sections that follow.

Using the mq_open Routine

The **`mq_open(3)`** library routine establishes a connection between a calling process and a message queue. Depending on flag settings, **`mq_open`** may create a message queue. The **`mq_open`** routine always creates and opens a new message-queue descriptor. Most other library routines that support message queues use this message-queue descriptor to refer to a message queue.

The specifications required for making the **mq_open** call are as follows:

```
#include <mqueue.h>

mqd_t mq_open(name, oflag [,mode, attr, remote_debug])

char            *name;
int             oflag;
mode_t          mode;
struct mq_attr  *attr;
int             remote_debug
```

The arguments are defined as follows:

name a null-terminated string that specifies the name of a message queue.

The general syntax is:

`[/]` <*ipc_name*>

ipc_name may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretations of it. If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that a leading slash character is provided.

Processes calling **mq_open** with the same value of *name* refer to the same message queue. If the *name* argument is not the name of an existing message queue and you did not request creation, **mq_open** fails and returns an error.

As a non-POSIX extension, remote message queues are also supported. The syntax for specifying a remote message queue is as follows:

`[/]` <*hostname*> / <*ipc_name*>

This specifies the message queue called *ipc_name* on the SBC named *hostname*. Remote message queues are only valid on closely-coupled systems. For more information refer to the section on "Remote Message Queues".

On Power Hawk 620/640 CCS systems, *hostname* must correspond to a valid SBC within the current cluster, and it must match one of the "`VME Hostname`" fields of one of the entries in the **/etc/dtables/nodes.vmeboot** configuration file.

On Power Hawk Series 700 CCS systems, if the message queue resides on a client SBC, then *hostname* must match the corresponding client profile file name located in the **/etc/profiles** directory. If the message queue resides on the server SBC, then *hostname* must match the nodename of the server SBC (the node name returned by **uname(1)** with the **-n** option).

hostname may contain a maximum of 255 characters. *hostname* may also be the host name of the local system. Note that the remote message queue syntax is valid even on systems that are not part of a cluster, if the local system's host name is specified.

When specifying a remote message queue, *ipc_name* is the message queue name and has the same interpretation as in the general syntax case.

oflag an integer value that shows whether the calling process has send and receive access to the message queue; this flag also shows whether the calling process is creating a message queue or establishing a connection to an existing one.

The *mode* a process supplies when it creates a message queue may limit the *oflag* settings for the same message queue. For example, assume that at creation, the message queue *mode* permits processes with the same effective group ID to read but not write to the message queue. If a process in this group attempts to open the message queue with *oflag* set to write access (**O_WRONLY**), **`mq_open`** returns an error.

The only way to change the *oflag* settings for a message queue is to call **`mq_close`** and **`mq_open`** to respectively close and reopen the message queue descriptor returned by **`mq_open`**.

Processes may have a message queue open multiple times for sending, receiving, or both. The value of *oflag* must include exactly one of the three following access modes:

O_RDONLY	Open a message queue for receiving messages. The calling process can use the returned message-queue descriptor with **`mq_receive`** but not **`mq_send`**. For information on **`mq_send`** and **`mq_receive`**, see "Using the mq_send Routine" and "Using the mq_receive Routine", respectively.
O_WRONLY	Open a message queue for sending messages. The calling process can use the returned message-queue descriptor with **`mq_send`** but not **`mq_receive`**.
O_RDWR	Open a message queue for both receiving and sending messages. The calling process can use the returned message-queue descriptor with **`mq_send`** and **`mq_receive`**.
O_CCS_DEBUG	This flag is only meaningful when opening a remote message queue. It indicates that a debug level will be provided, via the parameter **`remote_debug`**. The debug level remains in affect as long as the file is open. For more on remote message queue debug levels, refer to the section "Remote Message Queue Debugging".

The value of *oflag* <u>may</u> include any combination of the remaining flags:

O_CREAT	Create and open an empty message queue if it does not already exist. If message queue *name* is not currently open, this flag causes **`mq_open`** to create an empty message queue. If message queue *name* is already open on the system, the effect of this flag is as noted under **O_EXCL**. When you set the **O_CREAT** flag, you must also specify the *mode* and *attr* arguments. A newly-created message queue has its user ID set to the calling process's effective user ID and its group ID set to the calling process's effective group ID.
O_EXCL	Return an error if the calling process attempts to create an existing message queue. The **`mq_open`** routine fails if **O_EXCL** and **O_CREAT** are set and message queue *name* already exists. The **`mq_open`** routine succeeds if **O_EXCL** and **O_CREAT** are set and message queue *name* does <u>not</u> already exist. The **`mq_open`** routine ignores the setting of **O_EXCL** if **O_EXCL** is set but **O_CREAT** is not set.
O_NONBLOCK	On an **`mq_send`**, return an error rather than wait for a message slot to become free in a full message queue. On an **`mq_receive`**, return an error rather than wait for a message to arrive at an empty message queue.

mode
: an integer value that sets the read, write, and execute/search permission for a message queue if this is the **`mq_open`** call that creates the message queue. The **`mq_open`** routine ignores all other mode bits (for example, setuid). The setting of the file-creation mode mask, **`umask`**, modifies the value of *mode*. For more information on mode settings, see the **`chmod(1)`** and **`umask(2)`** system manual pages.

When you set the **O_CREAT** flag, you must specify the *mode* argument to **`mq_open`**.

attr
: the null pointer constant or a pointer to a structure that sets message-queue attributes--for example, the maximum number of messages in a message queue and the maximum message size. For more information on the **`mq_attr`** structure, see "Understanding the Message Queue Attribute Structure."

If *attr* is NULL, the system creates a message queue that contains 80 message slots of 40 bytes each. If *attr* is <u>not</u> NULL, the system creates the message queue with the attributes specified in this field. If *attr* is specified, it takes effect only when the message queue is actually created.

remote_debug
: This parameter is only meaningful when opening a remote message queue. It specifies the debug level. This parameter is only interpreted if

O_CCS_DEBUG is specified in *oflag*. The debug level remains in affect as long as the remote message queue is open.

For more on remote message queue debug levels, refer to the section "Remote Message Queue Debugging".

A return value of a message-queue descriptor shows that the message queue has been successfully opened. A return value of ((mqd_t) **-1**) shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_open(3)** system manual page for a listing of the types of errors that may occur.

Using the mq_send Routine

The **mq_send(3)** library routine writes a message into an empty message slot in a specific message queue. The **mq_send** routine is an *async-safe* operation; that is, you can call it within a signal-handling routine.

A successful **mq_send** to an empty message queue causes the system to wake the highest priority process that is blocked to receive from that message queue. If a message queue has a notification request attached and no processes blocked to receive, a successful **mq_send** to that message queue causes the system to send a signal to the process that attached the notification request. For more information, read about **mq_receive** in "Using the mq_receive Routine" and **mq_notify** in "Using the mq_notify Routine."

The specifications required for making the **mq_send** call are as follows:

```
#include <mqueue.h>

int mq_send(mqdes, msg_ptr, msg_len, msg_prio)

mqd_t        mqdes;
char         *msg_ptr;
size_t       msg_len;
unsigned int msg_prio;
```

The arguments are defined as follows:

mqdes a message-queue descriptor obtained from an **mq_open**. If the specified message queue is full and **O_NONBLOCK** is set in *mqdes*, the message is not queued, and **mq_send** returns an error. If the specified message queue is full and **O_NONBLOCK** is not set in *mqdes*, **mq_send** blocks until a message slot becomes available to queue the message or until **mq_send** is interrupted by a signal.

Assume that multiple processes are blocked to send a message to a full message queue. When a message slot becomes free in that message queue (because of an **mq_receive**), the system wakes the highest-priority process that has been blocked the longest. This process sends the next message.

For **mq_send** to succeed, the **mq_open** call for this message queue descriptor must have had **O_WRONLY** or **O_RDWR** set in *oflag*. For information on **mq_open**, see "Using the mq_open Routine."

msg_ptr	a string that specifies the message to be sent to the message queue represented by *mqdes*.
msg_len	an integer value that shows the size in bytes of the message pointed to by *msg_ptr*. The **`mq_send`** routine fails if *msg_len* exceeds the *mq_msgsize* message-size attribute of the message queue set on the creating **`mq_open`**. Otherwise, the **`mq_send`** routine copies the message pointed to by the *msg_ptr* argument to a message slot in the message queue.
msg_prio	an unsigned integer value that shows the message priority. The system keeps messages in a message queue in order by message priority. A newer message is queued before an older one only if the newer message has a higher message priority. The value for *msg_prio* ranges from 0 through 31, where 0 represents the least favorable priority. For correct usage, the message priority of an urgent message should exceed that of an ordinary message. Note that message priorities give you some ability to define the message-receipt order but not the message recipient.

Figure 5-2 illustrates message priorities within a message queue and situations where processes are either blocked or are free to send a message to a message queue. Specifically, the following facts are depicted:

- The operating system keeps messages in each message queue in order by message priority.
- Several messages within the same message queue may have the same message priority.
- By default, a process trying to send a message to a full message queue is blocked.

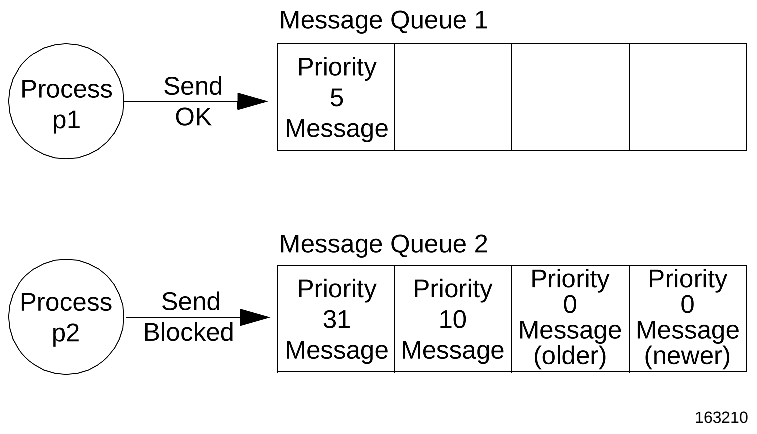


Figure 5-2. The Result of Two mq_sends

A return value of **0** shows that the message has been successfully sent to the designated message queue. A return value of **-1** shows that an error has occurred; **`errno`** is set to show the error. Refer to the **`mq_send(3)`** system manual page for a listing of the types of errors that may occur.

Using the mq_receive Routine

The **mq_receive(3)** library routine reads the oldest of the highest-priority messages from a specific message queue, thus freeing a message slot in the message queue. The **mq_receive** routine is an *async-safe* operation; that is, you can call it within a signal-handling routine.

A successful **mq_receive** from a full message queue causes the system to wake the highest-priority process that is blocked to send to that message queue. For more information, read about **mq_send** in "Using the mq_send Routine."

The specifications required for making the **mq_receive** call are as follows:

```
#include <mqueue.h>

int mq_receive(mqdes, msg_ptr, msg_len, msg_prio)

mqd_t               mqdes;
char                *msg_ptr;
size_t              msg_len;
unsigned int        *msg_prio;
```

The arguments are defined as follows:

mqdes a message-queue descriptor obtained from an **mq_open**. If **O_NONBLOCK** is set in *mqdes* and the referenced message queue is empty, nothing is read, and **mq_receive** returns an error. If **O_NONBLOCK** is not set in *mqdes* and the specified message queue is empty, **mq_receive** blocks until a message becomes available or until **mq_receive** is interrupted by a signal.

Assume that multiple processes are blocked to receive a message from an empty message queue. When a message arrives at that message queue (because of an **mq_send**), the system wakes the highest-priority process that has been blocked the longest. This process receives the message.

For **mq_receive** to succeed, the process's **mq_open** call for this message queue must have had **O_RDONLY** or **O_RDWR** set in *oflag*. For information on **mq_open**, see "Using the mq_open Routine."

Figure 5-3 shows two processes without **O_NONBLOCK** set in *mqdes*. Although both processes are attempting to receive messages, one process is blocked because it is accessing an empty message queue. In the figure, the arrows indicate the flow of data.

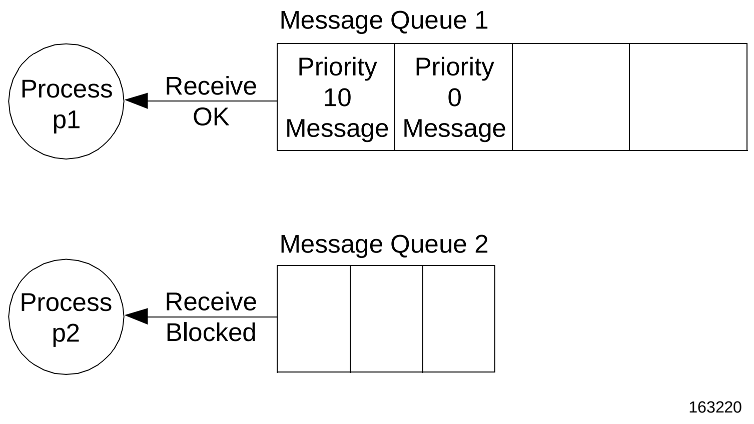


Figure 5-3. The Result of Two mq_receives

msg_ptr a pointer to a character array (message buffer) that will receive the message from the message queue represented by *mqdes*. The return value of a successful **`mq_receive`** is a byte count. This byte count is (1) the message length, (2) the number of characters transferred from the message slot, and (3) the number of characters overwritten in the message buffer. Neither the message nor the message buffer is padded; the message is not null-terminated.

msg_len an integer value that shows the size in bytes of the array pointed to by *msg_ptr*. The **`mq_receive`** routine fails if *msg_len* is less than the *mq_msgsize* message-size attribute of the message queue set on the creating **`mq_open`**. Otherwise, the **`mq_receive`** routine removes the message from the message queue and copies it to the array pointed to by the *msg_ptr* argument.

msg_prio the null pointer constant or a pointer to an unsigned integer variable that will receive the priority of the received message. If *msg_prio* is NULL, the **`mq_receive`** routine discards the message priority. If *msg_prio* is not NULL, the **`mq_receive`** routine stores the priority of the received message in the location referenced by *msg_prio*. The received message is the oldest, highest-priority message in the message queue.

A return value of **`-1`** shows that an error has occurred; **`errno`** is set to show the error and the contents of the message queue are unchanged. A non-negative return value shows the length of the successfully-received message; the received message is removed from the message queue. Refer to the **`mq_receive(3)`** system manual page for a listing of the types of errors that may occur.

Using the mq_notify Routine

The **`mq_notify(3)`** library routine allows the calling process to register for notification of the arrival of a message at an empty message queue. This functionality permits a process to continue processing rather than blocking on a call to **`mq_receive`** to receive a message from a message queue (see “Using the mq_receive Routine” for an explanation of this routine). Note that for a multithreaded program, a more efficient means of attaining this functionality is to spawn a separate thread that issues an **`mq_receive`** call.

At any time, only one process can be registered for notification by a message queue. However, a process can register for notification by each *mqdes* it has open except an *mqdes* for which it or another process has already registered. Assume that a process has already registered for notification of the arrival of a message at a particular message queue. All future attempts to register for notification by that message queue will fail until notification is sent to the registered process or the registered process removes its registration. When notification is sent, the registration is removed for that process. The message queue is again available for registration by any process.

Assume that one process blocks on **`mq_receive`** and another process registers for notification of message arrival at the same message queue. When a message arrives at the message queue, the blocked process receives the message, and the other process’s registration remains pending.

The specifications required for making the **`mq_notify`** call are as follows:

```
#include <mqueue.h>

int mq_notify(mqdes, notification)

mqd_t           mqdes;
struct sigevent *notification;
```

The arguments are defined as follows:

mqdes a message-queue descriptor obtained from an **`mq_open`**.

notification

the null pointer constant or a pointer to a structure that specifies the way in which the calling process is to be notified of the arrival of a message at the specified message queue. If *notification* is not **NULL** and neither the calling process nor any other process has already registered for notification by the specified message queue, **`mq_notify`** registers the calling process to be notified of the arrival of a message at the message queue. When a message arrives at the empty message queue (because of an **`mq_send`**), the system sends the signal specified by the *notification* argument to the process that has registered for notification. Usually the calling process reacts to this signal by issuing an **`mq_receive`** on the message queue.

When notification is sent to the registered process, its registration is removed. The message queue is then available for registration by any process.

If *notification* is **NULL** and the calling process has previously registered for notification by the specified message queue, the existing registration is removed.

If the value of *notification* is not **NULL**, the only meaningful value that *notification->sigevent.sigev_notify* can specify is **SIGEV_SIGNAL**. With this value set, a process can specify a signal to be delivered upon the arrival of a message at an empty message queue.

If you specify **SIGEV_SIGNAL**, *notification->sigevent.sigev_signal* must specify the number of the signal that is to be generated, and *notification->sigevent.sigev_value* must specify an application-defined value that is to be passed to a signal-handling routine defined by the receiving process. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file **`<signal.h>`**. The application-defined value may be a pointer or an integer value. If the process catching the signal has invoked the **`sigaction(2)`** system call with the **SA_SIGINFO** flag set prior to the time that the signal is generated, the signal and the application-defined value are queued to the process when a message arrives at the message queue. For complete information on signal management facilities, the **`sigevent`** structure, and support for specification of an application-defined value, refer to the *PowerMAX OS Programming Guide*.

A return value of **0** shows that the calling process has successfully registered for notification of the arrival of a message at the specified message queue. A return value of **-1** shows that an error has occurred; **`errno`** is set to show the error. Refer to the **`mq_notify(3)`** system manual page for a listing of the types of errors that may occur.

Using the mq_setattr Routine

The **`mq_setattr(3)`** library routine allows the calling process to set the attributes associated with a specific message queue.

The specifications required for making the **`mq_setattr`** call are as follows:

```
#include <mqueue.h>

int mq_setattr(mqdes, mqstat, omqstat)

mqd_t              mqdes;
struct mq_attr     *mqstat;
struct mq_attr     *omqstat;
```

The arguments are defined as follows:

mqdes a message-queue descriptor obtained from an **`mq_open`**. The **`mq_setattr`** routine sets the message queue attributes for the message queue associated with *mqdes.*

mqstat a pointer to a structure that specifies the flag attribute of the message queue referenced by *mqdes*. The value of this flag may be zero or an integer value that sets the following bit:

O_NONBLOCK	causes the **`mq_send`** and **`mq_receive`** operations associated with the message queue to operate in non-blocking mode
MQ_REG_PERSIST	causes the registration for notification to persist across notifications. If this flag is not set, the registration is removed when a notification is delivered.

The following fields are ignored on this call: *mqstat->mq_maxmsg*, *mqstat->mq_msgsize*, and *mqstat->mq_curmsgs*.

For information on the **`mq_attr`** structure, see "Understanding the Message Queue Attribute Structure." For information on the **`mq_send`** and **`mq_receive`** routines, see "Using the mq_send Routine" and "Using the mq_receive Routine," respectively.

omqstat the null pointer constant or a pointer to a structure to which information about the previous attributes and the current status of the message queue referenced by *mqdes* is returned. For information on the **`mq_attr`** structure, see "Understanding the Message Queue Attribute Structure."

A return value of **0** shows that the message-queue attributes have been successfully set as specified. A return value of **-1** shows that an error has occurred; **`errno`** is set to show the error. Refer to the **`mq_setattr(3)`** system manual page for a listing of the types of errors that may occur.

Using the mq_getattr Routine

The **`mq_getattr(3)`** library routine obtains status and attribute information associated with a specific message queue.

The specifications required for making the **`mq_getattr`** call are as follows:

```
#include <mqueue.h>

int mq_getattr(mqdes, mqstat)

mqd_t             mqdes;
struct mq_attr    *mqstat;
```

The arguments are defined as follows:

mqdes a message-queue descriptor obtained from an **`mq_open`**. The **`mq_getattr`** routine provides information about the status and attributes of the message queue associated with *mqdes*.

mqstat a pointer to a structure that receives current information about the status and attributes of the message queue referenced by *mqdes*. For information on the **`mq_attr`** structure, see "Understanding the Message Queue Attribute Structure."

A return value of **0** shows that the message-queue attributes have been successfully attained. A return value of **-1** shows that an error has occurred; **`errno`** is set to show the error. Refer to the **`mq_getattr(3)`** system manual page for a listing of the types of errors that may occur.

Using the mq_close Routine

The **`mq_close(3)`** library routine breaks a connection between a calling process and a message queue. The **`mq_close`** routine does this by removing the message-queue descriptor that the calling process uses to access a message queue. The **mq_close** routine does not affect a message queue itself or the messages in a message queue.

Note

If a process requests notification about a message queue and later closes its connection to the message queue, this request is removed; the message queue is available for another process to request notification. For information on notification requests via **mq_notify**, see “Using the mq_notify Routine.”

The specifications required for making the **mq_close** call are as follows:

```
#include <mqueue.h>

int mq_close(mqdes)

mqd_t     mqdes;
```

The argument is defined as follows:

mqdes a message-queue descriptor obtained from an **mq_open**.

A return value of **0** shows that the message queue has been successfully closed. A return value of **-1** shows that an error has occurred; **errno** is set to show the error. Refer to the **mq_close(3)** system manual page for a listing of the types of errors that may occur.

Using the mq_unlink Routine

The **mq_unlink(3)** library routine prevents further **mq_open** calls to a message queue. When there are no other connections to this message queue, **mq_unlink** removes the message queue and the messages in it.

CAUTION

If a process that is accessing a message queue receives a signal, the process may abort and leave the message queue locked. Before another process can use this message queue, a process must remove the message queue via **mq_unlink** and re-create it via **mq_open**. For more information about how message queues use spin locks and react to signals, see “Understanding Advanced Concepts.”

The specifications required for making the **mq_unlink** call are as follows:

```
#include <mqueue.h>

int mq_unlink(name)
```

```
char *name;
```

The argument is defined as follows:

name a null-terminated string that specifies the name of the message queue to be removed. This string must match the one specified on an mq_open call.

The general syntax is:

`[/]` *<ipc_name>*

i*pc_name* may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretations of it. If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that a leading slash character is provided.

If a process has message queue *name* open when **`mq_unlink`** is called, **`mq_unlink`** immediately returns; destruction of message queue *name* is postponed until all references to the message queue have been closed. A process can successfully remove message queue *name* only if the **`mq_open`** that created this message queue had a *mode* argument that granted the process both read and write permission.

As a non-POSIX extension, remote message queues are also supported. The syntax for specifying a remote message queue is as follows:

`[/]` *<hostname>* / *<ipc_name>*

This indicates that the message queue called *ipc_name* on the SBC named *hostname* is removed. Remote message queues are only valid on closely-coupled systems. For more information refer to the section on “Remote Message Queues".

On Power Hawk 620/640 CCS systems, *hostname* must correspond to a valid SBC within the current cluster, and it must match one of the `"VME Hostname"` fields of one of the entries in the **`/etc/dtables/nodes.vmeboot`** configuration file.

On Power Hawk Series 700 CCS systems, if the message queue resides on a client SBC, then *hostname* must match the corresponding client profile file name located in the **`/etc/profiles`** directory. If the message queue resides on the server SBC, then *hostname* must match the nodename of the server SBC (the node name returned by **`uname(1)`** with the **`-n`** option).

hostname may contain a maximum of 255 characters. *hostname* may also be the host name of the local system. Note that the remote message queue syntax is valid even on systems that are not part of a cluster, if the local system's host name is specified.

ipc_name is the message queue name and has the same interpretation as in the general syntax case.

A return value of **0** shows that a message queue has been successfully removed. A return value of **-1** shows that an error has occurred; **`errno`** is set to show the error. Refer to the **`mq_unlink(3)`** system manual page for a listing of the types of errors that may occur.

Using the mq_remote_timeout Routine

The **`mq_remote_timeout(3)`** library routine establishes a timeout value for a remote message queue descriptor.

Once established, the timeout value will be applied to every operation done to a remote message queue. A timeout occurs if a request is done remotely but a response is not received from the server within the specified time.

When a timeout occurs, the message queue library routines will fail and will return an errno value of ETIMEDOUT.

The specifications required for making the mq_remote_timeout call are as follows:

```
#include <mqueue.h>

int mq_remote_timeout(mqdes, timeout)

mqd_t   mqdes;
int     timeout;
```

The arguments are defined as follows:

mqdes
: a message queue descriptor obtained from **`mq_open`**. This must be a remote message queue (i.e. located on a remote SBC within a closely-coupled cluster). For more information on remote message queues refer to "Remote Message Queues".

  The **`mq_remote_timeout`** routine establishes a timeout value for the message queue descriptor, *mqdes*. Note that each descriptor for a common message queue can have a unique timeout value.

timeout
: timeout value in seconds. A value of zero indicates that timeouts will not occur. A negative value is not legal.

  A return value of 0 indicates that the timeout value has been modified. A return value of -1 shows that an error has occurred; errno is set to show the error. Refer to the **`mq_remote_timeout(3)`** manual page for a listing of the types of errors that may occur.

  The default timeout value is zero; i.e. no timeouts.

It should be noted that there are cases where a message queue routine would normally block. The two most prevalent are:

1. A **`mq_receive(3)`** call is done on an empty message queue and the **O_NONBLOCK** flag is not set.

In this case, the process blocks until a message arrives.

2. A **`mq_send(3)`** call is done on a full message queue and the **O_NONBLOCK** flag is not set.

   In this case, the process blocks until a message is remove.

When such operations are done on a remote message queue, the remote server will block. If a timeout value has been established, then the operation may timeout if the remote server blocks for a long period of time.

Remote Message Queue Debugging

Debugging information is available when using remote message queues. This information can be used to determine the status of messages sent to and replies received from the server located on the SBC where the message queue resides.

Debugging is enabled when the message queue is opened, using **`mq_open(3)`**. The flag **O_CCS_DEBUG** must be specified in the flags parameter. In this case, an additional parameter is provided that is the debug level. Debugging information, at the requested level, will be displayed for all operations on the message queue descriptor until it is closed. Messages are sent to stderr.

Debugging messages have different degrees of importance. Each message has an associated debug number. Lower numbered messages have more importance than higher numbered messages. A message will only be displayed if the debug number associated with the message is lower or equal to the current debug level.

Debug messages are numbered as follows:

10	connection failures vme-messaging mailbox reservation failures
20	connections/disconnections with remote servers
30	signal notification requests and removals
40	general client requests vme-messaging mailbox reservations
60	remote server replies to client requests host name lookups
70	vme-mailbox messages (sends and receives) nodes file operations

It should be noted that there is some amount of overhead involved when debugging is enabled. Debugging should not be enabled in time-critical operations.

It is also possible to obtain debugging information on the server side. This is done by specifying a debug level when invoking the daemon **`sbc_msgd`**. Refer to the **`sbc_msgd(3)`** manual page for more information.

6
Interprocess Synchronization

6
Interprocess Synchronization

Interprocess synchronization refers to coordinating the execution of processes. *Interprocess communication* refers to transferring data between processes. This chapter describes the tools that PowerMAX OS provides to meet a variety of interprocess synchronization needs. All of the interfaces that are described here provide the means for cooperating processes to synchronize access to shared resources. Chapter 5 describes support for real-time interprocess communication.

"Understanding POSIX Counting Semaphores" describes a group of interfaces that provide synchronization through the use of POSIX counting semaphores. These interfaces are based on IEEE Standard 1003.1b-1993. They provide a portable means of synchronizing processes.

"Understanding Synchronization Problems" describes the problems associated with synchronizing cooperating processes' access to data in shared memory. "Using Interprocess Synchronization Tools" describes the tools that have been developed by Concurrent to provide solutions to those problems, and it explains the procedures for using them. The interfaces that are described in "Using Interprocess Synchronization Tools" provide the most efficient means of avoiding priority inversion—a problem that is explained in "Understanding Synchronization Problems."

PowerMAX OS also supports the System V semaphore facilities. These facilities include a set of system calls that is described in the *PowerMAX OS Programming Guide*. These interfaces are much less efficient than the POSIX counting semaphores or the synchronization tools developed by Concurrent.

Understanding POSIX Counting Semaphores

Real-time applications require a synchronization mechanism that allows cooperating processes to coordinate access to the same set of resources—for example, a number of I/O buffers, units of a hardware device, or a critical section of code. A counting semaphore is an object that has an integer value and a limited set of operations defined for it. These operations and the corresponding POSIX interfaces include the following:

- An initialization operation that sets the semaphore to zero or a positive value—**`sem_init`** or **`sem_open`**
- A lock operation that decrements the value of the semaphore—**`sem_wait`**. If the resulting value is negative, the process performing the lock operation blocks.
- An unlock operation that increments the value of the semaphore—**`sem_post`**. If the resulting value is less than or equal to zero, one of the processes blocked on the semaphore is wakened. If the resulting value is greater than zero, no processes were blocked on the semaphore.

- A conditional lock operation that decrements the value of the semaphore only if the value is positive—**`sem_trywait`**. If the value is zero or negative, the operation fails.
- A query operation that provides a snapshot of the value of the semaphore—**`sem_getvalue`**

The lock, unlock, and conditional lock operations are atomic.

A counting semaphore may be used to control access to any resource that can be used by multiple cooperating processes. It is a global entity that is not associated with any process. A counting semaphore may be unnamed or named. A process creates an *unnamed semaphore* by allocating space for the semaphore in memory that can be shared by multiple processes and initializing the semaphore through a call to the **`sem_init`** routine. The semaphore is initialized to a value that is specified on the call. Any process that has access to the memory that contains the unnamed semaphore then has access to the semaphore. A process creates a *named semaphore* by invoking the **`sem_open`** routine and specifying a unique name that is simply a null-terminated string. The semaphore is initialized to a value that is supplied on the call to **`sem_open`** to create the semaphore. No space is allocated by the process for a named semaphore because the **`sem_open`** routine will include the semaphore in the process's virtual address space. Other processes can gain access to the named semaphore by invoking **`sem_open`** and specifying the same name. When an unnamed or a named semaphore is initialized, its value should be set to the number of available resources. To use a counting semaphore to provide mutual exclusion, the semaphore's value should be set to one.

A cooperating process that wants access to a critical resource must lock the semaphore that protects that resource. When the process locks the semaphore, it knows that it can use the resource without interference from any other cooperating process in the system. You must write your application so that the resource is accessed only after the semaphore that protects it has been acquired.

As long as the semaphore value is positive, resources are available for use; one of the resources is allocated to the next process that tries to acquire it. When the semaphore value is zero or negative, none of the resources are available; a process trying to acquire a resource must wait until one becomes available. If the semaphore value is negative, its absolute value is equal to the number of processes that are blocked waiting to acquire one of the resources. When a process completes use of a resource, it unlocks the semaphore, indicating that the resource is available for use by another process.

The concept of ownership does not apply to a counting semaphore. One process can lock a semaphore; another process can unlock it.

The semaphore unlock operation is *async-safe*; that is, a process can unlock a semaphore from a signal-handling routine without causing deadlock.

The absence of ownership prevents priority inheritance. Because a process does not become the owner of a semaphore when it locks the semaphore, it cannot temporarily inherit the priority of a higher-priority process that blocks trying to lock the same semaphore. As a result, unbounded priority inversion can occur. (The priority inversion problem and priority inheritance are explained in detail in “Understanding Synchronization Problems.” Procedures for using synchronization tools and the client system calls to construct sleepy-wait mutual exclusion mechanisms with bounded priority inversion are explained in “Using Interprocess Synchronization Tools.”)

Implementation Issue

Counting semaphores are implemented at user level with some kernel-level support. When a process performs a lock operation that does not require it to block, the process does not enter the kernel. This type of uncontested lock operation is performed very quickly. The counting semaphore implementation affects processes in the following way:

User-level spin locks are used inside the **`sem_wait(3)`** and **`sem_post(3)`** routines to synchronize access to the counting semaphore. While a spin lock is locked, most signals are blocked in order to prevent the application from aborting; however, certain signals cannot be blocked. If a signal causes an application that is using counting semaphores to abort while the spin lock is locked, the counting semaphore cannot be used by any process; all processes that are attempting to access the counting semaphore hang when attempting to gain access to the spin lock. In order for processes to be able to access the counting semaphore again, a process must remove the counting semaphore by invoking **`sem_destroy(3)`** or **`sem_unlink(3)`** and recreate it by invoking **`sem_init(3)`** or **`sem_open(3)`**.

Performance Issue

The counting semaphore implementation uses a rescheduling variable to protect the critical sections inside the semaphore interfaces. If you do not have the privilege required to initialize a rescheduling variable, then none will be used. In this case, the functionality of the semaphore interfaces does not change; however, it is possible to be preempted by higher priority processes from within critical sections. If such a preemption occurs, another LWP or process that is attempting an operation on the counting semaphore will block until the preempted LWP or process is able to run again and complete the critical section. Performance of the semaphore operations may be substantially degraded as a result.

The privilege that is required to initialize a rescheduling variable is `P_RTIME`. In time-critical applications, it is important to ensure that this privilege is granted to all of the processes that access a particular counting semaphore.

Note that the page where a rescheduling variable is located will be locked down in memory. This locked page may prevent the changing of a process's or LWP's CPU bias, if the new CPU bias value requires migrating to a different CPU board on a Night Hawk system.

The procedures for using rescheduling variables are fully explained in “Rescheduling Control” on page 6-18.

Remote Semaphores

A closely-coupled system is composed of multiple single board computers (SBC) that share a VMEbus. For more information on closely-coupled systems see the Power Hawk Series 600 or 700 *Diskless Systems Administrator’s Guide*. Several families of interfaces are supported for communicating between processes that running on separate SBCs in a closely-coupled system. One of those interface families is remote semaphores. Refer to the Power Hawk Series 600 or 700 *Closely-Coupled Programming Guide* for more information on the interfaces available in a closely-coupled configuration.

Remote semaphores allow processes on separate SBCs to synchronize their access to shared memory data structures or to asynchronously notify a process of some event. A remote semaphore is predefined to be resident on one of the SBCs in the closely-coupled configuration. Processes on other SBCs in the closely-coupled configuration can operate on the remote semaphore using standard Posix semaphore operations which result in RPC-like messages being passed to the SBC where the remote semaphore is resident. A test and set operation is performed locally, on the processor where the remote semaphore is resident, to guarantee that only one process can lock the semaphore at any given point in time.

The full functionality of the Posix semaphore interfaces is available when using a remote semaphore.

When programming with remote semaphores, the only difference in the Posix semaphore interfaces is in the creation and attaching to the remote semaphore. The name of a remote semaphore reflects the name of the SBC where the remote semaphore resides. The host name of the SBC where the remote semaphore is resident is prepended to the name of the semaphore. Refer to the section “Using the sem_open Routine” on page 6-7 for more information. The **`sem_init(3)`** routine cannot be used to create of access a remote semaphore.

The remote semaphore implementation utilizes a server process that runs on the SBC where a remote semaphore resides. A unique connection is established between a process that opens a remote semaphore and this server process. The server process responds to RPC requests to perform operations on the remote semaphore. The server process is named **`sbc_msgd(3)`**. On client SBCs, **`sbc_msgd(3)`** does not start up by default, so it must be configured to start up by enabling the CCS_IPC **`vmebootconfig(1M)`** subsystem. Refer to the **`sbc_msgd(3)`** and **`vmebootconfig(1M)`** manual pages for more information on this subject.

Interfaces

The sections that follow explain the procedures for using the POSIX counting semaphore interfaces. These interfaces are briefly described as follows:

`sem_init`	initialize an unnamed counting semaphore
`sem_destroy`	remove an unnamed counting semaphore
`sem_open`	create, initialize, and establish a connection to a named counting semaphore
`sem_close`	relinquish access to a named counting semaphore
`sem_unlink`	remove the name of a named counting semaphore
`sem_wait`	lock a counting semaphore
`sem_trywait`	lock a counting semaphore only if it is currently unlocked

sem_post	unlock a counting semaphore
sem_getvalue	obtain the value of a counting semaphore
sem_remote_timeout	specify the timeout value for a remote semaphore descriptor

Note that to use these interfaces, you must link your application with the threads library. You may link with this library statically or dynamically (refer to *Compilation Systems Volume 1 (Tools)* for information on static and dynamic linking). The following example shows the typical command-line instruction:

cc [*options*] **-D_REENTRANT** *file* **-lthread**

Using the sem_init Routine

The **sem_init(3)** library routine allows the calling process to initialize an unnamed counting semaphore by setting the semaphore value to the number of available resources being protected by the semaphore. To use an unnamed counting semaphore for mutual exclusion, the process sets the value to one.

If you wish to use an unnamed counting semaphore for interprocess synchronization, the semaphore must exist in memory that can be shared by multiple processes. After one process creates and initializes an unnamed semaphore, other cooperating processes can operate on the semaphore by using the **sem_wait**, **sem_trywait**, **sem_post**, and **sem_getvalue** library routines (see “Using the sem_wait Routine,” “Using the sem_trywait Routine,” “Using the sem_post Routine,” and “Using the sem_getvalue Routine,” respectively, for an explanation of these routines). A child process created by a **fork(2)** system call inherits access to an unnamed semaphore that has been initialized by the parent. A process loses access to an unnamed semaphore after invoking the **exec(2)** or **exit(2)** system calls.

CAUTION

The IEEE 1003.1b-1993 standard does not indicate what happens when multiple processes invoke **sem_init** for the same semaphore. Currently, the PowerMAX OS implementation returns an EEXIST error on **sem_init** calls that are made subsequent to the initial call to **sem_init**. Other implementations of **sem_init** may simply reinitialize the semaphore to the value specified on **sem_init** calls made subsequent to the initial call. To be certain that application code can be ported to any system that supports POSIX interfaces (including future Concurrent systems), cooperating processes that use **sem_init** should ensure that a single process initializes a particular semaphore and that it does so only once.

An unnamed counting semaphore is removed by invoking the **sem_destroy** routine (see “Using the sem_destroy Routine” for an explanation of this routine).

The **sem_init(3)** library routine cannot be used to create or access a remote semaphore.

The specifications required for making the **`sem_init`** call are as follows:

```
#include <semaphore.h>

int sem_init(sem, pshared, value)

sem_t *sem;
int pshared;
unsigned int value;
```

The arguments are defined as follows:

sem a pointer to a **`sem_t`** structure that represents the unnamed counting semaphore that is to be initialized

pshared an integer value that indicates whether or not the semaphore to which *sem* points is to be shared by other processes. If *pshared* is set to a non-zero value, then the semaphore is shared among processes. If *pshared* is set to zero, then the semaphore is shared only among threads within a process.

value zero or a positive integer value that initializes the semaphore value to the number of resources currently available. This number cannot exceed the value of **SEM_VALUE_MAX** (see the file <**`limits.h`**> to determine this value).

A return value of **0** indicates that the call to **`sem_init`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_init(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_destroy Routine

The **`sem_destroy(3)`** library routine allows the calling process to remove an unnamed counting semaphore.

CAUTION

An unnamed counting semaphore should not be removed until there is no longer a need for any process to operate on the semaphore and there are no processes currently blocked on the semaphore. Following a successful call to **`sem_destroy`**, attempts to operate on the semaphore will result in an error, and processes that are blocked waiting for the semaphore cannot be wakened.

The **`sem_destroy(3)`** library routine cannot be used to destroy a remote semaphore.

The specifications required for making the **`sem_destroy`** call are as follows:

```
#include <semaphore.h>

int sem_destroy(sem)
```

```
sem_t *sem;
```

The argument is defined as follows:

sem	a pointer to the unnamed counting semaphore that is to be removed. Only a counting semaphore that has been created through a call to **`sem_init(3)`** may be removed by invoking **`sem_destroy`**.

A return value of **0** indicates that the call to **`sem_destroy`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_destroy(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_open Routine

The **`sem_open(3)`** library routine allows the calling process to create, initialize, and establish a connection to a named counting semaphore. When a process creates a named counting semaphore, it associates a unique name with the semaphore. It also sets the semaphore value to the number of available resources being protected by the semaphore. To use a named counting semaphore for mutual exclusion, the process sets the value to one.

After a process creates a named semaphore, other processes can establish a connection to that semaphore by invoking **`sem_open`** and specifying the same name. Upon successful completion, the **`sem_open`** routine returns the address of the named counting semaphore. A process subsequently uses that address to refer to the semaphore on calls to **`sem_wait`**, **`sem_trywait`**, and **`sem_post`** (see “Using the sem_wait Routine,” “Using the sem_trywait Routine,” and “Using the sem_post Routine,” respectively, for an explanation of these routines). A process may continue to operate on the named semaphore until it invokes the **`sem_close`** routine or the **`exec(2)`** or **`exit(2)`** system calls. On a call to **`exec`** or **`exit`**, a named semaphore is closed as if by a call to **`sem_close`**. A child process created by a **`fork(2)`** system call inherits access to a named semaphore to which the parent process has established a connection.

If a single process makes multiple calls to **`sem_open`** and specifies the same name, the same address will be returned on each call unless (1) the process itself has closed the semaphore through intervening calls to **`sem_close`** or (2) some process has removed the name through intervening calls to **`sem_unlink`** (see “Using the sem_close Routine” and “Using the sem_unlink Routine,” respectively, for explanations of these routines).

If multiple processes make calls to **`sem_open`** and specify the same name, the address of the same semaphore object will be returned on each call unless some process has removed the name through intervening calls to **`sem_unlink`**. (Note that the same address will not necessarily be returned on each call.) If a process has removed the name through an intervening call to **`sem_unlink`**, the address of a new instance of the semaphore object will be returned.

The specifications required for making the **`sem_open`** call are as follows:

```
#include <semaphore.h>

sem_t *sem_open(name, oflag [ mode, value, remote_debug ])
```

```
char *name;
int oflag;
mode_t mode;
unsigned int value;
int remote_debug;
```

The arguments are defined as follows:

name a null-terminated string that specifies the name of a semaphore.

The general syntax is:

```
[/] <ipc_name>
```

ipc_name may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretations of it (**`/named_sema and named_sema`** are interpreted as the same name). If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that a leading slash character is provided.

As a non-POSIX extension, remote semaphores are also supported. The syntac for specifying a remote semaphore is as follows:

```
[/] <hostname> / <ipc_name>
```

This specifies the semaphore called *ipc_name* on the SBC named hostname. Remote semaphores are only valid on closely-coupled systems. For more information refer to the section on "Remote Semaphores".

On Power Hawk 620/640 CCS systems, *hostname* must correspond to a valid SBC within the current cluster, and it must match one of the "*VME Hostname*" fields of one of the entries in the **`/etc/dtables/nodes.vmeboot`** configuration file.

On Power Hawk Series 700 CCS systems, if the remote semaphore resides on a client SBC, then *hostname* must match the corresponding client profile file name located in the **`/etc/profiles`** directory. If the remote semaphore resides on the server SBC, then *hostname* must match the nodename of the server SBC (the node name returned by **`uname(1)`** with the **`-n`** option).

hostname may contain a maximum of 255 characters. *hostname* may also be the host name of the local system. Note that the remote semaphore syntax is valid even on systems that are not part of a cluster, if the local system's host name is specified.

When specifying a remote semaphore, *ipc_name* is the semaphore name and has the same interpretation as in the general syntax case.

oflag an integer value that indicates whether the calling process is creating a named counting semaphore or establishing a connection to an existing one. The following bits may be set:

O_CREAT causes the counting semaphore specified by *name* to be created if it does not exist. The semaphore's user ID is set to the effective user ID of the calling process; its group ID is set to the effective group ID of the calling process; and its permission bits are set as specified by the *mode* argument. The semaphore's initial value is set as specified by the *value* argument. Note that you must specify both the *mode* and the *value* arguments when you set this bit.

If the counting semaphore specified by *name* exists, setting **O_CREAT** has no effect except as noted for **O_EXCL**.

O_EXCL causes **`sem_open`** to fail if **O_CREAT** is set and the counting semaphore specified by *name* exists. If **O_CREAT** is not set, this bit is ignored.

Note that the **`sem_open`** routine returns an error if flag bits other than **O_CREAT** and **O_EXCL** are set in the *oflag* argument.

O_CCS_DEBUG

This flag is only meaningful when opening a remote semaphore. It indicates that a debug level will be provided, via the parameter **`remote_debug`**. The debug level remains in affect as long as the file is open.

For more on remote semaphore debug levels, refer to the section on "Remote Semaphore Debugging".

mode an integer value that sets the permission bits of the counting semaphore specified by *name* with the following exception: bits set in the process's file mode creation mask are cleared in the counting semaphore's mode (refer to the **`umask(2)`** and **`chmod(2)`** system manual pages for additional information). If bits other than the permission bits are set in *mode*, they are ignored. A process specifies the *mode* argument only when it is creating a named counting semaphore.

value zero or a positive integer value that initializes the semaphore value to the number of resources currently available. This number cannot exceed the value of **SEM_VALUE_MAX** (see the file <**`sys/limits.h`**> to determine this value). A process specifies the *value* argument only when it is creating a named counting semaphore.

remote_debug

This parameter is only meaningful when opening a remote semaphore. It specifies the debug level. This parameter is only interpreted if **O_CCS_DEBUG** is specified in *oflag*. The debug level remains in affect as long as the semaphore is open.

For more on remote semaphore debug levels, refer to the section on "Remote Semaphore Debugging".

If the call is successful, **`sem_open`** returns the address of the named counting semaphore. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_open(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_close Routine

The **`sem_close(3)`** library routine allows the calling process to relinquish access to a named counting semaphore. The **`sem_close`** routine frees the system resources that have been allocated for the process's use of the semaphore. Subsequent attempts by the process to operate on the semaphore may result in delivery of a SIGSEGV signal.

The count associated with the semaphore is not affected by a process's call to **`sem_close`**.

The specifications required for making the **`sem_close`** call are as follows:

```
#include <semaphore.h>

int sem_close(sem)

sem_t  *sem;
```

The argument is defined as follows:

sem	a pointer to the named counting semaphore to which access is to be relinquished. Only a counting semaphore to which a connection has been established through a call to **`sem_open(3)`** may be specified.

A return value of **0** indicates that the call to **`sem_close`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_close(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_unlink Routine

The **`sem_unlink(3)`** library routine allows the calling process to remove the name of a counting semaphore. A process that subsequently attempts to establish a connection to the semaphore by using the same name will establish a connection to a different instance of the semaphore. A process that has a reference to the semaphore at the time of the call may continue to use the semaphore until it invokes **`sem_close(3)`** or the **`exec(2)`** or **`exit(2)`** system call.

The specifications required for making the **`sem_unlink`** call are as follows:

```
#include <semaphore.h>

int sem_unlink(name)

char  *name;
```

The argument is defined as follows:

name a null-terminated string that specifies the name of the counting semaphore to be removed.

The general syntax is

`[/]` *<ipc_name>*

ipc_name may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretations of it (**`/named_sema`** and **`named_sema`** are interpreted as the same name). If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that a leading slash character is provided.

As a non-POSIX extension, remote semaphores are also supported. The syntac for specifying a remote semaphore is as follows:

`[/]` *<hostname>* / *<ipc_name>*

This indicates that the semaphore called ipc_name on the SBC named hostname is removed. Remote semaphores are only valid on closely-coupled systems. For more information refer to the section on "Remote Semaphores".

On Power Hawk 620/640 CCS systems, *hostname* must correspond to a valid SBC within the current cluster, and it must match one of the "`VME Hostname`" fields of one of the entries in the **`/etc/dtables/nodes.vmeboot`** configuration file.

On Power Hawk Series 700 CCS systems, if the remote semaphore resides on a client SBC, then *hostname* must match the corresponding client profile file name located in the **`/etc/profiles`** directory. If the remote semaphore resides on the server SBC, then *hostname* must match the nodename of the server SBC (the node name returned by **`uname(1)`** with the **`-n`** option).

hostname may contain a maximum of 255 characters. *hostname* may also be the host name of the local system. Note that the remote semaphore syntax is valid even on systems that are not part of a cluster, if the local system's host name is specified.

A return value of **0** indicates that the call to **`sem_unlink`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_unlink(3)`** system manual page for a listing of the types of errors that may occur.

When specifying a remote semaphore, *ipc_name* is the semaphore name and has the same interpretation as in the general syntax case.

Using the sem_wait Routine

The **`sem_wait(3)`** library routine allows the calling process to lock a named or unnamed counting semaphore. If the semaphore value is less than or equal to zero, the semaphore is already locked. In this case, the process blocks until it is interrupted by a signal or the semaphore is unlocked. If the semaphore value is greater than zero, the process locks the semaphore and proceeds. In either case, the semaphore value is decremented.

The specifications required for making the **`sem_wait`** call are as follows:

```
#include <semaphore.h>

int sem_wait(sem)

sem_t  *sem;
```

The argument is defined as follows:

sem a pointer to the named or unnamed counting semaphore that is to be locked

A return value of **0** indicates that the process has succeeded in locking the specified semaphore. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_wait(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_trywait Routine

The **`sem_trywait(3)`** library routine allows the calling process to lock a counting semaphore only if the semaphore value is greater than zero, indicating that the semaphore is unlocked. If the semaphore value is less than or equal to zero, the semaphore is already locked, and the call to **`sem_trywait`** fails. If a process succeeds in locking the semaphore, the semaphore value is decremented; otherwise, it does not change.

The specifications required for making the **`sem_trywait`** call are as follows:

```
#include <semaphore.h>

int sem_trywait(sem)

sem_t  *sem;
```

The argument is defined as follows:

sem a pointer to the named or unnamed counting semaphore that the calling process is attempting to lock

A return value of **0** indicates that the calling process has succeeded in locking the specified semaphore. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_trywait(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_post Routine

The **`sem_post(3)`** library routine allows the calling process to unlock a counting semaphore. If one or more processes are blocked waiting for the semaphore, the waiting process with the highest priority is wakened when the semaphore is unlocked.

The specifications required for making the **`sem_post`** call are as follows:

```
#include <semaphore.h>

int sem_post(sem)

sem_t  *sem;
```

The argument is defined as follows:

sem	a pointer to the named or unnamed counting semaphore that is to be unlocked

A return value of **0** indicates that the call to **`sem_post`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_post(3)`** system manual page for a listing of the types of errors that may occur.

Using the sem_getvalue Routine

The **`sem_getvalue(3)`** library routine allows the calling process to obtain the value of a named or unnamed counting semaphore. If the specified semaphore is locked at the time of the call, the value that the **`sem_getvalue`** routine returns will be zero or a negative number whose absolute value is equal to the number of processes that were blocked waiting for the semaphore at some unspecified time during the call.

The specifications required for making the **`sem_getvalue`** call are as follows:

```
#include <semaphore.h>

int sem_getvalue(sem, sval)

sem_t  *sem;
int  *sval;
```

The arguments are defined as follows:

sem	a pointer to the named or unnamed counting semaphore for which you wish to obtain the value
sval	a pointer to a location to which the value of the specified named or unnamed counting semaphore is to be returned. The value that is returned represents the actual value of the semaphore at some unspecified time during the call. It is important to note, however, that this value may not be the actual value of the semaphore at the time of the return from the call.

A return value of **0** indicates that the call to **`sem_getvalue`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sem_getvalue(3C)`** system manual page for a listing of the types of errors that may occur.

Using the sem_remote_timeout Routine

The **`sem_remote_timeout(3)`** library routine establishes a timeout value for a remote semaphore descriptor.

Once established, the timeout value will be applied to every operation done on a remote semaphore. A timeout occurs if a request is done remotely but a response is not received from the server within the specified time. When a timeout occurs, the semaphore interface routines will fail and will return an errno value of ETIMEDOUT.

The specifications required for making the sem_remote_timeout call are as follows:

```
#include <semaphore.h>

int sem_remote_timeout(sem, timeout)

sem_t    *sem;
int      timeout;
```

The arguments are defined as follows:

sem	a pointer to a remote semaphore as returned by **`sem_open`**. For more information on remote semaphores refer to the section on "Remote Semaphores".
timeout	timeout value in seconds. A value of zero indicates that timeouts will not occur. A negative value is not legal.

A return value of 0 indicates that the timeout value has been modified. A return value of -1 shows that an error has occurred; errno is set to show the error. Refer to the **`sem_remote_timeout(3)`** manual page for a listing of the types of errors that may occur.

The default timeout value is zero; i.e. no timeouts.

It should be noted that there are cases where a semaphore routine would normally block. The most prevalent is:

A **`sem_wait(3)`** call is done on a locked semaphore.

In this case, the process blocks until the semaphore can be obtained.

When this operation is done on a remote semaphore, the remote server will block. If a timeout value has been established, then the operation may timeout if the remote server blocks for a long period of time.

Remote Semaphore Debugging

Debugging information is available when using remote semaphores. This information can be used to determine the status of messages sent to and replies received from the server located on the SBC where the semaphore resides.

Debugging is enabled when the semaphore is opened, using **`sem_open(3)`**. The flag **O_CCS_DEBUG** must be specified in the flags parameter. In this case, an additional parameter is provided that is the debug level. Debugging information, at the requested level, will be displayed for all operations on the semaphore until it is closed. Messages are sent to stderr.

Debugging messages have different degrees of importance. Each message has an associated debug number. Lower numbered messages have more importance than higher numbered messages. A message will only be displayed if the debug number associated with the message is lower or equal to the current debug level.

Debug messages are numbered as follows:

10	connection failures inter-SBC messaging mailbox reservation failures
20	connections/disconnections with remote servers
40	general client requests inter-SBC messaging mailbox reservations
60	remote server replies to client requests host name lookups
70	inter-SBC mailbox messages (sends and receives) nodes file operations

It should be noted that there is some amount of overhead involved when debugging is enabled. Debugging should not be enabled in time-critical operations.

It is also possible to obtain debugging information on the server side. This is done by specifying a debug level when invoking the daemon **`sbc_msgd`**. Refer to the **`sbc_msgd(3)`** manual page for more information.

Understanding Synchronization Problems

Application programs that consist of two or more processes sharing portions of their virtual address space through use of shared memory need to be able to coordinate their access to shared memory efficiently. Two fundamental forms of synchronization are used to coordinate processes' access to shared memory: mutual exclusion and condition synchronization. Mutual exclusion mechanisms serialize cooperating processes' access to shared memory. Condition synchronization mechanisms delay a process's progress until an application-defined condition is met.

Mutual exclusion mechanisms ensure that only one of the cooperating processes can be executing in a critical section at a time. Three types of mechanisms are typically used to provide mutual exclusion—those that involve busy waiting, those that involve sleepy waiting, and those that involve a combination of the two when a process attempts to enter a locked critical section. Busy-wait mechanisms are appropriate if critical sections are quite short and delays are expected to be less than the time required for two context switches. Sleepy-wait mechanisms are preferable if critical sections are quite long and delays are expected to be lengthy.

Critical sections are often very short. To keep the cost of synchronization comparatively small, synchronizing operations performed on entry to and exit from a critical section cannot enter the kernel. The execution overhead associated with entering and leaving the critical section can be longer than the length of the critical section itself.

In order for busy-wait mutual exclusion to be an effective tool, the expected delay must be not only brief but also predictable. Such unpredictable events as page faults, signals, and CPU rescheduling cause the real elapsed time in a critical section to exceed the virtual execution time. At best, such events may cause other CPUs to delay longer than anticipated; at worst, they may cause deadlock. Although some of these events can be controlled with system calls, system call overhead is prohibitive. To ensure that delays are predictable, a low-overhead means of controlling such events needs to be provided.

In the context of mutual exclusion, the priority inversion problem surfaces. Priority inversion occurs when one or more low-priority processes prevent the progress of a high-priority process. The problem is illustrated by the following example. Assume that **H** is a high-priority process, **M** a medium-priority process, **L** a low-priority process, and **`mutex`** a mutual exclusion synchronizing variable; assume also that **L** is the currently running process.

> **L** locks **`mutex`** and enters a critical section
>
> **M** preempts **L**
>
> **M** begins a lengthy computation
>
> **H** preempts **M**
>
> **H** attempts to lock **`mutex`**
>
> **H** blocks, waiting for **L** to leave the critical section and release **`mutex`**
>
> **M** regains the CPU and continues its computation

H cannot continue until **L** leaves the critical section; **L** is prevented from running by **M**. For all practical purposes, **H** assumes **L**'s low priority and blocks for an unpredictable, unbounded period of time.

One approach to preventing priority inversion in such situations as this is to raise the priority of **L** when it enters a critical section. If raising the priority is inexpensive, this approach may be preferable when a critical section is short and busy-wait mutual exclusion can be used. If a critical section is long enough to warrant use of sleepy-wait mutual exclusion, however, this approach may be too conservative: it will defer the preemption of a low-priority process in situations in which inversion would not have occurred.

Another approach is to provide a means for correcting a priority inversion when it occurs. An approach known as *priority inheritance* involves passing the priority of a waiting process to the process that is in the critical section. The process in the critical section "inherits" the priority of the processes waiting to enter the critical section. The net effect is to allow the low-priority process to execute long enough to leave the critical section.

Priority inheritance needs to be transitive. If high-priority process **H** were blocked on a critical section occupied by medium-priority process **M** and if **M** were blocked on a different critical section occupied by low-priority process **L**, both **M** and **L** should inherit **H**'s priority.

Using Interprocess Synchronization Tools

To provide solutions to the problems described in "Understanding Synchronization Problems," PowerMAX OS supplies a variety of interprocess synchronization tools. These tools are based on the concept of a lightweight process. They are designed to be used by single-threaded processes or bound threads.

The synchronization tools that are supplied include tools for controlling an LWP's vulnerability to rescheduling, serializing LWPs' access to critical sections with busy-wait mutual exclusion mechanisms, and coordinating interaction among LWPs. From these tools, a mechanism for providing sleepy-wait mutual exclusion with bounded priority inversion can be constructed. Tools for providing rescheduling control are described in "Rescheduling Control." Tools for implementing busy-wait mutual exclusion are explained in "Busy-Wait Mutual Exclusion." Tools for coordinating interaction between LWPs are described in "Client-Server Coordination." Procedures for implementing sleepy-wait mutual exclusion are presented in "Constructing Sleepy-Wait Mutual Exclusion Tools." An example program that illustrates use of the interprocess synchronization tools is provided in Appendix B.

The interprocess synchronization tools can be used to synchronize execution of LWPs in any process in the system. In this respect, they differ from most other PowerMAX OS interfaces that operate on LWPs (**priocntl(2)**, **cpu_bias(2)**, **_lwp_suspend(2)**, for example) because those interfaces allow an LWP to operate only on LWPs in the same process. The rescheduling control and client-server coordination tools that are described in subsequent sections of this chapter use a unique, global identifier to identify an LWP. The global LWP identifier (**global_lwpid_t**), which is defined in <**sys/types.h**>, differs from the LWP identifier (**lwpid_t**) in that it uniquely identifies an LWP system-wide. Note that neither the global LWP ID nor the LWP ID is the same as the process identifier (PID).

You can obtain the global LWP ID for an LWP with the **_lwp_global_self(2)** system call. The specification required for making this call is as follows:

```
global_lwpid_t  _lwp_global_self(void)
```

The **_lwp_global_self** system call returns a nonzero value that is the unique global identifier assigned to the calling LWP. For additional information, refer to the system manual page **_lwp_global_self(2)**.

Rescheduling Control

To use busy-wait mutual exclusion effectively, lock hold times must be small and predictable. Rescheduling and signal handling are major sources of unpredictability. To provide you with the means to control rescheduling and signal handling, a rescheduling variable has been developed. You allocate the variable in your application, notify the kernel of its location, and manipulate it directly from your application to disable and re-enable rescheduling. While rescheduling is disabled, quantum expirations, preemptions, and certain types of signals are held.

A system call and a set of macros accommodate use of the rescheduling variable. In the sections that follow, the variable, the system call, and the macros are described, and the procedures for using them are explained.

Although these interfaces can be used by multithreaded processes, it is recommended that they be used only by single-threaded processes or bound threads. The reasons are explained as follows:

A rescheduling variable is valid only for the LWP that informs the kernel of its location (see "Using the resched_cntl System Call"and the explanation of the RESCHED_SET_VARIABLE command). The global LWP ID of a multiplexed thread changes according to the LWP on which the thread is currently scheduled. A multiplexed thread cannot reliably determine the LWP on which it is scheduled. As a result, a multiplexed thread may inform the kernel of the location of its rescheduling variable and then be assigned to a different LWP before it locks the variable (see "Using the Rescheduling Control Macros" and the explanation of the **resched_lock** macro). In such cases, a multiplexed thread can lock the wrong rescheduling variable. If it does so, it will fail to receive any of the protection that comes from disabling rescheduling. The problem is a serious one because the call to **resched_lock** will succeed; the multiplexed thread will have no indication that the wrong rescheduling variable has been locked.

For information on threads programming and threads management facilities, refer to the *PowerMAX OS Programming Guide*.

Understanding Rescheduling Variables

A rescheduling variable is a data structure that controls a lightweight process's vulnerability to rescheduling. In your application program, you must allocate one variable for each LWP for which you wish to defer rescheduling.

A rescheduling variable is defined as follows:

```
struct resched_var {
        pid_t rv_pid;
        global_lwpid_t rv_glwpid;
    ...
};
```

As shown, a rescheduling variable contains both the process identifier and the global LWP identifier of the LWP that owns the variable. These fields are included for your convenience. Some of the system calls that are used to coordinate interaction among LWPs require that you specify a global LWP identifier as an argument (see "Client-Server Coordination"). The remaining fields are not meant to be referenced directly by an application.

A system call, **resched_cntl(2)**, enables you to perform a variety of operations specific to the rescheduling variable. Use of this call is explained in "Using the resched_cntl System Call." A set of rescheduling control macros enables you to manipulate the variable from your application. Use of these macros is explained in "Using the Rescheduling Control Macros."

NOTE

The Ada environment provides functionality for these services through the `rescheduling_control` package. Refer to the "Additional Support Packages" chapter of the *HAPSE Reference Manual* for additional information.

Using the resched_cntl System Call

The **`resched_cntl`** system call enables you to perform a variety of rescheduling control operations. These include initializing a rescheduling variable, informing the kernel of its location, obtaining its location, and setting a limit on the length of time that rescheduling can be deferred.

The specifications required for using the **`resched_cntl`** call are as follows:

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

int resched_cntl(cmd, arg)

int cmd;
char *arg;
```

Arguments are defined as follows:

cmd	the operation to be performed
arg	a pointer to an argument whose value depends upon the value of *cmd*

Cmd can be one of the following. The values of *arg* that are associated with each command are indicated.

RESCHED_SET_VARIABLE
: inform the kernel of the location of the calling LWP's rescheduling variable, and initialize the variable

or

dissociate the calling LWP from an existing rescheduling variable

In the first case, *arg* points to the rescheduling variable; when the call is completed, the variable will be initialized, and the page(s) in which it is located will be locked in physical memory. The rescheduling variable must be located in a process private page, which excludes pages in shared memory segments or files that have been mapped MAP_SHARED. Note that two LWPs within a single process should not specify the same address for their rescheduling variables.

In the second case, *arg* is **NULL**; when the call is completed, the page(s) that contained the variable is unlocked.

After a **fork(2)**, the child process inherits a rescheduling variable from its parent. The rv_pid and rv_glwpid fields of the child's rescheduling variable are updated to the process ID and global LWP ID of the child. If a child process has inherited a rescheduling variable and it, in turn, forks one or more child processes, those child processes will inherit the rescheduling variable with the rv_pid and rv_glwpid fields updated. If a rescheduling variable is locked in the parent process at the time of the call to **fork**, the rescheduling variable will be locked in the child process.

Note that to use this command, the calling LWP must have the P_RTIME privilege (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **intro(2)** system manual page).

RESCHED_GET_VARIABLE — obtain the location of the calling LWP's rescheduling variable

In this case, *arg* must point to a rescheduling variable pointer. The pointer referenced by *arg* is set to **NULL** if the caller has no rescheduling variable; otherwise, it is set to the location of the rescheduling variable.

RESCHED_SET_LIMIT — define the maximum length of time that rescheduling of the calling LWP can be deferred

or

clear a previously specified time limit

In the first case, *arg* points to a **timeval** structure that contains the time limit; when this limit is exceeded, the SIGRESCHED signal is sent to the calling LWP. The default action of the signal is to terminate the LWP, but the signal can also be caught or ignored.

In the second case, *arg* is **NULL**; when the call is completed, a previously specified time limit is cleared.

This command is provided as a debugging tool.

RESCHED_YIELD — relinquish control of the CPU to other LWPs of equal priority. If there are no other LWPs of equal priority, the calling LWP retains control of the CPU. The value of *arg* must be zero.

A return of **0** indicates that the requested operation has been successful. A return of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. For additional information on the use of this call, refer to the system manual page **`resched_cntl(2)`**.

Using the Rescheduling Control Macros

A set of rescheduling control macros enables you to disable and re-enable rescheduling and to determine the number of rescheduling locks in effect. These macros are briefly described as follows:

`resched_lock`	increment the number of rescheduling locks held by the calling LWP, and disable rescheduling
`resched_unlock`	decrement the number of rescheduling locks held by the calling LWP. If the resulting number of rescheduling locks is zero, rescheduling is re-enabled.
`resched_nlocks`	return the number of rescheduling locks currently held by the calling LWP

The **`resched_lock`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>

void resched_lock(r)

struct resched_var *r;
```

The argument is defined as follows:

r	a pointer to the calling LWP's rescheduling variable

`Resched_lock` does not return a value; it increments the number of rescheduling locks held by the calling LWP. Calls to **`resched_lock`** can be nested. Rescheduling will be disabled as long as the number of locks held is not zero.

If the calling LWP does not enter the kernel, quantum expirations, preemptions, and signals other than those representing error conditions are deferred until all of the rescheduling locks held by the LWP are released. Signals that represent error conditions and other events that should not be deferred do not affect rescheduling locks. These signals are as follows: SIGILL, SIGTRAP, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGXCPU, SIGXFSZ, and SIGRESCHED.

If the LWP enters the kernel via a page fault or a system call, it will receive signals and be subject to context switching regardless of the number of rescheduling locks it holds.

The **`resched_unlock`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>

void resched_unlock(r)

struct resched_var *r;
```

The argument is defined as follows:

r	a pointer to the calling LWP's rescheduling variable

`Resched_unlock` does not return a value; it decrements the number of rescheduling locks held by the calling LWP. If the number of rescheduling locks is zero after the number is decremented, a pending context switch or signal will be serviced.

NOTE

If the number of **`resched_unlock`** requests exceeds the number of **`resched_lock`** requests, the number of locks held by the calling LWP will be negative. You may wish to use **`resched_nlocks`** to make the appropriate assertion in your application program. For information on use of **`assert(3X)`**, refer to the corresponding system manual page.

The **`resched_nlocks`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>

int resched_nlocks(r)

struct resched_var *r;
```

The argument is defined as follows:

r	a pointer to the calling LWP's rescheduling variable

`Resched_nlocks` returns the number of rescheduling locks currently held by the calling LWP. You may wish to use this call in an **`assert`** statement to determine whether or not the number of locks held is negative.

For additional information on the use of these macros, refer to the system manual page **`resched_cntl(2)`**.

Applying Rescheduling Control Tools

The following C program segment illustrates the procedures for controlling rescheduling by using the tools described in the preceding sections. This program segment defines a rescheduling variable (**`rv`**) as a global variable; initializes the variable with a call to **`resched_cntl`**; and disables and re-enables rescheduling with calls to **`resched_lock`** and **`resched_unlock`**, respectively.

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

   ...

   struct resched_var rv;
   ...

void
main()
{
   resched_cntl(RESCHED_SET_VARIABLE, &rv);

   resched_lock(&rv);

   ...     /* nonpreemptible code */

   assert(resched_nlocks(&rv) > 0);
   resched_unlock(&rv);
}
```

Rescheduling Variables and Ada

ADA applications or applications that make use of POSIX semaphores or POSIX message queues should not register rescheduling variables from initialization (`.initp`) sections. This is because the execution order of Ada initialization sections cannot be guaranteed to precede other user-defined initialization sections, and also because the threads library's internal use of rescheduling variables are delayed until needed, as previously mentioned.

Since ADA and the threads library might internally use rescheduling variables, these variables must be preserved. Before defining a rescheduling variable, use the **`RESCHED_GET_VARIABLE`** command to see if one already exists:

- If already defined, share rather than removing it to register your own.
- If not already defined, define and register your own rescheduling variable.

Rescheduling Variables and Processor Migration

The page where a rescheduling variable resides is locked down in memory by the kernel when the rescheduling variable is defined with the **`RESCHED_SET_VARIABLE`** command. In a multi-CPU board Night Hawk system, a rescheduling variable that is locked down in a local memory page will prevent the associated process and/or LWP from migrating to a different CPU board (see **`cpu_bias(2)`** or the *PowerMAX OS Programming Guide* Chapter 4, Process Management). Furthermore, since the address space of a multi-threaded application is shared between all threads in the process, any thread's rescheduling variable that is locked down in a local memory pool will prevent all other threads/LWPs in the process from migrating to another CPU board.

The threads library internally uses rescheduling variables for POSIX semaphores and message queues. The threads library delays defining internal rescheduling variables for all the threads in a process until the first thread in the process makes a **`sem_open(3)`**, **`sem_init(3)`**, **`sem_wait(3)`**, **`sem_post(3)`**, or **`mq_open(3)`** function call. The delaying of setting up the rescheduling variables is intended to aid in an application's ability to change the CPU bias mask of the process or the LWPs within the process before memory pages are locked down due to these internal rescheduling variables.

Therefore, for applications that make use of POSIX semaphores and/or POSIX message queues in a multi-CPU board Night Hawk system with local memory binding for their private data area, it is highly recommended that the application first change the CPU bias of the process or any thread within the process before calling the first **`sem_open(3)`**, **`sem_init(3)`**, **`sem_wait(3)`**, **`sem_post(3)`**, or **`mq_open(3)`** function call.

Also, applications that make use of both POSIX semaphores and/or message queues and also make direct use of a rescheduling variable should first call a least one of the **`sem_open(3)`**, **`sem_init(3)`**, **`sem_wait(3)`**, **`sem_post(3)`**, or **`mq_open(3)`** functions before checking to see if a rescheduling variable is already defined with the **`RESCHED_GET_VARIABLE`** command.

Since rescheduling variables are inherited by child processes across **`fork(2)`** calls, the creation of child processes that will be doing cross-CPU board migrations with local memory bindings on a Night Hawk system should be created via **`fork(2)`** by the parent process before calling any of the previously mentioned semphore or message queue routines or before an application defines its own rescheduling variable(s).

Busy-Wait Mutual Exclusion

Busy-wait mutual exclusion is achieved by associating a synchronizing variable with a shared resource. When an LWP wishes to gain access to the resource, it locks the synchronizing variable. When it completes its use of the resource, it unlocks the synchronizing variable. If another LWP attempts to gain access to the resource while the first LWP has the resource locked, that LWP must delay by repeatedly testing the state of the lock. This form of synchronization requires that the synchronizing variable be accessible directly from user mode and that the lock and unlock operations have very low overhead.

PowerMAX OS busy-wait mutual exclusion tools include a low-overhead busy-wait mutual exclusion variable (a spin lock) and a corresponding set of macros. In the sections that follow, the variable and the macros are defined, and the procedures for using them are explained.

The threads library, **`libthread`**, also provides a set of spin lock routines. Those routines are described in the *PowerMAX OS Programming Guide*. It is recommended that you use the macros described in this chapter instead of the routines because the macros are more efficient and they give you more flexibility; for example, the spin lock macros allow you to construct a synchronization primitive that is a combination of the busy-wait and sleepy-wait primitives. If you were to construct such a primitive, the primitive would gain access to the lock by spinning for some number of spins and then blocking if the lock were not available. The advantage that this type of lock offers is that you do not have to use rescheduling variables to prevent deadlock.

Understanding the Busy-Wait Mutual Exclusion Variable

The busy-wait mutual exclusion variable is a data structure known as a spin lock. This variable is defined in **`<sys/lwp_synch.h>`** as follows:

```
struct spin_mutex {

     ...

};
```

The spin lock has two states: locked and unlocked. When initialized, the spin lock is in the unlocked state.

If you wish to use spin locks to coordinate access to shared resources, you must allocate them in your application program and locate them in memory that is shared by the processes or LWPs that you wish to synchronize. You can manipulate them by using the macros described in “Using the Busy-Wait Mutual Exclusion Macros.”

NOTE

The Ada environment provides functionality for these services through the `spin_locks` package. Refer to the “Additional Support Packages” chapter of the *HAPSE Reference Manual* for additional information.

Using the Busy-Wait Mutual Exclusion Macros

A set of busy-wait mutual exclusion macros allows you to initialize, lock, and unlock spin locks and determine whether or not a particular spin lock is locked. These macros are briefly described as follows:

`spin_init`	initialize a spin lock to the unlocked state
`spin_trylock`	attempt to lock a specified spin lock
`spin_unlock`	unlock a specified spin lock
`spin_islock`	determine whether or not a specified spin lock is locked

It is important to note that none of these macros enables you to lock a spin lock unconditionally. You can construct this capability by using the tools that are provided.

CAUTION

Operations on spin locks are not recursive; an LWP can deadlock if it attempts to relock a spin lock that it has already locked.

You must initialize spin locks before you use them by calling the **`spin_init`** macro. You call **`spin_init`** only once for each spin lock. If the specified spin lock is locked, **`spin_init`** effectively unlocks it. The **`spin_init`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>

void spin_init(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m the starting address of the spin lock to be initialized

`Spin_init` does not return a value; it places the spin lock in the unlocked state.

The **`spin_trylock`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>

int spin_trylock(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m a pointer to the spin lock that you wish to try to lock

A return of `TRUE` indicates that the calling LWP has succeeded in locking the spin lock. A return of `FALSE` indicates that it has not succeeded. **`Spin_trylock`** does not block the calling LWP.

Note that the Concurrent C compiler generates in-line code for this routine if you specify the **`-F`** option when you invoke the compiler (for additional information on use of the **`-F`** option, refer to the system manual page **`cc(1)`**).

The **`spin_unlock`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>

void spin_unlock(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m a pointer to the spin lock that you wish to unlock

`Spin_unlock` does not return a value.

The **`spin_islock`** macro is specified as follows:

```
#include <sys/types.h>
#include <sys/lwp_synch.h>
```

```
int spin_islock(m)

struct spin_mutex *m;
```

The argument is defined as follows:

m	a pointer to the spin lock whose state you wish to determine

A return of `TRUE` indicates that the specified spin lock is locked. A return of `FALSE` indicates that it is unlocked. **`Spin_islock`** does not attempt to lock the spin lock.

For additional information on the use of these macros, refer to the system manual page **`spin_trylock(2)`**.

Applying Busy-Wait Mutual Exclusion Tools

Procedures for using the tools for busy-wait mutual exclusion are illustrated by the following code segments. The first segment shows how to use these tools along with rescheduling control to acquire a spin lock; the second shows how to release a spin lock. Note that these segments contain no system calls; they will contain no procedure calls when you invoke the Concurrent C compiler and specify the **`-F`** option (for additional information on use of the **`-F`** option, refer to the system manual page **`cc(1)`**).

The `_m` argument points to a spin lock, and the `_r` argument points to the calling LWP's rescheduling variable. It is assumed that the spin lock is located in shared memory. To avoid the overhead associated with paging and swapping, it is recommended that the pages that will be referenced inside the critical section be locked in physical memory (see the *PowerMAX OS Programming Guide* for an explanation of the procedures for using the **`mlock(3C)`** library routine and the **`shmctl(2)`** system call).

```
#define spin_acquire(_m,_r) \
{ \
   resched_lock(_r); \
   while (!spin_trylock(_m)) { \
      resched_unlock(_r); \
      while (spin_islock(_m)); \
      resched_lock(_r); \
   } \
}


#define spin_release(_m,_r) \
{ \
   spin_unlock(_m); \
   resched_unlock(_r); \
}
```

In the first segment, note the use of the **`spin_trylock`** and **`spin_islock`** macros. If an LWP attempting to lock the spin lock finds it locked, it waits for the lock to be released by calling **`spin_islock`**. This sequence is more efficient than polling directly with **`spin_trylock`**. The **`spin_trylock`** macro contains special instructions to perform test-and-set atomically on the spin lock. These instructions are less efficient than the simple memory read performed in **`spin_islock`**.

Note also the use of the rescheduling control macros. To prevent deadlock, an LWP disables rescheduling prior to locking the spin lock and re-enables it after unlocking the spin lock. An LWP also re-enables rescheduling just prior to the call to **`spin_islock`** so that rescheduling is not deferred any longer than necessary.

Client-Server Coordination

PowerMAX OS condition synchronization tools are based on the idea of a client-server relationship between cooperating LWPs. A *client* LWP is one that requests service from another LWP. A *server* LWP is one that satisfies a client's request for service. When a client requests service, it usually waits for a response from the selected server. When the server completes the request, it wakes the corresponding client. If another request is pending, the server handles that request; otherwise, it blocks to wait for the arrival of the next request. An LWP may act as both client and server during its lifetime.

When a client waits for a server, its priority is passed to that server. Priority inheritance requires that the priority of a server be at least as high as that of any of its clients.

Interaction between client and server LWPs is handled by two sets of system calls. The client system calls are described in "Using the Client System Calls"; the server system calls are described in "Using the Server System Calls." Examples of their use are provided in each case.

NOTE

The Ada environment provides functionality for these services through the `client_server_services` package. Refer to the "Additional Support Packages" chapter of the *HAPSE Reference Manual* for additional information.

Using the Client System Calls

A set of client system calls manipulates LWPs that are acting as clients. These calls provide a priority inheritance mechanism. They allow an LWP to block and establish a client-server relationship, and they allow one or more LWPs that are blocked waiting on a particular server to be wakened. These system calls are briefly described as follows:

`client_block`	block the calling LWP (a client) and pass its priority to another LWP (a server)
`client_wake1`	wake a single client that is blocked in the **`client_block`** call
`client_wakechan`	wake all clients that are members of a specified group and are blocked in the **`client_block`** call

CAUTION

These system calls should be used only by single-threaded processes or by bound threads. The global LWP ID of a multiplexed thread changes according to the LWP on which the thread is currently scheduled. If these interfaces are used by multiplexed threads, it is possible that the wrong thread will be wakened. For information on threads programming and threads management facilities, refer to the *PowerMAX OS Programming Guide*.

The specifications required for making the **`client_block`** call are as follows:

```
int client_block(server, chan, options, m, r, timeout)

global_lwpid_t server;
int chan, options;
struct spin_mutex *m;
struct resched_var *r;
struct timeval *timeout;
```

Arguments are defined as follows:

server	the global LWP ID of the LWP from which service is being requested
chan	an integer value that is used to categorize client LWPs. This argument is optional; it is used by the **`client_wakechan`** system call.
options	an integer value that indicates whether or not the LWP specified by *server* is to be wakened if it is blocked in the **`server_block`** system call. If the value is **1**, the server is wakened as provided by the **`server_wake1`** system call (see "Using the Server System Calls"); if the value is **0**, the server is not wakened.
m	a pointer to a spin lock located in a shared memory region. This argument is optional; its value can be **NULL**.
r	a pointer to the calling LWP's rescheduling variable. This argument is optional; its value can be **NULL**.
timeout	a pointer to a **`timeval`** structure that contains the maximum length of time that the calling LWP will be blocked. This argument is optional; its value can be **NULL**. If its value is **NULL**, there is no time out.

It is important to note that to use the **`client_block`** call, the real or effective user ID of the calling LWP must match the real or effective user ID of the LWP specified by *server*, or the calling LWP must have the `P_OWNER` privilege (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **`intro(2)`** system manual page).

If the Enhanced Security Utilities are installed and running, one of the following conditions must be met:

- The Mandatory Access Control (MAC) level of the calling LWP must equal the MAC level of the target LWP.

- The calling LWP must have the `P_MACWRITE` or the `P_COMPAT` privilege.
- The target LWP must have the `P_COMPAT` privilege.

`Client_block` releases the spin lock specified by *m*, decrements the number of rescheduling locks associated with the rescheduling variable specified by *rv*, and blocks the calling LWP. The unlock and block operations are executed atomically to ensure that the calling LWP does not miss a wake-up. While the client is blocked in this system call, the specified server will have a priority that is at least as high as the client's.

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Note that upon return, the calling LWP should retest the condition that caused it to block; there is no guarantee that the condition has changed.

The specifications required for making the **`client_wake1`** call are as follows:

```
int client_wake1(server, client, r)

global_lwpid_t server, client;
struct resched_var *r;
```

Arguments are defined as follows:

server	must be zero
client	the global LWP ID of a client that has blocked in **`client_block`** and specified the calling LWP as its server
r	a pointer to the calling LWP's rescheduling variable. This argument is optional.

`Client_wake1` wakes the specified client if it is blocked in the **`client_block`** call, requesting service from the calling LWP. If the client is not blocked in this call, **`client_wake1`** does not affect it. **`Client_wake1`** also decrements the number of rescheduling locks associated with a rescheduling variable specified by *r*.

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error.

The specifications required for making the **`client_wakechan`** call are as follows:

```
int client_wakechan(server, chan, r)

global_lwpid_t server;
int chan;
struct resched_var *r;
```

Arguments are defined as follows:

server	must be zero
chan	an integer value that identifies the category of clients to be wakened by the LWP specified by *server*. If this value is zero, all of the calling LWP's clients will be wakened.

r a pointer to the calling LWP's rescheduling variable. This argument is optional.

`Client_wakechan` wakes the clients in the category specified by *chan* if they are blocked in the **`client_block`** call, requesting service from the calling LWP. If the value of *chan* is zero, it wakes all of the server's clients. **`Client_wakechan`** also decrements the number of rescheduling locks associated with a rescheduling variable specified by *r*.

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error.

For additional information on the use of these calls, refer to the system manual page **`client_block(2)`**.

Constructing Sleepy-Wait Mutual Exclusion Tools

If the delay at entry to a critical section is expected to be lengthy, sleepy-wait mutual exclusion is preferable to busy-wait mutual exclusion, where the CPU idles for potentially long periods of time. Sleepy-wait mutual exclusion mechanisms can be provided by using both the busy-wait mutual exclusion tools and the client system calls described in the previous section.

A sleepy-wait mutual exclusion variable can be defined as follows:

```
   struct sleep_mutex {
            struct spin_mutex mx;
            global_lwpid_t owner;
            int waiters;
   };
```

The `mx` field specifies the spin lock that serializes access to the owner and waiters fields. The `owner` field identifies the LWP that holds the sleepy-wait mutual exclusion variable. The `waiters` field indicates whether or not LWPs are blocked on the mutex.

Using a rescheduling variable (**`rv`**) and the **`spin_acquire`** and the **`spin_release`** functions defined in "Applying Busy-Wait Mutual Exclusion Tools," a function for locking the sleepy-wait mutex can be defined as follows:

```
void
sleep_lock(s)
        struct sleep_mutex *s;
{
    spin_acquire(&s->mx, &rv);
    while (s->owner) {
            s->waiters = 1;
            client_block(s->owner, s, 0, &s->mx, &rv, 0);
            spin_acquire(&s->mx, &rv);
    }
    s->owner = rv.rv_glwpid;
    spin_release(&s->mx, &rv);
}
```

Note that in this function, in what is an unusual interpretation of the client-server relationship, the owner of the sleepy-wait mutual exclusion variable acts as server and the waiters as clients. The call to **`client_block`** ensures that the owner's priority will be at least as high as that of any of the waiters. In this example, it is assumed that a sleepy-wait mutual exclusion variable appears at the same location in the shared portion of every LWP's address space; consequently, the address of that variable is used to categorize the waiters.

A function for unlocking the sleepy-wait mutual exclusion variable can be defined as follows:

```
void
sleep_unlock(s)
        struct sleep_mutex *s;
{
    int were_waiters;

    spin_acquire(&s->mx, &rv);
    s->owner = 0;
    were_waiters = s->waiters;
    s->waiters = 0;
    spin_unlock(&s->mx);

    if (were_waiters)
            client_wakechan(0, s, &rv);
    else
            resched_unlock(&rv);
}
```

Note that in this function, when an owner releases the sleepy-wait mutual exclusion variable, it wakes all of the waiters with a call to **`client_wakechan`**. When they execute, the wakened waiters must recontend for the mutex. One of them will become the new owner; the others will block and establish new priority-inheritance relationships.

Using the Server System Calls

A set of server system calls has been developed to enable you to manipulate LWPs acting as servers. These system calls are briefly described as follows:

`server_block`	block the calling LWP only if no wake-up request has occurred since the last return from **`server_block`**
`server_wake1`	wake a single server that is blocked in the **`server_block`** system call; if the specified server is not blocked in this call, the wake-up request is applied to the server's next call to **`server_block`**
`server_wakevec`	wake a group of servers that are blocked in the **`server_block`** system call; if a specified server is not blocked in this call, the wake-up request is applied to the server's next call to **`server_block`**

CAUTION

These system calls should be used only by single-threaded processes or by bound threads. The global LWP ID of a multiplexed thread changes according to the LWP on which the thread is currently scheduled. If these interfaces are used by multiplexed threads, it is possible that the wrong thread will be wakened. For information on threads programming and threads management facilities, refer to the *PowerMAX OS Programming Guide*.

The specifications required for making the **`server_block`** call are as follows:

```
int server_block(options, r, timeout)

int options;
struct resched_var *r;
struct timeval *timeout;
```

Arguments are defined as follows:

options	The value of this argument must be zero.
r	a pointer to the calling LWP's rescheduling variable. This argument is optional; its value can be **NULL**.
timeout	a pointer to a **`timeval`** structure that contains the maximum length of time that the calling LWP will be blocked. This argument is optional; its value can be **NULL**. If its value is NULL, there is no time out.

The **`server_block`** system call will return immediately if the calling LWP has a pending wake-up request; otherwise, it will return when the calling LWP receives the next wake-up request. A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Note that upon return, the calling LWP should retest the condition that caused it to block; there is no guarantee that the condition has changed because the LWP could have been prematurely wakened by a signal.

`Server_wake1` is invoked to wake a server that is blocked in the **`server_block`** call.

The specifications required for making the **`server_wake1`** call are as follows:

```
int server_wake1(server, r)

global_lwpid_t server;
struct resched_var *r;
```

Arguments are defined as follows:

server	the global LWP ID of the server LWP that is to be wakened
r	a pointer to the calling LWP's rescheduling variable. This argument is optional; its value can be **NULL**.

It is important to note that to use the **`server_wake1`** call, the real or effective user ID of the calling LWP must match the real or saved user ID of the LWP specified by *server*, or the calling LWP must have the P_OWNER privilege (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **intro(2)** system manual page).

If the Enhanced Security Utilities are installed and running, one of the following conditions must be met:

- The Mandatory Access Control (MAC) level of the calling LWP must equal the MAC level of the target LWP.
- The calling LWP must have the P_MACWRITE or the P_COMPAT privilege.
- The target LWP must have the P_COMPAT privilege.

Server_wake1 wakes the specified server if it is blocked in the **server_block** call. If the server is not blocked in this call, the wake-up request is held for the server's next call to **server_block**. **Server_wake1** also decrements the number of rescheduling locks associated with the rescheduling variable specified by *r*.

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **errno** is set to indicate the error.

The **server_wakevec** system call is invoked to wake a group of servers blocked in the **server_block** call.

The specifications required for making the **server_wakevec** call are as follows:

```
int server_wakevec(servers, nservers, r)

global_lwpid_t *servers;
int nservers;
struct resched_var *r;
```

Arguments are defined as follows:

servers	a pointer to an array of the global LWP IDs of the server LWPs that are to be wakened
nservers	an integer value specifying the number of elements in the array
r	a pointer to the calling LWP's rescheduling variable. This argument is optional; its value can be **NULL**.

It is important to note that to use the **server_wakevec** call, the real or effective user ID of the calling LWP must match the real or saved user IDs of the LWPs specified by *servers*, or the calling LWP must have the P_OWNER privilege (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **intro(2)** system manual page).

If the Enhanced Security Utilities are installed and running, one of the following conditions must be met for each of the target LWPs:

- The Mandatory Access Control (MAC) level of the calling LWP must equal the MAC level of the target LWP.
- The calling LWP must have the P_MACWRITE or the P_COMPAT privilege.
- The target LWP must have the P_COMPAT privilege.

Server_wakevec wakes the specified servers if they are blocked in the **server_block** call. If a server is not blocked in this call, the wake-up request is applied to the server's next call to **server_block**. **Server_wakevec** also decrements the number of rescheduling locks associated with a rescheduling variable specified by *r*.

A return of **0** indicates that the call has been successful. A return of **–1** indicates that an error has occurred; **errno** is set to indicate the error.

For additional information on the use of these calls, refer to the system manual page **server_block(2)**.

Applying Condition Synchronization Tools

The rescheduling variable, spin lock, and server system calls can be used to design functions that enable a producer and a consumer LWP to exchange data through use of a mailbox in a shared memory region. When the consumer finds the mailbox empty, it blocks until new data arrives. After the producer deposits new data in the mailbox, it wakes the waiting consumer. An analogous situation occurs when the producer generates data faster than the consumer can process it. When the producer finds the mailbox full, it blocks until the data is removed. After the consumer removes the data, it wakes the waiting producer.

A mailbox can be represented as follows:

```
struct mailbox {
    struct spin_mutex mx;
    unsigned int data;
    int full;
    global_lwpid_t producer;
    global_lwpid_t consumer;
};
```

The `mx` field is used to serialize access to the mailbox. The `data` field represents the information that is being passed from the producer to the consumer. The `full` field is used to indicate whether the mailbox is full or empty. The `producer` field identifies the LWP that is waiting for the mailbox to be empty. The `consumer` field identifies the LWP that is waiting for the arrival of data.

Using the **spin_acquire** and the **spin_release** functions defined in "Applying Busy-Wait Mutual Exclusion Tools," a function to enable the consumer to extract data from the mailbox can be defined as follows:

```
void
mailbox_get (mb, data)
    struct mailbox *mb;
    unsigned int *data;
{
    global_lwpid_t producer;

    spin_acquire (&mb–>mx, &rv);


       while (! mb–>full) {
        mb–>consumer = rv.rv_glwpid;
        spin_unlock (&mb–>mx);
        server_block (0, &rv, 0);
        spin_acquire (&mb->mx, &rv);
    }

    *data = mb->data;
    mb->full = 0;

    producer = mb->producer;
    mb->producer = 0;

    spin_unlock (&mb->mx);

    if (producer)
        server_wake1 (producer, &rv);
    else
        resched_unlock (&rv);
}
```

Note that in this function, the consumer LWP locks the mailbox prior to checking for and removing data. If it finds the mailbox empty, it unlocks the mailbox to permit the producer to deposit data, and it calls **`server_block`** to wait for the arrival of data. When the consumer is wakened, it must again lock the mailbox and check for data; there is no guarantee that the mailbox will contain data—the consumer may have been wakened prematurely by a signal.

A similar function that will enable the producer to place data in the mailbox can be defined as follows:

```
void
mailbox_put (mb, data)
    struct mailbox *mb;
    unsigned int data;
{
    global_lwpid_t consumer;

    spin_acquire (&mb->mx, &rv);


       while (mb->full) {
        mb->producer = rv.rv_glwpid;
        spin_unlock (&mb->mx);
        server_block (0, &rv, 0);
        spin_acquire (&mb->mx, &rv);
    }

    mb->data = data;
    mb->full = 1;

    consumer = mb->consumer;
    mb->consumer = 0;

    spin_unlock (&mb->mx);

    if (consumer)
```

```
            server_wake1 (consumer, &rv);
        else
            resched_unlock (&rv);
}
```

In this function, the producer LWP waits for the mailbox to empty before depositing new data. The producer signals the arrival of data only when the consumer is waiting; note that it does so after unlocking the mailbox. The producer must unlock the mailbox first so that the wakened consumer can lock it to check for and remove data. Unlocking the mailbox prior to the call to **`server_wake1`** also ensures that the mutex is held for a short time. To prevent unnecessary context switching, rescheduling is disabled until the consumer is wakened.

7
Timing Facilities

7 Timing Facilities

This chapter provides an overview of some of the facilities that can be used for timing. Presented first is an overview of the POSIX clocks and timers interfaces. These interfaces are based on IEEE Standard 1003.1b-1993. The POSIX clock interfaces provide a high-resolution clock, which can be used for such purposes as time stamping or measuring the length of code segments. The POSIX timer interfaces provide a means of receiving a signal or process thread wakeup asynchronously at some future time. These interfaces also allow a thread of execution to block itself until a specified high-resolution time has elapsed.

This chapter also presents an overview of the high–resolution timing facility. This facility provides a means of accurately measuring the CPU time utilized by a process or LWP.

Understanding POSIX Clocks and Timers

Clocks provide a high-resolution mechanism for measuring and indicating time. Currently two system-wide clocks are available. They are defined as **CLOCK_UNIX** and **CLOCK_REALTIME** in the file <**`time.h`**>. The **CLOCK_UNIX** clock, a Harris Computer Systems extension based on the hardclock interrupt, is the timing source for the system time-of-day clock. The **CLOCK_REALTIME** clock, based on the hardware interval timer, is the timing source for POSIX timers.

POSIX timers provide a mechanism for signaling the lapse of a time period. They are created by and associated with a particular process. A process can create at most **PTIMER_MAX** timers as defined in the file **`/etc/conf/mtune.d/svc`**.

There are two types of timers: one-shot and periodic. They are defined in terms of an initial expiration time and a repetition interval. The initial expiration time indicates when the timer will first expire. The repetition interval indicates the amount of time that will elapse between one expiration of the timer and the next. The initial expiration time may be absolute (for example, at 8:30 a.m.) or relative to the current time (for example, in 30 seconds). If a timer is armed with an absolute time, a timer expiration notification is sent to the process when the clock associated with the timer reaches the specified time. If a timer is armed with a relative time, a timer expiration notification is sent to the process when the specified period of time, as measured by the clock associated with the timer, elapses.

A one-shot timer is armed with either an absolute or a relative initial expiration time and a repetition interval of zero. It expires only once--at the initial expiration time--and then is disarmed.

A periodic timer is armed with either an absolute or a relative initial expiration time and a repetition interval that is greater than zero. The repetition interval is always relative to the time at the point of the last timer expiration. When the initial expiration time occurs, the timer is reloaded with the value of the repetition interval and continues counting. The timer may be disarmed by setting its initial expiration time to zero.

Access to the clocks and timers is provided by a set of related POSIX library routines that are located within the C and thread libraries. The structures that are used to specify time to these routines are explained in “Understanding the Time Structures.” The clock routines are then presented in “Using the Clock Routines” and the timer routines in “Using the Timer Routines.”

Understanding the Time Structures

The POSIX library routines related to clocks and timers use two structures for time specifications: the **`timespec`** structure and the **`itimerspec`** structure. These structures are defined in the file **`<time.h>`**. The **`timespec`** structure specifies a single time value in seconds and nanoseconds. The **`itimerspec`** structure specifies the initial expiration time and the repetition interval for a timer.

The **`timespec`** structure is defined as follows:

```
struct timespec {
       time_t  tv_sec;
       long    tv_nsec;
};
```

The fields in the structure are described as follows.

`tv_sec`	specifies the number of seconds in the time value
`tv_nsec`	specifies the number of additional nanoseconds in the time value. The value of this field must be greater than or equal to zero and less than 1,000,000,000.

You supply a pointer to a **`timespec`** structure when you invoke the routines that allow you to set the time of a clock or obtain the time or resolution of a clock (for information on these routines, see “Using the Clock Routines”).

The **`itimerspec`** structure is defined as follows:

```
struct itimerspec {
       struct timespec it_interval;
       struct timespec it_value;
};
```

The fields in the structure are described as follows.

`it_interval`	specifies the amount of time that makes up the repetition interval of a timer
`it_value`	specifies the time of or the amount of time until a timer’s expiration

You supply a pointer to an **`itimerspec`** structure when you invoke the routines that allow you to set the time at which a timer expires or obtain information about a timer’s expiration time (for information on these routines, see “Using the Timer Routines”).

Using the Clock Routines

The POSIX library routines that allow you to perform a variety of functions related to clocks are briefly described as follows:

`clock_settime`	set the time of a specified clock
`clock_gettime`	obtain the time from a specified clock
`clock_getres`	obtain the resolution in nanoseconds of a specified clock

Procedures for using each of these routines are explained in the sections that follow.

CLOCK_UNIX is the system time-of-day clock. Its value serves as the time for file system creation and modification times, accounting and auditing record times, and IPC message queue and semaphore times. The following commands, library routines, and system calls read and set **CLOCK_UNIX**: **`date(1)`**, **`gettimeofday(3C)`**, **`settimeofday(3C)`**, **`stime(2)`**, **`time(2)`**, and **`adjtime(2)`**. These commands, library routines, and system calls do not affect **CLOCK_REALTIME** or the interval timer.

CLOCK_REALTIME is the timing source for POSIX timers. Its value is based on the hardware interval timer.

Using the clock_settime Routine

The **`clock_settime(3C)`** library routine allows you to set the time of a specified clock. Note that to use this routine, the calling process must have the **P_SYSOPS** privilege (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **`intro(2)`** system manual page).

For instructions about how to safely set a clock, see "Setting the Clock During System Initialization" and "Setting the Clock at Init State 2."

CAUTION

The **`clock_settime`** routine should not be used to set the time of **CLOCK_REALTIME**. The only time that you can use this routine without adversely affecting facilities that rely on the interval timer is at system initialization or during a change to **`init`** state 2. The reasons are explained in the following paragraphs.

If you set **CLOCK_REALTIME** after system start-up, the following facilities do not operate properly: the **`ktrace`** utility, the high-resolution timing facility, the performance monitor, and the high-resolution callout queue. The following paragraphs describe the problems that occur with each facility.

The **`ktrace(1)`** utility allows you to determine the amount of time that is devoted to processing within the operating system. It uses the interval timer to include in each trace record a time stamp that corresponds to the time at which a kernel event occurred. It then calculates relative times by computing the difference between time stamps in two records. Changing the value of the interval timer causes errors in the times that the **`ktrace`** utility reports. (For additional information on the **ktrace** utility, refer to the corresponding system manual page and Chapter 4 of this guide.) Note that in future releases, there will be other facilities that utilize the interval timer; these facilities will be adversely affected by changing **CLOCK_REALTIME**.

The high-resolution timing facility uses the system's interval timer as the timing source to keep track of each process's and LWP's elapsed system and elapsed user time. It uses the interval timer to associate time stamps with the times of entry into and exit from the kernel, the times of entry into and exit from interrupt service routines, and the time between context switches. It allows accurate measurement of CPU utilization. Changing the value of the interval timer causes errors in the times that the high-resolution timing facility records. (For additional information on the high-resolution timing facility, refer to "Using the High-Resolution Timing Facility.")

The performance monitor facility is a feature of PowerMAX OS that allows you to monitor use of the CPU by processes or LWPs that are scheduled on a frequency-based scheduler (the performance monitor and the frequency-based scheduler are documented in the *PowerMAX OS Guide to Real-Time Services*). On PowerMAX OS systems, the performance monitor relies on the high-resolution timing facility to obtain its timing values. Changing the value of the interval timer affects the performance monitor in the same way that it affects the high-resolution timing facility.

The high-resolution callout queue contains entries that specify routines that are to be called at some time in the future and the amount of time that is to elapse before they are called. The expiration time for an entry in the high-resolution callout queue is stored as an interval timer value. Changing the value of the interval timer affects the expiration times for entries in this queue in the following manner: (1) queue entries that are set to expire at an absolute time will still expire at that time according to the new time which has been set via **clock_settime(3C)** and (2) queue entries that are set to expire at a relative time will be adjusted so that they expire at the same relative time. Examples of routines that cause entries to be placed in the high-resolution callout queue include the POSIX **timer_settime(3C)** and **nanosleep(3C)** routines, which are presented in "Using the timer_settime Routine" and "Using the nanosleep Routine," respectively. (For additional information on the high-resolution callout queue, refer to Chapter 3 of this guide.)

If you set the system time-of-day clock, **CLOCK_UNIX**, after system start-up, the following times may not be accurate: file system creation and modification times, times in accounting and auditing records, the expiration times for callout queue entries, and the Time of Century setting. The following paragraphs describe the problems that can occur.

When you modify the system time, file system creation and modification times and times in accounting and auditing records that were recorded before the time change are no longer accurate relative to the new time-of-day system time.

Unlike high-resolution callout queue entries with expiration times based on the interval timer, callout queue entries (and **cron(1)** jobs) have expiration times based on the system time. Changing the system time value affects the expiration times for entries in this queue. Examples of routines that cause entries to be placed in the callout queue include the **setitimer(3C)** and **getitimer(3C)** routines. (For additional information on the callout queue, refer to Chapter 3 of this guide.)

During a system reboot, the system initializes **CLOCK_REALTIME** and **CLOCK_UNIX** to roughly the same time, the value of the Time of Century clock at a resolution of one second. Setting **CLOCK_UNIX** sets the Time of Century clock. Incorrectly setting **CLOCK_UNIX** invalidates the Time of Century clock value for the next system reboot.

The specifications required for making the **`clock_settime`** call are as follows:

```
#include <time.h>

int clock_settime(clock_id, tp)

clockid_t          clock_id;
struct timespec   *tp;
```

The arguments are defined as follows:

clock_id the identifier for the clock for which you wish to set the time. The value of *clock_id* must be **CLOCK_REALTIME** or **CLOCK_UNIX**.

tp a pointer to a structure that specifies the time to which the clock identified by *clock_id* is to be set. When *clock_id* is **CLOCK_REALTIME**, the interval timer is set to a new value. When *clock_id* is **CLOCK_UNIX**, the time-of-day clock is set to a new value. Note that you set **CLOCK_REALTIME** and **CLOCK_UNIX** independently.

The values in the structure represent the amount of time in seconds and nanoseconds that have passed since 00:00:00 GMT (Greenwich mean time), January 1, 1970. In reality, the system is not synchronized to any standard time reference; however, the application can treat this time as if it represents the number of seconds between the referenced time and 00:00:00 GMT, January 1, 1970. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated to the smaller multiple of the resolution. See the **`clock_getres(3C)`** system manual page and "Using the clock_getres Routine."

A return value of **`0`** indicates that the specified clock has been successfully set. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`clock_settime(3C)`** system manual page for a listing of the types of errors that may occur.

Setting the Clock During System Initialization

If you always wish to set **CLOCK_UNIX** or **CLOCK_REALTIME** as you boot the system into single user mode, perform the following steps:

1. Write a program that sets the clock or clocks. This program should be statically linked and located either in the **`/`** or **`/var`** file systems because the **`/usr`** file system will not be mounted when your program will be executed.

2. Edit the **`/etc/inittab`** file.

3. Search for the following line (where the 2 may be a 3):

   ```
   is:2:initdefault:
   ```

4. If your program is named **`/sbin/setclk`**, then insert the following line above the line you just found.

   ```
   sk::sysinit:/sbin/setclk >/dev/sysmsg
   ```

(For more information about adding entries in the **inittab** file, see the **inittab(4)** system manual page.)

5. Repeat steps 2 through 4 for the **/etc/conf/init.d/kernel** file.

During system initialization, **init** will dispatch your program for execution.

Setting the Clock at Init State 2

If you always wish to set CLOCK_UNIX or CLOCK_REALTIME as you enter **init** state 2, perform the following steps:

1. Write a program that sets the clock or clocks.
2. Change directory to **/etc/rc2.d**.
3. Create a script file with a name that adheres to the following convention for files in this directory.

   S*##name*

   where:

S	indicates that the script will be executed whenever the system enters **init** state 2 from **init** state 0.
##	is a two-digit number that indicates script execution order, where low numbers are executed first. If you wish to execute your set clock program after all other **init** 2 scripts have run, then use a number that is higher than those of existing scripts. For example, the **S75rpc** script will be executed before the **S85setclock** script.
name	is a user-supplied file name suffix.

4. Edit this script so that it will execute the program or programs that you wish to have run. The following example shows the possible contents of this script for a set clock program named **/sbin/setclk**.

```
#
# Run program to set the clock.
#
echo /sbin/setclk
/sbin/setclk
```

Assume that your set clock program runs at **init** state 2, and the script that calls it has the highest number in the directory. Networking and non-NFS file systems will both be available to your program. This is not true of programs started at system initialization.

Using the clock_gettime Routine

The **clock_gettime(3C)** library routine allows you to obtain the time from a specified clock.

The specifications required for making the **`clock_gettime`** call are as follows:

```
#include <time.h>

int clock_gettime(clock_id, tp)

clockid_t          clock_id;
struct timespec *tp;
```

The arguments are defined as follows:

clock_id the identifier for the clock from which you wish to obtain the time. The value of *clock_id* must be **CLOCK_REALTIME** or **CLOCK_UNIX**.

tp a pointer to a structure to which the time of the clock identified by *clock_id* is returned. The values returned represent the amount of time in seconds and nanoseconds that has passed since 00:00:00 GMT (Greenwich mean time), January 1, 1970. In reality, the system is not synchronized to any standard time reference; however, the application can treat this time as if it represents the number of seconds between the referenced time and 00:00:00 GMT, January 1, 1970. Note that you may change the value of this clock by using the **`clock_settime(3C)`** library routine (see "Using the clock_settime Routine" for an explanation of this routine).

A return value of **`0`** indicates that the call to **`clock_gettime`** has been successful. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`clock_gettime(3C)`** system manual page for a listing of the types of errors that may occur.

Using the clock_getres Routine

The **`clock_getres(3C)`** library routine allows you to obtain the resolution in nanoseconds of a specified clock.

Processes cannot set clock resolutions. The resolution for **CLOCK_REALTIME** is one microsecond (1,000 nanoseconds), and the resolution for **CLOCK_UNIX** on PowerMAX OS systems is 60 Hz (16,666,667 nanoseconds).

The specifications required for making the **`clock_getres`** call are as follows:

```
#include <time.h>

int clock_getres(clock_id, res)

clockid_t          clock_id;
struct timespec *res;
```

The arguments are defined as follows:

clock_id the identifier for the clock for which you wish to obtain the resolution. The value of *clock_id* must be **CLOCK_REALTIME** or **CLOCK_UNIX**.

res	a pointer to a structure to which the resolution of the clock identified by *clock_id* is returned

A return value of **`0`** indicates that the call to **`clock_getres`** has been successful. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`clock_getres(3C)`** system manual page for a listing of the types of errors that may occur.

Using the Timer Routines

The POSIX library routines that allow you to perform a variety of functions related to POSIX timers are briefly described as follows:

`timer_create`	create a timer using a specified clock
`timer_delete`	remove a specified timer
`timer_settime`	arm or disarm a specified timer by setting the expiration time
`timer_gettime`	obtain the repetition interval for a specified timer and the time remaining until the timer expires
`timer_getoverrun`	obtain the overrun count for a specified periodic timer
`nanosleep`	allow threads to block until a specified period of time elapses

Procedures for using each of these routines are explained in the sections that follow.

NOTE

Processes can create, remove, set, and query timers and may receive notification when a timer expires. When a process has multiple threads, each of these threads can access each timer the process creates. Signals delivered to a process are asynchronous in nature. This signal delivery follows the normal threads library method of selecting a thread within a process for receiving a signal (see “Programming with the Threads Library” in the *PowerMAX OS Programming Guide)*.

Using the timer_create Routine

The **`timer_create(3C)`** library routine allows the calling process to create a timer using a specified clock as the timing source. By default, the system is configured to allow 32 timers per process. A POSIX-compliant program can use a maximum of 32 timers only. You can use the **`config(1M)`** utility to change the **PTIMER_MAX** tunable parameter to allows up to 2048 timers per process. Note that after changing a tunable parameter, you must rebuild the kernel and then reboot your system. For an explanation of the procedures for using **config**, refer to the “Configuring and Building the Kernel” chapter of *System Administration Volume 2*.

A timer is disarmed when it is created. It is armed when the process invokes the **timer_settime(3C)** library routine (see “Using the timer_settime Routine” for an explanation of this routine).

It is important to note the following:

- When a process invokes the **fork** system call, the timers that it has created are not inherited by the child process.
- When a process invokes the **exec** system call, the timers that it has created are disarmed and deleted.
- When a process has multiple threads, each of these threads can access each timer the process creates.

The specifications required for making the **timer_create** call are as follows:

```
#include <time.h>
#include <signal.h>

int timer_create(clock_id, evp, timerid)

clockid_t        clock_id;
struct sigevent *evp;
timer_t         *timerid;
```

The arguments are defined as follows:

clock_id	the identifier for the clock that is to serve as the timing source for the timer. The value of *clock_id* must be **CLOCK_REALTIME**, which has a resolution of one microsecond.
evp	the null pointer constant or a pointer to a structure that specifies the way in which the calling process is to be asynchronously notified of the expiration of the timer. Table 7-1 summarizes these notification methods.

NOTE

The signal denoting expiration of the timer may cause the process to terminate unless it has specified a signal-handling routine. To determine the default action for a particular signal, refer to the **`signal(5)`** system manual page.

Table 7-1. Notification Mechanisms for Timer Expirations

Assignment	Effect and Required Actions
evp = **NULL**	The default signal, SIGALRM, is to be sent to the process when the timer expires. If the process is catching the SIGALRM signal and has set the **SA_SIGINFO** flag on a call to **`sigaction(2)`** to declare an action for the signal, then the expired timer's identifier is the value and **SI_TIMER** is the code that is delivered with the signal.
evp->sigev_notify = **SIGEV_SIGNAL**	A specified signal and application-defined value are to be sent to the process when the timer expires. Your application must set *evp->sigev_signo* to the signal number that is to be sent to the process upon expiration of the timer. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file **`<signal.h>`**. Your application must also set *evp->sigev_value* to an application-defined value that is to be used by a signal-handling routine. This value may be a pointer or an integer value. Assume that the process catching the signal invoked the **`sigaction(2)`** system call with the **SA_SIGINFO** flag set prior to the time that the timer is created. The signal, the application-defined value, and the **SI_TIMER** code are queued to the process when the timer expires.

continued on next page

Table 7-1. Notification Mechanisms for Timer Expirations (Cont.)

Assignment	Effect and Required Actions
evp->sigev_notify = **SIGEV_CALLBACK**	At the time of the **`timer_create`** call, a bound daemon thread is to be created for handling timer expirations. When the timer expires, this thread is to execute the process's timer-expiration routine. Your application must set *evp->sigev_func* to the address of a user-defined timer-expiration routine. Your application must also set *evp->sigev_value* to an application-defined value that is to be used by a signal-handling routine. This value may be a pointer or an integer value. The following interface describes how the timer-expiration routine will be called: `void` *user_expiration_routine* (void **evp->sigev_value*) If your timer-expiration routine calls other routines, the timer expiration thread must exit these routines via an explicit or implicit **`return`**. Expiration routines for periodic timers should not call **`thr_exit(3thread)`** because this will cause the timer to be deleted within the kernel. Additionally, applications should not call functions that would block or put the thread to sleep for long periods of time because timer overruns would be likely to occur in these situations (see **`timer_getoverrun(3C)`**). To help ensure that the timer-expiration thread functions properly, application code should not change the signal mask of the timer-expiration thread. This Harris Computer Systems extension, which is <u>not</u> POSIX-compliant, provides a much quicker and more deterministic notification method for timer expirations than the signal delivery of **SIGEV_SIGNAL** method. The **SIGEV_CALLBACK** method wakes up an already blocked, bound daemon thread in the kernel. This thread immediately returns to user space to execute your timer-expiration routine. If the **SIGEV_CALLBACK** notification method is used, then the application <u>must</u> link in the **`thread`** library. Failure to adhere to this requirement will result in a run-time error return of **`-1`** from the **`timer_create(3C)`** call with **`errno`** set to EINVAL. You may link this library either statically or dynamically. (For information about static and dynamic linking, see the "Link Editor and Linking" chapter in *Compilation Systems Volume 1 (Tools)*.) The following example shows the typical command-line format: **`cc`** [*options*] -**`D_REENTRANT`** *file* -**`lthread`**
evp->sigev_notify = **SIGEV_NONE**	No notification is to be delivered when the timer expires.

timerid a pointer to the location to which **`timer_create`** returns the identifier for the timer that is created. This identifier is required by the other POSIX timer routines and is unique within the calling process until the timer is deleted by the **`timer_delete(3C)`** routine.

CAUTION

If the real-time clock driver is not configured in the kernel that is currently executing or if the **`hrtconfig(1M)`** program has not been executed in order to set up a timer interrupt source for the high-resolution callout queue, then this routine fails and sets **`errno`** to **ENODEV**. (For additional information on the high-resolution callout queue, refer to Chapter 3 of this guide.)

A return value of **`0`** indicates that the call to **`timer_create`** has been successful. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`timer_create(3C)`** system manual page for a listing of the types of errors that may occur.

Using the timer_delete Routine

The **`timer_delete(3C)`** library routine allows the calling process to remove a specified timer. If the selected timer is armed when the process invokes this routine, the timer is first disarmed and then removed.

If the timer-expiration notification method is **SIGEV_SIGNAL**, then a signal that is already pending due to a previous timer expiration for this timer will still be delivered to this process.

If the timer-expiration notification method is **SIGEV_CALLBACK** and **`timer_delete`** successfully returns, then the timer-expiration routine will not be run again for the deleted timer. Additionally, if a timer-expiration thread is currently executing the timer-expiration routine at the time that the **`timer_delete`** call is made, the thread will be allowed to complete execution of the routine before the caller returns from **`timer_delete`**.

The specifications required for making the **`timer_delete`** call are as follows:

```
#include <time.h>

int timer_delete(timerid)

timer_t timerid;
```

The argument is defined as follows:

timerid the identifier for the timer that is to be removed. This identifier comes from a previous call to **`timer_create(3C)`** (see "Using the timer_create Routine" for an explanation of this routine).

A return value of **`0`** indicates that the specified timer has been successfully removed. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`timer_delete(3C)`** system manual page for a listing of the types of errors that may occur.

Using the timer_settime Routine

The **`timer_settime(3C)`** library routine allows the calling process to arm a specified disarmed timer by setting the time at which it will expire. A calling process can use this routine on an armed timer to (1) disarm the timer or (2) reset the time until the next expiration of the timer to the value specified on the call.

CAUTION

If the real-time clock driver is not configured in the kernel that is currently executing or if the **`hrtconfig(1M)`** program has not been executed in order to set up a timer interrupt source for the high-resolution callout queue, then this routine fails and sets **`errno`** to **ENODEV**. (For additional information on the high-resolution callout queue, refer to Chapter 3 of this guide.)

The specifications required for making the **`timer_settime`** call are as follows:

```
#include <time.h>

int timer_settime(timerid, flags, value, ovalue)

timer_t              timerid;
int                  flags;
struct itimerspec   *value;
struct itimerspec   *ovalue;
```

The arguments are defined as follows:

timerid the identifier for the timer that is to be set. This identifier comes from a previous call to **`timer_create(3C)`** (see "Using the timer_create Routine" for an explanation of this routine).

flags an integer value that specifies one of the following:

TIMER_ABSTIME	causes the selected timer to be armed with an absolute expiration time. The timer will expire when the clock associated with the timer reaches the *value->it_value* actual time. If this time has already passed, **`timer_settime`** succeeds, and the timer-expiration notification is made.
0	causes the selected timer to be armed with a relative expiration time. The timer will expire when the clock associated with the timer advances *value->it_value* seconds and nanoseconds from the time when the **`timer_settime`** call was made.

value a pointer to a structure that contains the repetition interval and the initial expiration time of the timer.

If you wish to have a one-shot timer, specify a repetition interval, *value->it_interval*, of zero. In this case, the timer expires once, when the initial expiration time occurs, and then is disarmed.

If you wish to have a periodic timer, specify a repetition interval, *value->it_interval*, that is not equal to zero. In this case, when the initial expiration time occurs, the timer is reloaded with the value of the repetition interval and continues to count.

In either case, you may set the initial expiration time, *value->it_value*, to a value that is equal to a certain time (for example, at 3:00 p.m.) or relative to the current time (for example, in 30 seconds). To set the initial expiration time to a certain time, you must have set the **TIMER_ABSTIME** bit in the *flags* argument. When the timer-expiration notification method is **SIGEV_CALLBACK**, all executions of the timer expiration routine that occur after the **`timer_settime`** call will be due only to expirations based on the new time setting. When the timer-expiration notification method is not **SIGEV_CALLBACK**, then any signal that is already pending due to a previous timer expiration for the specified timer will still be delivered to the process.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer are rounded up to the larger multiple of the resolution.

To disarm the timer, set the initial expiration time, *value->it_value*, to zero. When the timer-expiration notification method is **SIGEV_CALLBACK**, the timer-expiration routine will not be run again until a subsequent **`timer_settime`** call is made to set the timer. When the timer-expiration notification method is not **SIGEV_CALLBACK**, then any signal that is already pending due to a previous timer expiration for this timer will still be delivered to the process.

ovalue the null pointer constant or a pointer to a structure to which the previous repetition interval and initial expiration time of the timer are returned. If the timer has been disarmed, the value of the initial expiration time, *ovalue->it_value*, is zero. The members of *ovalue* are subject to the resolution of the timer and are the same values that would be returned by a **`timer_gettime(3C)`** call at that point in time.

A return value of **`0`** indicates that the specified timer has been successfully set. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`timer_settime(3C)`** system manual page for a listing of the types of errors that may occur.

Using the timer_gettime Routine

The **`timer_gettime(3C)`** library routine allows the calling process to obtain the repetition interval for a specified timer and the amount of time remaining until the timer expires.

The specifications required for making the **`timer_gettime`** call are as follows:

```
#include <time.h>

int timer_gettime(timerid, value)

timer_t              timerid;
struct itimerspec *value;
```

The arguments are defined as follows:

timerid	the identifier for the timer for which you wish to obtain the repetition interval and the amount of time remaining until the timer expires. This identifier comes from a previous call to **`timer_create(3C)`** (see "Using the timer_create Routine" for an explanation of this routine).
value	a pointer to a structure to which the repetition interval and the amount of time remaining on the timer are returned. The amount of time remaining is relative to the current time. If the timer is disarmed, the value is zero.

A return value of **`0`** indicates that the call to **`timer_gettime`** has been successful. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`timer_gettime(3C)`** system manual page for a listing of the types of errors that may occur.

Using the timer_getoverrun Routine

The **`timer_getoverrun(3C)`** library routine allows the calling process to obtain the overrun count for a particular periodic timer. The overrun count indicates the number of times that the timer has expired between:

- The generation (queueing) of the timer expiration signal and delivery of the signal. For each timer expiration signal that is generated, the overrun count is zero if there is no delay between generation and delivery of the signal.
- The beginning and end of a timer-expiration thread's execution of the timer-expiration routine.

Assume that a signal is already queued or pending for a process with a timer using timer-expiration notification **SIGEV_SIGNAL**. If this timer expires while the signal is queued or pending, a timer overrun occurs, and no additional signal is sent.

Assume that a timer-expiration routine is running for a timer using timer-expiration notification **SIGEV_CALLBACK**. If this timer expires while the timer-expiration routine is running, a timer overrun occurs, and no additional instance of the timer-expiration routine is run.

NOTE

You must invoke this routine from the timer-expiration signal-handling or timer-expiration thread execution routine. If you invoke it outside this routine, the overrun count that is returned is not valid for the timer-expiration signal last taken.

The specifications required for making the **`timer_getoverrun`** call are as follows:

```
#include <time.h>

int timer_getoverrun(timerid)

timer_t  timerid;
```

The argument is defined as follows:

timerid the identifier for the periodic timer for which you wish to obtain the overrun count. This identifier comes from a previous call to **`timer_create(3C)`** (see "Using the timer_create Routine" for an explanation of this routine).

If the call is successful, **`timer_getoverrun`** returns the overrun count for the specified timer. This count cannot exceed **DELAYTIMER_MAX** in the file **`<limits.h>`**. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`timer_getoverrun(3C)`** system manual page for a listing of the types of errors that may occur.

Using the nanosleep Routine

The **`nanosleep(3C)`** routine provides a high-resolution sleep mechanism that causes execution of the calling process or thread to be suspended until (1) a specified period of time elapses or (2) a signal is received and the associated action is to execute a signal-handling routine or terminate the process. The use of **`nanosleep`** has no effect on the action or blockage of any signal.

The sleep time resolution for calls made to **`nanosleep`** by multiplexed threads is less deterministic than calls made by bound threads. Therefore, use bound threads if your multithreaded application requires a high-resolution sleep time on **`nanosleep`** calls.

The specifications required for making the **`nanosleep`** call are as follows:

```
#include <time.h>

int nanosleep(rqtp, rmtp)

struct timespec *rqtp;
struct timespec *rmtp;
```

Arguments are defined as follows:

rqtp a pointer to a **`timespec`** structure that contains the length of time that the process is to sleep. The suspension time may be longer than requested because the *rqtp* value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. Except for the case of being interrupted by a signal, the suspension time will not be less than the time specified by *rqtp*, as measured by **CLOCK_REALTIME**. You will obtain a resolution of one microsecond on the blocking request.

rmtp	the null pointer constant or a pointer to a **`timespec`** structure to which the amount of time remaining in the sleep interval is returned if **`nanosleep`** is interrupted by a signal. If *rmtp* is **NULL** and **`nanosleep`** is interrupted by a signal, the time remaining is not returned.

CAUTION

If the real-time clock driver is not configured in the kernel that is currently executing or if the **`hrtconfig(1M)`** program has not been executed in order to set up a timer interrupt source for the high-resolution callout queue, then this routine fails and sets **`errno`** to **ENODEV**. (For additional information on the high-resolution callout queue, refer to Chapter 3 of this guide.)

A return value of **`0`** indicates that the requested period of time has elapsed. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`nanosleep(3C)`** system manual page for a listing of the types of errors that may occur.

Using the High-Resolution Timing Facility

This section provides an overview of the high–resolution timing facility and explains the procedures for using the related library routine, **`hirestmode(3C)`**.

Overview of the High-Resolution Timing Facility

The high–resolution timing facility is a feature of PowerMAX OS systems. It provides a means of measuring each process's or LWP's execution time. The high–resolution timing facility is available only if it is configured into the currently executing kernel. By default, it is not configured into the kernel. If the high-resolution timing facility is not configured, execution time is estimated by sampling the program counter 60 times a second. The high-resolution timing facility uses the interval timer to accurately measure the processor time that a process or LWP uses while executing in both system and user mode. It is important to note that the high-resolution timing facility adds a small amount of overhead to interrupt entry and exit, exception entry and exit, and context switches.

To configure the high–resolution timing facility, you must change the value of the `HIGHRESTIMING` system tunable parameter from 0 to 1. You can use the **`config`** utility to (1) determine whether the high-resolution timing facility is configured into you kernel, (2) change the value of the HIGHRESTIMING tunable parameter, and (3) rebuild the kernel. Note that you must be the root user to change the value of a tunable parameter and rebuild the kernel. After rebuilding the kernel, you must then reboot your system. For an explanation of the procedures for using **`config(1M)`**, refer to the “Configuring and Building the Kernel” chapter of *System Administration Volume 2*.

The high–resolution timing facility has two modes of operation: one that includes interrupt time in timing values and one that excludes interrupt time from timing values. The two modes are defined as follows:

Include interrupt time	A process's or LWP's user and system times total the elapsed time that accrues when the process or LWP is currently running. This elapsed time includes time spent servicing interrupts and performing context switches. Time spent servicing interrupts is added to the process's or LWP's system time. Time spent switching to a new process or LWP is included in the new process's or LWP's system time.
Exclude interrupt time	A process's or LWP's user and system times total the time that accrues when the process or LWP is currently running. This time excludes time spent servicing interrupts, but it includes time spent performing context switches. Time spent switching to a new process or LWP is included in the new process's or LWP's system time.

The default mode is to exclude interrupt time.

The high–resolution timing facility is supported by a library routine, **`hirestmode(3C)`**, that enables you to set the timing mode for the system. Procedures for using this routine are explained in "Using the hirestmode Library Routine."

When the high–resolution timing facility is configured, it affects the times reported by **`time(1)`**, **`times(2)`**, **`_lwp_info(2)`**, **`timex(1)`**, the performance monitor, and **`/proc`**. On a read of the /**`proc`**/*pid*/**`status`** file, the `pr_utime`, `pr_stime`, `pr_cutime`, and `pr_cstime` fields in the **`pstatus_t`** structure are affected. On a read of the /**`proc`**/*pid*/**`lwp`**/*lwpid*/**`lwpsinfo`** file, the `pr_time` field in the **`lpwsinfo_t`** structure is affected. (For additional information on these fields, refer to the **`proc(4)`** system manual page.)

In each case, the timing values that are returned and represent a process's and LWP's system time will be affected by the mode in which the high–resolution timing facility is operating; that is, they will either include or exclude time spent servicing interrupts. (To determine the type of information that is returned in each case, refer to the corresponding system manual pages.)

Using the hirestmode Library Routine

The **`hirestmode`** library routine enables you to set the timing mode or obtain the current timing mode for the high–resolution timing facility on a PowerMAX OS system. It is important to note that to use this call to set the timing mode, the calling process must have the P_RTIME privilege (for additional information on privileges, refer to the **`intro(2)`** System Manual and the *PowerMAX OS Programming Guide*.

CAUTION

The timing mode for the high–resolution timing facility is set system–wide. It affects all processes and LWPs running on all CPUs.

The specifications required for making the **`hirestmode`** call are as follows:

```
int hirestmode(mode)

int mode;
```

The value of *mode* indicates the type of operation that you wish to perform and must be one of the following:

<0	Get the current system timing mode.
0	Set the system timing mode to exclude time spent servicing interrupts.
>0	Set the system timing mode to include time spent servicing interrupts.

If the value of *mode* is less than zero, the **`hirestmode`** call returns the current system timing mode. If the value of *mode* is greater than or equal to zero, **`hirestmode`** returns the previous timing mode. A return value of 0 indicates that time spent servicing interrupts is excluded. A return value of 1 indicates that time spent servicing interrupts is included. A return value of –1 indicates that an error has occurred; **`errno`** is set to indicate the error. For a listing of the types of errors that may occur, refer to the system manual page **`hirestmode(3C)`**.

8
User-Level Interrupt Routines

8 User-Level Interrupt Routines

This chapter provides an overview of user-level interrupt routines and the ways in which they may be used. It describes the system calls, library routines, and utilities that have been developed to support the use of user-level interrupt routines, and it explains the procedures for using them.

Overview of User-Level Interrupt Routines

The PowerMAX OS operating system provides the support necessary to allow a process to define a connection to an interrupt vector generated by a selected device and to enable that connection. When a process defines an interrupt vector connection, it specifies the interrupt vector number to which it is connecting, the address of a user interrupt-handling routine that will be executed upon each occurrence of the connected interrupt, and a parameter tht is passed to that routine. When a process enables the connection to an interrupt vector, it blocks in the kernel; it no longer executes at normal program level. It executes only at interrupt level—executing the specified interrupt-handling routine when the connected interrupt becomes active.

Throughout this chapter, the term *user-level interrupt process* denotes the process or process thread that defines and enables an interrupt vector connection; the term *interrupt-handling routine* denotes the routine that is executed each time the connected interrupt occurs or becomes active. Constraints imposed on the user-level interrupt process are described in this section. Constraints imposed on the interrupt-handling routine are described in "Interrupt-Handling Routine Constraints."

For a single-threaded process, a user-level interrupt process may define a connection to only <u>one</u> interrupt vector at a time. For a multithreaded process, multiple interrupt connections may be made by using a separate bound thread for each connection. Only <u>one</u> user-level interrupt process may define a connection to a particular interrupt vector at a time.

The device driver for the real-time clock (**rtc**) provides support for connecting a process to an interrupt vector generated by the real-time clock. Prior to connecting a process to an interrupt vector generated by this device, you must ensure that the device is configured in your system. For information on the use of this device, refer to Chapter 12 of this guide and the **rtc(7)** system manual page.

On PowerMAXION systems, an application program can use the fifth real-time clock on each processor board as a watch-dog timer. When programmed as a watch-dog timer, this real-time clock's time-out generates a nonmaskable exception to the PPC604 processor on that board. The application can connect a user-level interrupt routine to this exception. When the timer expires, the operating system passes program control to the user-level interrupt routine at interrupt level.

The application sets up the watch-dog timer by performing the following steps:

1. The fifth real-time clock's interrupt must be disabled on the processor's interrupt controller. The application does this by mapping the interrupt controller's enable register using the shared memory mechanism. Refer to the “Interprocess Communication” chapter of the *PowerUX Programming Guide* for an explanation of the procedures for using shared memory. The physical addresses for the interrupt enable registers on the PowerMAXION are as follows:

   0x96200020 local processor
   0x9D000020 processor board 0
   0x9D100020 processor board 1
   0x9D200020 processor board 2
   0x9D300020 processor board 3

   The fifth real-time clock interrupt is disabled by resetting bit 17 in the enable register. This is a 32-bit register. The application must not change other bits in the interrupt controller's enable register. This can be achieved by reading the enable register, masking out bit 17 only, and then rewriting the contents to the enable register.

   The application is responsible for re-enabling this interrupt after use of the watch-dog timer is complete. This is achieved by setting bit 17 in the enable register. Failure to do so will prevent the fifth real-time clock on that processor board from being used as a timer.

2. The interrupt signal from the fifth real-time clock must be routed directly to a nonmaskable exception input on the PPC604 processor. The application does this by mapping the processor’s control and status register (PCSR) using the shared memory mechanism. This is a 16-bit register. The physical addresses for the PCSRs are as follows:

   0xB2000000 processor 0
   0xB2000008 processor 1
   0xB6000000 processor 2
   0xB6000008 processor 3

   Routing of the fifth real-time clock interrupt is achieved by setting bit 11 in the PCSR for the respective processor board. The application must not change other bits in the PCSR. This can be achieved by reading the register, setting bit 11, and rewriting the contents to the register.

   The application is responsible for restoring bit 11 of the PCSR to 0 after use of the watch-dog timer is complete. Failure to do so will prevent the fifth real-time clock on that processor board from being used as a timer.

3. The application must connect and enable the user-level interrupt routine. This is achieved by using the interfaces that are described in “Operating System Support” (page 8-3). The application must also lock all memory resources used by the user-level interrupt routine. These resources include shared memory segments, library text and data, and process text and data.

4. The fifth real-time clock must be programmed by the application with the correct count and frequency.

Prior to enabling an interrupt connection, a user-level interrupt process must lock into memory portions of its virtual address space that will be referenced by the interrupt-handling routine. Exceptions that occur during execution of the interrupt-handling routine are fatal (see “Interrupt-Handling Routine Constraints” for more detailed information).

It is recommended that a user-level interrupt process avoid using system calls and library routines that enable you to obtain an execution profile (for example, **`profil(2)`** and **`monitor(3C)`**) because the resulting information will be incorrect.

You may use local memory with a user-level interrupt process, but you should take into consideration the CPU that will be receiving the connected interrupt prior to defining and enabling the interrupt vector connection. Guidelines for using local memory with user-level interrupt processes are presented in “Using Local Memory.”

User-level interrupt routines provide a user-level process with the capability to react quickly and deterministically to an external event that arrives as an interrupt. Some of the ways in which you may use user-level interrupt routines are as follows:

- To write a user-level device driver
- To connect to an external interrupt for purposes of process scheduling or event notification

Configuration requirements related to the use of user-level interrupt routines are presented in “Configuration Requirements.” Operating system support for user-level interrupt routines is described in “Operating System Support.”

Configuration Requirements

The user-level interrupt module (**`ui`**) is optional. By default, the **`ui`** module is not configured. This means that the **`ui`** kernel driver containing the user-level interrupt kernel support code is not configured into the kernel. You can use the **`config(1M)`** utility to (1) determine whether or not the **`ui`** module is enabled in your kernel, (2) enable the **`ui`** module, and (3) rebuild the kernel. Note that you must be the root user to enable a module and rebuild the kernel. After rebuilding the kernel, you must then reboot your system. For an explanation of the procedures for using **`config(1M)`**, refer to the “Configuring and Building the Kernel” chapter of *System Administration Volume 2*.

You can also remove user-level interrupt support from the kernel by using **`config`**. In this case also, you must then rebuild the kernel and reboot your system.

Operating System Support

Operating system support for user-level interrupt routines consists of system calls, a utility, and C and threads library routines. The **`iconnect(3C)`** and the **`ienable(3C)`** library routines enable you to define a connection between a user-level interrupt process and an interrupt vector and to enable that connection. Use of these system calls is explained in “Connecting to an Interrupt Vector.” The operating system allows you to reserve an entry in the system interrupt vector table for a VME board whose interrupt vector is fixed. Procedures for doing so are also explained in “Connecting to an Interrupt Vector.”

The **uistat(1)** utility allows you to (1) display user-level interrupt vector connections that have been defined on your system, (2) remove interrupt vector connection definitions, and (3) disconnect user-level interrupt processes for which a connection has been enabled. Procedures for using this utility are explained in “Viewing User-Level Interrupt Connections.”

A set of **libud** library routines enables you to raise and lower a processor’s interrupt priority level from user level. These routines include **spl_map(3X)**, **spl_request(3X)**, and **spl_unmap(3X)**. Procedures for using these routines and the related **spl_request_macro** are explained in “Using the spl Support Routines.”

The **vme_address(3C)** library routine enables you to obtain a 32-bit physical address for a specified device’s A24 or A16 VME address. Use of this routine is described in “Using the vme_address Routine.”

Appendix C contains an example C program that demonstrates use of the system calls and selected library routines by a user program that executes a user-level interrupt process and interrupt-handling routine.

Connecting to an Interrupt Vector

Defining a connection between a user-level interrupt process and an interrupt vector and enabling that connection requires the following steps:

1. Provide for communication between the user-level interrupt process and other processes by creating or attaching a shared memory region. You can do so by using the **shmget(2)** and the **shmat(2)** system calls, respectively. Use of these calls is fully explained in the *PowerMAX OS Programming Guide*. Note that in a multithreaded process that does not require interprocess communication, this step is not necessary because the address space is shared among the threads.

2. Determine the interrupt vector to which you wish to connect the user-level interrupt process. Procedures for performing this step are explained in “Obtaining an Interrupt Vector.”

3. Set up an interrupt connection structure, and define a connection between the user-level interrupt process and the interrupt vector obtained in Step 2.

   The interrupt connection structure is defined in the header file <**sys/iconnect.h**>. Among the fields that this structure contains are those that specify the interrupt vector and the address of the interrupt-handling routine. You define the interrupt vector connection by using the **iconnect(3C)** library routine and specifying the interrupt connection structure. The fields in the structure and the procedures for using this call are explained in “Using the iconnect Library Routine.”

4. Lock the user-level interrupt process’s pages in physical memory. You can do so by using the **mlock(3C)** or **mlockall(3C)** library routines. It is

recommended that you lock the entire process in memory by using **`mlockall(3C)`**. Use of this routine is fully explained in the *PowerMAX OS Programming Guide*.

5. Enable the user-level interrupt process's interrupt vector connection defined in Step 3. You can do so by using the **`ienable(3C)`** library routine. Procedures for using this routine are explained in "Using the ienable Library Routine."

   Note that the user-level interrupt process will not return from this call unless an error occurs during the **`ienable(3C)`** call or another process or process thread disconnects it from the interrupt vector. It will execute only at interrupt level in the interrupt-handling routine when the connected interrupt occurs.

Obtaining an Interrupt Vector

To use the **`iconnect`** library routine to define an interrupt vector connection, you must be able to specify the interrupt vector number to be connected to the user-level interrupt process. You can obtain the interrupt vector number by using one of the following methods:

1. Use the **`ioctl`** system call, and specify the IOCTLVECNUM command.

   You use this method if you are connecting to an interrupt vector generated by a device that has a kernel device driver that supports the IOCTLVECNUM **`ioctl`** command. (The kernel device drivers for the real-time clock and edge-triggered interrupt support this command.) Procedures for using this method are explained in "Using the IOCTLVECNUM ioctl System Call."

2. Use the ICON_IVEC **`iconnect`** call to allocate an interrupt vector.

   You use this method if you are connecting to an interrupt vector generated by a device that allows its interrupt vector number to be programmed and does not have a kernel device driver that supports the IOCTLVECNUM **`ioctl`** command. Procedures for using this method are explained in "Allocating Interrupt Vectors."

   NOTE

   After using this method to allocate an interrupt vector, you must program the device so that it interrupts at that vector.

3. Reserve an interrupt vector by modifying the interrupt vector table associated with your machine.

   You use this method if you are connecting to an interrupt vector that is generated by a device that interrupts at a fixed vector number and does not have a kernel device driver that supports the IOCTLVECNUM **`ioctl`** command. Procedures for using this method are explained in “Modifying the Interrupt Vector Table.”

Using the IOCTLVECNUM ioctl System Call

To use the IOCTLVECNUM **`ioctl`** call, you must first make an **`open(2)`** call to obtain a file descriptor for the device special file corresponding to the real-time clock (**`rtc`**) or edge-triggered interrupt (**`eti`**) or RCIM distributed interrupt. For information on the device special file names associated with these devices, refer to Chapter 12 of this guide.

After you have obtained a file descriptor, use the following specifications to make the **`ioctl`** call:

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

int ioctl(fildes, IOCTLVECNUM, arg)
int fildes;
int *arg;
```

Arguments are defined as follows:

fildes	the file descriptor for the device special file corresponding to the selected device
IOCTLVECNUM	the command to place the interrupt vector number of the device in the location pointed to by *arg*
arg	a pointer to the location to which the interrupt vector number of the device will be returned

After you have obtained the interrupt vector, you may free the specified file descriptor by using the **`close(2)`** system call; you are not required to do so prior to defining and enabling the interrupt vector connection, however.

NOTE

Certain device drivers (the driver for **rtc**, for example) reset the device when the last **`close(2)`** is issued for the device’s file descriptor; as a result, you are advised to use caution when closing a device file descriptor.

Modifying the Interrupt Vector Table

On Series 6000 systems, there is one interrupt vector table: **`ivt`**, which is contained in file: **`/etc/conf/cf.d/ivt.s`**.

You can modify the interrupt vector table by using a text editor of your choice.

Procedures for modifying the interrupt vector table for Series 6000 systems are as follows:

1. Search the table for entries that are marked STRAY.

2. Select a STRAY entry that corresponds to one of the highest vector numbers, and modify it by entering the following:

   ```
   RESERVED
   ```

3. Repeat Step 2 for each of the interrupt vector entries that you need to reserve.

4. If you have modified an entry that includes a number of interrupt vectors (for example, `STRAY4, STRAY8, STRAY16, and STRAY32`, where `4, 8, 16`, and `32` denote the number of interrupt vector entries represented), then use the appropriate combination of `STRAY, STRAY4, STRAY8, STRAY16`, and `STRAY32` to modify succeeding `STRAY` entries to indicate the remaining unused vector numbers.

5. Run **`idbuild(1M)`** to rebuild the kernel. Refer to the **`idbuild(1M)`** system manual page for details.

Using the iconnect Library Routine

The **`iconnect`** library routine allows the calling process to perform the following functions:

- Define a connection between the user-level interrupt process and an interrupt vector.
- Disconnect a user-level interrupt process from an interrupt vector.
- Allocate and free interrupt vectors.
- Obtain information about the status of an interrupt vector.
- Lock or unlock memory pages internal to the C or threads library that are referenced during the entry to and exit from the user-level interrupt routine.

NOTE

To use the **`iconnect`** library routine, the kernel must have been configured with the **`ui`** kernel driver.

The specifications required for making the **`iconnect`** call are as follows:

```
#include <sys/types.h>
#include <sys/iconnect.h>

int iconnect(command, arg)

int command;

union {
    struct icon_conn *ic;
    struct icon_ivec *ii;
    struct icon_stat *is;
    int              vector;
} arg;
```

Arguments are defined as follows:

command	the operation to be performed
arg	an interrupt vector number or a pointer to a structure. The value of *arg* depends upon the operation specified by *cmd*.

Command can be one of the following. The values of *arg* that are associated with each command are indicated.

ICON_CONN	define an interrupt vector connection. *Arg* points to an **`icon_conn`** structure.
ICON_DISC	disconnect the connected interrupt process from the interrupt vector and remove the defined user-level interrupt vector connection. *Arg* specifies the interrupt vector number.
ICON_IVEC	allocate or free interrupt vector(s). *Arg* points to an **`icon_ivec`** structure.
ICON_STAT	obtain information about the status of an interrupt vector. Information can be obtained for the following types of interrupt vectors: (1) those that have been connected to a user-level interrupt process with an ICON_CONN **`iconnect`** call, (2) those for which an interrupt vector connection has been enabled with an **`ienable`** call, (3) those that have been allocated with an ICON_IVEC **`iconnect`** call, and (4) those that are undergoing a transition from the **`iconnected`** to the **`ienabled`** state. *Arg* points to an **`icon_stat`** structure.
ICON_LOCK	lock or unlock pages internal to the C or threads library that are referenced during the entry to and exit from a user-level interrupt routine.

Procedures for using these commands are explained in the sections that follow. Use of the ICON_CONN command to define an interrupt vector connection is explained in “Defining an Interrupt Vector Connection”; use of the ICON_DISC command is explained in “Disconnecting a Process from an Interrupt Vector”; use of the ICON_IVEC command is explained in “Allocating Interrupt Vectors”; use of the ICON_STAT command is explained in “Obtaining the Status of Interrupt Vectors”; use of the ICON_LOCK command is explained in “Locking Library Memory Pages”

Defining an Interrupt Vector Connection

To define a connection between an interrupt vector and a user-level interrupt process, you specify the ICON_CONN command on an **`iconnect`** call and provide a pointer to an **`icon_conn`** structure. In a multithreaded process, the calling thread must be a bound (THR_BOUND) thread. See **`thr_create(3thread)`** for more information. Note that to use the ICON_CONN command, the calling process or thread must have the P_USERINT privilege.

The **`icon_conn`** structure is defined as follows:

```
struct icon_conn {
       int   ic_version;
       u_int ic_flags;
       int   ic_vector;
       void  (*ic_routine)();
       int   ic_stack;
       int   ic_value;
       int   ic_ipl;
};
```

The fields in the structure are described as follows.

ic_version — must currently contain the value **IC_VERSION1** for Night Hawk and **IC_VERSION2** for Power Hawk (requires additional argument in I**CON_CONN** -- see **IC_IPL**)

ic_flags — contains zero or an integer value that sets one or more of the following bits:

IC_KROUTINE	causes the kernel device driver’s interrupt routine to be called for the specified interrupt <u>after</u> the user-level interrupt-handling routine has been executed
IC_NOSAVEXFP	prevents the floating-point registers from being saved before executing the user-level interrupt-handling routine and from being restored after executing the user-level interrupt-handling routine. This means that in return for faster interrupt response time, the user-level interrupt-handling routine must <u>not</u> modify any floating-point registers that it does not save and restore to their original values.
IC_SAVEFP	is silently ignored but exists for backward compatibility

IC_DEBUG	enables you to enter the console processor debugger on the first occurrence of the connected interrupt

ic_vector — contains the interrupt vector number to be connected to the user-level interrupt process. You must have obtained this number previously by using one of the following methods: (1) by making an IOCTLVECNUM **`ioctl`** system call, (2) by making an ICON_IVEC **`iconnect`** call, or (3) by reserving a particular interrupt vector. These methods are described in “Obtaining an Interrupt Vector.”

ic_routine — contains the virtual address of the process’s user-level interrupt-handling routine

ic_stack — contains the virtual address of the process’s data area that will be used as a stack area during execution of the user-level interrupt-handling routine (remember to specify the address of the top of the stack because the stack grows from higher addresses to lower addresses). At a minimum, allocate 1024 bytes for the stack area. If the routine makes many nested subroutine calls or involves routines that use many local variables, a larger stack size may be necessary.

ic_value — contains a value to be passed to the user-level interrupt-handling routine on each occurrence of the connected interrupt

ic_ipl — on Power Hawk systems only, specifies the processor interrupt priority level at which to execute the user-level interrupt-handling routine. Refer the file **`<sys/ipl.h>`** for a set of symbolic constants that has been dined to assist you in specifying this value.

A return value of **0** indicates that the ICON_CONN **`iconnect`** call has been successful. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`iconnect(3C)`** system manual page for a listing of the types of errors that may occur.

NOTE

After a user-level interrupt process has defined a connection to an interrupt vector by using the ICON_CONN command, no other process is allowed to define a connection to the same interrupt vector until a corresponding ICON_DISC command has been issued or the connected interrupt process has exited.

Disconnecting a Process from an Interrupt Vector

To disconnect a user-level interrupt process from an interrupt vector and remove the defined interrupt vector connection, you specify the ICON_DISC command on an **`iconnect`** call and provide the interrupt vector number to be disconnected. Note that to use this command, the calling process must have the P_USERINT privilege.

A user-level interrupt process that has defined an interrupt vector connection but has not yet made a related **`ienable`** call can use this command to remove the defined connection.

If a process that has defined an interrupt vector connection terminates prior to calling **`ienable`**, the defined connection is automatically removed from the system; in this case, another process is allowed to define an interrupt vector connection for the previously used interrupt vector number.

If the ICON_DISC **`iconnect`** call is successful, the return value is **0**. A process that is blocked in an **`ienable`** call for the specified interrupt is unblocked.

A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`iconnect(3C)`** system manual page for a listing of the types of errors that may occur.

Allocating Interrupt Vectors

To allocate or free an interrupt vector, you specify the ICON_IVEC command on an **`iconnect`** call and provide a pointer to an **`icon_ivec`** structure. Note that to use this command, the calling process must have the P_USERINT privilege.

The **`icon_ivec`** structure is defined as follows:

```
struct icon_ivec {
       u_int ii_flags;
       int   ii_nvect;
       int   ii_vector;
};
```

The fields in the structure are described as follows.

ii_flags — contains zero or an integer value that sets one or more of the following bits:

II_ALLOCATE	causes interrupt vectors to be allocated. This is the default action if neither II_ALLOCATE nor II_DEALLOCATE is set. The following two bits may be set in conjunction with the II_ALLOCATE bit:
II_EVEN_VECTOR	ensures that the first interrupt vector that is allocated is an even number. This bit is silently ignored if II_VECSPEC is also set.
II_VECSPEC	enables you to specify the first interrupt vector number to be allocated. The requested interrupt vector number is returned if it is available; otherwise, the call fails, and a value of **-1** is returned.
II_DEALLOCATE	causes the specified interrupt vectors to be freed

ii_nvect — specifies the number of interrupt vectors to be allocated or freed. Note that interrupt vectors are always allocated or freed in a contiguous block.

ii_vector specifies the first vector to be allocated if the II_VECSPEC bit is set in the **ii_flags** field or specifies the first vector to be freed if the II_DEALLOCATE bit is set.

If the specified number of interrupt vectors is successfully allocated, the ICON_IVEC **iconnect** call returns the first interrupt vector number that has been allocated. If the II_DEALLOCATE bit is set and the specified interrupt vectors are successfully freed, the ICON_IVEC **iconnect** call returns a value of zero. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **iconnect(3C)** system manual page for a listing of the types of errors that may occur.

After you have used the ICON_IVEC command to allocate an interrupt vector, you can use the ICON_CONN command to connect a user-level interrupt process to the allocated interrupt vector. It is important to note that the interrupt vector continues to be allocated until an ICON_IVEC **iconnect** call is made with the II_DEALLOCATE bit set. The process that calls **iconnect** to allocate an interrupt vector is not directly associated with the interrupt vector. Another process that has the appropriate privilege can connect to the interrupt vector or free it.

Obtaining the Status of Interrupt Vectors

To obtain information about the status of one or more interrupt vectors, you specify the ICON_STAT command on an **iconnect** call and provide a pointer to an **icon_stat** structure. Note that to use this command, the calling process must have the P_SYSOPS privilege.

The **icon_stat** structure is defined as follows:

```
struct icon_stat {
       u_int   is_flags;
       int     is_vector;
       pid_t   is_pid;
       u_int   is_state;
       lwpid_t is_lwpid;
};
```

The fields in the structure are described as follows.

is_flags contains zero or an integer value that sets one of the following mutually exclusive bits:

IS_SPEC enables you to indicate that status information about the interrupt vector specified in the **is_vector** field is to be returned

IS_SCAN enables you to indicate that the system table of user-level interrupt information is to be scanned for the next interrupt vector that has been allocated or the next interrupt vector for which an interrupt vector connection has been defined or enabled. The table is indexed by interrupt vector number and scanned in ascending order. The point at which scanning of the table begins is determined by the value contained in the **is_vector** field.

is_vector	contains the interrupt vector number for which status information is to be returned if the IS_SPEC bit in the **`is_flags`** field is set. If the IS_SPEC bit is <u>not</u> set or the IS_SCAN bit is set, this field contains **-1** or the interrupt vector number from which scanning is to begin. If this field contains **-1**, scanning begins with the entry for interrupt vector **0**; otherwise, scanning begins with the entry for the interrupt vector number obtained by adding **1** to the number specified in this field. In either case, on return, this field contains the next interrupt vector number that has been allocated or for which an interrupt vector connection has been defined or enabled. Note that to obtain information on the status of <u>all</u> of the interrupt vectors that have been allocated or for which an interrupt vector connection has been defined or enabled, you must repeatedly call **`iconnect(3C)`** and specify the same **`icon_stat`** structure until a value of **-1** is returned and **`errno`** is set to **ENOTCONN**.
is_pid	on return, contains **-1** or a process identification number (PID). This field contains **-1** if the IS_IVEC bit in the **`is_state`** field is set and the IS_CONN bit is <u>not</u> set; in this case, the interrupt vector contained in the **`is_vector`** field is not currently connected to a user-level interrupt process. If one or both of the IS_CONN and IS_ENABLED bits in the **`is_state`** field are set, this field contains the PID of the process that made the ICON_CONN **`iconnect`** call to connect a user-level interrupt process to the interrupt vector. If the IS_PENDING bit in the **`is_state`** field is set, this field contains the PID of the process that made the ICON_CONN **`iconnect`** call to connect a user-level interrupt process to the interrupt vector.
is_lwpid	on return, contains **-1** or a lightweight process identifier (LPW ID). This field contains **-1** if the IS_IVEC bit in the **`is_state`** field is set and the IS_CONN bit is <u>not</u> set; in this case, the interrupt vector contained in the **`is_vector`** field is not currently connected to a user-level interrupt process. If one or both of the IS_CONN and IS_ENABLED bits in the **`is_state`** field are set, this field contains the LWP ID associated with the bound thread that made the ICON_CONN **`iconnect`** call to connect a user-level interrupt process to the interrupt vector. If the IS_PENDING bit in the **`is_state`** field is set, this field contains the LWP ID associated with the bound thread that made the ICON_CONN **`iconnect`** call to connect a user-level interrupt process to the interrupt vector.
is_state	on return, contains an integer value that sets one or more of the following bits:

IS_CONN	indicates that the interrupt vector contained in the **`is_vector`** field has been successfully

	connected to a user-level interrupt process with an ICON_CONN **`iconnect`** call.
IS_ENABLED	indicates that an interrupt vector connection for the interrupt vector contained in the **`is_vector`** field has been successfully enabled with an **`ienable`** call.
IS_IVEC	indicates that the interrupt vector contained in the **`is_vector`** field has been successfully allocated with an ICON_IVEC **`iconnect`** call.
IS_PENDING	indicates that the interrupt vector contained in the **`is_vector`** field is currently undergoing a transition from the **iconnected** to the **ienabled** state. Because this transition state is short-lived, the IS_PENDING bit is not normally set.

Locking Library Memory Pages

One restriction placed on a user-level interrupt routine is that all memory references must be only to memory-locked pages. You may lock down the entire application's address space with the **`mlockall(3C)`** library routine or the **`memcntl(2)`** system call, or you may selectively lock portions of the application’s address space with **`mlock(3C)`**, **`userdma(2)`**, or **`memcntl(2)`**. If your application is large, you may wish to selectively lock down those portions of the user-level interrupt routine’s instructions and data that are referenced during the user-level interrupt routine’s execution.

In addition to the memory accessed by your user-level interrupt routine code, there are additional instruction and data accesses made within the C or threads library. These additional memory accesses are made as the user-level interrupt process or process thread enters into and exits out of your user-level interrupt routine.

In order to provide better selective page-locking support for user-level interrupt applications, a ICON_LOCK **`iconnect(3C)`** command may be used to either lock or unlock those pages of the program internal to the C or threads libraries that will be referenced as the process or process thread enters and exits the user-level interrupt routine. Note that to use this command, the calling process must have the P_PLOCK privilege.

Note that you will still be responsible for locking down those portions of the application that are referenced by the code that you have written that is executed by your user-level interrupt routine.

To lock or unlock the library pages, you specify the ICON_LOCK command on an **`iconnect`** call and provide a pointer to an **`icon_lock`** structure. The **`icon_lock`** structure is defined as follows:

```
struct icon_lock {
       u_int il_flags;
};
```

The field in the **`icon_lock`** structure is described as follows.

il_flags	contains a value that has one of the following mutually exclusive bits set:

IL_LOCK	Lock the library pages in memory.
IL_UNLOCK	Unlock the library pages.

When the IL_LOCK bit in the **`il_flags`** field is set, then all the data structures and instructions that are referenced within the C or threads library during a process's or process thread's entry into and exit from a user-level interrupt routine will be locked down. For multithreaded processes, if more than one interrupt connection definition exists, then all data structures used by all the currently existing interrupt connections will be locked down.

Multiple ICON_LOCK commands that are made with the IL_LOCK bit set will nest; that is, the same number of ICON_LOCK command calls with the IL_UNLOCK bit set will be required to remove the internal library page locks associated with user-level interrupt routines.

When the IL_UNLOCK bit in the **`il_flags`** field is set and the preceding nesting requirement has been met, then all instructions and data structures internally used by the C or threads library for user-level interrupt routine processing will be unlocked. For multithreaded applications that have more than one interrupt connection definition, all internal data structures will be unlocked. The unlock call should not be made until the process or process thread has been disconnected from the interrupt.

Using the ienable Library Routine

The **`ienable`** library routine allows the user-level interrupt process to enable a defined interrupt vector connection. The calling process must previously have made an **`iconnect`** call to define the interrupt vector connection. In a multithreaded process, the calling thread must be the same thread that made the previous corresponding **`iconnect`** call to define the interrupt vector connection.

The specifications required for making the **`ienable`** call are as follows:

```
#include <sys/types.h>
#include <sys/iconnect.h>

int ienable(vector_number)

int vector_number;
```

The argument is defined as follows:

vector_number	the interrupt vector number for which the defined connection is to be enabled. This number must be the same as that specified on a previous ICON_CONN **`iconnect`** call.

The **`ienable`** library routine places the calling process or process thread in a blocked state in the kernel and then enables the process's interrupt vector connection. While the process or process thread is in this state, all signals are ignored. The process or process thread no longer executes at normal program level. Each time the connected interrupt becomes active, the CPU that is receiving the interrupt switches to the context of the connected interrupt process or process thread within the kernel. The kernel then jumps to the beginning of the interrupt-handling routine with the connected interrupt still active. Although the connected interrupt is active, the process or process thread will be executing in user mode rather than kernel mode; all of the process's virtual address space that has previously been locked into physical memory is accessible.

The calling process will continue to block until another process or process thread makes an ICON_DISC **`iconnect`** call and specifies the interrupt vector to which the blocked process is connected. The blocked process can also be disconnected from the interrupt vector by using the **`uistat`** command and specifying the **`-d`** option (see "Viewing User-Level Interrupt Connections" for an explanation of the procedures for using this command). When the blocked process is wakened, the **`ienable`** routine returns a value of zero.

If the **`ienable`** routine is not successful, the user-level interrupt process does not block. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`ienable(3C)`** system manual page for a listing of the types of errors that may occur.

Viewing User-Level Interrupt Connections

The **`uistat(1)`** utility enables you to display information about the following types of interrupt vectors: (1) those that have been connected to a user-level interrupt process with an ICON_CONN **`iconnect`** call, (2) those for which an interrupt vector connection has been enabled with an **`ienable`** call, and (3) those that have been allocated with an ICON_IVEC **`iconnect`** call. It also allows you to disconnect a connected interrupt process from an interrupt vector and free an interrupt vector that has been allocated with the ICON_IVEC **`iconnect`** call.

The format for executing the **`uistat`** utility is as follows:

```
uistat  [ -d vector ]  [ -f vector ]
```

If you execute **`uistat`** without specifying an option, you can view information for each interrupt connection in the system. The following information is displayed on your terminal screen:

- The interrupt vector number
- The process identification number (PID) of the connected or enabled process
- The lightweight process identifier (LWP ID) of the connected or enabled process or process thread
- The status of the vector

The status that is displayed for user-level interrupt processes is either **iconnected** or **ienabled**. If a process has defined a connection to an interrupt vector, the status is **iconnected**. If a process has both defined a connection and enabled it, the status is **ienabled**. If an interrupt vector has been allocated by using the ICON_IVEC **`iconnect`** call, the status that is displayed is **ivec**; in this case, the PID and LWP ID are displayed only if a process has defined a connection to the interrupt vector.

If you execute **`uistat`**, information similar to the following is displayed on your terminal screen:

```
Vector #   Proc Id   LWP Id   Status
--------   -------   ------   ------
   101      14238       1      ienabled
   132      15482       1      iconnected
   152       2326       2      ienabled ivec
   153      14241       4      iconnected ivec
   154      -----     ---      ivec
```

The **`-d`** option enables you to disconnect a process that is in the **iconnected** or **ienabled** state from the interrupt vector and remove the defined user-level interrupt connection. The *vector* argument specifies the interrupt vector number from which the process is to be disconnected. When the process is disconnected, it will return from the **`ienable`** call.

The **`-f`** option enables you to free an interrupt vector that has been allocated by using an ICON_IVEC **`iconnect`** call but is not in the **iconnected** or **ienabled** state. The *vector* argument specifies the interrupt vector number that is to be freed. If you wish to free an interrupt vector that is in the **iconnected** or the **ienabled** state, you must first use the **`-d`** option to disconnect the process from the interrupt vector.

NOTE

To use **`uistat`** without any options, you must have the P_SYSOPS privilege. To use the **`-d`** and the **`-f`** options, you must have the P_USERINT privilege.

If an error occurs when you execute **`uistat`**, a message indicating the reason for the failure is displayed.

Using Local Memory

If a process binds some portion of its address space to local memory and then issues the **`iconnect(3C)`** and **`ienable(3C)`** calls in order to connect to an interrupt, the CPU that processes the interrupt may not be located on the same CPU board where the process's address space bindings were created. In this case, some of the local memory references that were not previously remote may now become remote memory references. Similarly, some of the previously remote local memory references may now no longer be remote references. In these cases, data incoherencies may occur when the user-level interrupt process references these portions of its address space.

Note that remote memory references are not an issue on Series 6000 systems that have only one processor board.

For those user-level interrupt applications that wish to bind some portion of their address space to local memory on a Series 6000 system that has more than one CPU board, the following steps must be taken in order to prevent data incoherencies.

1. Determine which CPU is receiving the interrupt to which you wish to connect the user-level interrupt process.

   You can do so by using one of the following methods: (1) use the **`intstat(1M)`** utility, or (2) invoke the **`mpadvise(3C)`** library routine from a program and specify the MPA_CPU_INTVEC or the MPA_CPU_VMELEV command. (For (H)VME interrupts, use the MPA_CPU_VMELEV command; for other interrupts, use the MPA_CPU_INTVEC command.) For additional information, refer to the **`intstat(1M)`** and **`mpadvise(3C)`** system manual pages.

2. Set the process's CPU bias to include, at most, those CPUs that reside on the same processor board where the interrupt is received.

   You can do so by using the **`mpadvise(3C)`** library routine and specifying the MPA_CPU_LMEM and MPA_PRC_SETBIAS or MPA_PRC_SETRUN commands as explained in the corresponding system manual page.

3. If desirable, create one or more shared memory regions that are bound to local memory.

   You can do so by using the **`shmget(2)`** system call, the **`shmdefine(1)`** utility, or the **`shmconfig(1M)`** utility as explained in the *PowerMAX OS Programming Guide*.

Data from local or global memory can be stored in a CPU's data cache; however data from remote memory cannot because there is no hardware mechanism to keep cached, remote data coherent. To prevent remote data from being stored in a CPU's data cache, the operating system builds virtual-to-physical translations to remote memory in a special way (a bit is set in the page table entry to inform the hardware that the data cache should not be used).

The terms *local* and *remote* describe a relationship between a page frame and a CPU. If translations were built with respect to one CPU and subsequently used from a different CPU, it is possible that data from remote memory would be stored in the cache. In most circumstances, the operating system's load-balancing and process-migration algorithms prevent the situation just described from occurring. It can occur, however, when a process running on one CPU board connects to an interrupt serviced by a different CPU board. It is for this reason that the steps outlined above are recommended for applications using local memory.

On Series 6000 systems, all external interrupts are connected to particular pins on the terminator board. Associated with each pin are CPU, priority, and vector attributes that are configured by the operating system at system initialization time. You can ensure that a certain logical CPU receives interrupts occurring from a particular pin by configuring the CPU attribute associated with that pin. To assign specific interrupts sources to specific CPUS, see the section "Assigning Interrupts to CPUs" in Chapter 2 of this manual.

NOTE

The kernel tunables that are associated with interrupt source CPU assignments are initially set to be round-robin by default. In these cases, the operating system will assign these interrupt sources to the available CPUs in the system in a round-robin fashion, in an attempt to spread out these configurable interrupt CPU assignments across all CPUs in the system. Therefore, note that changing certain interrupt source tunables to be set to specific CPUs may also have the side-affect of altering the CPU assignments made for one or more of the round-robin interrupt sources.

The local memory to which a process's address space is bound is that of the CPU on which the process is executing. You can ensure that the CPU on which the user-level interrupt process executes is the same as the CPU that will receive the connected interrupt by using the MPA_PRC_SETBIAS **mpadvise(3C)** library routine and specifying only that CPU or those CPUs on the same processor board in the process's bit mask.

An overview of local memory and an explanation of the procedures for using it are provided in the memory management chapter of the *PowerMAX OS Programming Guide*. If you wish to use local memory with user-level interrupt processes, it is recommended that you review that chapter and refer to the **memory(7)**, **memdefaults(2)**, **mpadvise(3C)**, and **shmget(2)** system manual pages for additional information.

Interrupt-Handling Routine Constraints

One parameter is passed to a user-level interrupt-handling routine: the value that is specified in the **ic_value** field of the **icon_conn** structure supplied on the **iconnect(3C)** call that defines the connection between the user-level interrupt process and an interrupt vector. The interrupt-handling routine is entered in user mode with the connected interrupt still active.

An interrupt-handling routine can reference any memory location that is in the virtual address space of the user-level interrupt process--including VME I/0 memory space to which the process's virtual address space has previously been bound. Note, however, that portions of the user-level interrupt process's address space that are referenced by the interrupt-handling routine must have been locked into physical memory prior to enabling the interrupt vector connection with an **ienable(3C)** call.

Any type of exception (page fault, floating-point exception, and so on) is fatal during execution of an interrupt-handling routine. In the kernel, the exception-handling code checks for interrupt-handling routines. If an interrupt-handling routine causes an exception, the kernel indicates this occurrence by printing a message on the console and specifying the type of trap and the value of the program counter; it then executes a system **panic()**. The system must panic because the state of the process that was executing at the time that the connected interrupt occurred cannot be recovered. An example of the output that is displayed on the system console is as follows:

```
System trap from user interrupt routine:
```

```
          type = Misaligned Access Exception, pc = 0x000209A8,
              interrupt vector = 93

      PANIC: User-level interrupt trap.
```

A multithreaded process that has one or more threads that are currently blocked in an **`ienable(3C)`** call (connected to an external interrupt) must not make a **`fork(2)`** or **`forkall(2)`** system call. In this case, the **`fork(2)`** or **`forkall(2)`** call returns **`-1`** and sets **`errno`** to EINTR. However, the **`fork1(2)`** system call is allowed from a multithreaded process that currently has one or more **`ienabled`** threads within it.

For multithreaded processes, all threads library functions that might block the calling thread, such as **`thr_yield(3thread)`** and **`thr_join(3thread)`**, must not be used within a user-level interrupt routine. The **`_spin_lock(3sync)`** and **`_spin_unlock(3sync)`** primitives may be called from a user-level interrupt routine, provided that the program-level threads have properly used the **`spl(3X)`** functions around the spin lock, thus avoiding any potential deadlock with the user-level interrupt routine's acquisition of the lock. All other threads synchronization primitives, such as **`mutexes`** and **`rwlocks`**, for example, should be avoided.

An interrupt-handling routine may make only two system calls: **`server_wake1(2)`** and **`server_wakevec(2)`**. These calls enable the calling process to wake one or more processes that are blocked in the **`server_block(2)`** system call. (Use of these system calls is fully explained in Chapter 6.)

If an interrupt-handling routine makes a system call other than **`server_wake1`** or **`server_wakevec`**, the kernel will set **`errno`** to EINVAL and return to the routine without executing the call.

If you write assembly language code that is executed during processing of the interrupt-handling routine, you must save and restore the nonvolatile registers that are used. Applicable registers are identified as follows:

- Register r16 must be saved and restored if it is used.
- If the IC_NOSAVEXFP flag is set, all floating-point registers must be saved and restored if they are modified by the interrupt-handling routine.

The interrupt-handling routine may call other routines, but it must eventually exit via an explicit or implicit return from inside the routine whose address is specified in the **`ic_routine`** field of the **`icon_conn`** structure supplied on the **`iconnect(3C)`** call.

Debugging the Interrupt-Handling Routine

Because the interrupt-handling routine executes at interrupt level, you may not use such user-level debuggers as **`adb(1)`** and **`nview(1)`** to debug it; however, you may use the console processor to obtain some debugging capability for this routine. You may use the console processor as an assembler-level debugger that enables you to display and modify the contents of registers and memory, set breakpoints, single step through instructions, disassemble instructions, and reference variables and routines by using their symbolic names. You can use it to debug multiple interrupt-handling routines simultaneously.

CAUTION

User-level interrupt-handling routines can potentially cause the system to hang or panic. You must initially debug them on a single-user system.

To debug an interrupt-handling routine, you must set the IC_DEBUG bit in the **`ic_flags`** field of the **`icon_conn`** structure that you specify on the ICON_CONN **`iconnect(3C)`** call. If you set this bit, the console processor is entered the first time that the connected interrupt becomes active. The console processor is entered in an intermediate C or threads library routine--several instructions before the beginning of your interrupt-handling routine. By using several single-step processor commands (the **`z`** command), you can place the process at the beginning of the user-level interrupt-handling routine.

Any console breakpoints that are set in order to debug the user-level interrupt code may also be encountered by other processes running the same program at user level. If any of these user level processes encounters one of the console breakpoints, the process will be core dumped and terminated with a SIGTRAP signal.

Therefore, care should be taken to not set breakpoints in code that is common to both interrupt level and non-interrupt level processes that are executing the same program at the same point in time. This can be accomplished by either:

- Temporarily creating a duplicate of the program for debugging the user-level interrupt routine

  or

- Rewriting the user-level interrupt routine so that it does not use any code that is executed at non-interrupt level by other processes.

If you have not stripped symbol table information from the object program that you are debugging (by specifying the **`-s`** option when executing the **`ld(1)`** command, for example), then you will be able to use the symbolic names of variables and routines while debugging the interrupt-handling routine. When you first enter the interrupt-handling routine, you must use the **`di %`** console processor command (disassemble instructions using the value of the program counter) to load the user's symbol table into the console processor's internal structures. If the kernel's symbol table were loaded at boot time and you do not issue this command, then the console processor attempts to use the kernel's symbol table.

In the kernel, the address and the symbol count of the user's symbol table must be stored at appropriate console processor memory locations upon entry of each connected interrupt. When the interrupt processing is completed, the kernel must restore the previous values of the symbol table address and symbol count in the console processor's memory. You should avoid this additional kernel interrupt processing overhead by removing the IC_DEBUG flag after you have debugged the interrupt-handling routine.

Interrupts that occur after the first connected interrupt are not automatically forced into the console processor. If you must debug additional interrupts, you must set and leave breakpoints in the interrupt-handling routine. You should remove all interrupt-handling routine breakpoints before the connected interrupt process is disconnected. The removal of these breakpoints should be done while the console processor has the CPU stopped within the interrupt-handling routine.

On multiprocessor systems, you should not attempt to set breakpoints or single step user-level code on a different processor unless it is currently executing a user-level interrupt-handling routine. If you wish to set breakpoints or single step an interrupt-handling routine on a different processor, then you must first switch the console processor to that CPU.

On PowerMAX OS systems, you should not attempt to set breakpoints in kernel text or display kernel data while in user mode. It is important to note that on PowerMAX OS systems, the query stack (**qs**) console command does not work properly if it is issued while in an interrupt-handling routine.

For additional information on procedures for using console processor debugging commands, refer to the console manual that is applicable to your machine (see the Preface for the list of Referenced Publications).

Understanding the Processor IPL

A process can ensure that interrupts at or below a certain level are held out by temporarily raising a processor's interrupt priority level (IPL). A CPU's IPL is normally set to zero for user-level program operation. By raising a CPU's IPL to **PL4**, for example, a process can hold out all VME level 4 interrupts and thereby prevent level 4 VME devices (the HPS Serial Controller, for example) from interrupting the CPU.

The IPL is specific to a particular CPU. Setting the IPL on one CPU does not affect the IPL on another CPU. Interrupts at or below the value to which the IPL is set on one CPU may continue to be received on other CPUs--whether or not such interrupts continue to be received on another CPU depends upon the value of the IPL on that CPU.

A set of symbolic constants has been defined to assist you in setting the processor interrupt priority level on the machine. These constants are defined in the file **/usr/include/sys/ipl.h**. They are presented in Table 8-1.

Table 8-1. IPL Values

Value	Effect
PLXCALL	Holds out most interrupts. These include (H)VME interrupts; edge-triggered interrupts; interprocessor interrupts; I/O bus error interrupts; sysfault and console wakeup interrupts; real-time clock interrupts; and hardclock interrupts. Use of this level is strongly discouraged.
PL8	Holds out (H)VME interrupts; edge-triggered interrupts; real-time clock interrupts; and hardclock interrupts. This level also holds out sysfault interrupts.
PLPROBE	Holds out (H)VME interrupts; edge-triggered interrupts; real-time clock interrupts; and hardclock interrupts. This level can be used to hold out all interrupts that are not critical.

Table 8-1. IPL Values (Cont.)

Value	Effect
PL7	Holds out (H)VME level 1-7 interrupts on the primary bus and the secondary bus. Note that this level and lower levels do not hold out any edge-triggered interrupts. This level does not hold out real-time clock and hardclock interrupts.
PL6	Holds out (H)VME level 6 interrupts on the primary bus and the secondary bus.
PL5	Holds out (H)VME level 5 interrupts on the primary bus on all systems and on the secondary bus. This is the lowest IPL level that will hold out the console terminal send and receive interrupts.
PL4	Holds out (H)VME level 4 interrupts on the primary bus on all systems and on the secondary bus.
PL3	Holds out (H)VME level 3 interrupts on the primary bus on all systems and on the secondary bus.
PL2	Holds out (H)VME level 2 interrupts on the primary bus on all systems and on the secondary bus.
PL1	Holds out (H)VME level 1 interrupts on the primary bus on all systems and on the secondary bus.
PLTIMEOUT	This is the lowest IPL level that will hold out the softclock interrupt. This level does not hold out (H)VME interrupts.
PLSWTCH	This is the lowest IPL level that will hold out context switch rescheduling interrupts. This level does not hold out (H)VME interrupts or edge-triggered interrupts.
PL0	This value is used to return the IPL to zero. The IPL is normally set to zero for user-level program operation. User-level programs that temporarily raise the IPL to values greater than **PL0** should return the IPL to this value.

For information on the interrupt levels assigned to specific (H)VME boards, refer to the architecture manual that is applicable to your system (see the Preface for the list of Referenced Publications).

Using the spl Support Routines

The user-level device driver library, **`libud`**, contains a set of library routines and a macro that allow you to modify the processor interrupt priority level (IPL) from user level. You can modify the IPL by mapping the physical address of the IPL register to a process's virtual address space and then writing directly to the hardware register. The routines and macro are documented in the section **`3X`** system manual pages and are briefly described as follows:

`spl_map`	map the physical address of the IPL register or processor level register to a process's virtual address space
`spl_request`	set the processor IPL to a specified level
`spl_request_macro`	set the processor IPL to a specified level
`spl_unmap`	unmap the IPL register or processor level register with an **`spl_map`** call

Use of the **`spl_map`** routine is explained in "Using the spl_map Routine." Use of the **`spl_request`** routine is explained in "Using the spl_request Routine." Use of the **`spl_request_macro`** is explained in "Using the spl_request_macro." Use of the **`spl_unmap`** routine is explained in "Using the spl_unmap Routine."

You typically use the **`spl`** routines and macro to allow a user-level process and an associated interrupt-handling routine to coordinate their access to shared data. You can ensure that a user-level process executing in user mode executes on its current CPU without being interrupted by an associated interrupt-handling routine by raising the IPL up to or higher than the interrupt priority level of the connected interrupt.

On multiprocessor systems, you must use an additional user-level spin lock to synchronize the processes' access to shared data among multiple CPUs. It is recommended that you use the **`spin_init`**, **`spin_trylock`**, **`spin_islock`**, and **`spin_unlock`** macros for this purpose. Procedures for using these macros are explained in Chapter 6 of this guide; reference information is provided in the **`spin_trylock(2)`** system manual page.

On a multiprocessor system, a user-level process that shares data with an associated interrupt-handling routine can synchronize its access to the data by performing the following steps:

STEP 1: Raise the IPL to the appropriate level.

STEP 2: Lock the user-defined spin lock.

STEP 3: Access or modify the shared data.

STEP 4: Unlock the user-defined spin lock.

STEP 5: Lower the IPL to zero.

The associated interrupt-handling routine can access the same data by performing the following steps:

STEP 1: Lock the user-defined spin lock.

STEP 2: Access or modify the shared data.

STEP 3: Unlock the user-defined spin lock.

An example C program that demonstrates how a program executing at user level and an associated interrupt-handling routine can synchronize access to shared data by using these steps and the **`spl`** support routines is provided in Appendix D.

Using the spl_map Routine

The **`spl_map(3X)`** routine enables you to map the physical address of the IPL register onto a process's virtual address space. The mapping is achieved by using the **`mmap(2)`** system call to establish a mapping to **`/dev/spl`**, the pseudo-device associated with the register. It is important to note that to use this routine, you must have read and write access to **`/dev/spl`** -- i.e. if not running as `root`, then the calling process must have P_DACWRITE and P_DACREAD privilege.

The specifications required for making this call are as follows:

```
#include <sys/types.h>
#include <sys/ipl.h>

caddr_t spl_map(addr)

caddr_t addr;
```

The argument is defined as follows:

addr	the virtual address at which the register is to be mapped. If you wish the kernel to select the address, specify a value of zero. If not, specify a nonzero value.

If no errors occur, the **`spl_map`** routine returns the virtual address of the IPL register. Otherwise, a value of **-1** is returned, and **`errno`** is set to indicate the error. Refer to the **`spl_request(3X)`** system manual page for a listing of the types of errors that may occur.

NOTE

Such library routines as those contained in a user-level device driver may make **`spl_map`** calls in addition to those made by the nonlibrary code portion of the program. Therefore, if a process makes more than one call to **`spl_map`** before making a call to **`spl_unmap`**, an attempt will be made to use the previously mapped register. In this case, **`spl_map`** will simply increment an internal reference count and return the previously bound virtual address. However, if the *addr* argument is specified on an additional **`spl_map`** call and the address does not match the previously bound **spl** register's address, an error will be returned.

Using the spl_request Routine

The **`spl_request(3X)`** routine enables you to set the IPL to a specified level. It is recommended that you lock the process's pages into physical memory prior to calling this routine. You can do so by using the **`mlock(3C)`** or **`mlockall(3C)`** library routines. Procedures for using these routines are explained in the *PowerMAX OS Programming Guide*.

The specifications required for calling **spl_request** are as follows:

```
#include <sys/types.h>
#include <sys/ipl.h>

void spl_request(value, addr)

u_int    value;
caddr_t  addr;
```

The arguments are defined as follows:

value an IPL value as defined in the file **<sys/ipl.h>** and presented in Table 8-1

addr the virtual address of the IPL or processor level register, which has been obtained on a previous call to **spl_map(3X)**

The **spl_request** routine returns the value that the IPL or processor level register contained prior to the call; it does not return an error status.

A process should raise the IPL to a value greater than zero only for a short time. While a process has the IPL raised, it should not perform any of the following operations:

- Make a system call.
- Reference a memory location that will produce a page fault.
- Set debugger breakpoints or single-step in a section of code where the IPL has been set to a value that is greater than zero.

CAUTION

A user-level process that raises the IPL should always reset it to zero. An interrupt-handling routine that temporarily raises the IPL should restore it to the value to which it was set at entry to the interrupt-handling routine--that is the same value that was returned on the interrupt-handling routine's initial call to **spl_request**.

Using the spl_request_macro

The **spl_request_macro** provides you with a faster means of modifying the value of the IPL or processor level register than the **spl_request(3X)** routine does. It can be called only from C programs; it cannot be called from Ada and Fortran programs.

The specifications required for calling this macro are as follows:

```
#include <sys/types.h>
#include <sys/ipl.h>

spl_request_macro(value, addr, pv)
```

```
u_int    value;
caddr_t  addr;
u_int    pv;
```

The arguments are defined as follows:

value	an IPL value as defined in the file **`<sys/ipl.h>`** and presented in Table 8-1
addr	the virtual address of the IPL or processor level register, which has been obtained on a previous call to **`spl_map(3X)`**
pv	a variable to which the macro returns the previous value of the IPL or processor level register. You must provide this argument.

Using the spl_unmap Routine

The **`spl_unmap(3X)`** routine enables you to unmap the IPL register that has been mapped on a previous call to **`spl_map(3X)`** (see “Using the spl_map Routine” for an explanation of this routine). Note that if the process invokes the **`exec(2)`** or **`exit(2)`** system call, the operating system removes the mapping.

The specifications required for calling this routine are as follows:

```
#include <sys/types.h>
#include <sys/ipl.h>

int spl_unmap(addr)

caddr_t  addr;
```

The argument is defined as follows:

addr	the virtual address of the IPL or processor level register that has been returned on a previous **`spl_map(3X)`** call

A return value of **0** indicates that the **`spl_unmap`** call has been successful. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`spl_request(3X)`** system manual page for a listing of the types of errors that may occur.

If the process has made multiple calls to **`spl_map`** before calling **`spl_unmap`**, the **spl** register will not actually be unmapped until the corresponding number of **`spl_unmap`** calls has been made.

Using the eti Support Routines

A set of C library routines have been developed to allow you to control a particular edge-triggered interrupt. You can do so by binding the physical address of the edge-triggered interrupt to a process's virtual address space and then directly performing the desired control operation. The routines are briefly described as follows:

`eti_map`	bind the physical address of an edge-triggered interrupt to a process's virtual address space
`eti_request`	perform a control operation for an edge-triggered interrupt
`eti_unmap`	detach a shared memory region that has been bound to the edge-triggered interrupt with an **eti_map** call

Use of the **`eti_map`** routine is explained in "Using the eti_map Routine." Use of the **`eti_request`** routine is explained in "Using the eti_request Routine." Use of the **`eti_unmap`** routine is explained in "Using the eti_unmap Routine."

Using the eti_map Routine

The **`eti_map(3C)`** routine enables you to map the physical address of an edge-triggered interrupt onto a process's virtual address space. The mapping is achieved by creating a shared memory region, binding it to the physical address of the edge-triggered interrupt, and attaching it to the process's virtual address space. It is important to note that to use this routine, you must have the P_SHMBIND privilege.

The specifications required for making this call are as follows:

```
#include <sys/types.h>
#include <sys/pin.h>

caddr_t eti_map(pin, addr)

int     pin;
caddr_t addr;
```

The arguments are defined as follows:

pin the pin number of the edge-triggered interrupt whose address you wish to map onto the process's virtual address space.

For Night Hawk systems, the following values may be specified on each system:

0-3	edge-triggered interrupts on CPU board 0
4-7	edge-triggered interrupts on CPU board 1

8-11	edge-triggered interrupts on CPU board 2
12-15	edge-triggered interrupts on CPU board 3

On Power Hawk Series 600/700/900 systems, edge-triggered interrupts are provided by the Real-Time Clocks and Interrupts Module (RCIM), if installed. Four edge-triggered interrupts are available on each RCIM.

0-3	RCIM edge-triggered interrupts

addr the virtual address at which the shared memory region is to be attached. If you wish the kernel to select the address, specify a value of zero. If not, specify a nonzero value. The address is automatically rounded to a SHMLBA boundary (SHMLBA is defined in **`<sys/shm.h>`**).

If no errors occur, the **`eti_map`** routine returns the virtual address that has been mapped to the interrupt controller's physical address space that controls the specified edge-triggered interrupt. Otherwise, a value of **-1** is returned, and **`errno`** is set to indicate the error.

Using the eti_request Routine

The **`eti_request(3C)`** routine enables you to perform a control operation for an edge-triggered interrupt pin that has been mapped onto a process's virtual address space. By using this routine, you can arm, disarm, enable, disable, and request a particular edge-triggered interrupt.

The specifications required for calling **`eti_request`** are as follows:

```
#include <sys/types.h>
#include <sys/pin.h>

void eti_request(func, addr)

u_int    func;
caddr_t  addr;
```

The arguments are defined as follows:

func a pin function as defined in the file <**`sys/pin.h`**>. This function may be one of the following:

PIN_ARM	allow interrupt to be recognized
PIN_DISARM	prevent interrupt from being recognized
PIN_ENABLE	allow interrupt to be received
PIN_DISABLE	prevent interrupt from being received
PIN_REQUEST	generate an interrupt on an enabled pin

addr the virtual address for controlling the edge-triggered interrupt, which has been obtained on a previous call to **`eti_map(3C)`**.

The **`eti_request(3C)`** routine does not return a value.

Using the eti_unmap Routine

The **`eti_unmap(3C)`** routine enables you to detach a shared memory region that has been attached to a process's virtual address space on a previous call to **`eti_map(3C)`** (see"Using the eti_map Routine" for an explanation of this routine). (Note that when a process exits or makes one of the **`exec(2)`** system calls, a shared memory region that is attached to its virtual address space is automatically detached.)

The specifications required for calling this routine are as follows:

```
#include <sys/types.h>
#include <sys/pin.h>

int eti_unmap(addr)

caddr_t addr;
```

The argument is defined as follows:

addr	the virtual address for controlling the edge-triggered interrupt, which has been returned on a previous **`eti_map(3C)`** call

A return value of **0** indicates that the **`eti_unmap`** call has been successful. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`eti_request(3C)`** system manual page for a listing of the types of errors that may occur.

Using the Distributed Interrupt Support Routines

There are a set of C library routines that are used to control distributed interrupts provided by the Real-Time Clocks and Interrupts Module (RCIM).

This is done by binding the physical address of the distributed interrupt control registers into the address space of a process and then directly performing the desired operation.

The routines provided are briefly describe:

distrib_intr_map	bind the physical address of a distributed interrupt into a process's virtual address space.
distrib_intr_request	perform a control operation for a distributed interrupt.
distrib_intr_unmap	detach a shared memory region that has been bound to a distributed interrupt with a **distrib_intr_map** call.

The man page **`distrib_intr_request(3c)`** provides a full description.

Distributed interrupts are similar to and handled in a similar fashion to edge-triggered interrupts. .See Section "Using the eti Support Routines" on page 8-28 for more information.

Using the vme_address Routine

The **vme_address(3C)** routine enables you to obtain a 32-bit physical address for an A16 or an A24 (H)VME address generated by a particular device. A 32-bit physical address is required when you use the **shmbind(2)** system call or the **shmconfig(1M)** command to bind a shared memory region to a section of physical (H)VME memory. (Procedures for using this system call and command are explained in the *PowerMAX OS Programming Guide*.) You may find this routine particularly helpful, for example, if you are writing a user-level device driver and wish to bind a shared memory region to the physical address of the (H)VME board.

The specifications required for calling this routine are as follows:

```
#include <sys/types.h>
#include <sys/pin.h>

u_int vme_address(addr, bus, type)

u_int addr;
int   bus;
int   type;
```

The arguments are defined as follows:

addr	a device's A16 or A24 address for which you wish to generate a 32-bit address
bus	an integer value that indicates the (H)VME bus to which the device is connected. A **0** denotes the primary bus; a **1** denotes the secondary bus.
type	an integer value that indicates whether the device's address specified by *addr* is an A16 or an A24 address. A **0** denotes an A16 address; a **1** denotes an A24 address.

If the call is successful, **vme_address** returns a 32-bit VME address. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **vme_address(3C)** system manual page for a listing of the types of errors that may occur.

9
Virtual Interrupt System

9
Virtual Interrupt System

Introduction

This chapter describes a facility for supporting asynchronous notification known as the *Virtual Interrupt System* (*VIS*), which provides a cohesive, symmetric subsystem for controlling software interrupts.

Intertask notification mechanisms can be controlled and coordinated through the Virtual Interrupt System (VIS), a generalized facility for supporting system-wide asynchronous notification.

The VIS provides a cohesive, symmetric subsystem for controlling software interrupts. Use the VIS to control asynchronous notification when performing new development under the PowerMAX OS. Although you need not impose the VIS on existing code, you may choose to do so in some cases to facilitate portability to future Concurrent platforms.

Understanding VIS Signals and Queued Signals

The *Virtual Interrupt System* (VIS) is a facility for the efficient use of signals and queued signals to provide asynchronous notification.

Signals are a software interrupt mechanism that notify one or more recipients of an event. A non-queued signal may not be reliable, i.e., if an instance of that particular signal is already pending for the recipient, the new instance is not sent. If sent, it may optionally have additional information that is delivered with the signal. Queued signals are reliable, i.e., even if other instances of the signal are pending, the new instance is sent. Queued signals can always be sent with additional information that is delivered with the signal. Queued signals conform to the POSIX 1003.1b standard.

VIS Overview

This section is an overview of the VIS interface and command-level VIS administration.

The Channel

A *virtual interrupt channel* (*channel*) is the object upon which software interrupts are sent (*sourced*) and received (*sensed*). This channel can be created by a user task, a device driver, or the kernel. (Currently, the PowerMAX OS kernel and device drivers do not create VIS channels.) Once created, the channel is identified by a unique handle and optionally by a unique name. The channel naming simplifies intertask access to the channel.

A channel is normally owned by the creating entity and hence exists for the life of its creator. It can, however, be adopted by the kernel, in which case it exists until it is explicitly removed or until the system is rebooted.

Sense and Source Connections

Entities can register to receive or send interrupts on a channel. Successfully registering to receive interrupts establishes a *sense connection* and results in the entity's joining the channel's *sense membership list.* The connection can subsequently be removed by either the connecting entity or by the channel owner. Otherwise, the connection exists for the life of the channel or the life of the connector, whichever is shorter.

The interrupt sensor also specifies (and provides) the notification mechanism, as well as the type of parameters to be received with that mechanism. The interrupt sensor accomplishes this by allocating and initializing an `ACTION` structure, which supplies all the information necessary for the kernel to generate a queued signal or a non queued signal on behalf of the sensor. Allocation and initialization of the `ACTION` structure may be performed with a library routine.

Successfully registering to send an interrupt establishes a *source connection*. A source connection causes an *input mapping* to the channel. (A single map of all source connections to all channels is maintained for each user task by VIS.) A mapping table of all source connections to channels is maintained for each user process by the VIS. This table and its usage is shared by all lightweight processes (LWPs) in a process. The (integer) *map index* that is returned by the source connection operation permits an efficient authentication of each subsequent request to post an interrupt.

A source connection can be removed by the sourcing entity as well as by the owner of the channel. Such removal is achieved by invalidating the mapping table slot that corresponds to the map index.

In addition to the per-process input map, a *source membership list* is maintained by the VIS for each channel. Each source connection causes a member to be added to this list, which serves as a reference for all source connections that target this channel.

VIS is designed to ensure graceful and coherent behavior under all conditions. The removal of channels or channel connections during usage is permitted without compromising system integrity. An application may, however, need to tolerate missed interrupts in this case.

At the time of its creation, a channel can be assigned attributes and permissions that regulate subsequent connections to it. In particular, the channel can be established for either *broadcast* sense connections, *event-monitoring* sense connections, or *timer* sense connections.

Event monitoring enables conditional receipt of interrupts, depending on the value of an *event word*. Each sense connection on the channel can register a particular logical test to be performed on the channel's event word. If the test evaluates true, the sense connection receives the interrupt.

Timer channels can only be created by the system or device driver, but allow arbitrary sense connections to exist on this type of channel. VIS timer channels allow efficient and hardware-independent support of time-based notification. (This feature is currently unused by PowerMAX OS.)

System Timer Channel

NOTE

System timer channels are currently not supported by PowerMAX OS; this may change in future releases.

The system timer channel is a kernel-established high-performance timing mechanism for the generation of one-shot and periodic system processing. The system timer channel supports VIS sense connections and POSIX 1003.4-conformant timers.

VIS timer channels internally maintain time for all associated sense connections via an aperiodic time base. An aperiodic time base reduces the interrupt load to the absolute minimum required for a given connection list while allowing time to be specified with a high degree of precision.

Connecting to a Timer Channel with vi_sense(2)

The **`vi_sense(2)`** function connects to a virtual interrupt channel as an interrupt receiver. **`vi_sense(2)`** can create a timer connection on a timer channel and define an initial pause and reload value.

VIS Calls and Routines

The two basic interfaces to VIS are:

- User
- Kernel/Driver

This manual describes only the user interface. See the device driver manual pages for information about the kernel/driver interface. The user task system calls and library routines are:

Table 9-1. VIS User Task System Calls and Routines

Call or Routine	Description
`vi_create(2)`	Creates a system-wide virtual interrupt channel for the sourcing and sensing of software interrupts.
`vi_delete(2)`	Deletes a system-wide virtual interrupt channel.
`vi_map(2)`	Establishes a map to a virtual interrupt channel for subsequent sending (or posting or sourcing) software interrupts.
`vi_mapsource(2)`	Sends (or post or sources) a software interrupt on a virtual interrupt channel.
`vi_unmap(2)`	Disconnects a map from a virtual interrupt channel.
`actsig(3)`	Initializes an `ACTION` structure as a signal specification.
`vi_sense(2)`	Connects to a virtual interrupt channel for receiving (or sensing) software interrupts.
`vi_nonsense(2)`	Disconnects an established sense connection from a virtual interrupt channel.
`vi_ctl(2)`	Performs control and housekeeping operations on a virtual interrupt channel. These operations include fetching and setting channel data (such as permissions, limitations, and statistics).

VIS Interface—Procedural Overview

This section presents a procedural overview of the VIS calls listed above.

Creating a Virtual Interrupt Channel

A user task creates a virtual interrupt channel by calling **`vi_create(2)`**. This establishes a system-wide virtual interrupt channel to which tasks and other system entities can subsequently connect. Pending creation, connections may be established upon a channel to sense or source interrupts.

The syntax for **`vi_create(2)`** is

```
#include <sys/vi.h>
int vi_create(name, maxcon, perm, [, evword])
unsigned char *name;
int maxcon;
unsigned int perm;
unsigned long evword;
```

where:

name if non-null, specifies a character string up to `MAXPATHLEN` characters in length, to be used as a system-wide tag to address the channel.

maxcon is the channel sense connection limit.

perm defines the following creation attributes and connection permissions:

`VIP_SO_ANY`
: Channel will allow unrestricted source connections.

`VIP_SO_PRIV`
: Caller is required to have *appropriate privilege* to connect as an interrupt source. Appropriate privilege requires either that the caller have an effective user ID of 0 (`root`) or that the current kernel allow users to assign real-time priorities to their tasks. The latter property is achieved by configuring a kernel with the `vis_privenb` parameter set to a value of 1, the PowerMAX OS default.

`VIP_SO_KERNEL`
: Connection for source is restricted to kernel.

`VIP_SE_ANY`
: Channel will allow unrestricted sense connections.

`VIP_SE_PRIV`
: Caller is required to have appropriate privilege to sense channel interrupts. See the description above of `VIP_SO_PRIV`.

`VIP_SE_KERNEL`
: Connection for sense is restricted to kernel.

`VIP_NODESPACE`
: Created channel is owned by the system and exists beyond the life of the creating task unless explicitly deleted. Normally an interrupt channel is removed when the creating task exits or executes. This operation requires the appropriate privilege. See the description above of `VIP_SO_PRIV`.

`VIP_DORMANT`
: Channel when created is dormant, which will pause or inhibit **`vi_map(2)`** attempts. A **`vi_ctl(2)`** call is required to activate the channel and allow source mappings to succeed. This mechanism allows sense connections to be established on a newly created channel before source interrupts occur; otherwise, these sense connections would effectively be lost.

`VIP_EVENT`
: *evword* is interpreted as the initial value to assign to the channel's event word; if `VIP_EVENT` is not specified, *evword* is ignored. Creation of a channel with `VIP_EVENT` is not required for event monitoring on the channel; it only indicates whether the channel's event word is to be assigned an initial value.

evword
: is an optional argument that specifies an initial value for the channel's event word. *evword* is interpreted only if `VIP_EVENT` is specified in *perm*.

Deleting a Virtual Interrupt Channel

A user task deletes a system-wide virtual interrupt channel by calling **`vi_delete(2)`**:

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

int vi_delete(chanid)
int chanid;
```

where *chanid*is removed after removing any connections previously established for source and sense, respectively. A return value of 0 indicates a successful removal; a return value of -1 indicates an error.

Establishing a Source Connection

A task that wants to source interrupts must establish an input mapping connection to the desired virtual interrupt channel by calling **`vi_map(2)`**:

```
#include <sys/vi.h>
int vi_map(chanid, flags, id)

int chanid;
unsigned int flags, id;
```

where:

chanid specifies the target virtual interrupt channel.

flags is used to define optional behavior for this call:

- If `flags & VIMAP_NOWAIT`, the call will not block and returns an error if *chanid* is currently dormant.
- If *flags* & `VIMAP_EVENT`, the source connection is established for event word access. This causes subsequent calls to **`vi_mapsource(2)`** to be interpreted as channel event word modifications rather than as broadcast interrupts.
- If *flags* & `VIMAP_ID` is `TRUE`, the field *id* is saved in the input map to facilitate identifying the map in subsequent **`vi_mapsource(2)`** calls. Visibility of the value is dependent on the `ACTION` convention chosen for the sense connection. See the **`action(5)`** man page for details.

The caller must meet the channel permission requirements established by **`vi_create(2)`**.

Input mappings are required to detect invalid channel handles (*chanid*)—and to authenticate valid ones—when they are used to post interrupts. This is accomplished by translating from a user-owned map structure to a virtual interrupt channel handle.

A user process has a private input mapping table used for **`vi_map(2)`** calls. This input mapping table is shared by all LWPs in a process. Upon the first **`vi_map(2)`** call, if an input mapping table has not been established for the caller, a table is allocated; by default, the table's size is `VIDEF_IM` mapping entries, as defined in `<sys/vi.h>`. As this table is fixed in size and will limit the number of outstanding input mapping connections, a task may allocate a table of sufficient size beforehand.

`vi_map(2)` returns an index into the caller's mapping table, which identifies the connection. This index is used by **`vi_mapsource(2)`** to source (post) an interrupt on a virtual interrupt channel. This index is common to all LWPs in a process and is analogous to file descriptors.

Sourcing an Interrupt

A task calls **`vi_mapsource(2)`** to source (post) an interrupt on, or modify the event word of, a virtual interrupt channel. The call takes the form:

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

int vi_mapsource(mapid, arg [, op, val])
int mapid;
int arg, op;
unsigned int val;
```

where:

mapid specifies a local input map to a virtual interrupt channel. If *mapid* is not established for event word access, *op* and *val* are ignored, and a broadcast interrupt is generated on *mapid*.

arg is offered to the connections sensing on the channel.

op is assumed to be a logical operation to be performed on *mapid*'s event word. The following table lists the operations defined for *op*:

op	**Operation**	**Result**
`EVM_ASSIGN`	*evword* = *val*	Assigns *val* to event word
`EVM_AND`	*evword* &= *val*	`AND` bits of *val* with event word
`EVM_OR`	*evword* |= *val*	`OR` bits of *val* with event word
`EVM_XOR`	*evword* ^= *val*	`XOR` bits of *val* with event word
`EVM_NAND`	*evword* = ~(*evword* & *val*)	`NAND` bits of *val* with event word
`EVM_NOR`	*evword* = ~(*evword* | *val*)	`NOR` bits of *val* with event word
`EVM_XNOR`	*evword* = ~(*evword* ^ *val*)	`XNOR` bits of *val* with event word

val is assumed to be a logical operation to be performed on *mapid*'s event word.

Removing a Source Channel

A task removes a channel input mapping by calling **`vi_unmap(2)`**:

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

int vi_unmap(mapid)
int mapid;
```

where *mapid* is the input mapping table entry to be disconnected from its virtual interrupt channel. The caller must meet the channel permission requirements established by **`vi_create(2)`**. **`vi_unmap(2)`** frees the map entry in the task's input mapping table claimed by **`vi_map(2)`**.

The action_t Structure

The interface for sensing an interrupt allows the choice of notification mechanism, specified with an `action_t` structure:

```
#include <sys/signal.h>
#include <sys/vi.h>

typedef union {
    int _pad[16];
    struct {
            int act_type;
            unsigned int act_flags;
            union {
                   struct {
                           int signum;
                           int arg;
                           } sig;
                   } act_spec;
            unsigned int (*act_init) ();
            union {
                    void *ptr;
                    int scal;
                    } act_arg;
            void *act_member;
            ushort act_limit;
            } act;
    struct sigevent sevt;
    } action_t;


#define  action_type act.act_type
#define  action_flags act.act_flags
#define  action_spec act.act_spec
#define  action_init act.act_init
#define  action_arg act.act_arg
#define  action_member act.act_member
#define  action_limit act.act_limit
```

The field `action_t.action_type` selects the mechanism from `ACT_SIG` and `ACT_QSIG`, which correspond to non-queued signals or queued signals.

The field `action_t.action_flags` defines optional behavior for the mechanism. `action_t.action_spec` is initialized by the caller before passing its address to the system as appropriate for the selected notification mechanism.

The field `action_t.action_init` and `action_t.action_arg` are reserved for use by the kernel; they should be ignored by the user.

The field `action_t.action_arg` is passed to a non-queued signal as the signal handler's argument. Queued signals will pass the argument used in a **`vi_mapsource(2)`** call to the signal handler.

Use of the `action_t` convention is facilitated by the library call **`actsig(3)`**. See the **`actsig(3)`** and **`action(5)`** man pages for details.

Allocating and Initializing an action_t Structure

The **`actsig(3)`** library call initializes an `action_t` structure as a signal specification

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

action_t *actsig(act, sig, flags, nqueue, func, arg)
action_t *act;
int sig;
unsigned int flags, nqueue;
void (*func) ();
unsigned int arg;
```

where:

act	if `NULL`, an `action_t` structure is allocated by **`actsig(3)`**; otherwise, *act* is assumed to be the address of a valid `action_t` structure.
sig	is the number of the signal to be depicted in the action.
nqueue	if 0 or 1, causes *act* to represent a conventional single recurrence signal (`ACT_SIG`). Otherwise, *nqueue* is taken to be the maximum number of signals that can be outstanding for *act* and the action is set to `ACT_QSIG`.
flags	If *flags* & `ACT_SIG_HAND` is specified, **`actsig(3)`** registers *func* to be a persistent user signal handler invoked when *sig* occurs. Additionally, *sig* is blocked during the duration of *func*. If *sig* is a member of the queued signals set, **`actsig(3)`** masks the set of `[`*sig*`...SIGRTMAX]` during *func* in order to enforce the delivery prioritization specified by POSIX 1003.1b. The priority masking may be overridden, if desired, by `flags & (ACT_SIG_NOPRI | ACT_SIG_HAND)`.

If `!(`*flags* `& ACT_SIG_HAND)`, the signal handler must be registered via one of **`signal(2)`**, **`sigset(2)`**, or **`sigaction(2)`** (preferred) if other than default behavior is desired for *`sig`*.

See the **`actsig(3)`** man page for details.

Establishing a Sense Connection

Assuming the `action_t` structure is initialized appropriately for the desired mechanism, a task passes its address by calling **`vi_sense(2)`**:

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

int vi_sense(chanid, pact, flags
[, arg1, arg2])
int chanid;
action_t *pact;
unsigned long flags;
unsigned int arg1, arg2;
```

where:

chanid is the virtual interrupt channel that **`vi_sense(2)`** attempts to connect to.

pact is an `action_t` structure specifying the asynchronous notification mechanism used to deliver the interrupt.

flags, *arg1*, and *arg2*
If `VISE_EVENT` is set in *flags*, **`vi_sense(2)`** creates an event monitor on *chanid* and interprets *arg1* and *arg2* as a logical operation and an event word mask, respectively. When an event is posted on *chanid*, the event word in that channel is masked against *arg2*, and the result is tested according to the operation specified by *arg1*. If the test results in `True`, an interrupt is received by the sense connection.

Upon successful completion, **`vi_sense(2)`** returns the connection ID unique for the selected channel.

Timer Connections

If *flags* and **`VISE_TIMER`** is specified and *chanid* is a multiplexed timer channel, **`vi_sense(2)`** will create a timer connection on the channel and interpret *arg1* as **`CLKSPEC *`***arg1*, which defines an initial pause and reload value:

```
typedef struct {
.
.
abstime_t cs_pause;/*delay to first connection expiration */
abstime_t cs_period;/*periodic reload value */
.
.
} clkspec_t;
```

The data structure **`abstime_t`** is intended to be wide enough to hold existing and foreseeable hardware-instituted representations of time as:

```
typedef union {
    unsigned long at_l;
    struct timestruc at_ts;
    struct uptime_clk at_utc;
    unsigned long at_longlong[2];
} abstime_t;
```

The actual data definition used on a particular timer channel is totally dependent on both the hardware and in-kernel driver servicing the channel.

Removing a Sense Connection

A task removes a sense connection by calling **`vi_nonsense(2)`**:

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

int vi_nonsense(chanid, conid)
int chanid;
int conid;
```

where:

chanid is the virtual interrupt channel

conid is the previously established channel connection.

`vi_nonsense(2)` attempts to remove the previously established channel connection *conid* from the virtual interrupt channel *chanid*. The caller must meet the channel permission requirements established by **`vi_create(2)`**.

VIS Control Operations

`vi_ctl(2)` provides user-level access for virtual interrupt channel administration:

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

int vi_ctl(cmd, arg1, arg2, arg3, arg4)
int cmd, arg1, arg2, arg3, arg4
```

where *`cmd`* specifies the desired functionality, currently defined as:

`VICTL_MAXCHAN`
Sets/gets the system-wide creation limit on virtual interrupt channels when *`arg1`* is zero/nonzero, respectively. When setting, *`arg2`* specifies the limit and the limit before modification is returned. Setting is a privileged operation.

`VICTL_MAXSECON`
Sets/gets the sense connection limit to be checked when creating a virtual interrupt channel when *`arg1`* is zero/nonzero, respectively. When setting, *`arg2`* specifies the limit and the limit before modification is returned. Setting is a privileged operation.

`VICTL_NCHAN`
Returns the number of system channels currently established.

VICTL_IMAPSZ
Creates/gets the calling task's input mapping table size when `arg1` is zero/nonzero, respectively. When creating, `arg2` specifies the size of the table. A table can be created only if one does not already exist.

VICTL_DELMYTASK
Deletes all virtual interrupt channels created by the caller.

VICTL_DELUSR
Deletes user-created virtual interrupt channels. The caller is required to have the appropriate privilege.

VICTL_DELALL
Deletes all nonpermanent user and system virtual interrupt channels. This is a privileged operation.

VICTL_MAXSENSE
Returns the sense connection limit for the `int` specified by *`arg1`* (the channel id).

VICTL_NSENSE
Returns the sense connection count for the `int` specified by *`arg1`* (the channel id).

VICTL_NMAP
Returns the source connection count for the `int` specified by *`arg1`* (the channel id).

VICTL_PERM
Returns the creation permissions for the `int` specified by *`arg1`* (the channel id).

VICTL_NAMETOID
Returns the virtual channel id `(int)` of the channel tagged as `unsigned char *arg1`, where `*arg1` is the name used to tag a virtual interrupt via **`vi_create(2)`**. If *`arg2`* is nonzero, **`vi_ctl(2)`** waits indefinitely for `*arg1` to exist. This forms a virtual interrupt channel intertask rendezvous mechanism keyed on a user-specified name.

VICTL_ACTIVATE
Activates/deactivates the virtual channel *`arg1`* when *`arg2`* is nonzero/zero, respectively. Activation is required when a channel has been created dormant. This mechanism will pause or inhibit **`vi_map(2)`** connections until all sense connections have been established.

VICTL_ARM
Sets the idle timer of the connection specified by the connection *`arg2`* to `clkspec_t *arg3` on the channel in *`arg1`*.

VICTL_IDLE
Disables the timer connection specified by *`arg2`* on the channel specified in *`arg1`*. The timer connection is not deleted by this operation.

VICTL_NEXTTIME

If *arg3* is not NULL, it is assumed to be a `clkspec_t*`*`arg3`*. If *arg4* is not NULL, it is assumed to be an `unsigned long *`*`arg2`*. For the timer connection whose connection id is in *arg2* on the channel *arg1,* return a snapshot of the timing parameters in `*`*`arg3`* and additionally the overrun count in `*`*`arg4`*.

Command-Level VIS Administration

NOTE

This command is not currently available on PowerMAX OS.

`vis(1)` provides a command-level interface for virtual interrupt channel administration. This facility lets you inspect and modify VIS resources: channels, channel creation and sense limits, channel activations, and system-wide VIS parameters.

The arguments to the **`vis(1)`** command belong to one of three categories: **`set`**, **`action`**, or **`options`**:

`vis` [**`set`**] [**`action`**] [**`options`**]

where:

`set` specifies resources to which the subsequent **`action`** arguments will be applied.

`action` specifies operations to be applied to the selected resources. Specifically:

- Channels can be activated and deactivated.
- Channel creation limits and channel sense limits can be modified.
- Channels can be deleted.
- Various resources can be examined.

`options` qualify the **`set`** and **`action`** flags in specific ways.

The following command prints out the number of channels that are active and owned by the system:

```
# vis -q -sys -act -v

VICHANID      OWNER     STATUS    SOURCE    SENSE    NAME
0xd0000020    system    active    1         0        DUARTCLOCK1
0xd000acec    system    active    1         0        DUARTCLOCK2
0xd000ad2c    system    active    1         0        DUARTCLOCK3
3 channels currently exist
```

In this command,

- `-sys` and `-act` are select flags.
- `-q` is an action flag requesting a one-line synopsis of each channel.
- `-v` is an option flag requesting verbosity.

The following command removes a dormant user-owned channel named `xyz`:

```
# vis -rm -usr -inact -nxyz
```

In this command:

- `-inact`, `-usr`, and `-nxyz` are select flags.
- `-rm` is an action flag requesting channel removal.

Used without arguments, **`vis(1)`** prints the current number of existing virtual interrupt channels. For example:

```
# vis
7 channels currently exist
```

See the **`vis(1)`** man page for more information.

10
Hardclock Interrupt Handling

10
Hardclock Interrupt Handling

This chapter describes the 60 Hz clock interrupt and its interrupt service routine. It describes the methods for controlling 60 Hz clock interrupt handling on particular CPUs in a system and the reasons for using them. An overview of the clock interrupt and the interrupt handler is provided in “Understanding Hardclock.” Procedures for controlling clock interrupt handling are explained in “Controlling Clock Interrupt Handling.” Resulting changes in the functionality of system calls, routines, commands, and utilities are described in “Understanding Functional Changes.”

Understanding Hardclock

The system-wide 60 Hz clock interrupts every CPU in the system 60 times per second. *Hardclock* refers to the operating system mechanism that services the 60 Hz clock interrupts on each processor.

Hardclock performs two sets of functions: those that are performed for the entire system and those that are performed for all currently running lightweight processes.

On the boot CPU only, hardclock carries out the following system-wide timing functions:

- Keeping the time of day
- Keeping the number of clock ticks since the last boot
- Keeping the number of seconds since 00:00:00 GMT (Greenwich mean time), January 1, 1970.
- Updating the time remaining for events in the local and global callout queues (The local callout queue contains functions that can run only on the local processor; the global callout queue contains functions that can run on any processor in the system.)
- Waking a daemon to run the expired events in the local callout queue
- Triggering a low–priority software interrupt to run the expired events in the global callout queue
- Keeping the amount of free memory

On all CPUs, hardclock performs the following local timing functions:

- Keeping process CPU usage statistics and sending the SIGXCPU signal
- Expiring the process virtual interval timer and sending the SIGVTALRM signal
- Expiring the process profiling timer and sending the SIGPROF signal

- Initiating process and kernel profiling functions
- Aging process rescheduling locks and sending the SIGRESCHED signal
- Keeping track of process resident memory usage
- Expiring the current quantum, which forces a context switch on interrupt completion. Quantum expiration is necessary for round–robin scheduling within a priority level.
- Calculating per process and per lightweight process user and system times

Mean and median times for **`hardclock()`** to service interrupts generated by the 60 Hz clock are nearly identical; however, worst case times can deviate from the mean by as much as 400 percent. The **`hardclock()`** routine's periodic and nondeterministic behavior can introduce unacceptable levels of jitter in applications with stringent timing requirements.

Some applications require the execution of a task within some period A_t. The task will execute for some duration A_x. Every period, there will be some slack time A_s, during which the task does not execute. This can be expressed as:

$$A_t = A_x + A_s$$

Like the task, the hardclock interrupt must also execute within some period, H_t, and for some duration H_x. If the hardclock interrupt and the tasks are to execute on the same CPU, the amount of CPU time available for execution of the task process will be reduced. If A_t is much smaller than H_t, there will be task periods where there is no loss in available CPU time and some where there is. If all tasks must complete in their allotted period, a task period must be long enough for execution of the task plus execution of the hardclock interrupt. This can be expressed as:

$$A_x + H_x < A_t = A_x + A_s$$

If the slack time in the task period, A_s, is larger than the execution time of the hardclock interrupt, H_x, all tasks have enough time to complete before the next task period. Otherwise, some tasks will not complete before the next task period, producing jitter in the application.

The purpose of **`hardclock(1M)`** is to disable local timing functions on CPUs where contention between the hardclock interrupt and a periodic real–time task is likely to occur. With $H_x \approx 0$, the second expression becomes:

$$A_x < A_t = A_x + A_s$$

This expression is always true where $A_s > 0$.

The jitter and overhead associated with servicing the 60 Hz clock interrupt can be reduced by disabling local timing functions on specified CPUs. Reducing the number of functions performed for a particular CPU will improve performance on that CPU.

An interface that allows you to specify dynamically which CPUs will service the clock interrupts is provided. If a CPU does not service the 60 Hz clock interrupt, hardclock functions for lightweight processes running on that CPU are not performed. This interface is described in "Controlling Local Timing Functions."

Controlling Clock Interrupt Handling

You can control clock interrupt handling in two ways: (1) by enabling a system daemon to perform system-wide timing functions on the boot CPU and (2) by disabling local timing functions on selected CPUs. “Controlling System-Wide Timing Functions” explains the procedures for using the system daemon. “Controlling Local Timing Functions” explains the procedures for disabling local timing functions on selected CPUs.

Controlling System-Wide Timing Functions

By default, hardclock system-wide timing functions are performed in an interrupt routine on the boot processor. You can configure your system so that the following system-wide timing functions are performed by a system daemon:

- Running the expired events in the global callout queue
- Keeping the amount of free memory

You can ensure that these functions are performed by a system daemon by setting the value of the `TODCINTRDAEMON` system tunable parameter to **`1`**.

On a multiprocessor system, you may also wish to ensure that the daemon runs on a particular processor. You can do so by setting the value of the `TODCINTR_ENGBIAS` system tunable parameter. The value of this parameter is a bit mask in which bits 0 through 7 correspond to processors 0 through 7. The default value for this parameter is -1, which denotes all available processors. Processors are selected by changing the value of the parameter to a hexadecimal value that sets the bit(s) corresponding to the desired processor(s) in the mask.

You can use the **`config(1M)`** utility to (1) determine whether the values of these parameters have been modified for your system, (2) change the values of these parameters, and (3) rebuild the kernel. Note that you must be the root user to change the value of a tunable parameter and rebuild the kernel. After rebuilding the kernel, you must then reboot your system.

It is important to note that the following system-wide timing functions are always performed at interrupt level:

- Keeping the time of day
- Keeping the number of clock ticks since the last boot
- Keeping the number of seconds since 00:00:00 GMT (Greenwich mean time), January 1, 1970.
- Updating the time remaining for events in the global callout queue
- On multiprocessor systems, updating the time remaining for events in the local callout queue
- On multiprocessor systems, waking a daemon to run the expired events in the local callout queue

Controlling Local Timing Functions

You can enable and disable local timing functions on all CPUs by using the **mpadvise(3C)** library routine or the **hardclock(1M)** command. Use of the **mpadvise** routine is explained in "Using the mpadvise Library Routine." Use of the **hardclock** command is explained in "Using the hardclock Command."

Using the mpadvise Library Routine

The **mpadvise(3C)** routine performs two functions pertaining to the 60 Hz clock interrupt. **Mpadvise** reads a CPU mask to determine which CPUs are currently handling the clock interrupt and writes a CPU mask to specify which CPUs are to service the clock interrupt.

The specifications required for using the **mpadvise** system call to handle the 60 Hz clock interrupt are as follows:

```
int mpadvise (cmd, which, who, mask)

int cmd, which, who;
cpuset_t *mask;
```

Arguments are defined as follows:

cmd the operation to be performed. To handle the 60 Hz clock interrupt, *cmd* must be one of the following:

MPA_CPU_GETHRDCLK	Return a mask indicating which CPUs are servicing the 60 Hz clock interrupt.
MPA_CPU_SETHRDCLK	Specify which CPUs are to service the 60 Hz clock interrupt. It is intended that this command be used in a real–time environment only. Services normally performed by the clock interrupt handler will not be performed on CPUs where clock interrupt handling is disabled. Services not performed are the local functions performed for a particular CPU that are described in "Understanding Hardclock." Note that the calling process must have the P_SYSOPS privilege to use this command.

which must be zero

who must be zero

mask A pointer to a location where a bit mask is returned if *cmd* is MPA_CPU_GETHRDCLK or a pointer to a bit mask that specifies which CPUs are to service the 60 Hz clock interrupt if *cmd* is MPA_CPU_SETHRDCLK. The system identifies CPUs by integers in the range 0 to 31. Collections of CPUs are specified by a 32–bit mask, where the value (1<<i) represents CPU i.

Upon successful completion of an MPA_CPU_GETHRDCLK **mpadvise** call, a mask indicating the CPUs that are handling the 60 Hz clock interrupt is returned to the location pointed to by *mask*. Upon successful completion of an MPA_CPU_GETHRDCLK or an MPA_CPU_SETHRDCLK **mpadvise** call, the return value of **mpadvise** is the number of bits set in *mask*. Attempts to enable clock interrupt handling for CPUs that are not active are silently ignored. If the call is not successful, the return value is **–1**, and **errno** is set to indicate the error. For additional information on the use of this routine, refer to the system manual page **mpadvise(3C)**.

Using the hardclock Command

The **hardclock(1M)** command allows you to perform two functions:

- Determine the CPUs that are servicing the 60 Hz clock interrupt.
- Specify the CPUs that are to service the 60 Hz clock interrupt.

The format for executing the **hardclock** command is as follows:

hardclock [*cpu–list*]

Cpu–list is an optional, comma–separated list of CPU IDs or CPU ID ranges (for example, **1,3–5,7**). *Cpu–list* can also be one of the constants **all**, **none**, or **boot**, where **all** represents all active CPUs, **none** represents no CPUs, and **boot** represents the boot CPU. Services normally performed by the **hardclock()** interrupt routine are not available on CPUs that have the handling of the 60 Hz clock interrupt disabled; these services are the functions performed for lightweight processes running on a particular CPU that are described in “Understanding Hardclock.” Note that you must have the P_SYSOPS privilege to specify *cpu–list*.

To determine the CPUs that are servicing the 60 Hz clock interrupt, use the **hardclock** command without specifying an argument.

The output will be similar to the following:

```
hardclock enabled:  0-3
```

To specify the CPUs on which handling of the 60 Hz clock interrupt is enabled, use the **hardclock** command, and provide *cpu–list*. The **hardclock** command enables clock interrupt handling for the specified CPUs and disables clock interrupt handling for the unspecified CPUs. It is intended that this command be used only in a real–time environment.

The **hardclock** command returns a CPU list specifying the CPUs on which clock interrupt handling was previously enabled and is currently enabled.

To enable clock interrupt handling on only the boot CPU, enter:

```
hardclock boot
```

The output will be similar to the following:

```
hardclock enabled:  old 0–3,  new 0
```

To enable clock interrupt handling on CPUs 0, 2, and 3, enter:

```
hardclock 0,2-3
```

The output will be similar to the following:

```
hardclock enabled:  old 1-3,  new 0,2-3
```

Understanding Functional Changes

Disabling **hardclock()** changes functionality in the PowerMAX OS process scheduler and in certain system calls, routines, commands, and utilities. The ways in which functionality is affected are described in the sections that follow.

The Process Scheduler

The time quantum associated with a process or an LWP is adjusted and expired by the local clock handler. If **hardclock()** is disabled on a CPU, the time that a process or an LWP spends executing on that CPU will not be accumulated. Consequently, the process's or LWP's quantum may expire late. The quantum of a process or LWP that runs only on a CPU on which **hardclock()** is disabled, will never expire; in this case, the process or LWP will run to completion unless it blocks or is preempted by a higher priority process.

Refer to the *PowerMAX OS Programming Guide* for complete information on the process scheduler; scheduler classes, POSIX scheduling policies, and scheduler priorities; and the program interfaces and commands that support process scheduling and management.

The Processor File System

Disabling **hardclock()** affects some of the values that are returned by the **/proc** file system. Selected fields in the **pstatus_t** and **lwpsinfo_t** structures are incorrect for processes and LWPs executing on a CPU on which clock interrupt handling is disabled. The fields that are affected in the **pstatus_t** structure are as follows: `pr_utime`, `pr_stime`, `pr_cutime`, and `pr_cstime`. The field that is affected in the **lwpsinfo_t** structure is `pr_time`.

System Calls, Routines, Commands, and Utilities

Disabling clock interrupt handling by using the **mpadvise(3C)** library routine or the **hardclock(1M)** command affects the system calls and routines that are presented in

Table 10-1. When **hardclock()** functions are disabled, the information presented in the table is not maintained.

Table 10-1. System Calls and Routines Affected by Disabling hardclock

System Calls	Effects

getrlimit(2)
setrlimit(2)
	When RLIMIT_CPU is set, **hardclock()** normally checks to see if the limit has been exceeded by the current process. If exceeded, **hardclock()** sends the SIGXCPU signal to the current process. While a process is executing on a CPU on which clock interrupt handling is disabled, time is not accumulated toward the CPU limit. The SIGXCPU signal may be delivered sometime after the limit has been exceeded, or it may not be delivered.

getitimer(2)
setitimer(2)
	The ITIMER_PROF and ITIMER_VIRTUAL timers are normally expired by **hardclock()**. For processes executing on CPUs on which clock interrupt handling is disabled, these timers expire late or do not expire, and the SIGPROF and SIGVTALRM signals are delivered late or are not delivered.

acct(2)	Accounting information for a process's physical memory usage is normally acquired in **hardclock()**. For processes executing on CPUs on which clock interrupt handling is disabled, the **ac_mem** information written to the accounting file is incorrect. Accounting information related to a process's CPU usage is normally acquired in **hardclock()**. For processes executing on CPUs on which clock interrupt handling is disabled, the **ac_utime** and **ac_stime** information written to the accounting file is incorrect.
profil(2)	The **profil(2)** system service works by sampling the application's program counter in the **hardclock()** interrupt routine. While a process is executing on a CPU on which **hardclock()** is disabled, no profiling will occur.
resched_cntl(2)	The RESCHED_SET_LIMIT command normally allows a process to specify the maximum length of time it expects to defer rescheduling. When this time limit is reached, the process is sent the SIGRESCHED signal. While a process is executing on a CPU on which **hardclock()** is disabled, time is not accumulated toward the rescheduling limit. Expiration of the set time limit and delivery of the SIGRESCHED signal will be late or will not occur.
times(2)	Fields in the **tms** structure, which are calculated from system and user process times, are incorrect. (This structure is defined in **<sys/times.h>**.)

Table 10-1. System Calls and Routines Affected by Disabling hardclock

System Calls	Effects
`_lwp_info(2)`	Fields in the **`lwpinfo_t`** structure, which are calculated from lightweight process times, are incorrect. These fields are: `lwp_utime` and `lwp_stime`. (This structure is defined in **`<sys/lwp.h>`**.)
`read(2)`	A **`read(2)`** operation on a file named /**`proc`**/*pid*/**`status`**, where *pid* represents any process in the system, provides the status for the process associated with *pid* and one of its LWPs. For a description of the effects of disabling **`hardclock()`**, refer to “The Processor File System.”

Any library routine or command that uses the system calls presented in Table 10-1 may produce unexpected results. In addition, commands and utilities that use the system’s timing facilities may not function correctly. Some of the library routines, commands, and utilities that may not function correctly are as follows:

`clock(3C)`	**`sh(1)`**	**`acct(1M)`**
`monitor(3C)`	**`time(1)`**	**`acctcms(1M)`**
`lastcomm(1)`	**`timex(1)`**	**`acctcon(1M)`**
`mpstat(1)`	**`top(1)`**	**`acctprc(1M)`**
`ps(1)`	**`truss(1)`**	**`prfpr(1M)`**
`prof(1)`	**`w(1)`**	**`sar(1M)`**
	`whodo(1)`	**`uucico(1M)`**
		`uuxqt(1M)`

For more information about these commands and utilities, refer to the corresponding system manual pages.

11
Disk I/O

11 Disk I/O

This chapter explains the procedures for performing direct disk I/O and the creation of contiguous files using **fallocate(3)** and the **contig(1M)** utility. This chapter also describes the virtual partition enhancement and provides an overview of the synchronized I/O interfaces that are based on IEEE Standard 1003.1b–1993. Real-time disk scheduling is discussed along with the disk I/O tunables.

NOTE

The synchronized I/O facility is an option available only on systems using the `_POSIX_SYNCHRONIZED_IO` option. “Configuring POSIX Synchronized I/O” (p. 11-7) contains the procedure to configure this option.

Direct Disk I/O

PowerMAX OS enables a user process to both read directly from--and write directly to--disk into its virtual address space, bypassing intermediate operating system buffering and increasing disk I/O speed. Direct disk I/O also reduces system overhead by eliminating copying of the transferred data.

To set up a disk file for direct I/O use the **open(2)** or **fcntl(2)** system call; for **xfs** disk files only, use **fadvise(3x)**. Use one of the following procedures:

- Invoke the **open** system call from a program; specify the path name of a disk file; and set the `O_DIRECT` bit in the *oflag* argument.
- For an open file, invoke the **fcntl** system call; specify an open file descriptor; specify the **F_SETFL** command, and set the `O_DIRECT` bit in the *arg* argument.
- For an open **xfs** file, invoke the **fadvise()** library routine and set the **NOREUSE** advisory. Enabling direct disk I/O ensures that subsequent **read(2)**, **readv(2)**, **aio_read(3)**, **pread(2)**, **write(2)**, **writev(2)**, **aio_write(3)**, and **pwrite(2)** calls using the same file descriptor bypass the buffer cache and transfer data directly to and from the user’s virtual address space.

Direct disk I/O transfers must meet all of the following requirements:

- The user buffer must be aligned on a byte boundary that is an integral multiple of the `_PC_REC_XFER_ALIGN` **pathconf(2)** variable.
- The current setting of the file pointer locates the offset in the file at which to start the next I/O operation. This setting must be an integral multiple of

the value returned for the `_PC_MIN_ALLOC_SIZE` **`pathconf(2)`** variable.

- The number of bytes transferred in an I/O operation must be an integral multiple of the value returned for the `_PC_MIN_ALLOC_SIZE` **`pathconf(2)`** variable.

CAUTION

For files under the **`ufs`** or **`sfs`** file system, a transfer byte count is allowed which is less than `_PC_MIN_ALLOC_SIZE`. This is not recommended as the operating system pads the remainder of the last sector with undefined padding data which affects write operations in the following ways:

- The padding data might overwrite existing data.
- The file pointer only advances to the end of the user-requested data written to the disk, so the padding data written does not extend the logical file size.

The maximum transfer size allowed for direct disk I/O defaults to the value of `MAXBIOSIZE` (defined in **`sys/param.h`**). However, for NCR controllers, the maximum transfer size can be raised by using the `NCRMAXTRNSFR` tunable.

The default and minimum values of `NCRMAXTRNSFR` are `MAXBIOSIZE` which is 0x20000 (128Kb). Other accepted values are 0x40000 (256Kb) and 0x80000 (512Kb). Any other values will be rounded down to one of these values with a minimum of `MAXBIOSIZE`.

PowerMAX OS supports direct I/O for regular files on the **`xfs`**, **`ufs`** and **`sfs`** file systems only. Note that **`fadvise(3X)`** is only available for files under the **`xfs`** file system. See “File Advisories” on page 11-4. Enabling direct I/O for files under other file systems or on nonregular files on one of the supported file systems returns an error. Trying to enable direct disk I/O on a file in a file system mounted with the file system-specific **`soft`** option also causes an error. The **`soft`** option specifies that the file system need not write data from cache to the physical disk until just before unmounting.

Although not recommended, you can open a file in both direct and cached (nondirect) modes simultaneously, at the cost of degrading the performance of both modes.

Using direct I/O does not ensure that a file can be recovered after a system failure. You must set the POSIX synchronized I/O flags to do so. For information on these flags, refer to “Using POSIX Synchronized I/O” (p. 11-8).

You cannot open a file in direct mode if a process currently maps any part of it with the **`mmap(2)`** system call. Similarly, a call to **`mmap()`** fails if the file descriptor used in the call is for a file opened in direct mode.

Whether direct I/O provides better I/O throughput for a task depends on the application:

- All direct I/O requests are synchronous, so I/O and processing by the application cannot overlap.

- Since the operating system cannot cache direct I/O, no read-ahead or write-behind algorithm improves throughput.

However, direct I/O always reduces system-wide overhead because data moves directly from user memory to the device with no other copying of the data. Savings in system overhead is especially pronounced when doing direct disk I/O between an embedded SCSI disk controller (a disk controller on the processor board) and local memory on the same processor board.

Contiguous Files

`xfs` is the only file system supported under PowerMAX OS that can create contiguous files. **`xfs`** allocates space in extents, where the size of the next extent is twice the size of the previous extent, up to a limit. This ensures that space remains as nearly contiguous as possible while minimizing wasted space. A contiguous file has all its disk space allocated as a single extent.

Creating Contiguous Files

Contiguous files can be created using **`fallocate(3X)`**, or by using the **`contig(1)`** utility. The **`fallocate(3X)`** call tries to allocate the specified amount of space in a single extent (if the offset is 0), thus producing a contiguous file. A successful return from **`fallocate`** ensures that subsequent attempts to write to the specified data do not fail due to lack of free space. You can add space to the file by writing beyond its end. **`xfs`** then tries to create a contiguous extent twice the size of the previous extent up to a limit. That limit is the larger of the first extent size and the configured extent size.

Setting the **FADV_SEQUENTIAL** advisory for the range ensures that a subsequent call to **`fallocate(3X)`** fails if the file system cannot allocate the entire amount as a single contiguous extent. See “File Advisories” on page 11-4.

Calling **`ffree(3X)`** clears the allocated space to 0, releasing extents when possible.

To create a contiguous file of 10 megabytes using **`contig(1)`**, you should enter:

```
contig -c <filename> 20480
```

The size in 512-byte blocks of the contiguous extent created will be reported. If the **`-c`** flag and the size argument are omitted, **`contig(1)`** can be used to report on whether or not an existing regular file is contiguous, and give the size in 512-byte blocks of the first extent in the file.

I/O With Contiguous Files

Use **`fallocate(3)`** to create contiguous files, which provide the following advantages:

- Write operations are faster, since they need not allocate space during the write.
- No indirect blocks are read, simplifying and speeding disk address mapping.
- Head movement for contiguous files is low, especially for sequential access, when there is no other disk activity.

File Advisories

File advisories use the **`fadvise(3X)`** call which is currently a part of a proposed POSIX draft standard. PowerMAX OS only supports the **`fadvise(3X)`** call for files on the **`xfs`** file system.

With advisories, an application advises the file system of its intended use of a file; based on the advisory information, the file system performs requests in the most efficient way. For example, if the application advises the file system that it shall access a data file sequentially, the file system pre-fetches the next sequential block of data to speed the operation.

This advisory approach enhances portability: an application's advisory need not change from one port to the next, even though the response to the advisory might differ substantially from one system to another, depending on the file system capabilities. For example, a file system that cannot prefetch ignores the sequential access advisory and still performs correctly, although more slowly, than a file system that does provide prefetching.

fadvise(3X)

To use the **`fadvise(3X)`** call, you must include the header file <**`sys/fs/xfs_space.h`**> in your source files. The call takes the following form:

```
int fadvise(
        int fd,
        off_t offset,
        size_t len,
        int advice
);
```

where:

fd is an open file descriptor. **`fadvise`** advises the file system on the expected use of the data in the file associated with *fd*.

offset specifies the start of the data to access.

len specifies the number of bytes of data to access. The range between *offset* and *len* need not currently exist in the file. If *len* is 0, specifies all data following *offset*.

advice indicates the expected behavior of the specified data, which can be:

`FADV_NORMAL`	The application has no advice on its use of the specified data. Note that `FADV_NORMAL` nullifies the effect of any previous **`fadvise()`** call on the specified range.
`FADV_SEQUENTIAL`	The application shall access the specified data sequentially, so the file system should do read-ahead caching.
`FADV_RANDOM`	The application shall access the specified data in a random order.
`FADV_WILLNEED`	The application shall need the specified data soon, so the file system should go ahead and cache it.
`FADV_DONTNEED`	The application shall not need the specified data anytime soon.
`FADV_NOREUSE`	The file system should not keep the data from the read or write cached after completing the request. When you specify this advisory and follow the rules for direct disk I/O, this advisory causes the file system to write data from user virtual address space directly to disk. See “Direct Disk I/O” on page 11-1.

Virtual Partition

PowerMAX OS provides an enhancement to mass storage called virtual partition. Virtual partition, **`vp(7)`**, is a pseudo disk device driver that combines multiple disk partitions into a single virtual partition. A configured virtual partition appears to the rest of the system as a single partition although it is actually made up of multiple member partitions.

A striped **`vp`** divides the contiguous data of the virtual partition into slices. The default slice size can be configured; it is 32K bytes by default. This size works best for environments with large, sequential disk accesses using a file system that has been created with a block size of 32K bytes.

Up to 16 partitions are combined into a single virtual partition by distributing slices across the member partitions in an alternating fashion. This is called *disk striping*. If a **`vp`** has two member partitions, all of the even-numbered slices are on the first partition, and all of the odd-numbered slices are on the second partition.

The advantages of striped virtual partitions are as follows:

- Disk striping for some applications can significantly increase performance by parallelizing large I/O operations across multiple disks.

  When multiple processes on different processors are writing to different partitions located on different physical disks, performance is improved because of the parallelism gained in performing the I/O operations.

- A single file system can be distributed across multiple disks, thereby increasing the maximum file system size (up to two gigabytes). Without virtual partition, the file system size is limited to the size of a single partition.

A disadvantage of striped virtual partitions is that the reliability of the file system as a whole is decreased because a single file system is distributed across multiple disks. (The probability of a file system's failing increases with the number of disks used by the file system.)

Your system may benefit from the use of virtual partitions if the file system is heavily used. By distributing a heavily-used file system across multiple disks, the system administrator can balance the use of disk drives in the system. If the heavily-used file system is accessed in single streams of large, sequential requests, the best performance will be obtained with the default configuration of a 32K slice size for the virtual partition and a 32K block size for the file system.

To improve performance, you can configure your system in several different ways. You can use the **`vpinit(1m)`** and **`vpstat(1m)`** utilities for this purpose. The **`vpinit`** utility is used to configure a **`vp`**. The **`vpstat`** utility displays information about a particular virtual partition. For an explanation of the procedure for configuring a virtual partition, refer to *System Administration Volume 2*.

Understanding POSIX Synchronized I/O

POSIX synchronized I/O provides the means for ensuring the integrity of an application's data and files. A synchronized output operation ensures that the data that are written to an output device are actually recorded by the device. A synchronized output operation is blocking—that is, the write function does not return control to the calling process until the write operation has been completed, or in the case of an asynchronous write operation, notification of I/O completion is not delivered to the calling process until the write operation has been completed. A synchronized input operation ensures that the data that are read from a device are an image of the data that currently reside on the device. Pending write requests that affect the data being read are completed prior to performing the read operation.

Synchronized I/O facilities allow you to specify that I/O operations performed on disk files are to be forced to the synchronized I/O completion state. You can specify whether the type of completion is to be synchronized I/O data integrity completion or synchronized I/O file integrity completion. The types of synchronized I/O completion are defined as follows:

<u>Synchronized I/O data integrity completion</u>

A synchronized I/O data integrity completion occurs for a read or a write operation when the operation has been successfully completed or the reason for failure has been diagnosed.

A read operation has been successfully completed only when an image of the data has been successfully transferred to the requesting process. If write requests affecting the data to be read are pending at the time that the synchronized read operation is requested, the data specified in the write requests are transferred to the disk prior to reading the data.

A write operation has been successfully completed only when the data specified in the write request and all file system information necessary to retrieve the data have been successfully transferred.

With either a read or a write operation, file attributes that are not necessary for data retrieval (for example, access time, modification time, and status change time) may not have been updated.

Synchronized I/O file integrity completion

A synchronized I/O file integrity completion occurs for a read or a write operation when the operation has been successfully completed or the reason for failure has been diagnosed.

A read operation has been successfully completed only when an image of the data has been successfully transferred to the requesting process and all file attributes related to the I/O operation have been successfully updated. If write requests affecting the data to be read are pending at the time that the synchronized read operation is requested, the data specified in the write requests are transferred to the disk prior to reading the data.

The write operation has been successfully completed only when the data specified in the write request have been successfully transferred and all file attributes related to the I/O operation have been updated.

NOTE

In the PowerMAX OS implementation of synchronized I/O, data integrity is functionally the same as file integrity.

Configuring POSIX Synchronized I/O

POSIX synchronized I/O must guarantee that data can be retrieved after a system crash. To guarantee that directory information is updated with current file information in all cases, the **`fsync(2)`** system call's flushing of directory data must be more thorough than is normally the case for the standard UNIX file system. This extra flushing causes performance of the **`fsync`** system call to be degraded. Consequently, extra flushing is performed only when `_POSIX_SYNCHRONIZED_IO` is configured in your system.

To enable the POSIX synchronized I/O facility, you must change the value of the `POSIX_SYNC_IO` tunable parameter from 0 to 1. You can modify the values of system tunable parameters by using the **`config(1M)`** utility. For an explanation of the procedures for using this utility, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*. Note that after changing a tunable parameter, you must rebuild the kernel and then reboot your system.

To determine whether or not the POSIX synchronized I/O facility is enabled in the currently executing kernel, you can use the **`sysconf(3C)`** routine. To determine whether or not POSIX synchronized I/O is supported for a particular file, you can use the

fpathconf(2) system call. For additional information on the use of **sysconf** and **fpathconf**, refer to the corresponding system manual pages.

Using POSIX Synchronized I/O

You can indicate that synchronized I/O is to be performed on a disk file by using one of the following functions: **open(2)**, **fcntl(2)**, **fdatasync(3C)**, or **fsync(2)**. The **open** and **fcntl** system calls allow you to ensure that every I/O operation that is performed on a specified file is forced to the synchronized I/O completion state. The **fdatasync** routine and the **fsync** system call allow you to ensure that every pending I/O operation that is associated with a file at the time of the call to **fdatasync** or **fsync** is forced to the synchronized I/O completion state.

Procedures for using **open(2)** and **fcntl(2)** are presented in “Using open and fcntl” (p. 11-8). Procedures for using **fdatasync(3C)** and **fsync(2)** are presented in “Using fdatasync” and “Using fsync,” respectively (p. 11-9).

An additional POSIX interface, **aio_fsync(3)**, allows you to perform asynchronous file synchronization. Procedures for using this routine are fully explained in Chapter 11 of this guide.

Using open and fcntl

Procedures for using the **open(2)** and the **fcntl(2)** system calls to indicate that synchronized I/O is to be performed on a disk file are as follows:

- Invoke the **open** system call from a program, specify the path name of a disk file, and set one or more of the following bits in the file status flag: **O_DSYNC**, **O_SYNC**, **O_RSYNC**.
- Invoke the **fcntl** system call from a program, specify the file descriptor for an open disk file, and specify the **F_SETFL** command. The **F_SETFL** command allows you to set the file status flags for an open file. For synchronized I/O, set one or more of the following: **O_DSYNC**, **O_SYNC**, **O_RSYNC**.

The synchronized I/O settings for the file status flags are defined as follows:

O_DSYNC	specifies that write operations are to be completed as defined for I/O data integrity completion.
O_SYNC	specifies that write operations are to be completed as defined for I/O file integrity completion.
O_RSYNC	specifies that read operations are to be completed with the same type of integrity as that specified by the `O_DSYNC` or the `O_SYNC` bit.

If both **O_RSYNC** and **O_DSYNC** are set, all I/O operations are to be completed as defined for I/O data integrity completion.

If both **O_RSYNC** and **O_SYNC** are set, all I/O operations are to be completed as defined for I/O file integrity completion.

Note that when you set the **O_RSYNC** bit, you must also set the **O_DSYNC** or the **O_SYNC** bit.

Note that using **`fcntl`** to indicate that synchronized I/O is to be performed on a disk file increases system overhead and causes performance of the **`fcntl`** system call to be degraded. It is recommended that you use the **`open`** system call instead of **`fcntl`** for this purpose when possible.

For additional information on use of the **`open`** and **`fcntl`** system calls, refer to the corresponding system manual pages.

Using fdatasync

The **`fdatasync(3C)`** interface is available if the system is configured with the `_POSIX_SYNCHRONIZED_IO` option. It allows the calling process to force all I/O operations associated with a particular disk file to the synchronized I/O completion state. All I/O operations are completed as defined for synchronized I/O data integrity completion (see "Understanding POSIX Synchronized I/O," p. 11-6, for an explanation of the meaning of this term). Only the I/O operations that are queued at the time of the call to **`fdatasync`** are certain to be forced to the completion state; subsequent operations associated with the file may not be.

The specifications required for making the **`fdatasync`** call are as follows:

```
#include <unistd.h>

int fdatasync(fildes)

int fildes;
```

The argument is defined as follows:

fildes	the file descriptor for the open file whose data are to be transferred to the storage device associated with the file.

A return value of **0** indicates that the call to **`fdatasync`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`fdatasync(3C)`** system manual page for a listing of the types of errors that may occur.

Using fsync

If the `_POSIX_SYNCHRONIZED_IO` option is enabled, the **`fsync(2)`** system call allows the calling process to force all I/O operations associated with a particular disk file to the synchronized I/O completion state. All I/O operations are completed as defined for synchronized I/O file integrity completion (see "Understanding POSIX Synchronized I/O" for an explanation of the meaning of this term). Only the I/O operations that are queued at the time of the call to **`fsync`** are certain to be forced to the completion state; subsequent operations associated with the file may not be. Both file data and directory information are guaranteed to be updated upon return from **`fsync`**.

If the `_POSIX_SYNCHRONIZED_IO` option is not enabled, the **`fsync`** function will flush data to the device as expected except when data are being written to a newly-created file. In this case, flushing of the directory entry for the file is not guaranteed upon return from **`fsync`**.

The specifications required for making the **`fsync`** call are as follows:

```
#include <unistd.h>

int fsync(fildes)

int fildes;
```

The argument is defined as follows:

fildes	the file descriptor for the open file whose data are to be transferred to the storage device associated with the file.

A return value of **0** indicates that the call to **`fsync`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`fsync(2)`** system manual page for a listing of the types of errors that may occur.

Real-Time Disk Scheduling

The disk scheduling algorithm used with NCR controllers is configurable. Four different algorithms are available, based on the state of two system tunables:

`DISK_PRIO_ORDER`	if set, disk requests are queued in process priority order, i.e., requests from higher priority processes are generally satisfied before those from lower priority processes. if not set, priority is not a factor in disk scheduling.
`DISK_FIFO_ORDER`	if set, disk requests are queued in first come, first served order (FCFS). if not set, disk requests are queued using a shortest seek time algorithm, `CSCAN`

With these tunables, 4 algorithms are available:

`CSCAN` (`DISK_PRIO_ORDER` set to zero, `DISK_FIFO_ORDER` set to zero)
All requests are scheduled using pure CSCAN algorithm. This is the default.

`FIFO` (`DISK_PRIO_ORDER` set to zero, `DISK_FIFO_ORDER` set to one)
All requests are scheduled using pure FCFS algorithm.

Priority ordering with `CSCAN` within priority band (`DISK_PRIO_ORDER` set to one, `DISK_FIFO_ORDER` set to zero)

Disk requests are scheduled in priority order with `CSCAN` used for requests at the same priority.

Priority ordering with FIFO within priority band (`DISK_PRIO_ORDER` set to one, `DISK_FIFO_ORDER` set to one)

Disk requests are scheduled in priority order with FCFS used for requests at the same priority.

For optimal real-time performance, one of the priority ordering algorithms should be selected, since these algorithms favor disk requests by higher priority processes. For best system-wide disk throughput, where real-time is less important, the `CSCAN` algorithm would generally be the best choice.

When priority ordering is enabled, a third tunable indicates which priority classes are affected:

`DISK_PRIO_ALL_CLASS`	if set and `DISK_PRIO_ORDER` is set, disk requests from <u>all</u> processes are priority ordered. Otherwise, only FP class requests are priority ordered (and are satisfied before other priority classes).

Miscellaneous Disk I/O Tunables

The following tunables apply only to NCR controllers:

`MCBS_PER_CONTROLLER`	specifies the number of Master Control Blocks (MCBs) per disk controller. This limits the total number of pending operations per controller. A higher number may help on disk intensive environments.
`MAX_IO_PER_DISK`	maximum number of queued I/O requests to a drive at one time. A higher number may help on disk intensive environments. However, as more requests are queued to a disk, the scheduling becomes less than optimal, because I/O requests can only be ordered prior to the handoff to the disk controller. This is especially true with priority disk scheduling.

12
Real-Time I/O

12
Real-Time I/O

This chapter describes PowerMAX OS asynchronous I/O facilities. These facilities allow a process to overlap CPU and I/O processing.

Asynchronous I/O is supported for:

- Regular disk files
- Special disk files
- Raw disk partitions

Asynchronous I/O is the most efficient means of performing large file transfers because I/O is performed directly to and from the user buffer without any buffering by the operating system.

Overview of Asynchronous I/O

The asynchronous I/O feature enhances performance by allowing applications to overlap processing with I/O operations. Using asynchronous I/O enables an application to have more CPU time available to perform other processing while the I/O is taking place.

With the asynchronous I/O capability, an application can submit an I/O request without waiting for its completion. It can perform other CPU work either until it is asynchronously notified of the completion of the I/O operation or until it wishes to poll for the completion. For applications that involve large amounts of I/O data, this CPU and I/O overlapping can offer significant improvement on throughput.

With asynchronous I/O, one process can have many I/O operations in progress while it is executing other code. In contrast, with synchronous I/O, the process will be blocked waiting for each I/O operation.

A **`read (2)`** system call is considered logically synchronous because the call cannot return to the user until the requested data are read into the specified buffer in the calling process's address space.

A **`write(2)`** system call is considered logically synchronous because the call does not return to the user until the requested data are written into the file system cache. After returning from the call, the user is free to reuse the buffers; however, the data are actually written to disk asynchronously at a later time. If the caller has set the **O_SYNC** flag on the call to **`open(2)`** or **`fcntl(2)`**, the call to **`write`** is truly synchronous and does not return to the user until the requested data are written out to disk. In contrast, a successful call to begin an asynchronous write queues a write request and returns immediately without waiting for the I/O to be completed. When returning from the call, the data are not copied from the user buffers, so the caller should not reuse the buffers until the I/O has completed.

An asynchronous read enables you to control the amount of read-ahead that is performed so that the data can already be available when needed by the application. Often the writing out of data can be done asynchronously although you may need to know when the I/O has completed. The notification mechanisms provided in PowerMAX OS fulfill this need.

Using Asynchronous I/O

PowerMAX OS provides support for threads-based asynchronous I/O and asynchronous I/O to raw disk partitions. Threads-based asynchronous I/O can be performed to regular files, device special files, and STREAMS-based files.

Access to both types of asynchronous I/O is provided by the POSIX asynchronous I/O interfaces, which are based on IEEE Standard 1003.1b–1993. These interfaces allow a process to perform asynchronous read and write operations, wait for completion of an asynchronous I/O operation, and cancel a pending asynchronous I/O operation. They provide the following additional features:

- Use of an asynchronous I/O control block, which specifies the parameters for an asynchronous I/O operation and, as the operation progresses, contains information about its status
- The ability to start an asynchronous read or write operation at a user–specified offset in a file
- The ability to initiate multiple asynchronous I/O operations with a single call
- The ability to specify which signal is used to notify a process of completion of an asynchronous I/O operation
- Support for asynchronous file synchronization

The POSIX asynchronous I/O interfaces are included in the threads library, `libthread`. They are briefly described as follows:

`aio_read`	perform an asynchronous read operation
`aio_write`	perform an asynchronous write operation
`lio_listio`	perform a list of asynchronous I/O operations
`aio_error`	obtain the error status of an asynchronous I/O operation
`aio_return`	obtain the return status of an asynchronous I/O operation
`aio_cancel`	cancel an asynchronous I/O operation
`aio_suspend`	wait for completion of an asynchronous I/O operation
`aio_fsync`	perform asynchronous file synchronization

The following asynchronous I/O interfaces are provided for use on files whose size exceeds (or may exceed) 2GB:

`aio_read64`	**`aio_return64`**
`aio_write64`	**`aio_cancel64`**
`lio_listio64`	**`aio_suspend64`**
`aio_error64`	**`aio_fsync64`**

These interfaces have the same functionality as the POSIX interfaces but the asynchronous I/O control block used as an argument to these functions should be declared as type `struct aiocb64`. The `aiocb64` structure is defined in **`aio.h`** and differs from `aiocb` only in that the *aio_offset* member is declared with type `off64_t` instead of `off_t`.

To support asynchronous I/O to raw disk partitions, PowerMAX OS provides two additional routines: **`aio_alignment(3)`** and **`aio_memlock(3)`**. The **`aio-_alignment`** routine allows the calling process to obtain the buffer alignment requirement for asynchronous I/O to raw disk partitions. The **`aio_memlock`** routine allows the calling process to lock in memory the portion of its virtual address space that is to be used for asynchronous I/O to raw disk partitions.

Table 12-1 indicates whether a particular asynchronous I/O interface is used for threads-based asynchronous I/0, asynchronous I/O to raw disk partitions, or both.

Table 12-1. Interfaces Supporting Asynchronous I/O

	Type of Asynchronous I/O	
Asynchronous I/O Interfaces	Threads-Based	Raw Disk
`aio_read(3)`	x	x
`aio_write(3)`	x	x
`lio_listio(3)`	x	x
`aio_error(3)`	x	x
`aio_return(3)`	x	x
`aio_cancel(3)`	x	
`aio_suspend(3)`	x	
`aio_fsync(3)`	x	x
`aio_alignment(3)`		x
`aio_memlock(3)`		x

The sections that follow describe the asynchronous I/O control block, threads-based asynchronous I/O, and asynchronous I/O to raw disk partitions. No reference to the large file interfaces is made in the sections that follow because each function behaves just like the corresponding standard POSIX asynchronous I/O function.

The Asynchronous I/O Control Block

The asynchronous I/O control block structure **aiocb** specifies the parameters for an asynchronous I/O operation. You supply a pointer to an **aiocb** structure when you invoke the POSIX asynchronous I/O routines that are described in "Using the POSIX Asynchronous I/O Interfaces."

The **aiocb** structure is defined in <**aio.h**> as follows:

```
typedef volatile struct aiocb aiocb_t;
struct aiocb {
    int                  aio_fildes;
    volatile void*       aio_buf;
    size_t               aio_nbytes;
    off_t                aio_offset;
    int                  aio_reqprio;
    struct sigevent      aio_sigevent;
    int                  aio_lio_opcode;
    ssize_t              aio__return;
    int                  aio__error;
    int                  aio_flags;
    void                 *aio__next;
    int                  aio_pad[1];
} aiocb_t;
```

The fields in the structure are described as follows.

aio_fildes	the file descriptor for the file from which data are to be read or to which data are to be written
aio_buf	the virtual address of the first byte of the I/O buffer into which data are to be read or from which data are to be written
aio_nbytes	the number of bytes to be read or written
aio_offset	the byte offset in the file where the asynchronous read or write operation is to begin
aio_reqprio	must be zero
aio_sigevent	specifies the way in which a process is to be notified of the completion of an asynchronous I/O operation. The value of **aio_sigevent.sigev_notify** must be **SIGEV_NONE**, **SIGEV_SIGNAL**, or **SIGEV_CALLBACK**.

SIGEV_NONE specifies that no notification is to be delivered upon completion of an asynchronous I/O operation. In this case, the error status and the return status for the operation are set as appropriate. A process can poll for completion of the operation by using the **aio_error(3)** and **aio_return(3)** library routines (see "The aio_error and aio_return Routines," p. 12-16 , for explanations of these library routines).

SIGEV_SIGNAL specifies that a signal is to be sent to the process upon completion of an asynchronous I/O operation. In this case, the `aio_sigevent.sigev_signal` component must specify the number of the signal that is to be sent to the process when the I/O operation is completed, and the `aio_sigevent.sigev_value` component must specify an application-defined value that is to be passed to the routine declared as the signal handlerl. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file <`signal.h`>. The application-defined value may be a pointer or an integer value. If the process catching the signal has invoked the `sigaction(2)` system call with the **SA_SIGINFO** flag set before the signal is generated, the signal and the application-defined value are queued to the process when the I/O operation is completed. If the value of `aio_sigevent.sigev_signo` is zero (the null signal), a signal is not sent to the process upon completion of the asynchronous I/O operation. For complete information on signal management facilities, the `sigevent` structure, and support for specification of an application-defined value, refer to the *PowerMAX OS Programming Guide*.

SIGEV_CALLBACK specifies that an application-defined call-back routine is to be called asynchronously upon completion of an asynchronous I/O operation. In this case, the `aio_sigevent.sigev_func` component specifies the address of the call-back routine, and the `aio_sigevent.sigev_value` component specifies an application-defined value that is to be passed as an argument to that routine.

CAUTION

The user-specified call-back routine must return rather then exit when processing is complete. The asynchronous I/O library invokes clean-up procedures after the call-back routine has completed and returned. Failure to return means that internal structures used by asynchronous I/O will not be freed.

`aio_lio_opcode` only on a call to the `lio_listio(3)` library routine, specifies the asynchronous I/O operation to be performed. The values that may be specified are as follows:

LIO_READ	indicates that an asynchronous read operation is to be performed
LIO_WRITE	indicates that an asynchronous write operation is to be performed

LIO_NOP	indicates that no operation is to be performed

For an explanation of the **lio_listio** library routine and more detailed information about the use of this field, refer to "The lio_listio Routine."

aio_flags	the flags that are associated with the asynchronous I/O request. Flag values are defined in <**aio.h**>. The **AIO_RAW** flag indicates that I/O is to be performed to a raw disk partition (for information on asynchronous I/O to raw disk partitions, see p. 12-7). The routines for which this flag may be set are as follows: **aio_read**, **aio_write**, and **lio_listio** (for information on the use of these routines, see "Using the POSIX Asynchronous I/O Interfaces").

The following fields in the **aiocb** structure are not intended to be used directly by a process. A process should obtain the information in the **aio__error** and the **aio__return** fields by using the **aio_error(3)** and the **aio_return(3)** library routines and supplying a pointer to the **aiocb** structure associated with the operation. The information that is obtained by using the library routines is fully described in "The aio_error and aio_return Routines."

aio__return	contains the return status of an asynchronous I/O operation
aio__error	contains the error status of an asynchronous I/O operation
aio__next	a pointer to a list of **aiocb** structures that the library uses
aio_pad[1]	is reserved for future use

Threads-Based Asynchronous I/O

This section describes how threads-based asynchronous I/O operates on regular files, device special files, and STREAMS-based files. Threads-based asynchronous I/O is not limited to specific file system types. Any device that supports **read** and **write** will accept asynchronous read and write requests.

The **aio_read** and **aio_write** interface routines correspond to the **read** and **write** system calls to support asynchronous read and write operations.

CAUTION

It is important that multiple threads performing I/O to the same file cooperate because the order of operations is nondeterministic. Even one thread should be careful not to mix synchronous I/O requests with asynchronous I/O requests because the order of operations is implementation-dependent.

The asynchronous read and write interfaces allow you to specify the file offset indicating where the I/O should begin. It is possible to specify absolute file offsets for each I/O request, and this type of access is defined to be random. It is also possible to indicate that the I/O should begin at the current file offset. Sequential access is defined to be multiple I/O requests from a single thread of control to the same file descriptor indicating that I/O should begin at the current file offset. See “The Asynchronous I/O Control Block” (p. 12-4) for more information.

Determining the value of the file pointer when asynchronous I/O operations are in progress is difficult. Considering that there may be multiple outstanding asynchronous I/O operations, each of which may update the file pointer at any time, things may become complex for an application. An application should be careful not to mix calls to **`read`** and **`write`** or **`lseek`**, for example, with asynchronous I/O operations. If sequential I/O is not requested, the order of I/O operations is indeterminate.

An application can request cancellation of one or more asynchronous I/O requests that the application has previously queued. The **`aio_cancel`** interface enables an application to cancel one or more outstanding asynchronous I/O operations. Those that have not been started are cancelled, but those in progress may or may not be cancelled.

An application can wait for asynchronous I/O completion. If an application has completed all its other work, it may relinquish control of the CPU until some of its outstanding asynchronous I/O requests have completed.

An application can obtain completion status information by calling **`aio_error`** and **`aio_return`**.

NOTE

The completion of an asynchronous write on any device has the same semantics as a synchronous write to that device; for example, completion of a write to a STREAMS-based device means that the data have been copied into the stream's queue and does not imply that the data have been written to a device.

When an asynchronous I/O request is made, the application can also request that it be notified when the I/O completes. This lets the application know immediately that the I/O has completed rather than having the application poll for completion status.

Asynchronous I/O may be used by single-threaded or multithreaded applications. In multithreaded applications, multiple threads within a process share the same address space and, therefore, have access to the asynchronous I/O control blocks of any other thread within that process. One thread, for example, may begin an asynchronous read on an open file, and another thread within that process may use the **`aio_suspend`** interface to wait for that read operation to be completed. If this occurs, the application must make sure that the cooperating threads are synchronized.

Asynchronous I/O to Raw Disk Partitions

Asynchronous I/O to raw disk partitions allows a user process to transfer data directly between its I/O buffers and disk, thereby bypassing intermediate operating system buffering. The benefits are higher performance and throughput.

To perform asynchronous I/O to a raw disk partition, you must meet the following constraints:

- The user I/O buffer must be properly aligned. The **aio_alignment(3)** routine allows you to obtain the buffer alignment requirement for your system (see p. 12-8 for an explanation of this routine and a code fragment that shows how to allocate an I/O buffer on the proper alignment).
- The size of the user I/O buffer must be a multiple of the system's logical disk sector size. The system's logical disk sector size, which is represented by the symbolic constant **NBPSCTR**, is defined in <**sys/param.h**>.
- The user I/O buffer must be locked in memory before performing an asynchronous I/O operation. The **aio_memlock(3)** routine is provided for this purpose (see p. 12-9 for an explanation of this routine).
- The `aio_flags` field of the `aiocb` structure supplied on a call to **aio_read**, **aio_write**, or **lio_listio** must be set to **AIO_RAW** (see p. 12-4 for an explanation of the `aiocb` structure). The upper limit on the size of a single raw asynchronous I/O transfer depends on both the virtual to physical translation of the I/O buffer and on the disk controller that controls the disk on which the file is stored. The reason is that a raw asynchronous I/O request is performed as a single disk transfer.

  The maximum number of physical memory fragments that can be handled on a single disk transfer is 32. Therefore, if the buffer specified for a raw asynchronous I/O request contains more than 32 physical fragments, the I/O request will return an error. To meet the restriction of 32 physical fragments in a single raw asynchronous I/O transfer, you have two options: (1) specify a buffer size that is less that 32 pages (131072 bytes), or (2) ensure that the physical memory to which the buffer is mapped is contiguous.

  You can create a physically contiguous buffer by first defining a reserved section of physical memory and then creating a shared memory segment and binding it to the reserved section of physical memory (for an explanation of the procedures, refer to the section entitled "Binding a Shared Memory Segment to Physical Memory" in the "Interprocess Communication" chapter of the *PowerMAX OS Programming Guide*).

  Furthermore, the maximum transfer size for disks on an ISE (Integral SCSI Ethernet) controller can never exceed 16 megabytes. The maximum transfer size for disks on a VIA (VME Interface Adapter) or an HSA (HVME SCSI Adapter) can never exceed 2 megabytes (for additional information on these adapters, refer to the **via(7)** and **hsa(7)** system manual pages).

The sections that follow explain the procedures for using the **aio_alignment** routine and the **aio_memlock** routine.

The aio_alignment Routine

The **aio_alignment(3)** routine allows the calling process to obtain the buffer alignment requirement for performing asynchronous I/O to raw disk partitions on the system.

The specification required for making the **`aio_alignment`** call is as follows:

```
int aio_alignment();
```

The return value is the buffer alignment requirement for your system.

You may use the following algorithm to allocate an I/O buffer of *size* bytes on the proper alignment for performing asynchronous I/O to raw disk partitions:

```
/* obtain correct alignment */
align = aio_alignment();

/* allocate enough space to adjust start of buffer */
addr = malloc(size + align);

/* addr is set to start of aligned buffer */
addr = ((int)addr + (align - 1)) & ~(align - 1);
```

The aio_memlock Routine

The **`aio_memlock(3)`** routine allows the calling process to lock in memory the pages within its virtual address space that are to be used for asynchronous I/O operations to raw disk partitions. Mappings for the pages may be private, writable mappings to files or any mapping to an unnamed memory object (for complete information on memory-mapping facilities, refer to the *PowerMAX OS Programming Guide*).

CAUTION

A process may invoke **`aio_memlock`** only once in its lifetime. After a specified range of the process's virtual pages has been locked in memory, it cannot be unlocked or changed. The range of pages will remain resident until the process exits. Invoking **`aio_memlock`** does not require special privilege.

The specifications required for making the **`aio_memlock`** call are as follows:

```
#include <aio.h>

int aio_memlock(avaddr, asize)

void     *avaddr;
size_t    asize;
```

The arguments are defined as follows:

avaddr	the starting address of the range of virtual address space that is to be locked in memory
asize	the length in bytes of the range of virtual address space that is to be locked in memory

The value of this argument depends on the number of asynchronous I/O requests that will be outstanding at one time. If the value of *asize* is too large, it may adversely affect system performance.

A return value of **0** indicates that the range of pages between *avaddr* and *avaddr* + *asize* **-1** is locked in memory. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **aio_memlock(3)** system manual page for a listing of the errors that may occur.

Using the POSIX Asynchronous I/O Interfaces

The sections that follow explain the procedures for using each of the POSIX asynchronous I/O interfaces. Note that to use these interfaces, you must link your application with the threads library, **libthread**.

Asynchronous I/O operations may be affected by the **fork(2)**, **forkall(2)**, **exit(2)**, **exec(2)**, and **close(2)** system calls. The ways in which they may be affected are described as follows:

fork and **forkall** — No asynchronous I/O is inherited by the child process after the **fork** and **forkall** system calls. Asynchronous I/O operations that occur after a **fork** or **forkall** call do not affect the copy of the **aiocb** control block in the child's address space, and the child does not receive any notification from the completion of the parent's I/O. If, for example, the parent process does a **forkall** while an I/O operation is in progress, the I/O completion will not be delivered to the child process.

exit and **exec** — The **exit** function will wait for all outstanding asynchronous I/O operations to complete before returning. It will unlock the area of memory locked by the **aio_memlock(3)** call.

The **exec** function will fail if there are any outstanding asynchronous I/O operations. Before calling **exec**, you must ensure that no requests are outstanding by waiting for the operations to be completed or by cancelling them.

The **sbrk(2)**, **brk(2)** and **shmdt(2)** functions will return EBUSY if the areas of locked memory are within the memory locked by **aio_memlock(3)**.

close — When using the **AIO_RAW** flag on raw disk partitions, **close** will block until all outstanding asynchronous I/O operations are completed. When the call returns, the application is free to reuse the control block and buffers.

If the **AIO_RAW** flag is not used, the **close(***fd***)** function will cancel all outstanding requests to *fd.*

The aio_read Routine

The **`aio_read(3)`** library routine allows the calling process to perform an asynchronous read operation. The call returns after the asynchronous read operation has been queued. If an error occurs while the operation is being queued, **`aio_read`** returns without queueing or initiating the operation.

The specifications required for making the **`aio_read`** call are as follows:

```
#include <aio.h>

int aio_read(aiocbp)

struct aiocb *aiocbp;
```

The argument is defined as follows:

aiocbp A pointer to an asynchronous I/O control block structure that specifies the parameters for the asynchronous read operation. Each of the fields in this structure is fully described in “The Asynchronous I/O Control Block” (see p. 12-4).

The control block to which *aiocbp* points should not be used by simultaneous asynchronous I/O operations.

After the asynchronous read operation has been queued (that is, **`aio_read`** has successfully returned), the value of the file offset cannot be determined. When the read operation has been successfully completed, the value of the file offset will be at the end of the file if the O_APPEND bit is set in the file status flag. Otherwise, on devices capable of seeking, the file offset will be equal to the offset specified by *aiocbp–>aio_offset* plus the number of bytes read. The file offset is undefined for devices incapable of seeking. If your application needs to issue multiple I/O requests, you must use append mode (by setting the O_APPEND bit on a call to **`open(2)`** or **`fcntl(2)`**), or you must carefully manage the offset specified for each asynchronous I/O operation in the *aiocbp–>aio_offset* field.

As the asynchronous read operation progresses, information about its status is placed in the *aiocbp–>aio_error* field. A process can obtain the contents of this field by invoking the **`aio_error(3)`** routine and specifying a pointer to this **`aiocb`** structure (see “The aio_error and aio_return Routines” for an explanation of this routine).

If the **O_RSYNC** and the **O_DSYNC** or the **O_RSYNC** and the **O_SYNC** bits have been set in the file status flag for the file specified by *aiocbp->aio_fildes*, completion of the **`aio_read`** operation occurs when an image of the data has been successfully transferred to the requesting process and all file system information has been updated. (For additional information on the use of these bits, refer to Chapter 10 of this guide and to the **`open(2)`** and **`fcntl(2)`** system manual pages.)

When the asynchronous read operation has been successfully completed, the number of bytes read is placed in the *aiocbp–>aio_return* field. A process can obtain the contents of this field by invoking the **aio_return(3)** routine and specifying a pointer to this **aiocb** structure (see "The aio_error and aio_return Routines" for an explanation of this routine).

CAUTION

The memory referenced by *aiocbp* or the buffer pointed to by *aiocbp–>aio_buf* must remain a valid part of the application's address space until the asynchronous I/O operation has been completed, or results will be undefined.

A return value of **0** indicates that the asynchronous read operation has been successfully queued. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **aio_read(3)** system manual page for a list of errors that may occur.

After the asynchronous read operation has been successfully queued, a process may cancel the operation, or an error may occur. A process can cancel an asynchronous I/O operation with a call to the **aio_cancel(3)** library routine (see "The aio_cancel Routine" for an explanation of this routine). If the process cancels the operation, the *aiocbp–>aio_error* field contains ECANCELED. If an error occurs, the *aiocbp–>aio_error* field contains one of the error codes that may be returned by the **read(2)** system call (see the corresponding system manual page for a listing of the types of errors that may occur).

The aio_write Routine

The **aio_write(3)** library routine allows the calling process to perform an asynchronous write operation.

The specifications required for making the **aio_write** call are as follows:

```
#include <aio.h>

int aio_write(aiocbp)

struct aiocb *aiocbp;
```

The argument is defined as follows:

aiocbp a pointer to an asynchronous I/O control block structure that specifies the parameters for the asynchronous write operation. Each of the fields in this structure is fully described in "The Asynchronous I/O Control Block" (see p. 12-4).

If the O_APPEND bit has been cleared in the file status flag for the file specified by *aiocbp–>aio_fildes*, the write operation begins at the position in the file specified by *aiocbp–>aio_offset*. If the O_APPEND bit has been set, the write operation appends the data to the end of the file.

After the asynchronous write operation has been queued (that is, **`aio_write`** has successfully returned), the value of the file offset cannot be determined. When the write operation has been successfully completed, the value of the file offset will be at the end of the file if the O_APPEND bit is set in the file status flag. Otherwise, on devices that are capable of seeking, the file offset will be equal to the offset specified by *aiocbp–>aio_offset* plus the number of bytes written. The file offset is undefined for devices incapable of seeking. If your application needs to issue multiple asynchronous I/O requests, you must either use append mode (by setting the O_APPEND bit on a call to **`open(2)`** or **`fcntl(2)`**) or carefully manage the offset specified for each asynchronous I/O operation in the *aiocbp–>aio_offset* field.

As the asynchronous write operation progresses, information about its status is placed in the *aiocbp–>aio_error* field. A process can obtain the contents of this field by invoking the **`aio_error(3)`** routine and specifying a pointer to this **`aiocb`** structure (see "The aio_error and aio_return Routines" for an explanation of this routine).

If the **O_DSYNC** or the **O_SYNC** bit has been set in the file status flag for the file specified by *aiocbp->aio_fildes*, completion of the **`aio_write`** operation occurs when the data specified on the call have been successfully transferred and all file system information has been updated. (For additional information on the use of these bits, refer to Chapter 10 of this guide and to the **`open(2)`** and **`fcntl(2)`** system manual pages.)

When the asynchronous write operation has been successfully completed, the number of bytes written is placed in the *aiocbp–>aio_return* field. A process can obtain the contents of this field by invoking the **`aio_return(3)`** routine and specifying a pointer to this **`aiocb`** structure.

CAUTION

The memory referenced by *aiocbp* or the buffer pointed to by *aiocbp–>aio_buf* must remain a valid part of the application's address space until the asynchronous I/O operation has been completed, or results will be undefined.

A return value of **0** indicates that the asynchronous write operation has been successfully queued. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`aio_write(3)`** system manual page for a listing of the types of errors that may occur.

After the asynchronous write operation has been successfully queued, a process may cancel the operation, or an error may occur. A process can cancel an asynchronous I/O operation with a call to the **`aio_cancel(3)`** library routine (see "The aio_cancel Routine" for an explanation of this routine). If the process cancels the operation, the *aiocbp–>aio_error* field contains ECANCELED. If an error occurs, the *aiocbp–>aio_error* field contains one of the error codes that may be returned by the **`write(2)`** system call (see the corresponding system manual page for a listing of the types of errors that may occur).

The lio_listio Routine

The **lio_listio(3)** library routine allows the calling process to initiate a list of asynchronous I/O operations with a single call. A process can obtain the maximum number of operations that can be included in a list by using the **sysconf(3C)** library routine and specifying **SC_AIO_LISTIO_MAX** as the *name*.

The specifications required for making the **lio_listio** call are as follows:

```
#include <aio.h>

int lio_listio(mode, aiocbp, count, signal)

int              mode;
struct aiocb   **aiocbp;
int              count;
struct sigevent *signal;
```

The arguments are defined as follows:

mode an integer value that indicates whether the routine is to return when the requested I/O operations have been queued or when they have been completed. This value may be one of the following:

LIO_WAIT	return after all of the requested I/O operations have been completed
LIO_NOWAIT	return as soon as the requested I/O operations have been queued. The *signal* argument specifies a signal that is to be delivered to the process when all of the I/O operations have been completed.

aiocbp a pointer to an array of pointers to asynchronous I/O control block structures. Elements in the array may contain the null pointer constant; such elements are ignored. Each control block to which an element in the array points specifies the parameters for an asynchronous I/O operation. The **aio_lio_opcode** field in the control block identifies the type of operation to be performed: asynchronous read (**LIO_READ**), asynchronous write (**LIO_WRITE**), or no operation (**LIO_NOP**). If this field contains **LIO_NOP**, the control block is ignored.

The other parameters specified by a control block include the file descriptor for the file from which data are to be read or to which data are to be written, the byte offset at which the read or write operation is to begin, the number of bytes to be read or written, and the virtual address of the I/O buffer into which data are to be read or from which data are to be written. The **aio_sigevent** field in each control block is used to notify the caller of completion of the individual I/O operation. See the description of the *signal* argument for information on the way in which the caller is notified of completion of the list of I/O operations.

If asynchronous I/O is to be performed to raw disk partitions, the **aio_flags** field in every control block in the list must be set to

AIO_RAW (for information on asynchronous I/O to raw disk partitions, see p. 12-7).

If the **O_RSYNC** bit and the **O_DSYNC** or **O_SYNC** bit have been set in the file status flag for the file specified by *the* **`aio_fildes`** field in the control block, completion of that I/O operation occurs as described in "The aio_read Routine" for **`aio_read(3)`**. If only the **O_DSYNC** or **O_SYNC** bit has been set, completion of that I/O operation occurs as described in "The aio_write Routine" for **`aio_write(3)`**.

count an integer value that indicates the number of elements in the array to which *aiocbp* points

signal the null pointer constant or a pointer to a structure that specifies the way in which the calling process is to be notified of the completion of the list of asynchronous I/O operations. The value of *signal–>sigev_notify* may be **SIGEV_NONE**, **SIGEV_SIGNAL**, or **SIGEV_CALLBACK**. **SIGEV_NONE** specifies that no notification is to be delivered upon completion of the I/O operations. **SIGEV_SIGNAL** indicates that a signal is to be sent to the process upon completion of the I/O operations. **SIGEV_CALLBACK** indicates that an application-defined routine is to be invoked upon completion of the I/O operations.

If you specify **SIGEV_SIGNAL**, *signal–>sigev_signo* specifies the number of the signal that is to be sent to the process when the list of asynchronous I/O operations has been completed, and *signal–>sigev_value* specifies an application–defined value that is to be used by a signal–handling routine defined by the receiving process. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file <**`signal.h`**>. The application-defined value may be a pointer or an integer value. If the process catching the signal has invoked the **`sigaction(2)`** system call with the **SA_SIGINFO** flag set prior to the time that the signal is generated, the signal and the application-defined value are queued to the process when the I/O operations are completed. For complete information on signal management facilities, the **`sigevent`** structure, and support for specification of an application–defined value, refer to the *PowerMAX OS Programming Guide*.

If you specify **SIGEV_CALLBACK**, *signal->sigev_func* specifies the address of an application-defined call-back routine that is to be called asynchronously upon completion of an asynchronous I/O operation, and *signal->sigev_value* specifies an application-defined value that is to be passed to that routine.

If the value of *mode* is **LIO_WAIT**, the *signal* argument is ignored. If the value of *mode* is **LIO_NOWAIT** and the value of *signal* is **NULL**, the value of *signal–>sigev_notify* is **SIGEV_NONE**, or the value of *signal–>sigev_signo* is zero (the null signal), no signal is delivered to the calling process when the asynchronous I/O operations have been completed.

If the value of the *mode* argument is **LIO_WAIT**, a return value of **0** indicates that all of the asynchronous I/O operations have been successfully completed. If the value of the *mode* argument is **LIO_NOWAIT**, a return value of **0** indicates that the asynchronous I/O operations have been successfully queued. In either case, a return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`lio_listio(3)`** system manual page for a listing of the types of errors that may occur.

Whether the value of *mode* is **LIO_WAIT** or **LIO_NOWAIT**, the return value does not indicate the status of the individual asynchronous I/O operations. If any of the operations has failed, the **`lio_listio`** routine returns a value of **–1**. If the value of *mode* is **LIO_WAIT**, **`errno`** is set to EIO. If the value of *mode* is **LIO_NOWAIT**, **`errno`** is set to EFAIL. A process can determine the status of a particular asynchronous I/O operation by invoking the **`aio_error(3)`** routine and specifying a pointer to the asynchronous I/O control block associated with that operation (see “The aio_error and aio_return Routines” for an explanation of this routine).

The aio_error and aio_return Routines

The **`aio_error(3)`** and **`aio_return(3)`** routines allow the calling process to obtain the status of a particular asynchronous I/O request. The process identifies the asynchronous I/O request by specifying a pointer to the **`aiocb`** structure associated with the request.

A process first invokes **`aio_error`** to determine whether the operation has been successfully completed. If it has not been successfully completed, **`aio_error`** returns the error code. When **`aio_error`** indicates that the operation has been successful, the process can invoke **`aio_return`** to determine the number of bytes that have been transferred.

The specifications required for making the **`aio_error`** call are as follows:

```
#include <aio.h>

int aio_error(aiocbp)

struct aiocb *aiocbp;
```

The argument is defined as follows:

aiocbp	a pointer to the asynchronous I/O control block structure associated with the I/O operation for which the status is to be returned

If the specified asynchronous I/O operation has been successfully completed, the return value is **0**. If the operation is in progress, the return value is EINPROGRESS. If an error has occurred, the return value is the associated error code as defined in the file <**`errno.h`**>. For a description of the types of errors that may occur, refer to the **`aio_read(3)`**, **`aio_write(3)`**, and **`aio_fsync(3)`** system manual pages.

The specifications required for making the **`aio_return`** call are as follows:

```
#include <aio.h>

int aio_return(aiocbp)

struct aiocb *aiocbp;
```

The argument is defined as follows:

aiocbp	a pointer to the asynchronous I/O control block structure associated with the I/O operation for which the status is to be returned

If the asynchronous I/O operation is still in progress or has resulted in an error, the return value is **–1**. If the operation has been successfully completed, the return value is the same as that defined for the **`aio_read(3)`**, **`aio_write(3)`**, and **`aio_fsync(3)`** routines. For **`aio_read`**, it is the number of bytes of data that have been read. For **`aio_write`**, it is the number of bytes of data that have been written. For **`aio_fsync`**, it is zero. See "The aio_read Routine," "The aio_write Routine," and "The aio_fsync Routine," respectively, for explanations of these routines.

The aio_cancel Routine

The **`aio_cancel(3)`** routine allows the calling process to cancel one or more asynchronous I/O operations.

The specifications required for making the **`aio_cancel`** call are as follows:

```
#include <aio.h>

int aio_cancel(fildes, aiocbp)

int            fildes;
struct aiocb *aiocbp;
```

The arguments are defined as follows:

fildes	the file descriptor for the file for which asynchronous I/O operations are to be cancelled
aiocbp	the null pointer constant or a pointer to the asynchronous I/O control block structure associated with the operation that is to be cancelled. If the value of *aiocbp* is **NULL**, then all of the asynchronous I/O operations for the file specified by *fildes* are cancelled.

If the asynchronous I/O operation or operations for the specified file descriptor have been successfully cancelled, the return value is AIO_CANCELED. If one or more of the asynchronous I/O operations for the specified file descriptor cannot be cancelled because they are still in progress, the return value is AIO_NOTCANCELED. The process can determine the status of such asynchronous I/O operations by invoking the **`aio_error(3)`** routine and specifying a pointer to the appropriate asynchronous I/O control block (see "The aio_error and aio_return Routines" for an explanation of this routine). If all of the asynchronous I/O operations for the specified file descriptor have already been completed, the return value is AIO_ALLDONE.

If an error occurs on the call, the return value is **–1**; **`errno`** is set to indicate the error. Refer to the **`aio_cancel(3)`** system manual page for a listing of the types of errors that may occur.

The aio_suspend Routine

The **`aio_suspend(3)`** routine allows the calling process to suspend execution until one of the following occurs:

- One or more of the specified asynchronous I/O operations are completed.
- The process is interrupted by a signal.
- A specified period of time elapses.

If any of the specified operations has already been completed at the time of the call, the routine returns without suspending the process. If the process is interrupted by a signal, note that the signal may have been generated because one of the specified asynchronous I/O operations has been completed.

This routine is flexible because it enables any thread within a process to ask about one or more outstanding asynchronous I/O operations and to specify how long to wait.

The specifications required for making the **`aio_suspend`** call are as follows:

```
#include <aio.h>

int aio_suspend(aiocbp, count, timeout)

struct aiocb    **aiocbp;
int               count;
struct timespec *timeout;
```

The arguments are defined as follows:

aiocbp	a pointer to an array of pointers to asynchronous I/O control block structures. Each of the control blocks to which the elements in the array point must have been specified on a call to **`aio_read(3)`**, **`aio_write(3)`**, or **`lio_listio(3)`** (see "The aio_read Routine," "The aio_write Routine," and "The lio_listio Routine," respectively, for explanations of these calls).
count	an integer value that indicates the number of elements in the array to which *aiocbp* points
timeout	the null pointer constant or a pointer to a structure that specifies the length of time that the process is to wait for completion of the specified asynchronous I/O operations. If the structure to which *timeout* points contains zeros or if the specified period of time elapses prior to completion of any of the specified operations, an error occurs.

A return value of **0** indicates that one or more of the specified asynchronous I/O operations has been completed. A process can determine whether or not a particular operation has been successful by invoking the **`aio_return(3)`** routine and specifying a pointer to the appropriate asynchronous I/O control block. It can obtain the status of the operation by invoking the **`aio_error(3)`** routine. (See "The aio_error and aio_return Routines" for explanations of these routines).

A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`aio_suspend(3)`** system manual page for a listing of the types of errors that may occur.

The aio_fsync Routine

The **`aio_fsync(3)`** routine allows the calling process to force all I/O operations associated with a particular disk file to the synchronized I/O completion state. You can specify whether the type of completion is to be synchronized I/O data integrity completion or synchronized I/O file integrity completion. POSIX synchronized I/O completion states are fully explained in Chapter 10 of this guide.

Only the I/O operations that are queued at the time of the call to **`aio_fsync`** are certain to be forced to the specified completion state; subsequent I/O operations associated with the file may not be.

The specifications required for making the **`aio_fsync`** call are as follows:

```
#include <aio.h>

int aio_fsync(opcode, aiocbp)

int           opcode;
struct aiocb *aiocbp;
```

The arguments are defined as follows:

opcode an integer value that specifies the type of synchronized I/O completion to be performed. This value may be one of the following:

O_DSYNC	specifies synchronized I/O data integrity completion
O_SYNC	specifies synchronized I/O file integrity completion

aiocbp a pointer to an asynchronous I/O control block structure that specifies the file descriptor for the disk file for which synchronized I/O completion is to be performed. The *aiocbp–>aio_sigevent* field may specify (1) the type of notification mechanism that is to be used when all of the I/O operations associated with the specified file have achieved synchronized I/O completion, (2) the number of a signal that is to be used to notify the process that all of the I/O operations associated with the specified file have achieved synchronized I/O completion, and (3) an application–defined value that is to be passed to the signal–handling routine.

The following fields are ignored on this call: *aiocbp–>aio_offset*, *aiocbp–>aio_buf*, *aiocbp–>aio_nbytes*, *aiocbp–>aio_reqprio*, *aiocbp–>lio_opcode*, *aiocbp–>aio_return*.

The control block to which *aiocbp* points must be a valid address within the calling process's virtual address space. It is important to note that the control block must remain valid until the asynchronous file synchronization operation completes; otherwise, a memory fault may occur.

A process can obtain the error status of this operation by invoking the **aio_error(3)** routine and specifying a pointer to this **aiocb** structure (see "The aio_error and aio_return Routines" for an explanation of this routine).

For a detailed description of each of the fields in the **aiocb** structure, refer to "The Asynchronous I/O Control Block."

A return value of **0** indicates that the asynchronous file synchronization operation has been successfully queued. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **aio_fsync(3)** system manual page for a listing of the types of errors that may occur.

Using Notification Mechanisms

You can synchronize process execution with completion of asynchronous I/O operations by polling for completion of the operations, by using call-back notification, or by using a signal to notify the process that an operation has been completed. "Polling" (p. 12-20) illustrates use of polling. "Call-Back Notification" (p. 12-22) illustrates use of call-back notification. "Signal Notification" (p. 12-24) illustrates use of signal notification.

Polling

A process can poll for completion of a particular asynchronous I/O operation by using the **aio_error(3)** routine to check the error status of the operation and the **aio_return(3)** routine to obtain the return status of the operation (refer to "The aio_error and aio_return Routines" for explanations of these routines). While the operation is in progress, **aio_error** returns **EINPROGRESS**; when the operation has been successfully completed, **aio_return** returns the number of bytes of data that have been transferred.

The following C program segment illustrates the procedures for using these routines to poll for completion of an asynchronous write operation.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <siginfo.h>
#include <signal.h>
#include <aio.h>

#define    FILE_SIZE  500


main()
{
     /* open() */
     int                      fi_desc;

     /* aio_write() */
     aiocb_t                  aiocbp_x;
```

```
char                    out_buf[FILE_SIZE];


/* misc */
int               i;
int               error;
int               ret;


/*
 * Fill up the buffer that will be written out
 */
for(i=0; i<FILE_SIZE;i++)
{
     out_buf[i] = ’x’;
}


/*
 * Create the output file
 */
fi_desc = open(”async.output”, O_CREAT | O_WRONLY);
if (fi_desc == -1)
{
     printf(”cannot open file\n”);
     exit(1);
}

/*
 * Initialize fields in the asynchronous I/O control block
 */
aiocbp_x.aio_offset                 = 0;
aiocbp_x.aio_buf                    = out_buf;
aiocbp_x.aio_nbytes                 = FILE_SIZE;
aiocbp_x.aio_reqprio                = 0;
aiocbp_x.aio_sigevent.sigev_notify  = SIGEV_NONE;
aiocbp_x.aio_fildes                 = fi_desc;


/*
 * Write to the file
 */
error = aio_write(&aiocbp_x);
if (error < 0)
{
     perror(”aio_write”);
     exit(1);
}


/*
 * Wait for completion (polling)
 */
while (aio_error(&aiocbp_x) == EINPROGRESS)
{
/*do work*/
     printf(”.”);
}

/*
 *  Check for errors
 */
if ((ret = aio_return(&aiocbp_x)) == -1)
{
     printf(”aio_write error errno=%d\n”, aio_error(&aiocbp_x));
```

```
          exit(1);
     }

     printf("aio_write wrote %d bytes\n", ret);

}
```

Call-Back Notification

A process can arrange for an application-defined call-back routine to be invoked upon completion of a single asynchronous I/O operation or a list of asynchronous I/O operations by specifying **SIGEV_CALLBACK** as the notification mechanism and by providing the address of the call-back routine on a call to **`aio_read(3)`**, **`aio_write(3)`**, or **`lio_listio(3)`**. In addition to providing the address of the call-back routine, it can specify an application–defined value that is to be passed to the call-back routine. (See "The aio_read Routine," "The aio_write Routine," and "The lio_listio Routine" for explanations of the **`aio_read`**, **`aio_write`**, and **`lio_listio`** routines.)

The C program segment that follows shows how a process can arrange for a call-back routine to be executed when an asynchronous read operation has been completed.

```
#include <stdio.h>
#include <errno.h>
#include <aio.h>
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>

/*
 * MAXBSIZE is the size of the receiving buffer
 * MAGIC_NUMBER can be used for verification of the received call-back
 */
#define    MAXBSIZE        8192
#define    MAGIC_NUMBER    5317

/*
 * The input file will be read into this buffer
 */
char       buffer[MAXBSIZE];


/*
 * Call-back Handler
 */
void
callback_handler(cb_info)
union sigval cb_info;


{
     /* misc */
     int  i;

     printf("callback_handler: callback received\n");
     printf("infop->si_signo = %d\n", infop->si_signo);
     printf("infop->si_code = %d\n", infop->si_code);
     printf("infop->si_value = %d\n", infop->si_value);
     printf("cb_info.sival_int = %d\n", cb_info.sival_int);
     /*
```

```
         * Verify what was read
         */
        printf(”\nThe following was read:\n”);
        for(i=0; i<MAXBSIZE; i++)
        {
             printf(”%c”, buffer[i]);
        }
   }


   main()
   {
        /* aio_read() */
        aiocb_t  aiocbp_x;
        int      buffsize = MAXBSIZE;
        int      error;


        /* fopen() & fclose() */
        FILE   *fi;
        int    fi_desc;

        /*
         * Open the input file
         */
        fi_desc = open(”async.input”, O_RDONLY);
        if (fi_desc == -1)
        {
             printf(”cannot open file\n”);
             exit(-1);
        }


        /*
         * Initialize fields in the asynchronous I/O control block
         */
        aiocbp_x.aio_fildes  = fi_desc;
        aiocbp_x.aio_offset  = 0;
        aiocbp_x.aio_buf     = buffer;
        aiocbp_x.aio_nbytes  = buffsize;
        aiocbp_x.aio_reqprio = 0;
        aiocbp_x.aio_sigevent.sigev_notify = SIGEV_CALLBACK;
        aiocbp.x.aio_sigevent.sigev_func = callback_handler;
        aiocbp_x.aio_sigevent.sigev_value.sival_int = MAGIC_NUMBER;

        /*
         * Do the read
         */
        error = aio_read(&aiocbp_x);
        if (error < 0)
        {
             perror(”aio_read”);
             exit(-1);
        }

        /* Perform unrelated work */
        /* . . . */

        exit(0);
   }
```

Signal Notification

A process can arrange for delivery of a signal to notify it of completion of a single asynchronous I/O operation or a list of asynchronous I/O operations by specifying **SIGEV_SIGNAL** as the notification mechanism and by providing the number of the desired signal on a call to **aio_read(3)**, **aio_write(3)**, or **lio_listio(3)**. In addition to providing the signal number, it can specify an application–defined value that is to be passed to the signal–handling routine when the signal is delivered. (See "The aio_read Routine," "The aio_write Routine," and "The lio_listio Routine" for explanations of the **aio_read**, **aio_write**, and **lio_listio** routines.) To use this method of notification, the process must define a signal–handling routine and declare that routine as the handler for the signal.

The C program segment that follows shows how a process can arrange for delivery of a signal to notify it that an asynchronous read operation has been completed. It illustrates some aspects of POSIX real–time signal behavior that are described in the *PowerMAX OS Programming Guide*. It shows how to declare the signal handler by invoking the **sigaction(2)** system call and setting the **SA_SIGINFO** flag. It also shows how to define the signal–handling routine with the particular interface that is required when the **SA_SIGINFO** flag is set.

The **SA_SIGINFO** flag is set to indicate (1) that the signal–handling routine is to be passed a **siginfo_t** structure providing information about the signal and (2) that when a subsequent occurrence of a pending signal is generated, another **siginfo_t** structure is queued with that instance of the signal. Note that the conditions for queuing a signal are met: the **aio_read** routine sends queued signals, and the process that is receiving the signal invokes **sigaction** with the **SA_SIGINFO** set prior to the time that the signal is generated. The signal number, a code that identifies the reason for the signal (**SI_ASYNCIO**), and the application–defined value are passed to the signal–handling routine in the **siginfo_t** structure.

```
#include <stdio.h>
#include <errno.h>
#include <aio.h>
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>

/*
 * MAXBSIZE is the size of the receiving buffer
 * MAGIC_NUMBER can be used for verification of the received signal
 */
#define    MAXBSIZE        8192
#define    MAGIC_NUMBER    5317

/*
 * The input file will be read into this buffer
 */
char       buffer[MAXBSIZE];


/*
 * Signal Handler
 */
void
sigaction_handler(sig, infop, ucp)
int              sig;
siginfo_t       *infop;
```

```
ucontext_t      *ucp;


{
     /* misc */
     int  i;

     printf(”sigaction_handler: signal received\n”);
     printf(”sig = %d\n”, sig);
     printf(”infop->si_signo = %d\n”, infop->si_signo);
     printf(”infop->si_code = %d\n”, infop->si_code);
     printf(”infop->si_value = %d\n”, infop->si_value);

     /*
      * Verify what was read
      */
     printf(”\nThe following was read:\n”);
     for(i=0; i<MAXBSIZE; i++)
     {
          printf(”%c”, buffer[i]);
     }
}


main()
{
     /* aio_read() */
     aiocb_t  aiocbp_x;
     int      buffsize = MAXBSIZE;
     int      error;

     /* sigaction() */
     struct sigaction  action;


     /* fopen() & fclose() */
     FILE   *fi;
     int    fi_desc;

     /*
      * Open the input file
      */
     fi_desc = open(”async.input”, O_RDONLY);
     if (fi_desc == -1)
     {
          printf(”cannot open file\n”);
          exit(-1);
     }


     /*
      * Initialize fields in the asynchronous I/O control block
      */
     aiocbp_x.aio_fildes  = fi_desc;
     aiocbp_x.aio_offset  = 0;
     aiocbp_x.aio_buf     = buffer;
     aiocbp_x.aio_nbytes  = buffsize;
     aiocbp_x.aio_reqprio = 0;
     aiocbp_x.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
     aiocbp_x.aio_sigevent.sigev_signo = SIGUSR1;
     aiocbp_x.aio_sigevent.sigev_value.sival_int = MAGIC_NUMBER;

     /*
      * Set signal handler
      */
     action.sa_handler  = sigaction_handler;
     sigemptyset(&action.sa_mask);
     action.sa_flags     = SA_SIGINFO;
     sigaction(SIGUSR1, &action, 0);

     /*
      * Do the read
      */
     error = aio_read(&aiocbp_x);
     if (error < 0)
     {
          perror("aio_read");
          exit(-1);
     }

     /* Perform unrelated work */
     /* . . . */

     exit(0);
}
```

13 Peripherals

13
Peripherals

This chapter contains the procedures for using a variety of devices.

- The real–time clock can be used for timing and frequency control functions. An overview of the device and the user interface to it is provided in “Using a Real–Time Clock.”
- The edge–triggered interrupt device enables a computer system to detect an external interrupt coming into the system from a user device that generates a signal pulse. A description of the edge–triggered interrupt and the user interface to it is provided in “Using an Edge–Triggered Interrupt Device.”
- The Real-Time Clocks and Interrupts Module (RCIM) provides distributed interrupts. These are interrupts sent to all systems connected via a RCIM chain. Distributed interrupts are similar to edge-triggered interrupts. A description of distributed interrupts and the user interface to them is provided in “Using a Distributed Interrupt Device”
- The High–Speed Data Enhanced interface, or HSDE, is provided to accommodate use of 32–bit external devices that use the Encore High Speed Data (HSD) interface Model 9132 protocol. The HSDE provides you with the ability to perform data chaining and command chaining. Procedures for using the HSDE are presented in “Using the High–Speed Data Enhanced Device (HSDE).” Memory mapping requirements are explained in “Memory Mapping for HSDE and DR11W.”
- The DR11W emulator is provided to accommodate use of external devices that use the Digital Equipment Corporation DR11W interface. An overview of the device and the procedures for using its associated user-level device driver are presented in “Using a DR11W Emulator.” Memory mapping requirements are explained in “Memory Mapping for HSDE and DR11W.”
- The 1553 Advanced Bus Interface (ABI) is a real-time interface between a PowerMAX OS system and a MIL-STD-1553 bus. It provides the host system with such advanced 1553 interface capabilities as independent and simultaneous bus controller simulation, remote terminal simulation, and 1553 bus monitoring. An overview of the 1553 ABI and the user interface to it are provided in “Using the 1553 Advanced Bus Interface.”
- The SYSTECH High Performance Serial (HPS) controller has the capability of being used with a standard STREAMS-based TTY driver and a special real-time device driver. This capability makes it possible for standard TTY activities and real-time communications to occur on the same port at different times. The information needed to use a real-time TTY device on the HPS controller is provided in “Using Real–Time Serial Communications.”

Using a Real–Time Clock

In this section, two types of information are provided to facilitate use of a real–time clock. An overview of the real–time clock device, **rtc**, is presented in “Understanding the Real–Time Clock Device.” A description of the user–interface to the device is provided in “Understanding the User Interface.”

Understanding the Real–Time Clock Device

The real–time clock device, **rtc**, is designed to be used for a variety of timing and frequency control functions. It provides a range of clock count values and a set of resolutions that taken together produce many different timing intervals—a feature that makes it particularly appropriate for frequency–based scheduling.

On Power Hawk Series 600/700/900 systems, there are integral real-time clocks located on the CPU board and additional clocks available through the Real-Time Clocks and Interrupts Module (RCIM), if installed.

On Power Hawk Series 600 systems, the integral clocks consist of tick timers and Zilog Z8536 timers. On Power Hawk Series 700/900 systems, the integral clocks consist of only tick timers. The number of each available varies based on system type. Only the tick timers can be used as the timing source for a frequency-based scheduler. For complete information on both types of timers refer to the **`rtc(7)`** system manual page and also to the system header file **`/usr/include/sys/rtc.h`**.

If the Real-Time Clocks and Interrupts Module (RCIM) is installed, it provides four additional real-time clocks. When multiple SBCs are connected via an RCIM chain, up to four real-time clocks may be designated to be distributed, i.e. its interrupts are sent to all connected systems. A distributed real-time clock may be located on any SBC within the RCIM chain.

More information can be found in the **`rtc(7)`** manual page.

On Power Hawk Series 600/700/900 systems, the device special files for the real-time clocks are as follows (where ***n*** specifies the clock number):

`/dev/rrtc/0c`*`n`*	rtc (character) special files - tick timer
`/dev/rrtc/1c`*`n`*	rtc (character) special files - Z8536 (Power Hawk Series 600 Systems only.)
`/dev/rrtc/2c`*`n`*	rtc (character) special files - RCIM

On PowerMAX OS systems, the real–time clock controller is integral to the system. Each CPU board has one real–time clock controller. On Model 6200 and 6800 systems, five real–time clocks are provided on the first CPU board (board 0). Three real–time clocks are provided on each additional CPU board. The HVME real–time clock controller is not supported. On PowerMAXION systems, five real-time clocks are provided on each of the four CPU boards.

On PowerMAX OS systems, device special files for real–time clocks have names of the form **/dev/rrtc/***m***c***n*, where *m* specifies a controller number that ranges from zero to three and corresponds to the CPU board on which the clock resides; **c** stands for clock, and *n* specifies a real–time clock number that ranges from zero to four on the first CPU board and zero to two on additional CPU boards. The names of the device special files on the first board must be as follows:

```
/dev/rrtc/0c0

/dev/rrtc/0c1

/dev/rrtc/0c2

/dev/rrtc/0c3

/dev/rrtc/0c4
```

The names of the device special files on the second board must be as follows:

```
/dev/rrtc/1c0

/dev/rrtc/1c1

/dev/rrtc/1c2
```

The names of the device special files on the third board must be as follows:

```
/dev/rrtc/2c0

/dev/rrtc/2c1

/dev/rrtc/2c2
```

The names of the device special files on the fourth board must be as follows:

```
/dev/rrtc/3c0

/dev/rrtc/3c1

/dev/rrtc/3c2
```

On PowerMAXION systems, device special files for real–time clocks have names of the form **/dev/rrtc/***m***c***n*, where *m* specifies a controller number that ranges from zero to three and corresponds to the CPU board on which the clock resides; **c** stands for clock, and *n* specifies a real–time clock number that ranges from zero to four on each CPU board. The names of the device special files on the first board must be as follows:

```
/dev/rrtc/0c0

/dev/rrtc/0c1

/dev/rrtc/0c2

/dev/rrtc/0c3

/dev/rrtc/0c4
```

The names of the device special files on the second board must be as follows:

/dev/rrtc/1c0

/dev/rrtc/1c1

/dev/rrtc/1c2

/dev/rrtc/1c3

/dev/rrtc/1c4

The names of the device special files on the third board must be as follows:

/dev/rrtc/2c0

/dev/rrtc/2c1

/dev/rrtc/2c2

/dev/rrtc/2c3

/dev/rrtc/2c4

The names of the device special files on the fourth board must be as follows:

/dev/rrtc/3c0

/dev/rrtc/3c1

/dev/rrtc/3c2

/dev/rrtc/3c3

/dev/rrtc/3c4

On PowerMAX OS systems, each real–time clock is connected to a particular pin on the interrupt terminator board. The hardware controls the interrupt priority associated with each pin. The real-time clock interrupts are handled on the CPU board on which they reside. They cannot be routed to other CPU boards.

NOTE

To use a real–time clock on a PowerMAX OS system on which the Enhanced Security Utilities are installed, device special files must be created in the **/dev/rrtc** directory. Refer to the "Trusted Facility Management" chapter of *System Administration Volume 1* for an explanation of the procedures for using device files when the Enhanced Security Utilities are installed.

A real–time clock operates in one of two modes: default mode or direct mode. If the clock is in default mode, you can control the following:

- Whether the clock counts up or down
- What the value of the clock count is

- What the resolution per clock count is
- Whether the clock automatically starts counting again when the clock count reaches its terminal count (zero or 65,535)

If the clock is in direct mode, you can directly program the hardware registers. Doing so requires information on the system timing chip and its registers. The information needed can be obtained by special request. Directly programming the hardware registers also requires information that is provided in the system manual page **`rtc(7)`**.

You can use a real-time clock for triggering events in the high-resolution callout queue. The **`hrtconfig(1M)`** command is provided for this purpose. When you configure a real-time clock for use with the high-resolution callout queue, you cannot use it for any other purpose. Refer to Chapter 3 for additional information on the high-resolution callout queue and use of the **`hrtconfig`** command. Note that using a real-time clock with the high-resolution callout queue does <u>not</u> affect the resolution of other clocks on the same controller.

Understanding the User Interface

The real–time clock can be directly controlled by using the standard PowerMAX OS system calls: **`open(2)`**, **`close(2)`**, and **`ioctl(2)`**.

NOTE

This device does not support the **`read(2)`** and **`write(2)`** system calls.

A set of **`ioctl`** commands enables you to perform a variety of operations that are specific to the device. These commands are summarized as follows:

RTCIOCSET	set the mode and the count and resolution values for a real–time clock
RTCIOCGET	obtain the mode and the count and resolution values for a real–time clock
RTCIOCINFO	obtain information about the specified real-time clock.
RTCIOCSETCNT	set the count value for a real–time clock
RTCIOCMODCNT	stop the counting of a real–time clock, modify the count value for the clock, and start the counting of the clock
RTCIOCGETCNT	obtain the current count value for a real–time clock
RTCIOCRES	obtain the current resolution value for a real–time clock
RTCIOCSTART	start the counting of a real–time clock
RTCIOCSTOP	stop the counting of a real–time clock

RTCIOCWAIT	wait for the count value for a real–time clock to reach zero
RTCIOCFBS	set the mode for a real–time clock to frequency–based scheduling
IOCTLVECNUM	obtain the interrupt vector number of a real–time clock

Detailed descriptions of these commands and the specifications required for using them are presented in the system manual page **`rtc(7)`**.

Watch-Dog Timer Function

The fifth RTC on the first board can be used as watch-dog timer or as an interrupting real time clock. When used as an interrupting clock the output of the RTC that indicates a time out continues to be connected to the interrupt control logic. However, when the RTC is being used as a watch-dog timer, its interrupt is disabled via software issuing a disarm interrupt command to the RTC’s interrupt level. The RTC’s time-out output is also connected to the logic Processor Control and Status Register (PCSR).

The RTC used in the watch-dog timer function is programmed by the application using the facilities provided under PowerMAX OS. For more information on software control capability of the RTC, refer to the manual pages section -rtc.

It is recommended that the RTC watch-dog timer be used in the default mode and programmed to have a clock resolution of 1 millisecond. This time gives a time out range of from 1 millisecond to 65.535 seconds. In the event of a time-out, the hardware generates the SRESET signal to the PPC604 processor. This signal causes the processor to save the machine state in its Save and Restore Registers (SRR) and start execution of a soft reset exception. This execution’s execution starts at physical location 0x00000100. The exception handler then tests the MODULE_NO_GO register flag to find out if the cause of the soft reset is the watch-dog timer time-out. If it is, processing of the soft reset continues by resetting of the MODULE_NO_GO bit followed by a reset of the SRESET register bit in the PCSR. Control is now passed to a user defined exception handler.

Using the watch-dog function, the application can monitor the health of its processes. To accomplish this the application must program for the watch-dog interrupt in the following manner:

1. The fifth real time clock's interrupt must be disabled on the processor's interrupt controller. The application does this by mapping the interrupt controller's enable register using the shared memory mechanism.The physical addresses for the interrupt enable registers on the PowerMAXION are:

   ```
   0x96200020 local processor
   0x9D000020 processor 0
   0x9D100020 processor 1
   0x9D200020 processor 2
   0x9D300020 processor 3
   ```

   The fifth real time clock interrupt is disabled by resetting bit 17 in the 32-bit enable register.

The application must take care to not change other bits in the interrupt controller's enable register. This can be achieved by reading the enable register, masking out only bit 17, and re-writing the contents back to the enable register.

The application has the responsibility of re-enabling this interrupt once use of the watch-dog timer is complete. This is achieved by setting bit 17 in the enable register. Failure to do so will preclude the fifth real time clock on processor board one from being used as a timer.

2. The interrupt signal from the fifth real-time clock must be routed to the PPC604 processor. The application does this by mapping the processor's 16 bit control and status register (PCSR) using the shared memory mechanism. The physical addresses for the PCSR's are as follows:

   ```
   0xB2000000 processor 0
   0xB2000008 processor 1
   0xB6000000 processor 2
   0xB6000008 processor 3
   ```

   Routing of the fifth real-time clock interrupt is achieved by setting bit 11 in the PCSR for the respective processor board. The application must take care to not change other bits in the PCSR. This can be achieved by reading the register, setting bit 11, and re-writing the contents back to the register.

   The application has the responsibility of restoring bit 11 of the PCSR to 0 once use of the watch-dog timer function is complete. Failure to do so will preclude the fifth real time clock on processor board one from being used as a timer.

3. The application must connect and enable the user level interrupt routine. This is achieved using the **`iconnect(3C)`** and **`ienable(3C)`** routines. The application must also lock all memory resources used by the user level interrupt routine. These resources include shared memory regions, library text and data, process text and data. See chapter 7 in this manual for a description of user level interrupts.

4. The fifth real time clock must be programmed by the application with the correct count and frequency. PowerMAX OS supplies a user interface to the real-time clocks.

Using an Edge–Triggered Interrupt Device

This section contains the information needed to use an edge–triggered interrupt. An overview of the edge–triggered interrupt device, **eti**, is presented in “Understanding the Edge–Triggered Interrupt Device.” A description of the user–interface to the device is provided in “Understanding the User Interface.”

Understanding the Edge–Triggered Interrupt Device

The edge–triggered interrupt device, **eti**, provides a means for the computer system to detect an external interrupt coming into the system from any user device that generates a signal pulse.

On PowerMAX OS systems, edge–triggered interrupts are integral to the system. Four edge–triggered interrupts are provided for each CPU board. One to four CPU boards may be configured; as a result, the number of edge–triggered interrupts per system ranges from four to 16.

The edge-triggered interrupts, by default, are automatically configured according to the number of CPU boards configured. If you do not wish the edge-triggered interrupts to be configured in your system, you can edit the **`/etc/conf/sdevice.d/eti`** file and change the value in the **`conf`** field to **`N`**.

On PowerMAX OS systems, device special files for the integral edge–triggered interrupts have names of the form **`/dev/reti/eti`***n*, where *n* specifies an edge–triggered interrupt number ranging from zero to 15. The numbers **0**–**3** are the edge–triggered interrupts on CPU board 0; **4**–**7** are the edge–triggered interrupts on CPU board 1; **8**–**11** are the edge–triggered interrupts on CPU board 2; and **12**–**15** are the edge–triggered interrupts on CPU board 3. If a CPU board in a specified slot is marked down or is not present, the numbering scheme is not affected. If a system contains a CPU board in slot 0 and a CPU board in slot 3, for example, the edge–triggered interrupts on the first board are numbered **0**–**3**, and the edge–triggered interrupts on the second board are numbered **12**–**15**

On PowerMAX OS systems, each edge–triggered interrupt is connected to a particular pin on the terminator board. Edge-triggered interrupts are handled on the CPU board on which they reside. They cannot be routed to other CPU boards.

For detailed information on the edge–triggered interrupt hardware and the conditions that are required for using it, refer to the system manual page **`eti(7)`**, the *HN6200 Architecture Manual*, the *HN6800 Architecture Manua*l, the *PowerMAXION Architecture Manual* and the *TurboHawk Architecture* Manual.

On Power Hawk Series 600/700/900 systems, edge-triggered interrupts are provided by the Real-Time Clocks and Interrupts Module (RCIM), if installed. There are four edge-triggered interrupts available to each SBC (single-board computer) that has an RCIM. When multiple SBCs are connected via an RCIM chain, up to four ETIs may be designated to be distributed, i.e. its interrupts are sent to all connected systems. A distributed ETI may be located on any SBC within the RCIM chain.

The kernel tunable **`RCIM_DISTRIB_ETIS`** specifies which ETIs are distributed.

On Power Hawk Series 600/700/900 systems, ETI device special files are <u>only</u> available if an RCIM module is installed. They have the following format:

`/dev/reti/eti0`*n*	eti (character) special files

For more information, refer to the system manual page **`eti(7)`**.

Understanding the User Interface

The edge–triggered interrupt device can be directly controlled by using the following standard PowerMAX OS system calls: **`open(2)`**, **`close(2)`**, and **`ioctl(2)`**.

NOTE

This device does not support the **`read(2)`** and **`write(2)`** system calls.

A set of **`ioctl`** commands enables you to perform a variety of operations that are specific to the device. These commands are summarized as follows:

ETI_ARM	arm the edge–triggered interrupt
ETI_DISARM	disarm the edge–triggered interrupt.
ETI_ENABLE	enable the edge–triggered interrupt.
ETI_DISABLE	disable the edge–triggered interrupt.
ETI_INFO	obtain information about the specified edge-triggered interrupt.
ETI_REQUEST	generate a software–requested interrupt along the edge–triggered interrupt. Note that the edge–triggered interrupt must previously have been armed and enabled.
ETI_ATTACH_SIGNAL	attach the specified signal number to the edge–triggered interrupt. The signal will be generated on every interrupt.
ETI_VECTOR	place the edge–triggered interrupt vector number in the specified location. Note that any device that is being attached to a frequency–based scheduler must support this command.

Detailed descriptions of these commands and the specifications required for using them are presented in the system manual page **`eti(7)`**.

Using a Distributed Interrupt Device

This section contains information needed to use distributed interrupts. An overview is presented in "Understanding Distributed Interrupts". A description of the user interface is provided in "Understanding the User Interface".

Understanding Distributed Interrupts

The Real-Time Clocks and Interrupts Module (RCIM) provides eight distributed interrupts. These are interrupts that are sent to all SBCs connected via a RCIM chain.

The source of the device that generates a distributed interrupt may be on any SBC within the RCIM chain. Sources of distributed interrupts are the real-time clocks, edge-triggered interrupts or priority interrupt generators located on an RCIM. For more information, refer to the **`distrib_intr(7)`** manual page.

Distributed interrupts are similar to edge-triggered interrupts. The '**eti**' driver must be enabled to access edge-triggered interrupts.

There are several system tunables that are used to configure distributed interrupts.

The tunables **RCIM_DISTRIB_ETIS**, **RCIM_DISTRIB_RTCS** and **RCIM_DISTRIB_PIGS** specify the set of ETIs, RTCs and PIGs from the local system only; that will have its interrupts distributed to all systems.

Generally, only one SBC within the RCIM chain should distribute the same device (in other words, no more than one SBC should distribute **`eti0`**, no more than one SBC should distribute **`eti1`**, etc.)

The tunable **RCIM_DISTRIB_INTR[N]** specify the source for each distributed interrupt.

Generally, these tunable values should be the same on all systems within a RCIM chain.

The device files for distributed interrupts are:

```
/dev/distrib_intrN       (where N=0..7)
```

Note: Both the **`rcim`** and **`eti`** kernel modules must be enabled to use distributed interrupts.

Understanding the User Interface

Distributed interrupts can be directly controlled using the following standard PowerMAX OS system calls: **`open(2)`**, **`close(2)`** and **`ioctl(2)`**.

Note: This device does not support **`read(2)`** and **`write(2)`** system calls.

A set of **`ioctl`** commands enables you to perform a variety of operations that are specific to distributed interrupts. These commands are summarized below:

DISTRIB_INTR_ARM	arm the distributed interrupt.
DISTRIB_INTR_DISARM	disarm the distributed interrupt.
DISTRIB_INTR_ENABLE	enable the distributed interrupt.

DISTRIB_INTR_DISABLE	disable the distributed interrupt.
DISTRIB_INTR_INFO	obtain information about the specified distributed interrupt, including its source.
DISTRIB_INTR_ATTACH_SIGNAL	attach the specified signal number to the distributed interrupt. The signal will be generated on every interrupt.
DISTRIB_INTR_VECTOR	obtain distributed interrupt vector number.

More information can be found in the **`distrib_intr(7)`** system manual page.

Using the High–Speed Data Enhanced Device (HSDE)

If your PowerMAX OS system is linked to a 32–bit external device that uses the Encore High Speed Data (HSD) Interface Model 9132 protocol, you may wish to take advantage of the features of the high–speed data enhanced device driver that are designed to improve the performance of real–time applications. The high–speed data enhanced device performs DMA transfers directly to and from the user's virtual address space; it provides you with the ability to perform command chaining and data chaining.

An overview of the high–speed data enhanced device, **hsde**, is presented in "Using the High–Speed Data Enhanced Device (HSDE)." A description of the user interface to the device is provided in "Understanding the HSDE User Interface." A protocol for coordinating communication and exchange of data between programs is presented in "Using a Master–Slave Transfer Protocol." Procedures for using the command chaining mode are described in "Using the HSDE Command Chaining Mode." Procedures for using the data chaining mode are described in "Using the HSDE Data Chaining Mode." In order to use the high–speed data device, you must ensure that the user's I/O buffer is bound to a contiguous section of physical memory. An explanation of the procedures for complying with memory mapping requirements is presented in "Memory Mapping for HSDE and DR11W."

Understanding the High–Speed Data Enhanced (HSDE) Device

The high–speed data enhanced device, **hsde**, is an enhanced channel interface which provides a bidirectional DMA link for transferring control, status, and data between a PowerMAX OS system and any 32–bit external device that uses the Encore High Speed Data (HSD) Interface Model 9132 protocol. The **hsde** is "enhanced" by the fact that it contains a dedicated Motorola 68020 microprocessor coupled with on–board firmware to perform the HSD protocol handshaking and a VIC068 VME Interface Controller (VIC) to control DMA operations to/from the PowerMAX OS system host. These enhancements give the **hsde** its high-end performance. The **hsde** does not support the Encore InterBus Link (IBL) protocol.

HSDE device channels are supported on the (H)VME bus. The **hsde** can be connected to any device that is compatible with the Encore HSD Interface Model 9132, including another **hsde**. Eight (H)VME address ranges are reserved for use by the **hsde**. The addresses associated with the **hsde** devices configured into your system may be found in the **`/etc/conf/sadapters.d/kernel`** file.

Device special files for **hsde** devices have names of the form **`/dev/hsde`***#*, where *#* represents a minor device number ranging from zero to the number of **hsde** devices on your system.

You can control operation of a **hsde** device by setting any of fourteen modes. Of the fourteen configuration modes, four are generally accessed. These four modes are described below:

Master/slave mode	enables you to indicate whether the high–speed data enhanced device is to serve as the master or the slave device
Transfer size mode	enables you to specify whether data are to be transferred one byte, one word, or one long-word at a time
Command chaining mode	enables you to indicate whether or not multiple operations can be performed on a single **`read`** or **`write`** system call
Extended I/O Control	
Block mode	enables you to specify whether the extended version of the HSD protocol's I/O Control Blocks (IOCBs) should be utilized during transfer operations. More information on IOCBs will be provided in the following sections.

Table 13-1. Mode Default Values

Mode	Default Value
Transfer size	`HSDE_LONG`
Master/Slave	`HSDE_MASTER`
Command chaining	Value Zero (Disabled)
Extended IOCB	Value Zero (Disabled)

You can set these modes as well as the additional ten modes to different values by making an `HSDE_SET_MODE` **`ioctl`** call. The values that can be specified for each mode and the specifications required for this call are presented in the system manual page **`hsde(7)`**.

Understanding the HSDE User Interface

The **hsde** device is controlled by using the following standard PowerMAX OS system calls: **`open(2)`**, **`close(2)`**, **`ioctl(2)`**, **`read(2)`**, **`write(2)`**, **`readv(2)`**, and **`writev(2)`**.

On the open call in `HSDE_MASTER` mode, the **hsde** device may be opened by multiple processes. It is the responsibility of the cooperating processes to synchronize access to the **hsde**. On the **`open`** call in `HSDE_SLAVE` mode, the **hsde** device is opened exclusively and cannot be opened by another process.

On the **`close`** call, the **hsde** device is automatically reset. Therefore, all configuration values are set to their default values as specified in the **`hsde(7)`** manual page.

A set of **`ioctl`** commands enables you to perform a variety of operations that are specific to the device. These commands are summarized as follows:

HSDE_GET_MODE	query the **hsde** device configuration modes
HSDE_SET_MODE	set the **hsde** device configuration modes
HSDE_COMMAND	send a high–speed data device command
HSDE_GET_CMD	receive a high–speed data command
HSDE_STATUS	send a high–speed data device status request command
HSDE_PULSE_SIG	send an IOR or TDV signal via the **hsde**
HSDE_VECTOR	obtain the base interrupt vector of the **hsde**
HSDE_DUMP	query all readable **hsde** on–board registers
HSDE_RESET	reset the **hsde** device and send IOR to external device
HSDE_LOCAL_STAT	query the **hsde** device for local status information
HSDE_ABORT_OP	abort the current operation executing on the **hsde**
HSDE_DIAG	run internal diagnostics on the **hsde**
HSDE_CYCLE_CHAIN	reexecute the previously executed command chain
HSDE_LOAD_UPROG	download a user–developed MC68020 data conversion program
HSDE_ENABLE_UPROG	enable/disable a downloaded MC68020 data conversion program

Detailed descriptions of these commands and the specifications required for using them are presented in the system manual page **`hsde(7)`**.

If you elect to use command chaining mode, you must use special forms of the **`read`** and **`write`** system calls. Command chaining mode and the specifications required for these calls are explained in detail in "Using the HSDE Command Chaining Mode."

Using a Master–Slave Transfer Protocol

If you are using the **hsde** device to communicate with another high–speed data device, one of the devices must be placed in slave mode, and one must be placed in master mode. Because the default mode for the **hsde** device is **HSDE_MASTER**, master/slave mode for the slave device must be set to its appropriate slave mode.

The role of the master device is to issue requests for status, issue commands, and send and receive data. The role of the slave is to update status, get commands, and receive and send data. Commands may be of two types: device commands and read or write commands. Communication and exchange of data between programs must be governed by a protocol. The minimal protocol that must be followed is presented in Table 13-2.

Table 13-2. Master/Slave Protocol

Device	Action
Slave HSD	1. Enable slave mode
	2. Get the next command
Master HSD	3. Issue a device status request
	4. Send a device command or Read or Write
Slave HSD	5. Process a device command or Write or Read
Master HSD	6. Go to Step 3 or Step 4
Slave HSD	7. Go to Step 2

Each action is performed by making a related **`ioctl(2)`**, **`read(2)`**, **`write(2)`**, **`readv(2)`**, or **`writev(2)`** system call. The specific calls that are involved in performing each action are described as follows.

Enable Slave Mode (1)

Master/slave mode is set to `HSDE_SLAVE` on the slave device by using the `HSDE_SET_MODE` **`ioctl`** command.

Get the next command (2)

The process that controls the slave HSDE receives a command by using the HSDE_GET_CMD **ioctl** command. This command enables the slave HSDE's external function command interrupt and thereby allows the process that controls the master HSDE to send a command.

Issue a Device Status Request (3)

The process that controls the master HSDE synchronizes operation with the process that controls the slave HSDE and queries the state of that process by using the HSDE_STATUS **ioctl** command to read the slave HSDE's device status register.

Send a Device Command or Read or Write (4)

The process that controls the master HSDE sends a device command by using the HSDE_COMMAND **ioctl** command. It sends a read command by using a **read** or **readv** system call and a write command by using a **write** or **writev** system call.

Process a Device Command or Write or Read (5)

When the process that controls the master HSDE sends a read command, the process that controls the slave HSDE makes a **write** or a **writev** system call. Alternatively, when it sends a write command, the slave HSDE makes a **read** or a **readv** system call. Device commands are specific to the application.

Detailed explanations of each of these actions and the corresponding system calls are provided in the system manual page **hsde(7)**. Example programs that demonstrate use of the protocol to transfer a file from a master HSDE to a slave HSDE are included in Appendix E.

Using the HSDE Command Chaining Mode

Command chaining mode enables a master high–speed data device to initiate device commands, I/O transfer commands, or a combination of the two with a single **read** or **write** system call. Device commands are specific to the external device. I/O transfer commands instruct the high–speed data device to initiate a DMA transfer to or from an external device. The number of commands that you can initiate cannot exceed the value of HSDE_MAX_IOCL, which is set in the tunable parameters configuration file, **/etc/conf/mtune.d/hsde**.

NOTE

When you enable command chaining mode, you must also set master/slave mode to HSDE_MASTER.

When you select command chaining mode, you must use special forms of the **read(2)** and **write(2)** system calls. The **read** and **write** system calls must be specified as follows:

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

read(fildes, iocl, niocb)

write(fildes, iocl, niocb)

int               fildes;
hsde_iocb_t       iocl[];
int               niocb;
```

Arguments are defined as follows:

fildes	the file descriptor for the file to or from which data are being transferred
iocl[]	a pointer to a high–speed data device I/O command list (hereinafter referred to as an IOCL), which is an array of high–speed data device I/O command block (hereinafter referred to as IOCB) structures
niocb	the number of high–speed data device commands in the specified IOCL

The **hsde** device I/O command block structure, **`hsde_iocb`**, is defined in the header file <**`sys/h3300_channel.h`**> as presented in the lines that follow.

```
/*
**  Hsde I/o Command Block Structure:
*/
typedef unsigned char   ui8;
typedef unsigned long   ui32;


typedef struct    hsde_iocb {
      /*    IOCB Word 0 */
      union {
            ui32            I_iocb0;   /* iocb word 0 */
            struct {
                  ui8       I_opcode;  /* hsd opcode */
                  ui8       I_info;    /* device dependent info. */
                  ui8       I_tc;      /* transfer count */
            } Iocb0_s ;
      } Iocb0_u ;
      /*    IOCB Word 1 */
      union {
            ui32            I_iocb1;   /* iocb word 1 */
            /* I/o Transfer Command Format */
            ui32            I_ta;      /* transfer address */
            /* Device Command Format */
            ui32            I_command; /* device dependent command */
     } Iocb1_u ;
     /*     IOCB Word 2 */
     union {
            ui32            I_iocb2;        /* iocb word 2 */
            ui32            I_la;           /* link address */
     } Iocb2_u ;
     /*     IOCB Word 3:  Used with extended IOCB. */
     union {
            ui32            I_iocb3;        /* iocb word 3 */
            struct {
                     ui8    I_xfer_opcode; /* xfer op code */
                     ui8    I_user_param; /* user prog param */
```

```
                    ui8     I_xfer_addr_modifier; /* vme xfer addr mod */
                    ui8     I_link_addr_modifier; /* vme link addr mod */
            } Iocb3_s ;
     } Iocb3_u ;
} h3300_channel_t ;


/*    IOCB Word 0 Defines:    */
#define     i_iocb0         Iocb0_u.I_iocb0
#define     i_opcode        Iocb0_u.Iocb0_s.I_opcode
#define     i_info          Iocb0_u.Iocb0_s.I_info
#define     i_tc            Iocb0_u.Iocb0_s.I_tc


/*    IOCB Word 1 Defines:    */
#define     i_iocb1         Iocb1_u.I_iocb1
#define     i_ta            Iocb1_u.I_ta
#define     i_command       Iocb1_u.I_command


/*    IOCB Word 2 Defines:    */
#define     i_iocb2         Iocb2_u.I_iocb2
#define     i_la            Iocb2_u.I_la


/*    IOCB Word 3 Defines:    */
#define     i_iocb3                 Iocb3_u.I_iocb3
#define     i_xfer_opcode           Iocb3_u.Iocb3_s.I_xfer_opcode
#define     i_user_param            Iocb3_u.Iocb3_s.I_user_param
#define     i_xfer_addr_modifer     Iocb3_u.Iocb3_s.I_xfer_addr_modifier
#define     i_link_addr_modifer     Iocb3_u.Iocb3_s.I_link_addr_modifier
```

In the definition of IOCB Word 1, the **`hsd_iocb`** structure definition shows that there are two formats for IOCBs: one for device commands and one for I/O transfer commands. The format of an IOCB is determined by the value of **`i_opcode`**, which is defined in IOCB Word 0. For device commands, the value of **`i_opcode`** must be **HSDE_OP_COMMAND**. For I/O transfer commands, the value of **`i_opcode`** must be either **HSDE_OP_READ** or **HSDE_OP_WRITE**.

NOTE

The i_opcode field of all but the last IOCB in the command chain must have the command chain bit (**HSDE_OP_CCHAIN**) set—for example:

```
iocl[0].i_opcode = HSDE_OP_READ | HSDE_OP_CCHAIN;
```

The fields that are included in the IOCB format for device commands are described as follows:

i_opcode	contains the high–speed data device opcode `HSDE_OP_COMMAND`
i_info	may contain device dependent information. This field is not interpreted by the **hsde** or the **hsde** device driver.
i_tc	not used

i_command	contains the device–dependent command. This command is meaningful only to the external hsd device. This field is not interpreted by the **hsde** or the **hsde** device driver.
i_iocb2	is an unused longword in the IOCB. It should be set to zero.

The fields that are included in the IOCB format for I/O transfer commands are described as follows:

i_opcode	contains the **hsde** device opcode `HSDE_OP_READ` or `HSDE_OP_WRITE`
i_info	may contain device dependent information. This field is not interpreted by the **hsde** or the **hsde** device driver.
i_tc	contains the desired I/O transfer count—that is, the number of bytes, words, or longwords to be transferred. The transfer count must be related to the transfer size mode that has been selected. If, for example, transfer size mode has been set to **HSDE_LONG**, this field must contain the longword count. It is important to note that the transfer count cannot exceed 65,535 bytes, words, or longwords.
i_ta	contains the transfer address—that is, the virtual base address of the desired I/O transfer
i_iocb2	is an unused longword in the IOCB. It should be set to zero.

If the extended IOCB option is enabled in the **hsde** configuration, the fourth longword of the IOCB structure will be recognized by the **hsde** interface. The main reason for desiring this option is to utilize user–developed MC68020 assembly language data conversion programs. These data conversion programs can be downloaded onto the **hsde** via the HSDE_LOAD_UPROG **`ioctl`**. Such programs can be used to convert data either being sent from or received on the **hsde**. These conversion programs must be assembled MC68020 binaries. For more information on the usage of MC68020 conversion programs refer to the **`hsde(7)`** manual page. The fields that are included in the IOCB format for I/O transfer commands are described as follows:

i_xfer_opcode	contains a flag (IUC defined in <**`sys/hsde.h`**>) indicating whether a user–developed data conversion routine previously loaded should be activated.
i_user_param	contains a parameter which may be sent to the user–developed data conversion routine. Such a parameter may be used to select a specific path of action in the conversion routine. If no parameter is needed, then it should be set to zero.
i_xfer_addr_modifer	specifies the (H)VME address modifier code to use for data transfers. This value should usually be left as the default value specified in the **hsde** configuration (refer to the configuration information in the **`hsde(7)`** manual page).

i_link_addr_modifer	specifies the (H)VME address modifier code to use for accessing an IOCB. This value should usually be left as the default value specified in the **hsde** configuration (refer to the configuration information in the **`hsde(7)`** manual page).

It is not necessary to directly access the fields of the fourth longword of an IOCB when dealing with user–developed data conversion routines. The HSDE_ENABLE_UPROG **`ioctl`** command provides a simple interface for modifying the data conversion routine–specific fields.

Example programs that demonstrate use of command chaining to transfer simple data from a slave HSDE to a master HSDE are included in Appendix E.

Using the HSDE Data Chaining Mode

The **hsde** data chaining mode allows the user to initiate up to `HSDE_MAX_IOCL` I/O transfer commands with a single **`readv(2)`** or **`writev(2)`** system call. Depending on the system call, all IOCBs in the data chain will be of that type (i.e., all read or all write). With data chaining, the **`hsde`** can perform scatter–gather I/O operations or perform a single data transfer operation on a block of data whose size is greater than the maximum allowable HSD single transfer size limit of 64K–1 bytes, 64K–1 words, or 64K–1 longwords.

The standard forms of the **`readv(2)`** and **`writev(2)`** system calls are as follows:

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

readv(int fildes, struct iovec *iov, int iov_cnt)
writev(int fildes, struct iovec *iov, int iov_cnt)
```

Arguments are defined as follows:

fildes	the file descriptor for the file to or from which data are being transferred
iov	an array of **`iovec`** structures each containing a base (virtual) address of a user data buffer and the size in bytes of that data buffer. The byte size must be a multiple of the data type width (i.e., byte(1), word(2), longword(4)).
iov_cnt	the number of **`iovec`** structures in the array. This value must not exceed HSDE_MAX_LOCK.

For scatter–gather I/O operations, each **`iovec`** structure will contain information concerning the base address of a buffer and its size. The size of each buffer must be less than or equal to the HSD protocol–specified maximum single transfer size (64K–1 bytes, 64K–1 words, or 64K–1 longwords). For transfer sizes greater than the maximum single HSD transfer size, the suggested method is to overlay the region containing the transfer data with buffers equivalent to the maximum transfer size. If a portion of the data cannot fill an entire maximum transfer size buffer, it can be overlaid by a buffer of exact size. Thus the data region will be broken down into a series of “chunks.” The size of each “chunk” will be the maximum single transfer size depending on the data path width. The *iov* array will then contain a series of pointers to these data region “chunks”.

Only one side in a high–speed data interface to high–speed data interface connection can utilize a data chain for a transfer operation. The HSD protocol requires that the normal I/O External Function (EF) handshake signaling across the interface cable take place only once at the start of a data chain operation and not at the start of each individual I/O operation within the data chain. In this fashion, the master HSD device will signal once of an impending I/O operation, and then proceed to send individual I/O requests for various sizes of data until the end of its request chain is reached. Because of the single EF handshake signal, a data chain operation has the appearance of a single I/O operation to the remote HSD device. As such, the remote device can only utilize a single block transfer operation in response to the single EF handshake signal initiated by the data chain operation. For example, a master HSD(E) can utilize the **`readv(2)`** call to read data sent by a slave HSD(E) into a series of data buffers chained together. The slave HSD(E) must utilize a single **`write(2)`** call to transfer an entire block of data whose size is equivalent to the sum of the sizes of the data buffers utilized by the HSD(E) master. The HSD(E) master will initiate the first I/O operation in the chain by sending a request for enough data to fill its first associated buffer. The slave HSD(E) will respond by carrying out that request by sending the exact amount of data from its total transfer buffer. The master HSD(E) will then issue the next I/O request in its data chain and the slave will respond in kind. This operation will continue until the master's requests have been completed.

The **hsde** emulates only the "Function 1" type data chaining described in the Encore HSD protocol specification. With this type of data chaining, the next block of data represented by an IOCB is not processed until an on–board data buffer (known as the First In First Out buffer or FIFO) has emptied after a preceding I/O operation. In this way, the PowerMAX OS system is synchronized with the **hsde** device interface on a data block by data block basis.

Example programs that demonstrate use of data chaining to transfer simple data from a slave HSDE to a master HSDE are included in Appendix E.

CAUTION

Although the **hsde** device supports data chaining, not all HSD–specific devices to which it may be connected may support data chaining. Consult available documentation provided with the other HSD device to determine if it does support data chaining.

Using a DR11W Emulator

An external device that uses the Digital Equipment Corporation DR11W interface can be connected to a PowerMAX OS system via the DR11W emulator. This emulator and its associated driver are designed to provide improved performance for related real–time applications. The DR11W emulator performs DMA transfers directly to and from the user's virtual address space, provides three attention interrupt notification mechanisms, and supports asynchronous I/O operations.

Use of the DR11W emulator is supported by the DR11W user–level device driver. The DR11W user–level device driver enables you to access a DR11W emulator directly from user space. It consists of a library of routines that allow you to perform a variety of operations and a configuration program that performs the initialization of the DR11W device necessary for the user's application.

An overview of the DR11W emulator, **dr11w**, is presented in "Understanding the DR11W Emulator." An overview of the DR11W user–level device driver is provided in "Understanding the DR11W User-Level Device Driver." Configuration requirements are described in "Configuration and Installation Requirements." An introduction to the user interface is presented in "Understanding the User Interface." Procedures for using the driver routines are explained in "Using the Driver Routines."

In order to use the DR11W emulator and its associated user–level device driver, you must ensure that the user's I/O buffer is bound to a contiguous section of physical memory. An explanation of the procedures for complying with memory mapping requirements is presented in "Memory Mapping for HSDE and DR11W."

Understanding the DR11W Emulator

The DR11W emulator, **dr11w**, is a high–speed 16–bit parallel DMA interface between a PowerMAX OS system and a device that uses the Digital Equipment Corporation DR11W interface, including another DR11W emulator.

On all PowerMAX OS systems, the DR11W emulator is located on an optional board that plugs into an (H)VME bus. Up to eight DR11W emulators are allowed on each system. They may be located on one or both of the (H)VME buses.

Device special files for the DR11W emulators have names of the form **`/dev/ud/dr11w`***n*, where *n* specifies a particular **dr11w** and ranges from zero to seven. Each emulator supports only one device; consequently, the value of *n* is also the minor number of the **dr11w** device.

All DR11W emulators are associated with the same interrupt level. If more than one is present, they are processed in first–come, first–served order (**fifo**).

The **dr11w** supports three DMA transfer modes. These modes are described as follows:

External device mode	instructs the **dr11w** to wait for the attached device's DMA cycle request signal before initiating the data transfer
Interprocessor link mode	instructs the **dr11w** to initiate a data transfer only during a write operation.Setting this mode prevents the loss or corruption of data when the **dr11w** is connected to another DR11W emulator.
DMA mode	instructs the **dr11w** to initiate the data transfer for both read and write operations. This mode may be used for loop–back DMA testing.

Understanding the DR11W User-Level Device Driver

The DR11W user–level device driver has several features that are designed to increase the real–time effectiveness of a DR11W emulator. Features of the DR11W emulator that are supported by the user–level driver are the capability to perform DMA transfers directly to and from the user's virtual address space, provision for two I/O completion and attention interrupt notification mechanisms, and support for asynchronous I/O operations. The size of a DMA transfer can range from a minimum of 2 bytes to a maximum of 32 megabytes. The DR11W user–level driver provides a means of performing device I/O operations without having to enter and exit the kernel, and it provides support for performing basic device and driver control operations.

Because of restrictions imposed by the hardware, the DR11W user–level device driver does not support the following DR11W emulator I/O modes:

- Control over incrementing the bus address and range counters
- Byte DMA transfers
- Change of transfer direction within a DMA block
- DMA transfers that are not aligned on a word boundary or are not physically contiguous

A user–level process's use of the DR11W user–level driver is bound by the following restrictions:

- To use a DR11W emulator, the user's I/O buffer must be bound to a contiguous section of physical memory. An explanation of the procedures for complying with memory mapping requirements is presented in "Memory Mapping for HSDE and DR11W."
- To use the DR11W user–level device driver, you must have the P_PLOCK, P_SHMBIND, and P_USERINT privileges (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **`intro(2)`** system manual page).
- The DR11W user–level driver ensures that only one process has access to a particular DR11W emulator at a time.
- If a user process makes a **`fork(2)`** system call after opening a DR11W emulator that is controlled by the user–level driver, the child process should <u>not</u> attempt to access the DR11W user–level device driver.
- An application program that uses the DR11W user–level device driver is permitted to initiate only one asynchronous I/O request at a time.

Configuration and Installation Requirements

Use of the DR11W user-level device driver requires that the user-level interrupt module (**`ui`**) be configured into the kernel. You can ensure that this module is configured into your kernel by using the **`config(1M)`** utility. Note that after configuring a module, you must rebuild the kernel and then reboot your system. For an explanation of the procedures for using **config(1M)**, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*.

Before using a DR11W emulator, you must ensure that the dr11w package is installed on your system. For an explanation of the procedures for installing software packages, refer to the *PowerMAX OS Version 2.2 Release Notes* and the **pkgadd(1M)** man page.

When the **dr11w** package is installed, you will be prompted to enter the number of DR11W emulators to be configured in the primary and secondary (H)VME buses, respectively. A **dr11w** script that is based on your responses is built and placed in the **/etc/dinit.d** directory. This script contains calls to the DR11W user-level driver configuration program, **dr11wconfig(1M)**. When your system is rebooted, **dr11wconfig** will automatically be invoked from this script for each configured DR11W emulator.

The driver configuration program has a set of standard options. The functions associated with each option are described as follows:

-c	create the shared memory segments required by the driver and initialize the device.
–i	create the user–level interrupt process
–r	reset the device
–d	display debug and status information
–x	remove the association of the user–level driver to the device, and restore the device to its initial state

The **-c** and the **-i** options are specified in the script. You may specify the **-r**, **-d**, and **-x** options by invoking **dr11wconfig** from the command line.

The DR11W user–level device driver uses the operating system's support for user–level interrupt routines (the user–level interrupt routine facility is fully described in Chapter 8 of this guide). Specifying the **–i** option creates a user–level interrupt process that connects an interrupt handling routine to an interrupt vector that corresponds to the interrupt generated by a DR11W emulator. The user–level interrupt process is responsible for servicing device interrupts for the DR11W user–level device driver routines.

The user–level interrupt process will continue to block and service interrupts until it is disconnected from the interrupt vector. You can disconnect it by invoking the **uistat(1)** command from the shell and specifying the **–d** option (see Chapter 8 for an explanation of the procedures for using this command). In order to determine the interrupt vector to disconnect, use the **dr11wconfig** program with the **–d** option to display the DR11W user–level device driver information; this information will include the interrupt vector of the device in one of the fields. The user–level interrupt process is automatically disconnected when you use the **dr11wconfig** program and specify the **–x** option to remove the association of the DR11W device to a DR11W user–level device driver. You should not attempt to kill the user–level interrupt process. When the user–level interrupt process is disconnected from the interrupt vector, the DR11W user–level driver will not be able to clean up its internal state information if the SIGKILL signal is pending.

Refer to the system manual pages **dr11wconfig(1M)** and **uistat(1)** for further description of these commands.

Understanding the User Interface

The DR11W emulator is controlled by using the routines contained in the DR11W user–level driver library, **/usr/lib/libdr11w.a**. These routines enable you to open and close a device, perform asynchronous I/O operations, and perform a variety of control operations that are specific to the device. Each routine in the library is explained in detail in "Using the Driver Routines." Application requirements are described in "Application Requirements." Compiling and linking procedures are explained in "Compiling and Linking Procedures."

Application Requirements

To use the DR11W user–level device driver, you must have the P_PLOCK, P_SHMBIND, and P_USERINT privileges. To use the structure definitions for the DR11W user–level driver, you must include **<ud/dr11w.h>** in your application program.

To use the DR11W user-level device driver with your application, you must invoke the **udbufalloc(3X)** library routine to obtain a description of the physical memory associated with the user I/O buffer. This routine invokes the **userdma(2)** system call. As a result, to use this routine you must have the P_PLOCK privilege. Although the user buffer is locked in memory upon return from **udbufalloc**, this locking is not necessary because you are required to have previously reserved a contiguous section of physical memory and bound the I/O buffer to it. Procedures for using **udbufalloc** are explained in *Device Driver Programming*.

Compiling and Linking Procedures

To use the DR11W user–level device driver, you must statically link the following libraries to the application:

```
/usr/lib/libdr11w.a

/usr/lib/libud.a
```

To compile and statically link a C program, the command line instruction is as follows:

```
cc source_file.c -Zlink=static –ldr11w –lud
```

For additional information, refer to the system manual pages **ld(1)** and **cc(1)** and the "Link Editor and Linking" chapter in *Compilation Systems Volume 1 (Tools)*. Refer to the appropriate language reference manual for the procedures for calling C routines from programs written in other languages.

Using the Driver Routines

The DR11W user–level driver routines provide access to a DR11W emulator. They enable you to perform such basic operations as the following: (1) open and close a DR11W emulator; (2) perform asynchronous read and write operations; (3) poll or wait for completion of an I/O request; (4) poll or wait for occurrence of an attention interrupt; (5) enable and disable interrupts; and (6) perform a variety of control operations that are specific to the device.

In the sections that follow, all of the driver routines contained in the **`libdr11w`** library are presented in alphabetical order. Figure 13-1 illustrates the approximate order in which you might invoke some of the basic routines from an application program.

dr11w_acheck

The **`dr11w_acheck`** routine allows a user process to obtain the status of an asynchronous I/O operation. It is called if the user process wishes to poll rather than wait for completion of an I/O request.

Specification

```
int dr11w_acheck(dr11w, req_id, count)

int dr11w;
int req_id;
int *count;
```

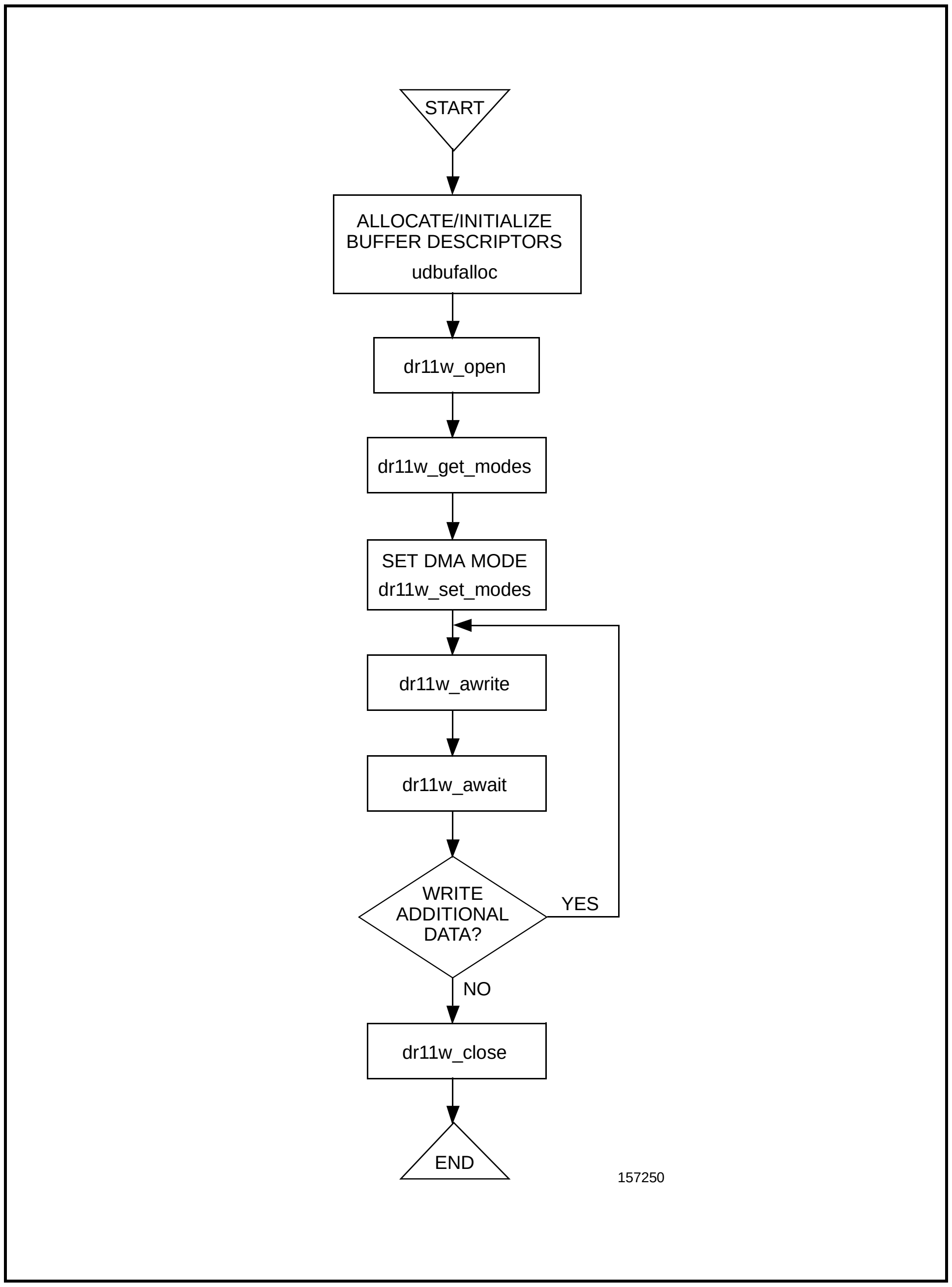


Figure 13-1. Library Call Sequence for Driver Routines

Parameters

dr11w	the identifier for the DR11W emulator to or from which the asynchronous I/O operation is being performed. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).
req_id	the request identifier of the asynchronous I/O operation for which the status is being requested. This identifier is allocated by the driver if a pointer is

supplied on a call to the **`dr11w_aread`** or **`dr11w_awrite`** routine (see "dr11w_aread" and "dr11w_awrite" for explanations of these routines).

count	a pointer to the location to which the number of bytes transferred by the specified I/O operation is returned

Return Value

The **`dr11w_acheck`** routine returns the following error codes:

EUD_NOERROR	The specified asynchronous I/O operation has been completed.
EUD_INPROGRESS	The operation has not been completed.
EUD_INVAL	An I/O operation has not been initiated.
EUD_INTR	A reset occurred to the board.

dr11w_aread

The **`dr11w_aread`** routine allows a user process to perform an asynchronous read of data from a particular DR11W emulator.

Specification

```
int dr11w_aread(dr11w, udbuf, count, req_id)

int      dr11w;
udbuf_t  *udbuf;
int      count;
int      *req_id;
```

Parameters

dr11w	the identifier for the DR11W emulator from which data are to be read. This identifier is allocated on a call to the **`dr11w_open`** routine (see "dr11w_open" for an explanation of this routine).
udbuf	a pointer to the user I/O buffer that describes the physical locations into which data are to be read
count	the number of bytes to be read
req_id	a pointer to the location to which the request identifier of the asynchronous read operation is returned.

The user process can use *req_id* to obtain the status of the operation. If *req_id* contains **NULL**, information about the status of the request is not maintained, and you may not use **`dr11w_acheck`** or **`dr11w_await`** to obtain the completion status. If *req_id* contains a pointer, you must perform a **`dr11w_acheck`** or **`dr11w_await`** using the *req_id* identifier (see "dr11w_attn_check" and "dr11w_await" for explanations of these routines). If interrupts are not enabled, you must provide a *req_id* pointer, and you must call the **`dr11w_acheck`** routine.

Return Value

The **`dr11w_aread`** routine returns the following error codes:

EUD_NOERROR	The DMA request has been successfully initiated.
EUD_INVAL	Completion status has not been requested when interrupts have been disabled, or a transfer of less than one word (two bytes) has been requested. Not requesting a completion status is a valid option.
EUD_IOREQ	The number of DMA requests has exceeded the limits imposed by the driver. The driver is limited to one outstanding DMA request at a time.
EUD_BUFFER	The transfer buffer is not contiguous; the transfer count is greater than the buffer; or an invalid buffer address has been specified by the user buffer descriptor.

dr11w_attn_check

The **`dr11w_attn_check`** routine allows a user process to check for an occurrence of an attention interrupt since the last call to **`dr11w_attn_check`** or **`dr11w_attn_wait`** (see “dr11w_await” for an explanation of the **`dr11w_attn_wait`** routine).

Specification

```
int dr11w_attn_check(dr11w, intr_occurred)

int dr11w;
int *intr_occurred
```

Parameters

dr11w	the identifier for the DR11W emulator you wish to check for the occurrence of an attention interrupt. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).
intr_occurred	a pointer to a flag that indicates whether or not an attention interrupt has occurred. The flag is set to TRUE if an attention interrupt has occurred since the last call to either **`dr11w_attn_check`** or **`dr11w_attn_wait`** (see “dr11w_attn_wait” for an explanation of the **`dr11w_attn_wait`** routine).

Return Value

The **`dr11w_attn_check`** routine returns **EUD_NOERROR**. It also returns TRUE or FALSE to the location referenced by the *intr_occurred* parameter.

dr11w_attn_wait

The **`dr11w_attn_wait`** routine allows a user process to wait for an attention interrupt. This routine blocks the calling process until the attention interrupt occurs, or it returns immediately if an attention interrupt has occurred since the last call to **`dr11w_attn_check`** or **`dr11w_attn_wait`**.

A process waiting for the occurrence of an attention interrupt will not be wakened if a DMA completion interrupt occurs. The DMA completion status will be saved for a future call to the **`dr11w_acheck`** or **`dr11w_await`** routine (see “Specification” and “dr11w_await” for explanations of these routines).

Specification

```
int dr11w_attn_wait(dr11w, intr_occurred)

int          dr11w;
int          *intr_occurred;
```

Parameters

dr11w	the identifier for the DR11W emulator whose attention interrupt you wish to await. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).
intr_occurred	a pointer to a flag that indicates whether or not an attention interrupt has occurred. The flag is set to TRUE if an attention interrupt has occurred since the last call to either **`dr11w_attn_check`** or **`dr11w_attn_wait`** (see “dr11w_attn_check” for an explanation of the **`dr11w_attn_check`** routine).

Return Value

The **`dr11w_attn_wait`** routine returns the following error codes:

EUD_NOERROR	An attention interrupt has been detected.
EUD_INVAL	Interrupts have been disabled.
EUD_INTR	The process was interrupted by an external event such as the removal of the interrupt routine or removal of the association of the user–level driver to the device.

This routine also returns TRUE or FALSE to the location referenced by the *intr_occurred* parameter.

dr11w_await

The **`dr11w_await`** routine allows a user process to wait for a pending asynchronous I/O operation to be completed.

Specification

```
int dr11w_await(dr11w, req_id, count)

int dr11w;
int req_id;
int *count;
```

Parameters

dr11w	the identifier for the DR11W emulator to or from which the asynchronous I/O operation is being performed. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).
req_id	the request identifier of the asynchronous I/O operation for which the process is waiting. This identifier is allocated by the driver if a pointer is supplied on a call to the driver’s **`dr11w_aread`** or **`dr11w_awrite`** routine (see “dr11w_aread” and “dr11w_awrite” for explanations of these routines).
count	a pointer to the location to which the number of bytes transferred by the specified I/O operation is returned

Return Value

The **`dr11w_await`** routine returns the following error codes:

EUD_NOERROR	The specified asynchronous I/O operation has been completed.
EUD_INVAL	An I/O operation has not been initiated.
EUD_INTR	The process was interrupted by an external event such as the removal of the interrupt routine or removal of the association of the user–level driver to the device.
EUD_NOINTR	The user–level interrupt process is not available.
EUD_INPROGRESS	A DMA is still in progress

dr11w_awrite

The **`dr11w_awrite`** routine allows a user process to perform an asynchronous write of data to a particular DR11W emulator.

Specification

```
int dr11w_awrite(dr11w, udbuf, count, req_id)

int      dr11w;
udbuf_t  *udbuf;
int      count;
int      *req_id;
```

Parameters

dr11w	the identifier for the DR11W to which data are to be written. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).
udbuf	a pointer to the user I/O buffer that describes the physical locations from which data are to be written
count	the number of bytes to be written
req_id	a pointer to the location to which the request identifier of the asynchronous write operation is returned.

The user process can use *req_id* to obtain the status of the operation. If *req_id* contains NULL, information about the status of the request is not maintained, and you may not use **`dr11w_acheck`** or **`dr11w_await`** to get the completion status. If *req_id* contains a pointer, you must perform a **`dr11w_acheck`** or **`dr11w_await`** using the *req_id* identifier (see “Specification” and “dr11w_await” for explanations of these routines). If interrupts are not enabled, you must provide a *req_id* pointer, and you must call the **`dr11w_acheck`** routine.

Return Value

The **`dr11w_awrite`** routine returns the following error codes:

EUD_NOERROR	The DMA request has been successfully initiated.
EUD_INVAL	Completion status has not been requested when interrupts have been disabled, or a transfer of less than one word (two bytes) has been requested. Not requesting a completion status is a valid option.
EUD_IOREQ	The number of DMA requests has exceeded the limits imposed by the driver. The driver is limited to one outstanding DMA request at a time.
EUD_BUFFER	The transfer buffer is not contiguous; the transfer count is greater than the buffer; or an invalid buffer address has been specified by the user buffer descriptor.
EUD_INPROGRESS	A DMA is still in progress

dr11w_close

The **`dr11w_close`** routine allows a user process to close a DR11W emulator that has been opened in preparation for I/O or control operations. The **`close`** routine does not return until pending I/O operations have been completed.

The identifier for a particular DR11W emulator allocated by the driver should not be used after the **`dr11w_close`** routine has been invoked. A user–level device driver has no way to prevent access to the device after the **`dr11w_close`** call; unauthorized access to the device can produce unexpected results.

Specification

```
int dr11w_close(dr11w)

int dr11w;
```

Parameter

dr11w	the identifier for the DR11W emulator for which the close operation is being performed

Return Value

The **dr11w_close** routine returns the following error codes:

EUD_NOERROR	The close operation has been successfully completed.
EUD_BADD	An invalid device identifier has been provided.
EUD_INPROGRESS	A DMA is still in progress.

dr11w_disable_interrupts

The **dr11w_disable_interrupts** routine allows a user process to disable interrupts on a particular DR11W emulator. Interrupts will not be disabled while a DMA operation is in progress or a process is blocked waiting for an interrupt to occur.

Specification

```
int dr11w_disable_interrupts(dr11w)

int          dr11w;
```

Parameters

dr11w	the identifier for the DR11W emulator whose interrupts are to be disabled. This identifier is allocated on a call to the **dr11w_open** routine (see “dr11w_open” for an explanation of this routine).

Return Value

The **dr11w_disable_interrupts** routine returns the following error codes:

EUD_NOERROR	Interrupts have been successfully disabled.
EUD_INPROGRESS	A DMA operation is in progress. (Interrupts cannot be disabled while a DMA operation is in progress.)

dr11w_dump

The **dr11w_dump** routine allows a user process to obtain the current values of the registers associated with a particular DR11W emulator.

Specification

```
int dr11w_dump(dr11w, dumpptr)

int          dr11w;
dr11w_dump_t *dumpptr;
```

Parameters

dr11w the identifier for the DR11W emulator whose registers are to be dumped. This identifier is allocated on a call to the **dr11w_open** routine (see “dr11w_open” for an explanation of this routine).

dumpptr A pointer to a structure to which the current values of the registers associated with the specified device are returned.

The fields defined in the **dr11w_dump** data structure are as follows:

```
unsigned short  d_sr;/* status register (SR) */
unsigned short  d_data;/* DMA data register (input/output) */
unsigned char   d_modifier;/* VME address modifier register */
unsigned char   d_vector;/* interrupt vector register */
unsigned short  d_dma_addr_lo;/* low word of DMA addr */
unsigned short  d_dma_addr_hi;/* high word of DMA addr */
unsigned short  d_dma_range_lo;/* low word of DMA range (xfer) count */
unsigned short d_dma_range_hi;/* high byte of DMA range (xfer) count */
```

Return Value

The **dr11w_dump** routine does not return an error code.

dr11w_enable_interrupts

The **dr11w_enable interrupts** routine allows you to enable interrupts on a particular DR11W emulator. Note that you must have previously initialized the user–level interrupt process by executing the **dr11wconfig** program and specifying the **-i** option (see “Configuration and Installation Requirements” for an explanation of this option). Interrupts cannot be enabled while a DMA operation is in progress.

Specification

```
int dr11w_enable_interrupts(dr11w)

int          dr11w;
```

Parameters

dr11w the identifier for the DR11W emulator whose interrupts are to be enabled. This identifier is allocated on a call to the **dr11w_open** routine (see “dr11w_open” for an explanation of this routine).

Return Value

The **dr11w_enable_interrupts** routine returns the following error codes:

EUD_NOERROR	Interrupts have been successfully enabled.

EUD_NOINTR	The user–level interrupt process is not available.
EUD_INPROGRESS	A DMA operation is in progress. (Interrupts cannot be enabled while a DMA operation is in progress.)

dr11w_get_modes

The **`dr11w_get_modes`** routine allows you to obtain the value of the DMA transfer mode associated with a particular DR11W emulator.

Specification

```
int dr11w_get_modes(dr11w, mode)

int            dr11w;
dr11w_modes_t *mode;
```

Parameters

dr11w the identifier for the DR11W emulator for which the current DMA transfer mode is to be returned. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).

mode a pointer to a structure to which the value of the DMA transfer mode currently associated with the specified device is returned.

Return Value

The **`dr11w_get_modes`** routine returns **EUD_NOERROR**. The DMA transfer mode information is returned to the location referenced by the *mode* parameter (see “dr11w_set_modes” for a description of the different modes).

dr11w_get_status

The **`dr11w_get_status`** routine allows you to obtain the current values of the status bits associated with a particular DR11W emulator.

Specification

```
int dr11w_get_status(dr11w, statusptr)

int dr11w;
dr11w_status_t *statusptr;
```

Parameters

dr11w the identifier for the DR11W emulator for which the values of the status bits are to be returned. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).

statusptr a pointer to a structure to which the current values of the status bits associated with the specified device are returned.

The **`dr11w_status`** data structure is defined as follows:

```
union dr11w_status {
    struct {
        unsigned statusA : 1;
        unsigned statusB : 1;
        unsigned statusC : 1;
        unsigned attentionH : 1;
    } s_status;

    struct {
        unsigned function1 : 1;
        unsigned function2 : 1;
        unsigned function3 : 1;
    } f_status;
}
```

The `status` fields should be accessed only after using the **`dr11w_get_status`** routine; they represent the DR11W's status bits as set by the attached device. Conversely, the `function` bits should be initialized prior to using **`dr11w_set_status`**; they represent the values sent to the attached device's status lines/bits. Using these fields in any other manner may cause unpredictable behavior.

Return Value

The **`dr11w_get_status`** routine returns **EUD_NOERROR**. The values of the status bits are returned to the location referenced by the *statusptr* parameter.

dr11w_ienabled

The **`dr11w_ienabled`** routine allows a user process to determine whether or not interrupts are enabled on a particular DR11W emulator.

Specification

```
int dr11w_ienabled(dr11w)

int          dr11w;
```

Parameters

dr11w	the identifier for the DR11W emulator. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).

Return Value

The **`dr11w_ienabled`** routine returns TRUE if interrupts are enabled; it returns FALSE if interrupts are not enabled.

dr11w_open

The **`dr11w_open`** routine allows a user process to open a DR11W emulator in preparation for I/O or control operations.

Specification

```
int dr11w_open(dr11w_ptr, path, oflags)

int         *dr11w_ptr;
char        *path;
int         oflags;
```

Parameters

dr11w_ptr	a pointer to the location to which an identifier for the opened DR11W emulator is returned. This identifier is allocated by the user–level device driver.
path	a pointer to the path name of the device special file associated with the DR11W emulator to be opened.
oflags	driver status flag. The following flag may be specified:

UD_DEBUG	Indicates that the specified device is to be opened for debugging purposes. In this mode of operation, the DR11W user–level device driver prints any messages regarding errors in the library routines as they occur.

Return Value

The **`dr11w_open`** routine returns the following error codes:

EUD_NOERROR	The device has been successfully opened.
EUD_NODEV	An invalid path name has been provided.
EUD_RESOURCE	The maximum number (currently 8) of open DR11W devices in the process has been reached; additional devices cannot be opened.
EUD_SHMID	A shared memory identifier is not available for the specified path name because the device has not been created by the **`dr11wconfig`** program.
EUD_BUSY	The device is busy (that is, another process has opened it); access is denied. If the device is hung and another process does not have it open, the **`dr11wconfig`** program should be used to reset the device.
EUD_SHMAT	Attachment of the shared memory segment has failed, possibly because the user may not have permission to access the shared memory segments.
EUD_INIT	Initialization of the driver by the **`dr11wconfig`** program has failed, and the driver is not initialized properly. Run the **`dr11wconfig`** program with the **`–x`** option, and then run it again with the **`–c`** option.
EUD_SPLMAP	The process is unable to map the SPL register.

EUD_MEMLOCK	Unable to lock pages of the DR11W user–level device driver library into memory.
EUD_SHMLOCK	Unable to lock the shared memory segments.
EUD_CREAT	The shared memory segment for the DR11W status buffer already exists.
EUD_SHMBIND	Shared memory bind to board registers failed.
EUD_IO	The DR11W emulator did not respond to a probe.

dr11w_pio_read

The **`dr11w_pio_read`** routine allows a user process to read the value in a DR11W emulator's date input register.

Specification

```
int dr11w_pio_read(dr11w, data)

int           dr11w;
unsigned short *data
```

Parameters

dr11w	the identifier for the DR11W emulator whose data input register is to be read. This identifier is allocated on a call to the **`dr11w_open`** routine (see "dr11w_open" for an explanation of this routine).
data	a pointer to the location to which the value of the specified device's data input register is returned.

Return Value

The **`dr11w_pio_read`** routine returns the following error codes:

EUD_NOERROR	The read is successful.
EUD_IO	A DMA operation is in progress.

dr11w_pio_write

The **`dr11w_pio_write`** routine allows you to write a value to a DR11W emulator's data output register.

Specification

```
int dr11w_pio_write(dr11w, data)

int            dr11w;
unsigned short *data;
```

Parameters

dr11w	the identifier for the DR11W emulator whose data output register is to be written. This identifier is allocated on a call to the **`dr11w_open`** routine (see "dr11w_open" for an explanation of this routine).
data	a pointer to the location that contains the value to be written to the specified device's data output register.

Return Value

The **`dr11w_pio_write`** routine returns the following error codes:

EUD_NOERROR	The write is successful.
EUD_IO	A DMA operation is in progress, and a write to the data output register will disrupt the DMA operation.

dr11w_reset

The **`dr11w_reset`** routine allows a user process to reset the DR11W hardware. It performs a master clear of the device and verifies that the interrupt vector and VME modifiers are set correctly. This routine also clears up any pending DMA completion status and halts any DMA operation in progress. It sends an INIT H signal to the external device. If the user–level interrupt process is available, interrupts are enabled. This routine also initializes all of the data structures that are maintained by the driver.

Specification

```
int dr11w_reset(dr11w)

int            dr11w;
```

Parameters

dr11w	the identifier for the DR11W emulator that is to be reset. This identifier is allocated on a call to the **`dr11w_open`** routine (see "dr11w_open" for an explanation of this routine).

Return Value

This **`dr11w_reset`** routine returns **EUD_NOERROR**.

dr11w_sendgo

The **`dr11w_sendgo`** routine allows a user process to send a GO signal to the attached external device and to enable DMA transfers.

Specification

```
int dr11w_sendgo(dr11w)

int          dr11w;
```

Parameters

dr11w	the identifier for the DR11W emulator from which a GO signal is to be sent. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).

Return Value

The **`dr11w_sendgo`** routine returns **EUD_NOERROR**. The routine returns **EUD_WOULDBLOCK** if a DMA operation is in progress.

dr11w_sendintr

The **`dr11w_sendintr`** routine allows a user process to send an attention interrupt to the attached external device.

Specification

```
int dr11w_sendintr(dr11w)

int          dr11w;
```

Parameters

dr11w	the identifier for the DR11W emulator from which an attention interrupt is to be sent. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).

Return Value

The **`dr11w_sendintr`** routine returns **EUD_NOERROR**. The routine returns **EUD_WOULDBLOCK** if a DMA operation is in progress.

dr11w_set_modes

The **`dr11w_set_modes`** routine allows a user process to set the DMA transfer modes for a particular DR11W emulator. The three valid modes for the DR11W user–level device driver are as follows:

DR11W_LINKMODE	instructs the **`dr11w`** to initiate a data transfer only during a write operation. Setting this mode prevents the loss or corruption of data when the **`dr11w`** is connected to another DR11W emulator.
DR11W_EXTDEVMODE	instructs the **`dr11w`** to wait for the attached device's DMA cycle request signal before initiating the data transfer
DR11W_STARTDMAMODE	instructs the **`dr11w`** to initiate the data transfer for both read and write operations. This mode may be used for loop–back DMA testing.

Specification

```
int dr11w_set_modes(dr11w, mode)

int dr11w;
dr11w_modes_t *mode;
```

Parameters

dr11w	the identifier for the DR11W emulator for which the DMA transfer mode is to be set. This identifier is allocated on a call to the **`dr11w_open`** routine (see "dr11w_open" for an explanation of this routine).
mode	a pointer to a structure that contains the value to which the DMA transfer mode for the specified device is to be set.

Return Value

The **`dr11w_set_modes`** routine returns **EUD_NOERROR**. The routine returns **EUD_TRANSMODE** if a bad mode value is specified.

dr11w_setsdir

The **`dr11w_setsdir`** routine allows a user process to set the value of the transfer direction control bit associated with a particular DR11W emulator.

Specification

```
    int dr11w_setsdir(dr11w, sdirptr)

    int           dr11w;
    dr11w_sdir_t  *sdirptr
```

Parameters

dr11w the identifier for the DR11W emulator for which the transfer direction control bit is to be set. This identifier is allocated on a call to the **`dr11w_open`** routine (see “dr11w_open” for an explanation of this routine).

sdirptr a pointer to a structure that contains the value to which the transfer direction control bit for the specified device is to be set.

The **`dr11w_sdir`** data structure is defined as follows:

```
struct dr11w_sdir {
    unsigned sw_xferdir : 1;
};
```

The valid values for the `sw_xferdir` field are as follows:

```
    #define  DR11W_READ     0x01
    #define  DR11W_WRITE    0x00
```

This command is typically used only when the incoming C1 CNTL control signal from the attached device is not stable and the function 1 bit (see **`dr11w_get_status`** in “dr11w_get_status”) is being used for another purpose.

Return Value

The **`dr11w_setsdir`** routine returns **EUD_NOERROR**. The routine returns **EUD_WOULDBLOCK** if a DMA operation is in progress.

dr11w_set_status

The **`dr11w_set_status`** routine allows a user process to set the values of the function bits associated with a particular DR11W emulator.

Specification

```
    int dr11w_set_status(dr11w, statusptr)

    int dr11w;
    dr11w_status_t *statusptr
```

Parameters

dr11w the identifier for the DR11W emulator whose function bits are to be set. This identifier is allocated on a call to the **`dr11w_open`** routine (see "dr11w_open" for an explanation of this routine).

statusptr a pointer to a structure that contains the values to which the function bits associated with the specified device are to be set.

Refer to "dr11w_get_status" for the definition of the **`dr11w_status`** data structure.

Return Value

The **`dr11w_set_status`** routine returns **EUD_NOERROR**. The routine returns **EUD_WOULDBLOCK** if a DMA operation is in progress.

Using the 1553 Advanced Bus Interface

This section contains information needed to use the 1553 Advanced Bus Interface (ABI). The 1553 Advanced Bus Interface (ABI) is a single-card, real-time MIL-STD-1553 interface between a PowerMAX OS system and a MIL-STD-1553 bus.

An overview of the 1553 ABI, **abi**, is presented in "Understanding the 1553 Advanced Bus Interface." A description of the user interface to the 1553 ABI is provided in "Understanding the User Interface." Procedures for using the 1553 ABI user–level device driver are provided in "Using the 1553 ABI User-Level Device Driver." Procedures for using the user-level driver routines are explained in "Using the 1553 ABI User-Level Driver Routines."

Understanding the 1553 Advanced Bus Interface

The 1553 ABI, **`abi`**, is a real-time interface between a PowerMAX OS system and a MIL-STD-1553 bus. The 1553 ABI provides the host system with advanced 1553 interface capabilities. These capabilities include independent and simultaneous bus controller simulation, remote terminal simulation, and 1553 bus monitoring. The 1553 ABI can simulate as many as 31 remote terminals, and it can perform two modes of bus monitoring: sequential monitoring and map monitoring. With sequential monitoring, raw 1553 traffic is stored in double buffers for real-time data recording and analysis. With map monitoring, linked-list data buffers are set up and polled for 1553 data. The 1553 ABI also has a wide range of event interrupt capabilities that are related to bus controller simulation, remote terminal simulation, monitoring, and system functions.

The 1553 ABI board uses a bit-slice microcomputer that has 8Kx40 bit microcode instructions containing programs customized for real-time 1553 applications. When installed in a host computer, the 1553 ABI appears as a contiguous block of virtual or physical 64Kx16 bit memory. This pseudo dual-port memory resides on the 1553 ABI and is mapped or connected to the host system for access by application programs.

On all PowerMAX OS systems, the 1553 ABI is located on an optional board that plugs into an (H)VME bus or a standard 6U VME backplane without the optional 9U carrier. Up to eight 1553 ABI boards are allowed on each system. They may be located on either of the (H)VME buses.

Device special files for the 1553 ABI have names of the form **`/dev/ud/abi`***n*, where *n* specifies a particular **abi** device and ranges from zero to seven.

All of the 1553 ABI boards are associated with the same interrupt level. The interrupt level is established by jumper settings on the board.

The 1553 ABI supports 16-bit word-aligned transfers only; it does not have the capability of performing odd-byte addressing.

A detailed overview of the 1553 ABI architecture and operations is provided in the *1553 ABI Reference Manual* that is shipped with the 1553 ABI board. Included are descriptions of the ABI memory organization; control registers; and functions and data structures associated with bus controller and remote terminal simulation and bus monitoring. Also included are the procedures for handling ABI interrupts. It is recommended that you carefully review this manual prior to using the 1553 ABI.

For on-line information about the 1553 ABI, refer to the system manual page **`1553abi(7)`**,

Understanding the User Interface

Use of the 1553 ABI is supported by the 1553 ABI user–level device driver and the 1553 ABI library.

The 1553 ABI user–level device driver enables you to access a 1553 ABI directly from user space. It consists of a library of routines that allow you to perform a variety of operations and a configuration program that performs the initialization of the 1553 ABI device necessary for the user's application. These routines are contained in the library**`/usr/lib/lib1553drv.a`**. Procedures for using the 1553 ABI user-level device driver are explained in "Using the 1553 ABI User-Level Device Driver." Each routine in the **`lib1553drv`** library is described in detail in "Using the 1553 ABI User-Level Driver Routines."

The 1553 ABI library is a set of C library routines that is supplied by the board manufacturer. This library has been ported to the PowerMAX OS by Concurrent. The following types of routines are included in this library: device management routines, remote terminal routines, bus controller routines, bus monitor routines, interrupt management routines, BIT (Built-in Test) management routines, and low-level routines. These routines are contained in the library **`/usr/lib/libabi.a`**. They are documented in the *1553 Interface Libraries* manual that is shipped with the 1553 ABI board.

Note that if you wish to use the 1553 ABI library, you must use it in conjunction with the 1553 user-level device driver.

Using the 1553 ABI User-Level Device Driver

Features of the 1553 ABI that are supported by the user-level device driver are the capability to perform basic device and driver-control operations; to read from and write to the 1553 ABI control registers and data structure area directly from the user's virtual address space; and to handle 1553 ABI event interrupts in two ways: by fielding the interrupt or by disabling interrupts on the 1553 ABI hardware and using software polling. When 1553 ABI interrupts are enabled, the user-level driver also provides the ability to wait for the occurrence of an event interrupt.

A user–level process's use of the 1553 ABI user–level driver is bound by the following restrictions:

- To use the 1553 ABI user–level device driver, you must have the P_PLOCK, P_SHMBIND, and P_USERINT privileges (for additional information on privileges, refer to the *PowerMAX OS Programming Guide* and the **`intro(2)`** system manual page).
- The 1553 ABI user–level driver attempts to ensure that only one process has access to a particular 1553 ABI device at a time. The driver will not operate correctly when commands are issued simultaneously from more than one source.
- If a user process makes a **`fork(2)`** system call after opening a 1553 ABI that is controlled by the user–level driver, the child process should not attempt to access the 1553 ABI user–level device driver. A multithreaded process should ensure that only one thread issues commands to the 1553 ABI at any time.

Configuration and installation requirements are presented in "Configuration and Installation Requirements." Application requirements are described in "Application Requirements" (p. 13-45). Compiling and linking procedures are explained in "Compiling and Linking Procedures" (p. 13-46). The driver routines contained in the **`lib1553drv`** library are presented in "Using the 1553 ABI User-Level Driver Routines" (p. 13-46).

Configuration and Installation Requirements

Use of the 1553 ABI user-level device driver requires that the user-level interrupt module (**`ui`**) be configured into the kernel. You can ensure that this module is configured into your kernel by using the **`config(1M)`** utility. Note that after configuring a module, you must rebuild the kernel and then reboot your system. For an explanation of the procedures for using **`config(1M)`**, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*.

Before using a 1553 ABI, you must ensure that the 1553drv package is installed on your system. For an explanation of the procedures for installing software packages, refer to the *PowerMAX OS Version 2.2 Release Notes* and the **`pkgadd(1M)`** man page.

When the 1553drv package is installed, you will be prompted to enter the number of 1553 ABIs to be configured in the primary and secondary (H)VME buses, respectively. A 1553 ABI script that is based on your responses is built and placed in the **`/etc/dinit.d`** directory. This script contains calls to the 1553 ABI user-level driver configuration program, **abiconfig(1M)**. When your system is rebooted, **abiconfig** will automatically be invoked from this script for each configured 1553 ABI.

The driver configuration program has a set of standard options. The functions associated with each option are described as follows:

-c	create the shared memory segments required by the driver and initialize the device.
–i	create the user–level interrupt process
–r	reset the 1553 ABI
–d	display debug and status information
–x	remove the association of the user–level driver to the device, and restore the 1553 ABI to its initial state

The **-c** and the **-i** options are specified in the script. You may specify the **-r**, **-d**, and **-x** options by invoking **abiconfig** from the command line.

The 1553 ABI user–level device driver uses the operating system's support for user–level interrupt routines (the user–level interrupt routine facility is fully described in Chapter 8 of this guide). Specifying the **–i** option creates a user–level interrupt process that connects an interrupt handling routine to an interrupt vector that corresponds to the interrupt generated by a 1553 ABI. The user–level interrupt process is responsible for servicing event interrupts for the 1553 ABI user–level device driver routines.

The user–level interrupt process will continue to block and service interrupts until it is disconnected from the interrupt vector. You can disconnect it by invoking the **uistat(1)** command from the shell and specifying the **–d** option (see the "Viewing User-Level Interrupt Connections" section of Chapter 8 for an explanation of the procedures for using this command). In order to determine the interrupt vector to disconnect, use the **abiconfig** program with the **–d** option to display the 1553 ABI user–level device driver information; this information will include the interrupt vector of the device in one of the fields. The user–level interrupt process is automatically disconnected when you use the **abiconfig** program and specify the **–x** option to remove the association of the 1553 ABI to a 1553 ABI user–level device driver. You should <u>not</u> attempt to kill the user–level interrupt process. When the user–level interrupt process is disconnected from the interrupt vector, the 1553 ABI user–level driver will not be able to clean up its internal state information if the SIGKILL signal is pending.

Refer to the system manual pages **abiconfig(1M)** and **uistat(1)** for further description of these commands.

Application Requirements

To use the 1553 ABI user–level device driver, you must have the P_PLOCK, P_SHMBIND, and P_USERINT privileges. To use the structure definitions for the 1553 ABI user–level driver, you must include **<abi.h>** in your application program.

The 1553 ABI supports one type of interrupt: the event interrupt that notifies the processor of bus controller, remote terminal, bus monitoring, and system events that have occurred on the 1553 bus. The 1553 ABI manages interrupts by using double-buffered queues and interrupt control registers. You may determine that an interrupt has occurred by using the user-level device driver's **`abi_pio_read`** () routine (see p. 13-52) to poll the interrupt control registers with the hardware interrupts disabled or by using the user-level device driver's interrupt service routine. In either case, to handle the interrupt, you must use the interrupt management routines contained in the 1553 ABI library (see p. 13-43) or write an interrupt-handling routine that is designed to meet the needs of your application.

Compiling and Linking Procedures

To use the 1553 ABI user–level device driver, you must statically link the following libraries to the application:

`/usr/lib/lib1553drv.a`

`/usr/lib/libud.a`

To compile and statically link a C program, the command line instruction is as follows:

```
cc source_file.c -Zlink=static –l1553drv –lud
```

Note that if you wish to use the 1553 ABI library with the user-level device driver, you must also statically link the **`/usr/lib/libabi.a`** library to your application.

For additional information, refer to the system manual pages **`ld(1)`** and **`cc(1)`** and the "Link Editor and Linking" chapter in *Compilation Systems Volume 1 (Tools)*. Refer to the appropriate language reference manual for the procedures for calling C routines from programs written in other languages.

Using the 1553 ABI User-Level Driver Routines

The 1553 ABI user-level driver routines provide access to a 1553 ABI. They enable you to perform such basic operations as the following: (1) open and close a 1553 ABI; (2) poll or wait for occurrence of an event interrupt; (3) enable and disable interrupts; and (4) perform a variety of control operations that are specific to the device.

In the sections that follow, all of the driver routines contained in the **`lib1553drv`** library are presented in alphabetical order.

abi_attn_check

The **`abi_attn_check`** routine allows a user process to check for an occurrence of an event interrupt since the last call to **`abi_attn_check`** or **`abi_attn_wait`** (see "abi_attn_wait" for an explanation of the **`abi_attn_wait`** routine).

Specification

```
int abi_attn_check(abi, intr_occurred)

int abi;
int *intr_occurred
```

Parameters

abi	the identifier for the 1553 ABI you wish to check for the occurrence of an event interrupt. This identifier is allocated on a call to the **`abi_open`** routine (see "abi_open" for an explanation of this routine).
intr_occurred	a pointer to a flag that indicates whether or not an event interrupt has occurred. The flag is set to TRUE if an event interrupt has occurred since the last call to either **`abi_attn_check`** or **`abi_attn_wait`** (see "abi_attn_wait" for an explanation of the **`abi_attn_wait`** routine).

Return Value

The **`abi_attn_check`** routine returns **EUD_NOERROR**. It also returns TRUE or FALSE to the location referenced by the *intr_occurred* parameter.

abi_attn_wait

The **`abi_attn_wait`** routine allows a user process to wait for an event interrupt. This routine blocks the calling process until the event interrupt occurs, or it returns immediately if an event interrupt has occurred since the last call to **`abi_attn_check`** or **`abi_attn_wait`**.

Specification

```
int abi_attn_wait(abi, intr_occurred)

int         abi;
int         *intr_occurred;
```

Parameters

abi	the identifier for the 1553 ABI whose event interrupt you wish to await. This identifier is allocated on a call to the **`abi_open`** routine (see "abi_open" for an explanation of this routine).

intr_occurred	a pointer to a flag that indicates whether or not an event interrupt has occurred. The flag is set to TRUE if an event interrupt has occurred since the last call to either **`abi_attn_check`** or **`abi_attn_wait`** (see "abi_attn_check" for an explanation of the **`abi_attn_check`** routine).

Return Value

The **`abi_attn_wait`** routine returns the following error codes:

EUD_NOERROR	An event interrupt has been detected.
EUD_INVAL	Interrupts have been disabled.
EUD_INTR	The process was interrupted by an external event such as the removal of the interrupt routine or removal of the association of the user–level driver to the device.

This routine also returns TRUE or FALSE to the location referenced by the *intr_occurred* parameter.

abi_close

The **`abi_close`** routine allows a user process to close a 1553 ABI that has been opened in preparation for I/O or control operations. The **`close`** routine does not return until pending I/O operations have been completed.

The identifier for a particular 1553 ABI allocated by the driver should not be used after the **`abi_close`** routine has been invoked. A user–level device driver has no way to prevent access to the device after the **`abi_close`** call; unauthorized access to the device can produce unexpected results.

Specification

```
int abi_close(abi)

int abi;
```

Parameter

abi	the identifier for the 1553 ABI for which the close operation is being performed

Return Value

The **`abi_close`** routine returns the following error codes:

EUD_NOERROR	The close operation has been successfully completed.
EUD_BADD	An invalid device identifier has been provided.
EUD_INPROGRESS	An operation is in progress.

abi_disable_interrupts

The **`abi_disable_interrupts`** routine allows a user process to disable interrupts on a particular 1553 ABI. Interrupts will not be disabled while a 1553 ABI operation is in progress or a process is blocked waiting for an interrupt to occur.

Specification

```
int abi_disable_interrupts(abi)

int           abi;
```

Parameters

abi the identifier for the 1553 ABI whose interrupts are to be disabled. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).

Return Value

The **`abi_disable_interrupts`** routine returns the following error codes:

EUD_NOERROR	Interrupts have been successfully disabled.
EUD_INPROGRESS	An operation is in progress. (Interrupts cannot be disabled while an operation is in progress.)

abi_dump

The **`abi_dump`** routine allows a user process to obtain the current values of the registers associated with a particular 1553 ABI.

Specification

```
int abi_dump(abi, dumpptr)

int           abi;
abi_dump_t    *dumpptr;
```

Parameters

abi the identifier for the 1553 ABI whose registers are to be dumped. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).

dumpptr A pointer to a structure to which the current values of the registers associated with the specified device are returned.

The fields defined in the **`abi_dump`** data structure are as follows:

```
unsigned int  d_csr;      /* control/status register (CSR) */
unsigned int  d_resp;     /* response to command register */
unsigned int  d_iqcnt1;   /* interrupt queue count register 1 */
unsigned int  d_m2ptr;    /* sequential monitor buffer 2 */
unsigned int  d_bccptr;   /* current BC program block pointer */
unsigned int  d_bclptr;   /* last BC program block pointer */
```

```
     unsigned int  d_bcervl;   /* last BC error table entry */
     unsigned int  d_brtcnt;   /* number retries for error */
     unsigned int  d_brtbus;   /* defines bus A or bus B */
     unsigned int  d_brtcmd;   /* last command retried */
     unsigned int  d_brtrtc;   /* actual number of retried messages */
     unsigned int  d_iqptr1;   /* offset to interrupt queue 1 */
     unsigned int  d_iqptr2;   /* offset to interrupt queue 2 */
     unsigned int  d_swtptr;   /* status response table pointer */
     unsigned int  d_atptr;    /* address table pointer */
     unsigned int  d_ftptr;    /* filter table pointer */
```

Return Value

The **`abi_dump`** routine does not return an error code.

abi_enable_interrupts

The **`abi_enable_interrupts`** routine allows you to enable interrupts on a particular 1553 ABI. Note that you must have previously initialized the user–level interrupt process by executing the **`abiconfig`** program and specifying the **`-i`** option (see “Configuration and Installation Requirements” for an explanation of this option). Interrupts cannot be enabled while a 1553 ABI operation is in progress.

Specification

```
     int abi_enable_interrupts(abi)

     int           abi;
```

Parameters

abi	the identifier for the 1553 ABI whose interrupts are to be enabled. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).

Return Value

The **`abi_enable_interrupts`** routine returns the following error codes:

EUD_NOERROR	Interrupts have been successfully enabled.
EUD_NOINTR	The user–level interrupt process is not available.
EUD_INPROGRESS	An operation is in progress. (Interrupts cannot be enabled while an operation is in progress.)

abi_ienabled

The **`abi_ienabled`** routine allows a user process to determine whether or not interrupts are enabled on a particular 1553 ABI.

Specification

```
int abi_ienabled(abi)

int         abi;
```

Parameters

abi	the identifier for the 1553 ABI. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).

Return Value

The **`abi_ienabled`** routine returns TRUE if interrupts are enabled; it returns FALSE if interrupts are not enabled.

abi_open

The **`abi_open`** routine allows a user process to open a 1553 ABI in preparation for I/O or control operations.

Specification

```
int abi_open(abi_ptr, path, oflags)

int         *abi_ptr;
char        *path;
int         oflags;
```

Parameters

abi_ptr	a pointer to the location to which an identifier for the opened 1553 ABI is returned. This identifier is allocated by the user–level device driver.
path	a pointer to the path name of the device special file associated with the 1553 ABI to be opened.
oflags	driver status flag. The following flag may be specified:

UD_DEBUG	Indicates that the specified device is to be opened for debugging purposes. In this mode of operation, the 1553 ABI user–level device driver prints any messages regarding errors in the library routines as they occur.

Return Value

The **`abi_open`** routine returns the following error codes:

EUD_NOERROR	The device has been successfully opened.
EUD_NODEV	An invalid path name has been provided.
EUD_RESOURCE	The maximum number (currently 8) of open 1553 ABIs in the process has been reached; additional devices cannot be opened.
EUD_SHMID	A shared memory identifier is not available for the specified path name because the device has not been created by the **`abiconfig`** program.
EUD_BUSY	The device is busy (that is, another process has opened it); access is denied. If the device is hung and another process does not have it open, the **`abiconfig`** program should be used to reset the device.
EUD_SHMAT	Attachment of the shared memory segment has failed, possibly because the user may not have permission to access the shared memory segments.
EUD_INIT	Initialization of the driver by the **`abiconfig`** program has failed, and the driver is not initialized properly. Run the **`abiconfig`** program with the **`-x`** option, and then run it again with the **`-c`** option.
EUD_SPLMAP	The process is unable to map the IPL register.
EUD_MEMLOCK	Unable to lock pages of the 1553 ABI user–level device driver library into memory.
EUD_SHMLOCK	Unable to lock the shared memory segments.
EUD_CREAT	The shared memory segment for the 1553 ABI status buffer already exists.
EUD_SHMBIND	The shared memory bind to board registers failed.
EUD_IO	The 1553 ABI did not respond to a probe.

abi_pio_read

The **`abi_pio_read`** routine allows a user process to read the value in a control register or data structure area on the 1553 ABI.

Specification

```
int abi_pio_read(abi, offset, data)

int           abi;
unsigned long offset;
unsigned short *data;
```

Parameters

abi	the identifier for the 1553 ABI whose control register or data structure area is to be read. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).
offset	an offset from the board’s base address to the location of the desired control register or data structure area on the 1553 ABI
data	a pointer to the location to which the value of the specified device’s control register or data structure area is returned.

Return Value

The **`abi_pio_read`** routine returns the following error codes:

EUD_NOERROR	The read is successful.
EUD_IO	An operation is in progress.

abi_pio_write

The **`abi_pio_write`** routine allows you to write a value to a control register or data structure area on the 1553 ABI.

Specification

```
int abi_pio_write(abi, offset, data)

int           abi;
unsigned long offset;
unsigned short *data;
```

Parameters

abi	the identifier for the 1553 ABI whose control register or data structure area is to be written. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).
offset	an offset from the board’s base address to the location of the desired control register or data structure area on the 1553 ABI
data	a pointer to the location that contains the value to be written to the specified device’s control register or data structure area.

Return Value

The **`abi_pio_write`** routine returns the following error codes:

EUD_NOERROR	The write is successful.
EUD_IO	An operation is in progress, and a write to a register will disrupt the current operation.

abi_reset

The **`abi_reset`** routine allows a user process to reset the 1553 ABI hardware. It performs a master clear of the device and verifies that the interrupt vector and VME modifiers are set correctly. This routine also clears up any pending event completion status and halts any 1553 ABI operation in progress. If the user–level interrupt process is available, interrupts are enabled. This routine also initializes all of the data structures that are maintained by the driver.

Specification

```
int abi_reset(abi)

int          abi;
```

Parameters

abi the identifier for the 1553 ABI that is to be reset. This identifier is allocated on a call to the **`abi_open`** routine (see “abi_open” for an explanation of this routine).

Return Value

This **`abi_reset`** routine returns **EUD_NOERROR**.

Using Real–Time Serial Communications

All serial controllers have a standard STREAMS-based TTY driver. Certain serial controllers have the capability of being used with a special real-time device driver. This allows both standard TTY activities and real–time communications to occur on the same port at different times. In addition, different ports on the same controller can be used by the standard TTY driver and the real-time driver simultaneously. Currently, the only controller that has this capability is the SYSTECH High Performance Serial (HPS) controller.

This section contains the information needed to use a real-time TTY device on the HPS controller. An overview of the HPS is presented in “Understanding the HPS Controller.” Configuration and installation requirements are described in “Configuration and Installation Requirements.” A description of the user interface to the HPS is provided in “Understanding the User Interface.” Recommendations for optimizing performance are presented in “Optimizing the Performance of Real-Time TTY Devices.”

Understanding the HPS Controller

The HPS controller is a VMEbus interface card that provides up to 16 serial ports. On the serial ports, it supports baud rates as high as 38,400. The HPS controller contains two Octal UARTs (Universal Asynchronous Receiver–Transmitter), 256K of local RAM (Random Access Memory), 64K of local EPROM (Erasable Programmable Read–Only Memory), 16K of dual–ported RAM, and a 68020 microprocessor. The dual–ported RAM is memory on the HPS board that can be read from or written to by both the on–board firmware and the host driver. The HPS controller is capable of performing microprocessor–controlled DMA (Direct Memory Access) to host memory using 8–, 16–, or 32–bit transfers. The HPS board fits in a standard (H)VME–bus slot and is connected to the physical port cabling via Concurrent–designed spreader cables.

The HPS on–board processor allows the host to off-load a large portion of the character handling to the board. The download code works in combination with a kernel driver that has been optimized to reduce overhead and minimize time at interrupt level. The net result is reduced system overhead for the standard TTY devices and an especially optimized path for the real–time TTY devices.

Device special files for the standard TTY ports on the HPS controllers have names of the form **`/dev/tty`***c_nn*, where c specifies an HPS controller number ranging from **0** to **7** and *nn* specifies a port number ranging from **00** to **15**. Device special files for the real-time TTY ports on the HPS controllers have names of the form **`/dev/rtty`***c_nn*, where c specifies an HPS controller number ranging from **0** to **7** and *nn* specifies a port number ranging from **00** to **15**.

Configuration and Installation Requirements

Before using an HPS controller, you must ensure that the hps package is installed on your system. For an explanation of the procedures for installing software packages, refer to the the appropriate *PowerMAX OS Release Notes* and the **`pkgadd(1M)`** man page.

Use of the HPS real-time driver requires that the following modules be configured into the kernel: (1) **`hpsrt`**, the real-time serial device-dependent driver module for the HPS controller, and (2) **`rtserial`**, the real-time serial device-independent driver module. You can ensure that these modules are configured into your kernel by using the **`config(1M)`** utility. Note that after configuring a module, you must rebuild the kernel and then reboot your system. For an explanation of the procedures for using **`config(1M)`**, refer to the “Configuring and Building the Kernel” chapter of *System Administration Volume 2*.

To use the real-time driver, you must also ensure that the lines that correspond to the real-time TTY devices in the **`/etc/conf/node.d/hpsrt`** file are uncommented.

Note that after changing any of these files, you must rebuild the kernel using **`idbuild(1M)`** and reboot the system. It is recommended that you see your system administrator for assistance.

Understanding the User Interface

The HPS drivers support the following standard PowerMAX OS system calls on both the standard and the real-time TTY devices: **`open(2)`**, **`close(2)`**, **`ioctl(2)`**, **`read(2)`**, and **`write(2)`**.

The **`ioctl`**, **`read`**, and **`write`** system calls operate differently on the real-time TTY devices. Use of the **`ioctl`** system call is explained in “Using the Ioctl System Call.” Use of the **`read`** and **`write`** system calls is explained in “Using Read and Write System Calls.”

Using the Ioctl System Call

A set of **`ioctl`** commands allows you to perform a variety of operations that are specific to asynchronous serial devices. **`Ioctl`** commands for controlling terminal operations are defined by the System V general terminal interface. They are described in the **`termio(7)`** system manual page. The capabilities provided by these commands are also available through the function call interfaces that are described in the **`termios(2)`** system manual page. The function call interfaces are the preferred user interface.

On standard TTY devices, the HPS driver supports all of the **`ioctl`** commands defined by the general terminal interface. On real–time TTY devices, it does not support the following **`ioctl`** commands: TIOCGPGRP, TIOCSPGRP, TIOCGSID, TIOCGWINSZ, and TIOCSWINSZ.

The **`ioctl`** commands that are supported by the HPS drivers on both standard and real-time TTY devices are summarized as follows:

TCGETS	Get the **`termios`** parameters associated with the specified port
TCSETS	Set the **`termios`** parameters associated with the specified port
TCSETSW	Wait for the output to drain, and set the **`termios`** parameters associated with the specified port
TCSETSF	Wait for the output to drain, flush the input queue, and set the **`termios`** parameters associated with the specified port
TCGETA	Get the **`termio`** parameters associated with the specified port
TCSETA	Set the **`termio`** parameters associated with the specified port
TCSETAW	Wait for the output to drain, and set the **`termio`** parameters associated with the specified port
TCSETAF	Wait for the output to drain, flush the input queue, and set the **`termio`** parameters associated with the specified port
TCSBRK	Wait for the output to drain, and send a break
TCXONC	Start and stop control
TCFLSH	Flush input and output queues
TIOCMBIS	Set selected modem control signals
TIOCMBIC	Clear selected modem control signals
TIOCMGET	Get current modem control signals
TIOCMSET	Set modem control signals

The following PowerMAX OS **`ioctl`** commands, which are not defined by System V, are also supported by the HPS driver on real–time TTY devices.

TCGETRAWQ	Get the raw queue character limit associated with the specified port
TCSETRAWQ	Set the raw queue character limit associated with the specified port
TCGETMAXRAWQ	Get the system–wide maximum raw queue character limit
TCGETOBUFS	Get the current output buffers
TCSETOBUFS	Reconfigure output buffers

You must supply a **`termios`** structure on TCGETS, TCSETS, TCSETSW, and TCSETSF **`ioctl`** calls. This structure is defined in <**`sys`**/**`termios.h`**> as follows:

```
/*
 * Ioctl control packet
*/

struct termios
     tcflag_t  c_iflag;       /* input modes */
     tcflag_t  c_oflag;       /* output modes */
     tcflag_t  c_cflag;       /* control modes */
     tcflag_t  c_lflag;       /* line discipline modes */
     cc_t      c_cc[NCCS];    /* control chars */
};
```

On real–time TTY devices, only certain fields in the **`termios`** structure are used. The other fields are ignored on TCSETA, TCSETS,TCSETSW, TCSETSF, TCSETAW, and TCSETAF **`ioctl`** calls; their contents are not defined on TC_GETA **`ioctl`** calls. The **`termios`** fields that are used on real–time TTY devices are described in the paragraphs that follow. It is recommended that you review the **`termio(7)`** system manual page for detailed information on use of the **`ioctl`** commands and these fields.

Control characters that are defined by the **`c_cc`** array and used on real–time TTY devices are described as follows:

VEOF	Terminate a read if ICANON is set in the **`c_lflag`** field Note that if ICANON is not set, this element of the array contains the value of MIN (the minimum number of characters used to satisfy a read request).
VEOL	Terminate a read if ICANON is set in the **`c_lflag`** field Note that if ICANON is not set, this element of the array contains the value of TIME (the timeout value used to satisfy a read request).
VSTART	Resume output
VSTOP	Suspend output

The input control flags that are defined by the **`c_iflag`** values and used on real–time TTY devices are described as follows:

IGNBRK	Ignore break condition
IGNPAR	Ignore parity errors
PARMRK	Mark parity errors
INPCK	Enable input parity check
ISTRIP	Strip character
IXON	Enable start and stop output control
IXANY	Enable any character to restart output
IXOFF	Enable start and stop input control

None of the output control flags that are defined by the **`c_oflag`** values are used on real–time TTY devices. All user settings are ignored.

All of the hardware terminal control flags that are defined by the **`c_cflag`** values are used on real–time TTY devices. These include baud rates, character sizes, parity types, and so on.

The line discipline control flag that is defined by the **`c_lflag`** value and used on real–time TTY devices is described as follows:

ICANON	Enable use of EOF (end of file) and EOL (end of line) as line delimiters on read operations Note that this is the only use of this flag on real–time TTY devices.

On real–time TTY devices, the ICANON flag has limited functionality. If ICANON is set, receipt of the EOL or EOF character terminates a read operation; characters up to and including the EOL or EOF are passed to the user buffer. Subsequent input is saved for the next **`read(2)`** call. Other types of canonical processing are not performed; for example, the NL (newline) character has no effect, and erase and kill edit functions are not performed. The 256–character line length is no longer in effect.

If ICANON is not set on real–time TTY devices, read requests are satisfied when at least MIN characters have been received or when the timeout value TIME expires. Refer to the **`termio(7)`** system manual page for additional information on the operation of these two parameters.

Using Read and Write System Calls

On real–time TTY devices, the **`read(2)`** and **`write(2)`** system calls do not operate in the same way that they do in the standard TTY mode. The differences are outlined in the paragraphs that follow.

The **read(2)** system call transfers data from the specified HPS port to the user buffer and returns the number of characters placed in the buffer. The return value is determined as follows:

- If the file associated with the port has been opened with the O_NDELAY or O_NONBLOCK flag set, then any data that are available at the time of the **read(2)** call are placed in the user buffer, and control is returned immediately to the user process. If data are not available at the time of the call, the return value is zero.
- If the ICANON flag has been set on an **ioctl(2)** call and an EOL or EOF character is encountered, all data up to and including that character are placed in the user buffer. Subsequent input, even if it is available at the time of the **read(2)** call, is saved for the next **read(2)** call.
- If sufficient data are available to fill the user buffer with the number of bytes specified on the **read(2)** call, then those data are transferred. The return value is the specified byte count.
- If the ICANON flag has not been set on an **ioctl(2)** call, then the read operation terminates in accordance with the setting of the MIN and TIME control parameters (refer to the **termio(7)** system manual page for an explanation of the use of these parameters); for example, control may be returned to the user process when at least MIN characters have been received. The return value is the number of characters placed in the user buffer.

The **write(2)** system call transmits the specified number of bytes to a particular HPS port. If the amount of available buffer space is not sufficient to accept all of the data at the time of the **write(2)** call, the user process blocks unless the file associated with the port has been opened with the O_NDELAY or the O_NONBLOCK flag set. In either of these cases, control is returned immediately to the user process. The return value is zero if the O_NDELAY flag has been set; it is **–1** if the O_NONBLOCK flag has been set, and **errno** is set to EAGAIN.

Optimizing the Performance of Real-Time TTY Devices

You can maximize the I/O throughput of the HPS real-time TTY devices by turning off certain input control flags and the line discipline control flag. These flags are defined by the **c_iflag** and the **c_lflag** fields in the **termios** structure; their use is explained in "Using the Ioctl System Call." It is recommended that you turn off the following flags:

- BRKINT
- ISTRIP
- INPCK and PARMRK
- ICANON

  If you turn off this flag, you should also set MIN as high as possible (the highest permitted value is 254) or set a large timeout.

Memory Mapping for HSDE and DR11W

In order to use the high–speed data enhanced device (HSDE) or the DR11W emulator, you must ensure that the user's I/O buffer is bound to a contiguous section of physical memory. You can do so by performing the following steps:

1. Define a reserved section of physical memory.
2. Create a shared memory segment, and bind it to the reserved section of physical memory.
3. Obtain the shared memory identifier associated with the segment.
4. Attach the segment of shared memory to the user's virtual address space.

Reserving Physical Memory

To use the HSDE or the DR11W emulator, a section of main memory can be reserved by placing an entry in the `res_sects[]` array in the **`/etc/conf/pack.d/mm/space.c`** file.

Your entry will describe the starting address and the desired length of the reserved section of memory. Initially the `res_sects[]` array appears as follows:

```
struct res_sect res_sects[] = {
/*    r_start,    r_len,    r_flags */
     { 0, 0, 0 } /* This must be the last line, DO NOT change it.*/
};
```

For each section of physical memory that you wish to reserve, place an entry in the `res_sects[]` array. The `r_start` field specifies the starting physical address, and the `r_len` field specifies the length in bytes. The `r_flags` field must always be zero.

It is recommended that you specify as the `r_start` address one of the higher addresses from the range of addresses that comprise main memory. The value of `r_len` is bound by the size of the largest single data transfer that can be made using either the high–speed data device or the DR11W emulator. If you are using the high–speed data device, this size is dependent upon the transfer mode that you have specified—that is, **HSDE_BYTE**, **HSDE_WORD**, or **HSDE_LONG**; the maximum transfer size is 65,535 (64K–1) bytes, words, or longwords. The amount of reserved memory (`r_len`) should be based on the largest single transfer that the application will make.

After changing the **`space.c`** file, you must rebuild the kernel using **`idbuild(1M)`** and reboot your system.

Binding a Shared Memory Segment to Physical Memory

To use the HSDE or the DR11W emulator, you can create a segment of shared memory and bind it to the reserved section of physical memory by adding a line to the **`shmconfig`** script in the **`/etc/init.d`** directory. The format that you should use to add a line is as follows:

`/usr/sbin/shmconfig -p`*paddr* **`-s`***size* [**`-u`***user*] [**`-g`***group*] [**`-m`***mode*] *key*

The shared memory segment will be created in global memory by default. The **`-p`** option enables you to specify the starting address (represented by *paddr*) of the section of physical memory that you have reserved, and the **`-s`** option enables you to specify the size in bytes of that section. It is recommended that you specify the **`-u`**, **`-g`**, and **`-m`** options to identify the user and group associated with the segment and to set the permissions controlling access to it. A suggested value for the *key* argument is the file name for the device: **`/dev/hsde`***n* for the high–speed data device or **`/dev/dr11w`***n* for the DR11W emulator, where *n* represents the **`hsd`** controller number or the **`dr11w`** emulator number.

For additional information on use of the **`shmconfig`** command, refer to the system manual page **`shmconfig(1M)`** and to the *PowerMAX OS Programming Guide*.

Obtaining an Identifier for a Shared Memory Segment

If you use the **`shmconfig`** command to create a shared memory segment and bind it to a section of physical memory, you must use the **`shmget(2)`** system call to obtain an identifier for the shared memory segment. This identifier is required by other system calls for manipulating shared memory segments. The format for the **`shmget`** call is as follows:

`shmget(`*key*, *size*, *shmflg***`)`**

Note that the value of *key* is determined by the value of the *key* argument specified with the **`shmconfig`** command (see “Binding a Shared Memory Segment to Physical Memory”). If the value of the *key* argument was an integer, that integer must be specified as *key* on the call to **`shmget`**. If the value of the *key* argument was a path name, you must first call the **`ftok`** subroutine to obtain an integer value that is based on the path name to specify as *key* on the call to **`shmget`**. Examples of the format that must be used to invoke **`ftok`** for the high–speed data devices, the DR11W emulator, and HVME reflective memory, respectively, are provided by the following:

```
ftok(”/dev/hsd0”, 0)
ftok(”/dev/hsde0”,0)
ftok(”/dev/dr11w0”, 0)
```

The value of *size* must be equal to the number of bytes specified by the **`-s`***size* argument to the **`shmconfig`** command (see “Binding a Shared Memory Segment to Physical Memory”). Because the shared memory segment has already been created by using the **`shmconfig`** command, the value of *shmflg* should be zero.

For additional information on use of the **`shmget(2)`** system call, refer to the corresponding system manual page and to the *PowerMAX OS Programming Guide*. For assistance in using **`ftok`**, see the system manual page for **`stdipc(3C)`**.

NOTE

If you wish to access shared memory from a FORTRAN program, you must use the **`shmdefine(1)`** utility to perform the **`shmget`** function. For additional information, refer to the corresponding system manual page and to the *PowerMAX OS Programming Guide*.

Attaching a Shared Memory Segment

You can attach the newly created shared memory segment to your virtual address space with the **`shmat`** system call. The format of this call is as follows:

`shmat(`*shmid***`,`** *shmaddr***`,`** *shmflg***`)`**

The value of *shmid* is the identifier for the shared memory segment that has been returned by the **`shmget(2)`** system call.

`Shmat` will return the virtual address at which the kernel has attached the shared memory segment. For additional information on use of the **`shmat`** system call, refer to the system manual page **`shmop(2)`** and to the *PowerMAX OS Programming Guide*.

NOTE

If you wish to access shared memory from a FORTRAN program, you must use the **`shmdefine(1)`** utility to perform the **`shmat`** function. For additional information, refer to the corresponding system manual page and to the *PowerMAX OS Programming Guide.*

14
STREAMS Network Buffers

14
STREAMS Network Buffers

Overview

This chapter describes the PowerMAX OS commands to create and work with STREAMS network buffers, and provides examples of their use.

STREAMS network buffers save CPU power by:

- Minimizing copying of data during output to a STREAMS network device. When a process asks to send network data, the OS copies data between user and kernel address space at the stream head. Such copying uses a lot of CPU power during bulk data transfers, especially when using large data units. Using a STREAMS network buffer for output minimizes copying of data and use of CPU power. It does so by sharing data buffers in the kernel between the application and STREAMS.
- Avoiding the overhead of repeatedly allocating and freeing network buffer pages with the **`kmem_alloc()`** and **`kmem_free()`** calls during output requests, because the operating system maintains a pool of network buffers rather than allocating buffer space at the time of the output request.

System Call

STREAMS network buffers extend the **`streamio(7)`** interface with the additional **`ioctl`** function **`I_NBUFF`**. Network buffers map into the virtual address spaces of both the application program and the operating system. Sharing this data space between the application code and operating system minimizes the memory copying that normally occurs when outputting to a STREAMS device.

`ioctl` (*fd*, **`I_NBUFF`**, *arg,,,*)

where:

`ioctl` is the function that contains **`I_NBUFF`**

fd is the file descriptor opened for network output

`I_NBUFF` is the **`ioctl`** command

arg is a pointer to a str_netbuff_info structure.

Understanding the Network Buffer Information Structure

The **`I_NBUFF ioctl`** function allocates and controls the use of STREAMS network buffers. **`arg`** points to the following **`str_netbuff_info`** structure:

```
struct str_netbuff_info {
    netbuff_req_t req_type;
    netbuff_t buff_type;
    int length_requested; /*bytes requested by the user*/
    int length_given; /*length allocated*/
    vaddr_t address; /*start address of network buffer*/
    int ref_count; /*for fixed translation buffers*/
};
```

The following components comprise **`str_netuff_info`**:

`netbuff_req_t` **`req_type`**`;`

Specifies a network buffer action with one of the following commands:

- **`NBUFF_ALLOC`**
- **`NBUFF_FREE`**
- **`NBUFF_REF_COUNT`**
- **`NBUFF_WAIT`**

`netbuff_t` **`buff_type`**`;`

Specifies one of the following network buffer re-use types:

- **`BUFF_FIXED_GLOBAL`**
- **`BUFF_RELOAD_GLOBAL`**
- **`BUFF_FIXED_LOCAL_CACHED`**
- **`BUFF_FIXED_LOCAL_NOCACHE`**

`int` **`length_requested`**`;`

Specifies in bytes the desired buffer to allocate. If this length is not an integral multiple of the page size, then PowerMAX OS rounds it up to the next higher value.

`int` **`length_given`**`;`

Returns the actual length assigned by PowerMAX OS in response to a buffer allocation request. **`length_requested`** <= **`length_given`**

`vaddr_t` **`address`**`;`

Specifies the starting address of the network buffer.

`int` **`ref_count`**`;`

The number of references returned from calling **`NBUFF_REF_COUNT`**.

Understanding the Network Buffer Commands

The **`I_NBUFF ioctl`** function commands use the *address* parameter to identify buffers (by their starting address):

`NBUFF_FREE`

Frees a network buffer, un-maps its address space, and returns its memory to either the kernel heap or the network buffer free pool.

`NBUFF_ALLOC`

Allocates a network buffer. The user specifies the size of the buffer in the *length_requested* field, and the page reuse type in the *buff_type* field. The page reuse type controls how the operating system maps the address space. It also controls whether global or local memory (see **`memory(7)`**) is used for the buffers, and in the case of local memory buffers, the page reuse type also specifies the cache mode (see **`cache(7)`**) to be used. The page reuse types are:

- **`BUFF_RELOAD_GLOBAL`**
  Provides a new virtual mapping to the network buffer on each output operation using the network buffer. Such page swapping re-maps the address space for each output request to a new physical address. This frees the process from having to confirm the buffer is free. See the following section on “Understanding Network Buffer Types” for more information. This page reuse type will use cache enabled translations to global memory pages for the network buffers.

- **`BUFF_FIXED_GLOBAL`**
  Provides a constant virtual to physical mapping of the network buffer. Such a fixed address translations only maps the address space once. To avoid overwriting a network buffer, the process must either call **`NBUFF_WAIT`** or include **`NBUFF_REF_COUNT`** in a routine to check if the buffer is free. See the following section on “Understanding Network Buffer Types” for more information. This page reuse type will use cache enabled translations to global memory pages for the network buffers.

- **`BUFF_FIXED_LOCAL_CACHED`**
  This page reuse type provides the same functionality as the **`BUFF_FIXED_GLOBAL`** type, except that cache enabled translations to local memory pages will be used for the network buffers. The same care must be taken to ensure that these buffers are not overwritten. This reuse type is valid only for Night-Hawk platforms configured with local memory.

  When this page reuse type is used, the calling LWP must have already set its CPU bias to a value that contains CPUs that are located on only one CPU board (see **`cpu_bias(2)`** and **`mpadvise(3C)`**). The local memory pool that is located on the CPU board where the LWP is executing at the time of this **`ioctl(2)`** call will be used for the buffer allocation.

Due to the fact that cached remote local memory CPU accesses are not kept cache coherent in the kernel, data corruption may result if this reuse type is specified. Therefore, general use of this buffer reuse type is highly discouraged. Currently, this reuse type is only officially supported with Data Link Layer Provider (DLPI) STREAMS stacks. See the *STREAMS Modules and Drivers* manual for example code and a discussion on the restrictions regarding this reuse type.

PowerMAX OS also supports a configurable kernel feature that enables use of a per-STREAM CPU bias mask for controlling the set of CPUs that may execute each STREAM's service procedures. This support may be useful in controlling the set of CPUs that execute service procedures of a STREAM that is making use of local memory networking buffers; thus reducing the risk of remote local memory CPU accesses occurring within the STREAM's service procedure memory references. For more information about this feature, see Chapter 2: Improving Response Time, in the section called "Controlling STREAMS Scheduling".

- **`BUFF_FIXED_LOCAL_NOCACHE`**
  This page reuse type provides the same functionality as the **`BUFF_FIXED_LOCAL_CACHED`** type, except that cache inhibited translations to local memory pages are used. Care must be taken to ensure that these buffers are not overwritten. This reuse type is valid only for NightHawk platforms configured with local memory.

  Unlike the **`BUFF_FIXED_LOCAL_CACHED`** type, there are no CPU bias restrictions placed on the calling LWP. While data cache incoherences will not result from using this buffer reuse type, higher performance will usually be observed when the application references the network buffer from CPUs that are non-remote to the local memory page(s).

  Possible application or system-wide performance improvements due to the use of the **`BUFF_FIXED_LOCAL_NOCACHE`** buffer reuse type instead of the **`BUFF_FIXED_GLOBAL`** reuse type will depend upon the application mix, system configuration, and upon the STREAMS stack(s) being used.

NOTE

Once the user application references local memory Streams Networking Buffers, these pages will be locked down in memory. This action will prevent cross-CPU board migrations (see **`memadvise(3C)`**) of that process or any of its LWPs until that stream is closed, or until the local memory buffers are removed with a **`I_NBUFF NBUFF_FREE`** **`ioctl(2)`** call. Therefore, it is recommended that process migrations be performed by the application prior to the setup of local memory Networking Streams Buffers.

`NBUFF_REF_COUNT`

Lets the application poll to check if the operating system has released the fixed network buffer specified in the call. The number of references to a network buffer is the number of outstanding I/O operations on that buffer. When the number reaches zero, the network buffer is free.

`NBUFF_WAIT`

Lets the calling application sleep until the operating system releases the fixed network buffer specified in the call.

Understanding Network Buffer Types

PowerMAX OS supports two types of network buffers:

- Reload
- Fixed.

When the operating system first allocates a reload network buffer, it assigns a permanent virtual address and then maps the virtual address to a physical address space. Each subsequent I/O request to the buffer virtual address re-maps it to a new physical address space. This simplifies the programming model because applications need not check if the operating system released the buffer. Data within a network buffer is no longer available to the application once a new I/O request uses the buffer, since the virtual address space gets re-mapped to a new physical address space. Compared to fixed network buffers, reload buffers impose a minor penalty in throughput for two reasons:

- The operating system must remap virtual to physical memory for each request to a reload network buffer.
- The pages of a reload network buffer tend to become discontiguous in kernel virtual address space over time.

When the operating system first allocates a fixed network buffer, it assigns a permanent virtual address and then maps the virtual address to a physical address space. Unlike reload buffers, in fixed buffers the mapping persists as long as the buffer exists.

Since some network protocols return control to the application before the I/O transfer completes, a network buffer might still be in use when control returns to the application. To issue another I/O request using the same network buffer, the application must either poll the operating system to see when I/O operations on a given buffer complete or wait for all operations to complete. An application can issue multiple I/O requests to the same fixed network buffer by using different address ranges within the network buffer. To segment a fixed buffer and interact with its parts, a process must use its own logic and track how much to offset each request from the starting address. This is because the network buffer commands only recognize buffers in their entirety.

Much less system overhead is incurred doing I/O with either variety of network buffers than with doing I/O in the standard way. This is because data copying from user space to kernel space at the STREAMS head is eliminated.

Note

Some STREAMS protocol stacks and drivers force synchronous completion of I/O request.

Example of a System Call

System call to allocate a network buffer on an open stream with file descriptor fd:

```
{struct str_netbuff_info str_netbuff_info;

    fd = open("/dev/udp", O_RDWR);
    str_netbuff_info.buff_type = BUFF_FIXED_GLOBAL;
    str_netbuff_info.length_requested = 8192;
    str_netbuff_info.req_type = NBUFF_ALLOC;
    error = ioctl(fd, I_NBUFF, &str_netbuff_info);

    if (error == -1)
    {
        perror("NBUFF_ALLOC failed");
        exit(1);
    }
}
```

After a successful call, `str_netbuff_info.address` contains the user virtual address of the network buffer and `str_netbuff_info.length_given` contains the length allocated by STREAMS (sometimes larger than requested as STREAMS rounds up the size to a page boundary.)

Example of Double-Buffering

The following program demonstrates how **`NBUFF_WAIT`** can be used with network buffers to do double buffering. **`NBUFF_REF_COUNT`** can be used in the same way, except it returns immediately with the number of references to the network buffer. An application can do other work while waiting for the buffer to become free by periodically checking **`NBUFF_REF_COUNT`**.

```
/*
 * example program - double buffering with network buffers using NBUFF_WAIT
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <sys/stropts.h>

struct network_buffer {
     struct str_netbuff_info str_netbuff_info;
     vaddr_t vaddr; /* start address of network buffer */
     int *header; /* for use with writev() */
};

main(argc, argv)
int argc;
char *argv[];

{
struct network_buffer network_buffer[2];
int buff_no, block_size;
int fdu, loopcount, i, buffer_length;
struct iovec iov[2];
vaddr_t block_addr, end_address;

     if (( fdu = open(argv[1],O_RDWR)) < 0 )
     {
          perror(" Failed to open user stream");
          exit(1);
     }

     buffer_length = 32768;
     /* allocate two fixed network buffers */
     for ( buff_no = 0; buff_no < 2; buff_no++ )
     {
          network_buffer[buff_no].str_netbuff_info.buff_type =
               BUFF_FIXED_GLOBAL;
          network_buffer[buff_no].str_netbuff_info.req_type = NBUFF_ALLOC;
          network_buffer[buff_no].str_netbuff_info.length_requested =
               buffer_length;
          ioctl(fdu, I_NBUFF, &network_buffer[buff_no].str_netbuff_info);
          network_buffer[buff_no].vaddr = network_buffer[buff_no].
               str_netbuff_info.address;
          network_buffer[buff_no].header = (int *)malloc(128);

     }

     printf("enter: loopcount  block_size \n");
     /* send <loopcount> blocks of size block_size down the stream */
     while (scanf("%d %d", &loopcount, &block_size) > 0)
     {
          /* start with buffer number 0 */
          buff_no = 0;
          iov[0].iov_base = (void *)network_buffer[buff_no].header;
          iov[0].iov_len = 128;
          block_addr = (vaddr_t)network_buffer[buff_no].vaddr;
          end_address = (vaddr_t)network_buffer[buff_no].vaddr +
               buffer_length;

          for (i = 0; i < loopcount; i++)
          {

               iov[1].iov_base = (void *)block_addr;
               iov[1].iov_len = block_size;
               /* write to network buffer */
               *(int *)block_addr = 12345678;
               writev(fdu, iov, 2);
               block_addr += block_size;
               if ( ( block_addr + block_size ) >= end_address )
               { /* buffer full - start using the other one */

                    buff_no = ( buff_no == 0 ) ? 1 : 0;
                    /* wait for buffer to become free */
                    network_buffer[buff_no].str_netbuff_info.req_type =
                         NBUFF_WAIT;
                    ioctl(fdu, I_NBUFF,
                         &network_buffer[buff_no].str_netbuff_info);
                    block_addr = (vaddr_t)network_buffer[buff_no].vaddr;
                    end_address = (vaddr_t)network_buffer[buff_no].vaddr +
                         buffer_length;

               }

          }
          printf("enter: loopcount  block_size \n");
   }
}
```

Kernel Tunables

Additional kernel tunables can further optimize the OS for STREAMS buffered network output. Refer to **`config(1)`** for more information on kernel tunables.

Table 14-1. Kernel Tunables

Kernel Tunable	Description
STR_MAX_NBUFF	Maximum number of pages per network buffer. Limits the size of a network buffer allocated by a **NBUFF_ALLOC** call.
STR_NPAGES_FPMAX	Maximum number of pages per memory pool in the network buffer free pool. If set to 0, allocates all new pages from the kernel heap.
STR_NPAGES_TMAX	Maximum number of network pages allocated at any one time, including pages: - Available in network buffer free pools - In use by PowerMAX OS for network buffers - In use by applications.
STR_NBUFF_COPYSIZE	Minimum size in bytes of I/O transfers allowed to use STREAMS network buffering. Below this threshold, the operating system copies data from user space into kernel space (as before) rather than using the network buffer as a shared resource.

15
Controlling Periodic Kernel Daemons

15 Controlling Periodic Kernel Daemons

Understanding Kernel Daemons

There are three types of kernel daemons that PowerMAX OS will execute:

1. periodic daemons - these run at a fixed frequency regardless of the state of the system. In some cases, tunables are provided that can regulate the frequency.
2. reactive daemons - these run only in response to a specific event.
3. interrupt daemons - run only in response to a specific device interrupt.

All of the daemons can be controlled to some degree via tunables. These tunables are used to specify a CPU mask, scheduling class and scheduling priority for each daemon.

To a lesser degree, a system developer does have some control over when the reactive daemons and interrupt daemons actually run. Generally, the events that trigger these daemons can be regulated.

In some real-time environments, especially those that do not have shielded processors, controlling the execution of periodic daemons is crucial to providing a deterministic program execution environment.

Enabling and Disabling Periodic Kernel Daemons

The periodic kernel daemons generally provide important functionality during specific important system states. At other times, this functionality may be much less critical. For this reason, the periodic kernel daemons may be disabled in real-time environments if the loss of functionality would not impact system integrity and performance.

However, disabling the periodic kernel daemons should not be done without a certain amount of consideration of the impact.

Furthermore, over time, the probability increases that a disabled kernel daemon becomes a deterrent to full system operation. For this reason, it is highly recommended that kernel daemons not be disabled permanently. Instead, at appropriate times, these daemons should be enabled, or perhaps run just once (this functionality is provided).

Daemoncntl

`Daemoncntl(1m)` is a command that can be used to control the periodic kernel daemons. There are four functions that this command provides:

- query the periodic kernel daemons to determine which are enabled or disabled.
- disable a selected list of the periodic kernel daemons.
- enable a selected list of the periodic kernel daemons.
- run a selected list of the periodic kernel daemons once. Refer to “Enabling and Disabling Periodic Kernel Daemons” on page 15-1.

Refer to the **`daemoncntl(1m)`** man page for a description of how to run this command.

Refer to the **`daemon_cntl(2)`** man page for a description of the system service that is used by **`daemoncntl(1m)`** and which can be directly called by a user’s application.

Description of Periodic Kernel Daemons

The following describes each of the periodic kernel daemons plus the possible impact when the daemon is disabled.

- `fsflush`: this daemon is responsible for periodically writing modified file system buffers and pages to disk. The frequency of this daemon is controlled by the tunable `FDFLUSHR`. The default is to run every second.

  The impact of disabling this daemon is that it increases the chance that data written to disk will be lost if the system crashes. In environments where few file system operations are done, it would be safe to disable this daemon. The risk increases as the number of operations increases.

- `kma_giveback`: this daemon returns kernel memory blocks from per-CPU “local” pools to system-wide “global” pools. There is one daemon per CPU and the daemons generally run every 30 seconds when memory is sufficient.

  There is no risk associated with disabling this daemon, as the “local” pools would no longer be used. However, there could be a slight performance penalty.

  It should be noted that this daemon can be disabled permanently by use of the `KMA_GBACK_DISABLE` tunables.

- `wallclock_ager`: searches for processes that need to have their address space aged. Address space aging is the process whereby pages, which have not been referenced, are marked as likely candidates for swap out from memory to swap space on disk. Runs once per second.

On a system with sufficient memory, there is minimal impact from disabling this daemon. However, in memory critical conditions, the memory manager may make poor decisions regarding page/process swapouts.

- `poolrefresh`: this daemon scans several “private” pools of memory and grows or shrinks the pools as needed. Private memory pools are preallocated data structures used by various kernel subsystems to speed the allocation of these structures. Runs once per second.

  On a system with sufficient memory, there is minimal impact from disabling this daemon. However, in memory critical conditions, this daemon can help alleviate the problem by making more memory available. This daemon is most important when the amount of kernel activity fluctuates during the execution of an application.

- `autounload`: this daemon locates and unloads inactive dynamic loadable modules (DLMs). Runs every 60 seconds.

  On most systems, there is little risk associated with disabling this daemon, as generally, few if any DLMs would need to be unloaded. If there were any and memory was critical, then this daemon may be able to alleviate the condition.

- `timepct_daemon`: this daemon computes the percentage of time that each lwp uses a CPU. Runs every 5 seconds.

  Very little risk associated with disabling this daemon. The CPU utilization numbers presented by **`top(1m)`** would be the only effect of disabling this daemon.

- `sleeper_search`: this daemon locates long term sleeper lwps that can be swapped or aged. Generally runs between 2 and 4 seconds with 2 seconds being the default.

  On a system with sufficient memory, there is minimal impact from disabling this daemon. However, in memory critical conditions, the memory manager may make poor decisions regarding page/process swapouts.

A
Example Program - Message Queues

This appendix contains an example program that illustrates use of the POSIX message queue facilities. The program is written in C. In this program, a parent process spawns a child process to offload some of its work. The parent process also creates a message queue and attaches a notification request to the message queue. When the child process completes its work, it sends the results to the parent process via the message queue. At this point, the parent process receives a signal and a value indicating that the child has sent a message.

```
#include <stdio.h>
#include <sys/ucontext.h>
#include <sys/siginfo.h>
#include <signal.h>
#include <errno.h>
#include <mqueue.h>

 #define MAXMSGS          5
#define MSGSIZE          40

 void sighandler();

 volatile int done = 0;

 void
main()
{
    mqd_t               mqdescr;
    int                 retval;
    int                 cpid;
    char                msg_ptr[MSGSIZE];
    struct sigevent     notif;
    struct mq_attr      attr;
    sigset_t            set;
    struct sigaction    sa;

     char *mqname = "test-mq";

     attr.mq_maxmsg = MAXMSGS;
    attr.mq_msgsize = MSGSIZE;

     /* open the message queue */
    mqdescr = mq_open( mqname, O_CREAT | O_RDWR, 0700, &attr );
    if( mqdescr == (mqd_t)-1 ) {
        perror( "mq_open" );
        exit( -1 );
    }

     cpid = fork();

     if( cpid < 0 ) {
        perror( "fork" );
        mq_close( mqdescr );
        mq_unlink( mqname );
        exit( -1 );
    }
```

```
if( cpid == 0 ) {
   /* child */

    /* perform some work for parent */
   sleep(1);

    /* ... */

    strcpy( msg_ptr, ”Results of work” );

    /* Send results to the parent via the inherited message queue.*/
   retval = mq_send( mqdescr, msg_ptr, strlen( msg_ptr ) + 1, 20 );

    if( retval ) {
       perror( ”mq_send (child)” );
       mq_close( mqdescr );
       exit( -1 );
   }

    mq_close( mqdescr );
   exit( 0 );

} else {

    /* parent */
   /*
    * Define the actions the parent process should take when
    * SIGRT1 is received.  It is not necessary to block other
    * signals, but it is necessary for the signal information
    * block to be delivered with SIGRT1.  The SA_SIGINFO flag
    * causes signals to be queued and a value to be sent with
    * the signal.
    */

    sigemptyset( &set );
   sa.sa_handler = sighandler;
   sa.sa_flags = SA_SIGINFO;
   sa.sa_mask = set;
   sigaction( SIGRT1, &sa, NULL );

    /*
    * Attach a notification request to the message queue
    * so a SIGRT1 informs the parent process that a message
    * has arrived from the child process.
    */

   notif.sigev_notify = SIGEV_SIGNAL;
   notif.sigev_signo = SIGRT1;
   notif.sigev_value.sival_int = (int)mqdescr;
   retval = mq_notify( mqdescr, &notif );
   if( retval < 0 ) {
       perror( ”mq_notify” );
       exit( -1 );
   }

    /* Do not attempt to receive a message from the child
    * process until it arrives.  Perform parent workload
    * while waiting for results.
    */

    while( !done ) {

        /* ... */

    }
```

```
        mq_close ( mqdescr );
        mq_unlink ( mqname );
        exit( 0 );
    }
}

 /*
 * This routine reacts to a SIGRT1 user-selected notification
 * signal by receiving the child process’s message.
 */

 void
sighandler( sig, sip, ucp )
int sig;
siginfo_t *sip;
ucontext_t *ucp;
{
    mqd_t               mqdescr;
    char                msg_ptr[MSGSIZE];
    unsigned int        msg_prio;
    int                 retval;

     mqdescr = (mqd_t)sip->si_value;

     /* read the message that was received */
    retval = mq_receive( mqdescr, msg_ptr, MSGSIZE, &msg_prio );
    if( retval == -1 )
        perror( ”mq_receive (parent)” );
    else
        printf( ”msg received: %s ; at priority %d\n”, msg_ptr, msg_prio );

     /* do processing for received message */

     done++;
}
```

B Example Program - Synchronization Tools

This appendix contains an example program that illustrates use of the interprocess synchronization tools. The program is written in C and shows how producer and consumer tasks can exchange data through use of a mailbox in a shared memory segment. To simplify the mailbox functions, queues are not used. For purposes of illustration, sleepy-wait mutual exclusion is used to serialize access to the mailbox; because the mailbox critical sections are very short, busy-wait mutual exclusion is preferable.

```
 /*
 * Name
 *   pc -- Producer/Consumer Example
 *
 * Purpose
 *      Illustrate use of process synchronization tools.
 *
 * Procedure
 *      A single producer task and a single consumer task communicate
 *      through a mailbox in shared memory.  The producer task sends
 *      random data, and the consumer task checks the received data
 *      against an independently running but identical random number
 *      generator.  The program runs until it is terminated by typing
 *      ^C.  The synchronization tools introduced in Chapter 5 of
 *      this guide are used to control access to the mailbox.
 */

#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/lwp_synch.h>

#include <stdio.h>
#include <assert.h>
/*************** Spin Locks **********************************************/

 #definespin_acquire(_m,_r) \
{ \
     resched_lock(_r); \
     while (! spin_trylock(_m)) { \
          resched_unlock(_r); \
          while (spin_islock(_m)); \
          resched_lock(_r); \
     } \
}

 #definespin_release(_m,_r) \
{ \
     spin_unlock(_m); \
     resched_unlock(_r); \
}

 /*************** Sleep Locks *********************************************/

 struct resched_var rv;/* Rescheduling variable. */
 struct sleep_mutex {
     struct spin_mutex mx;/* Serializes access to lock. */
     global_lwpid_t owner;/* Identifies the owner. */
     int waiters;    /* True iff there are waiters. */
};

 /*
 * Initialize a sleep lock.
 */
void
sleep_init (s)
     struct sleep_mutex *s;
{
     spin_init (&s->mx);
     s->owner = 0;
     s->waiters = 0;
}
/*
 * Acquire a sleep lock.
 */
void
sleep_lock (s)
     struct sleep_mutex *s;
{
     spin_acquire (&s->mx, &rv);
     while (s->owner) {
          s->waiters = 1;
          client_block (s->owner, (int)s, 0, &s->mx, &rv, 0);
          spin_acquire (&s->mx, &rv);
     }
     s->owner = rv.rv_glwpid;
     spin_release (&s->mx, &rv);
}

 /*
 * Release a sleep lock.
 */
void
sleep_unlock (s)
     struct sleep_mutex *s;
{
     int were_waiters;

     spin_acquire (&s->mx, &rv);
     s->owner = 0;
     were_waiters = s->waiters;
     s->waiters = 0;
     spin_unlock (&s->mx);

     if (were_waiters)
          client_wakechan (0, (int)s, &rv);
     else
          resched_unlock (&rv);
}
/************** Mailboxes **********************************************/

 struct mailbox {
     struct sleep_mutex smx;/* Serializes access to mailbox. */
     unsigned int data;/* Contents of the mailbox. */
     int full;      /* Mailbox full/empty flag. */
     global_lwpid_t producer;/* Identifies waiting producer. */
     global_lwpid_t consumer;/* Identifies waiting consumer. */
};
```

```
 /*
 * Initialize a mailbox.
 */
void
mailbox_init (mb)
     struct mailbox *mb;
{
     sleep_init (&mb->smx);
     mb->data = 0;
     mb->full = 0;
     mb->producer = 0;
     mb->consumer = 0;
}

 /*
 * Put data into a mailbox.  Wait for the mailbox to empty, put the
 * new data into the mailbox, and awaken any waiting consumer.
 */
void
mailbox_put (mb, data)
     struct mailbox *mb;
     unsigned int data;
{
     global_lwpid_t consumer;

     sleep_lock (&mb->smx);

     while (mb->full) {
          mb->producer = rv.rv_glwpid;
          sleep_unlock (&mb->smx);
          server_block (0, 0, 0);
          sleep_lock (&mb->smx);
     }


     mb->data = data;
     mb->full = 1;

     consumer = mb->consumer;
     mb->consumer = 0;

     sleep_unlock (&mb->smx);

     if (consumer)
          server_wake1 (consumer, 0);
}

 /*
 * Get data from a mailbox.  Wait for the mailbox to fill, get the
 * data from the mailbox, and awaken any waiting producer.
 */
void
mailbox_get (mb, data)
     struct mailbox *mb;
     unsigned int *data;
{
     global_lwpid_t producer;

     sleep_lock (&mb->smx);

     while (! mb->full) {
          mb->consumer = rv.rv_glwpid;
          sleep_unlock (&mb->smx);
          server_block (0, 0, 0);
          sleep_lock (&mb->smx);
     }

     *data = mb->data;
     mb->full = 0;

     producer = mb->producer;
     mb->producer = 0;

     sleep_unlock (&mb->smx);

     if (producer)
          server_wake1 (producer, 0);
}
/*************** Producer/Consumer Tasks **********************************/

 struct mailbox *mb;/* Points to a mailbox in shared memory. */

 /*
 * The producer task puts data obtained from a random number generator
 * into the mailbox.
 */
void
producer ()
{
     unsigned int data;

     srand48 (1);
     if (resched_cntl (RESCHED_SET_VARIABLE, (char *)&rv)) [
          perror("resched_cnt");
          exit(1);
     }

     while (1) {
          data = (unsigned int) lrand48();
          mailbox_put (mb, data);
     }
}

 /*
 * The consumer task compares the data obtained from the mailbox
 * with an independently running but identical random number
 * generator.
 */
void
consumer ()
{
     unsigned int i, data;

     srand48 (1);
     if (resched_cntl (RESCHED_SET_VARIABLE, (char *)&rv)) [
          perror("resched_cnt");
          exit(1);
     }

     for (i=1; 1; i++) {
          mailbox_get (mb, &data);
          if (data != (unsigned int) lrand48())
               printf (”%u: data comparison failed\n”, i);
          else if ((i % 250) == 0)
               printf (”%u: received %u\n”, i, data);
     }
}
/*************** Main ****************************************************/

void
```

```
main ()
{
     int i;
     pid_t pid;

     /*
      * Create a shared memory segment for the mailbox.  The
      * segment will disappear when the last process using it
      * exits.
      */

     i = shmget (IPC_PRIVATE, sizeof (*mb), 0600);
     if (i == -1) {
          perror ("shmget");
          exit (1);
     }

     mb = (struct mailbox *) shmat (i, 0, 0);
     if (mb == (struct mailbox *) -1) {
          perror ("shmat");
          exit (1);
     }

     shmctl (i, IPC_RMID, 0);

     /*
      * Initialize the mailbox.
      */

     mailbox_init(mb);

     /*
      * Create the producer process.
      */

     if ((pid = fork ()) < 0) {
          perror ("fork");
          exit (1);
     }

     if (pid == 0) {
          producer ();
          _exit (0);
     }

     /*
      * Run the consumer process.
      */

     consumer ();
     exit (0);
}
```

C
Example 1 - User-Level Interrupt Routines

This appendix contains an example C program that demonstrates use of the user-level interrupt routine facility by a user program that executes a user-level interrupt process and interrupt-handling routine. It is intended as an example of a typical single-threaded process that uses a shared memory region to communicate with other processes—in this case, a parent and child.

```
/*
 *  - uses one rtc interrupt
 *  - uses fork() to create one child
 *  - child connects to rtc interrupt
 *  - parent starts the rtc repetively counting
 *  - a shared memory segment is used for communication
 *    between the parent and child
 *  - uses the server_block() and sever_wake1() services
 */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/lwp.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/thread_synch.h>
#include <sys/ioctl.h>
#include <sys/rtc.h>
#include <sys/iconnect.h>


/*
 * Error messages
 */
extern char *sys_errlist[];

/*
 * Error messages
 */
char *badshmget   = "shmget() call failed";
char *badshmat    = "shmat() call failed";
char *badshmctl   = "shmctl() call failed";
char *baddevopen  = "dev open() call failed";
char *badioctl    = "IOCTLVECNUM ioctl() call failed";
char *badicon     = "iconnect() call failed";
char *badienable  = "ienable() call failed";
char *badlock      = "child's mlockall() call failed";
char *badfork      = "fork() failed";
char *badrtcstart  = "RTCIOCSTART ioctl() failed";
char *badrtcset    = "RTCIOCSET ioctl() failed";
char *badserverblk = "server_block() failed";

/*
 * Flag for err_rtn()
 */
#define ERR_DISC0x1/* disconnect the child process */

/*
 * shmget()
 */
#define DEF_PERM 0644
#define SHMFLG   DEF_PERM

/*
 * Shared memory segment layout.
 */
struct dcom {
    volatile int enabled;
    volatile int int_count;
    volatile int status;
    volatile int vec_num;
    volatile global_lwpid_t lid;/* lwp id of parent JUDYJUDYJUDY */
};
struct dcom *dcom_p; /* for sharing data */
void *shmaddr;/* virtual address of dcom shared memory segment */

/*
 * Values for status field of dcom struct.
 */
#define BAD_VALUE   1 /* bad value was passed to routine */
#define WAIT_FOR_ME 2 /* wait till ienabled */
#define BAD_SETUP   3 /* child didn't succesfully ienable() */

/*
 * User-level interrupt routine values and variables.
 */
#define VALUE        0x12345678 /* Passed as arg to interrupt routine */
#define NUM_REQUESTS 100        /* number of interrupts to force */
#define STACK_SIZE   4096       /* Stack size, in bytes. */

int done = 0;                  /* set to 1 when NUM_REQUESTS reached */
char interr_stack[STACK_SIZE];/* interrupt stack. */

/*
 * rtc device file.
 * Arbitrarily use the fourth clock on the first CPU board.
 */
char *dev_string = "/dev/rrtc/0c3"; /* device file name */

int file_dev = -1;/* -1 for cleanup code to skip close() */
```

```
main()
{
    int   shm_id;
    pid_t child;
    struct icon_conn ic;
    struct rtc       rtc;
    extern int       shmget();
    extern void      rtcintr();


    /*
     * Create a private shared memory segment to shared
     * data between interrupt routine and program level process.
     */
    shm_id = shmget(IPC_PRIVATE, sizeof(struct dcom), SHMFLG|IPC_CREAT);
    if (shm_id == -1)
        err_rtn(badshmget, 0);

    /*
     * Attach to the shared memory segment.
     */
    if ((shmaddr = shmat(shm_id, 0, 0)) == (void *)-1)
        err_rtn(badshmat, 0);

    /*
     * Remove shmid when detached.
     */
    if ((shmctl(shm_id, IPC_RMID, 0)) == -1)
        err_rtn(badshmctl, 0);

    /*
     * Initialize shared memory segment.
     */
    dcom_p            = (struct dcom *)shmaddr;
    dcom_p->enabled   = 0;
    dcom_p->int_count = 0;
    dcom_p->vec_num   = 0;
    dcom_p->status    = WAIT_FOR_ME;

    /*
     * Fork off a child process to receive the rtc interrupts.
     */
    if ((child = fork()) == -1)
        err_rtn(badfork, 0);
    if (!child) {
        /*
         * The child process.  Open the rtc device file.
         */
        if ((file_dev = open(dev_string, O_RDWR)) == -1) {
            dcom_p->status = BAD_SETUP;
            err_rtn(baddevopen, 0);
        }

        /*
         * Get the interrupt vector number for rtc.
         */
        if (ioctl(file_dev, IOCTLVECNUM, &dcom_p->vec_num) == -1) {
            dcom_p->status = BAD_SETUP;
            err_rtn(badioctl, 0);
        }

        /*
         * Setup the connection structure.
         */
        ic.ic_version = IC_VERSION1;
        ic.ic_flags   = 0;
        ic.ic_vector  = dcom_p->vec_num;
        ic.ic_routine = rtcintr;
        ic.ic_stack   = (int)(&interr_stack[STACK_SIZE]);
        ic.ic_value   = VALUE;

        /*
         * Create an interrupt connection definition.
         */
        if (iconnect(ICON_CONN, &ic) == -1) {
            dcom_p->status = BAD_SETUP;
            err_rtn(badicon, 0);
        }

        /*
         * Close the rtc device file.
         */
        close(file_dev);
        file_dev = -1;

        /*
         * Lock the program down.
         */
        if (mlockall(MCL_CURRENT) == -1) {
            dcom_p->status = BAD_SETUP;
            err_rtn(badlock, ERR_DISC);
        }

        /*
         * Indicate to parent that child is
         * ready to receive interrupts.
         */
        dcom_p->status = 0;
        /*
         * Enable the interrupt connection.
         */
        if (ienable(dcom_p->vec_num) == -1) {
            dcom_p->status = BAD_SETUP;
            err_rtn(badienable, ERR_DISC);
        }

        /*
         * Detach from shared memory segment, and exit.
         */
        shmdt(shmaddr);
        printf("Interrupt routine child process exiting\n");

        exit(0);
    }

    /*
     * Parent process.  Get my lwp id for the server_wake1() call.
     */
    dcom_p->lid = _lwp_global_self();

    /*
     * Wait for the user interrupt process to get ienabled.
     */
    sleep(1);/* let the child run */
    while (dcom_p->status == WAIT_FOR_ME)
        sleep(1);

    if (dcom_p->status == BAD_SETUP) {
        /*
         * Child didn't make it into ienable().
         */
        shmdt(shmaddr);
        exit(1);
    }

    /*
     * Open up the rtc device file.
     */
    if ((file_dev = open(dev_string, O_RDWR)) == -1)
        err_rtn(baddevopen, 2);

    /*
     * Set the real time clock to fire
     * repetitively once every 100 milliseconds.
     */
    rtc.r_modes  = RTC_DEFAULT | RTC_REPEAT;
    rtc.r_res    = MSEC;
    rtc.r_clkcnt = 100;

    if (ioctl(file_dev, RTCIOCSET, &rtc) == -1)
        err_rtn(badrtcset, 2);
    /*
     * Start the clock counting.
     */
    if (ioctl(file_dev, RTCIOCSTART, 0) == -1)
        err_rtn(badrtcstart, 2);

    /*
     * Block until the user interrupt routine wakes us up.
     */
    if (server_block(0, 0, 0) == -1)
        err_rtn(badserverblk, 2);

    /*
     * Interrupt routine woke us up.
     */
    if (dcom_p->status == BAD_VALUE) {
        /*
         * Interrupt routine received a bad arg parameter.
         */
        printf("Bad interrupt handler routine argument value.");
    }
    else if (NUM_REQUESTS != dcom_p->int_count) {
        /*
         * Check that the interrupt routine updated the count.
         */
        printf("Count mismatch failure: count = %d, expected %d\n",
            dcom_p->int_count, NUM_REQUESTS);
    }
    else
        printf("Completed test successfully\n");

    /*
     * Disconnect the child from the eti.
     */
    iconnect(ICON_DISC, dcom_p->vec_num);

    /*
     * Stop the rtc clock and close the file.
     * Detach the shared memory segment.
     */
    ioctl(file_dev, RTCIOCSTOP, 0);
    close(file_dev);
    shmdt(shmaddr);

    exit(0);
}
```

```
/*
 * Routine to output passed error message, and the errno meaning.
 */
err_rtn(mesg, flag)
char *mesg;
int flag;
{
    fprintf(stderr, "example1: %s\n", mesg);
    fprintf(stderr, "          %s\n", sys_errlist[errno]);

    if (flag == ERR_DISC)
        iconnect(ICON_DISC, dcom_p->vec_num);

    if (shmaddr)
        shmdt(shmaddr);

    if (file_dev != -1)
        close(file_dev);

    exit(1);
}


/*
 * User interrupt routine.
 */
void
rtcintr(value)
int value;
{
    extern int rtcincr();


    /*
     * Check the value parameter.
     */
    if (value != VALUE) {
        /*
         * Something is wrong.  Wake up parent to shut down test.
         */
        dcom_p->status = BAD_VALUE;
        server_wake1(dcom_p->lid, 0);
        return;
    }
    /*
     * Increment the count.
     */
    if (!done && (rtcincr())) {
        /*
         * Count has reached the end.  Wakeup parent.
         */
        dcom_p->status = 0;
        server_wake1(dcom_p->lid, 0);
    }
}


int
rtcincr()
{
    dcom_p->int_count++;

    if (dcom_p->int_count == NUM_REQUESTS) {
        done = 1;
        return(1);
    }
    else
        return(0);
}
```

D
Example 2 - User-Level Interrupt Routines

This appendix contains an example C program that demonstrates use of the user-level interrupt routine facility by a multithreaded program that creates two interrupt connections to two separate real-time clocks. For example purposes, the **spl_request(3X)** routines are used, along with the threads library **_spin(3synch)** lock mechanism, to coordinate accesses to a shared data counter that is located within the address space of the process.

This program should be built with both the **thread** and **ud** libraries:

$ **cc -D_REENTRANT [***options***]** *file* **-lthread -lud**

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <synch.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/ipl.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <sys/rtc.h>
#include <sys/iconnect.h>


/*
 * One thread connection structure for each interrupt connection.
 */
struct thrconn {
    int   vector;    /* interrupt vector number */
    int   stack;     /* thread's interrupt stack location */
    int   fd;        /* file descriptor */
    u_int flags;     /* state flags */
    char  *filename; /* device file name */
    int   count;     /* interrupt counter */
};

/*
 * flags field
 */
#define READY 0x1      /* thread is ready - ienable()ed */
#define DONE  0x2      /* number of interrupts has been reached */

#define INT_COUNT 100  /* receive 100 interrupts before quitting */
```

```
/*
 * Interrupt stacks.
 */
#define STACK_SIZE 4096

char istack1[STACK_SIZE];
char istack2[STACK_SIZE];

/*
 * rtc device files
 * Arbitrarily use the fourth and fifth clocks on the first CPU board.
 */
char dev_string1[] = "/dev/rrtc/0c3";
char dev_string2[] = "/dev/rrtc/0c4";

#define NUM_INTS 2 /* connecting to two interrupts */

struct thrconn thrconnects[] = {
    0, (int)&istack1[STACK_SIZE], 0, 0, (char *)&dev_string1, 0,
    0, (int)&istack2[STACK_SIZE], 0, 0, (char *)&dev_string2, 0 };

/*
 * Shared data that is modified by both interrupt routine threads.
 * Protected with a _spin(3synch) spin lock.
 */
spin_t spinlock;

/*
 * Address of spl register mapping.
 * Used for raising ipl with spl_request_macro().
 */
caddr_t spl_addr;

/*
 * At the end of the test, this count should equal the
 * sum of all the interrupts:  INT_COUNT * NUM_INTS.
 */
int total_int_count;


main()
{
    int i, done, status;
    struct rtc      rtc;
    struct thrconn  *tcnp;
    struct timespec ts;
    thread_t threadid;
    extern void     run_test(void *);
```

```
/*
 * Map the spl register into our address space. We don't care
 * where the virtual address for the spl register is bound.
 */
spl_addr = spl_map(0);
if (spl_addr == (caddr_t)-1) {
    printf("spl_map() failure, errno = %d\n", errno);
    exit(1);
}

/*
 * Initialize the spin lock.
 * The second parameter is reserved for future use.
 */
if ((status = _spin_init(&spinlock, (void *)NULL)) != 0) {
    printf("_spin_init() failure, returned %d\n", status);
    exit(1);
}

/*
 * Setup each interrupt connection structure.
 */
for (i = 0, tcnp = thrconnects; i < NUM_INTS; i++, tcnp++) {
    /*
     * Open the rtc device file.
     */
    if ((tcnp->fd = open(tcnp->filename, O_RDWR)) == -1) {
        printf("rtc open() failure, errno = %d\n", errno);
        exit(1);
    }

    /*
     * Get the interrupt vector number for this rtc.
     */
    if (ioctl(tcnp->fd, IOCTLVECNUM, &tcnp->vector) == -1) {
        printf("rtc ioctl() IOCTLVECNUM failure, errno = %d\n",
            errno);
        exit(1);
    }
}
```

```
    /*
     * Create a bound thread for each interrupt connection.
     * Each thread will connect to the specified interrupt.
     */
    for (i = 0, tcnp = thrconnects; i < NUM_INTS; i++, tcnp++) {
        status = thr_create(
            (void *)0, /* stack address - use default */
            (size_t)0,/* stack size - use default */
            (void *(*) (void *))run_test,
                    /* routine address */
            (void *)tcnp, /* arg to pass to routine */
            THR_BOUND, /* flags - create a bound thread */
            (thread_t *)&threadid);
                    /* returns thread id */

        if (status != 0) {
            printf("thr_create() failure, returned %d\n", status);
            exit(1);
        }
    }

    /*
     * Wait for each thread to get connected to the interrupt.
     */
    done = 0;
    while (!done) {
        done = 1;
        for (i = 0, tcnp = thrconnects; i < NUM_INTS; i++, tcnp++) {
            if ((tcnp->flags & READY) == 0) {
                sleep(1);
                done = 0;
                break;
            }
        }
    }

    /*
     * Now lock the entire program down.
     */
    if (mlockall(MCL_CURRENT) == -1) {
        printf("mlockall() failure, errno = %d\n", errno);
        exit(1);
    }
```

```
/*
 * Loop for each rtc clock interrupt.
 */
for (i = 0, tcnp = thrconnects; i < NUM_INTS; i++, tcnp++) {
    /*
     * Set the real time clock to fire
     * repetitively once every 100 milliseconds.
     */
    rtc.r_modes = RTC_DEFAULT | RTC_REPEAT;
    rtc.r_res = MSEC;
    rtc.r_clkcnt = 100;

    if (ioctl(tcnp->fd, RTCIOCSET, &rtc) == -1) {
        printf("rtc ioctl() RTCIOCSET failure, errno = %d\n", errno);
        exit(1);
    }

    /*
     * Start the clock counting.
     */
    if (ioctl(tcnp->fd, RTCIOCSTART, 0) == -1) {
        printf("rtc ioctl() RTCIOCSTART failure, errno = %d\n", errno);
        exit(1);
    }
}

/*
 * Wait for each thread to get the required number of interrupts.
 */
ts.tv_sec = 0;
ts.tv_nsec = 250000000;/* 1/4th a second wait */

done = 0;

while (!done) {
    done = 1;
    for (i = 0, tcnp = thrconnects; i < NUM_INTS; i++, tcnp++) {
        if ((tcnp->flags & DONE) == 0) {
            (void) nanosleep(&ts, (struct timespec *)0);
            done = 0;
            break;
        }
    }
}
```

```
    /*
     * Shut things down.
     */
    for (i = 0, tcnp = thrconnects; i < NUM_INTS; i++, tcnp++) {
        /*
         * Stop the rtc clock and close the file.
         */
        (void) ioctl(tcnp->fd, RTCIOCSTOP, 0);
        close(tcnp->fd);

        /*
         * Disconnect the child from the rtc.
         */
        (void) iconnect(ICON_DISC, tcnp->vector);
    }

    /*
     * Remove the spl register binding from our address space.
     */
    if (spl_unmap(spl_addr) == -1) {
        printf("spl_unmap() failure, errno = %d\n");
        exit(1);
    }

    /*
     * Check that the total count is correct.
     */
    if (total_int_count != (NUM_INTS * INT_COUNT)) {
        printf("total_int_count = %d, expected %d\n",
            total_int_count, NUM_INTS * INT_COUNT);
        exit(1);
    }

    /*
     * Successful test.
     */
    exit(0);
}
```

```
/*
 * Each created thread starts execution here.
 * The thrconn structure for the interrupt is passed in.
 */
void
run_test(void *thrconnp)
{
    int status;
    struct thrconn *tcnp;
    struct icon_conn ic;
    void extern rtcintr(struct thrconn *);


    /*
     * Setup the connection structure.
     * Pass thrconn structure to the interrupt routine.
     */
    tcnp = (struct thrconn *)thrconnp;
    ic.ic_version = IC_VERSION1;
    ic.ic_flags   = 0;
    ic.ic_vector  = tcnp->vector;
    ic.ic_routine = rtcintr;
    ic.ic_stack   = (int)tcnp->stack;
    ic.ic_value   = (int)tcnp;

    /*
     * Create an interrupt connection definition.
     */
    if (iconnect(ICON_CONN, &ic) == -1) {
        printf("ICON_CONN failure, vector = %d, errno = %d\n",
            tcnp->vector, errno);
        exit(1);
    }

    /*
     * Indicate to program level thread that
     * this thread is ready to receive interrupts.
     */
    tcnp->flags = READY;

    /*
     * Enable the interrupt connection.
     */
    if (ienable(tcnp->vector) == -1) {
        printf("ienable() failure, vector = %d, errno = %d\n",
            tcnp->vector, errno);
        exit(1);
    }

    printf("Interrupt vector %d thread exiting\n", tcnp->vector);

    status = 0;
    thr_exit((void *)&status);
}
```

```
/*
 * User-level interrupt routine.
 * All rtc interrupts come here when they become active.
 */
void
rtcintr(struct thrconn *tcnp)
{
    pl_t old_pl, junk_pl;


    if (tcnp->count < INT_COUNT) {
        tcnp->count++;/* private increment */

        /*
         * Raise ipl to prevent other rtc interrupts
         * from comming in on this CPU.
         */
        spl_request_macro(PL8, spl_addr, old_pl);

        /*
         * Acquire the spin lock to serialize with other CPUs.
         */
        _spin_lock(&spinlock);

        /*
         * Increment shared counter.
         */
        total_int_count++;

        /*
         * Let go of the spin lock.
         */
        _spin_unlock(&spinlock);

        /*
         * Drop ipl to the previous level.
         */
        spl_request_macro(old_pl, spl_addr, junk_pl);

        /*
         * Let program level thread know that we're done.
         */
        if (tcnp->count == INT_COUNT)
            tcnp->flags |= DONE;
    }
}
```

E
HSDE Example Programs

This appendix contains example programs that have been developed to illustrate use of the high–speed data device, HSDE. The programs are written in C and demonstrate use of the master–slave protocol described in "Using a Master–Slave Transfer Protocol" in Chapter 12 to transfer files from a master high–speed data device to a slave high–speed data device.

"HSDE Device Command and Status Definitions" contains a listing of the device command and status definitions. "HSDE Attach Routine" contains a listing of a routine that is used by all example programs to attach virtual address space to a physically contiguous high–speed data device (HSDE) I/O buffer. "Master HSDE Control Program" contains a listing of the program that controls the master HSDE. Section "Slave HSDE Control Program" contains a listing of the program that controls the slave HSDE.

"Master HSDE Data Chain Program" contains a listing of a simple program that demonstrates the techniques for data chain operations with a master mode HSDE. "Slave HSDE Data Chain Program" contains a listing of a simple program that demonstrates the techniques for data chain operations with a slave mode HSDE.

"Master HSDE Command Chain Program" contains a listing of a simple program that demonstrates the techniques for command chain operations with a master mode HSDE. "Slave HSDE Command Chain Program" contains a listing of a simple program that demonstrates the techniques for command chain operations with a slave mode HSDE.

HSDE Device Command and Status Definitions

```
/*
 * HSDE device command definitions:
 */

#define HSDE_PUT        1       /* Put a file from master to slave */
#define HSDE_EOF        2       /* End of file, transfer complete */

/* macro to roundup 'val' to next multiple of 'x',
 * where 'x' is a power of two (2).
 */
#define ROUNDUP(val, x) (((val)+((x)-1))&~((x)-1)) /* x is a power of 2! */
```

HSDE Attach Routine

```
/*
 * hsde_attach -- attach to a physically contiguous hsde i/o buffer.
 *
 * Note: this routine assumes that a physically contiguous shared memory
 *       segment has already been created with the following attributes:
 *
 *                       key = ftok(hsde)
 *                       size >= 'buf_size' bytes
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

extern int errno;

char *
hsde_attach(char *hsde_name, int buf_size)
{
        int     shm_key;
        int     shm_id;
        char    *shm_vaddr;

        /* Map the hsde file name into a unique IPC key */
        shm_key = ftok(hsde_name, 0);

        /* Get a shared mememoy identifier associated with shm_key */
        shm_id = shmget(shm_key, buf_size, SHM_R|SHM_W);
        if (shm_id < 0) {
                printf("hsde_attach: shmget() failed, errno = %d\n", errno);
                exit(1);
        }

        /* attach to the physically contiguous hsde i/o buffer */
        shm_vaddr = (char *)shmat(shm_id, 0, SHM_R | SHM_W);
        if ((int)shm_vaddr == -1) {
                printf("hsde_attach: shmat() failed, errno = %d\n", errno);
                exit(1);
        }

        /* return hsde i/o buffer address */
        return(shm_vaddr);
}
```

Master HSDE Control Program

```
/*
 * hsde_master_put -- Put a file across an hsde to hsde link.
 *                    NOTE:  This program assumes that a longword transfer
 *                           is to be undertaken.  Therefore, both the file
 *                           name and the file size must be multiples of the
 *                           size of a longword.
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/hsde.h>

#include "hsdeput_ex.h"

int     hsde_fd;                      /* hsde file descriptor */
char    *hsde_buf;                    /* pointer to hsde i/o buffer */
char    *hsde_name = "/dev/hsde0";    /* hsde special file name */

extern  int errno;

main()
{
        int          file_fd;        /* local file file descriptor */
        int          hsde_bc;        /* read/write byte count */
        int          args;           /* number of input arguments */
        int          bc;             /* byte count */
        configuration_block_t hsde_modes;      /* hsde mode structure */
        hsde_iocb_t  hsde_iocb;      /* hsde i/o command block structure */
        char         line[81];       /* line input buffer */
        char         local_name[81]; /* local file name */
        u_long       hsde_status;    /* slave hsde status */

        /* Open the hsde */
        hsde_fd = open(hsde_name, O_RDWR);
        if (hsde_fd == -1) {
                perror("hsde open");
                exit(1);
        }
        /* Get and set the hsde configuration */
        if (ioctl(hsde_fd, HSDE_GET_MODE, &hsde_modes) == -1) {
                perror("ioctl HSDE_GET_MODE");
                exit(1);
        }
        hsde_modes.hsde_operation_mode = HSDE_MASTER;
        hsde_modes.hsde_data_path_width = HSDE_LONG;
        if (ioctl(hsde_fd, HSDE_SET_MODE, &hsde_modes) == -1) {
                perror("ioctl HSDE_SET_MODE");
                exit(1);
        }
        /* Attach to the hsde shared memory segment.
         * Use the HSDE common hsde_attach() routine.
         * This also assumes that the /etc/rc hsde shmconfig
         * entries have been established.
         */
        hsde_buf = (char *) hsde_attach(hsde_name, 0x40000);

main_loop:

        /* Prompt for local/remote filenames */
        printf("Local filename [Remote filename] ? ");
        if (gets(line) == NULL)
                exit(0);

        /* Scan in local and remote file names */
        args = sscanf(line, "%s %s", local_name, hsde_buf);
        if (args == 1)
                strcpy(hsde_buf, local_name);

        /* Open local file */
        file_fd = open(local_name, O_RDWR);
        if (file_fd == -1) {
                printf("hsde master: can not open %s.\n", local_name);
                goto main_loop;
        }

        /* Make sure slave hsde is connected */
        if (ioctl(hsde_fd, HSDE_STATUS, &hsde_status) == -1)
        {
                perror("ioctl HSDE_STATUS");
                exit(1);
        }

        hsde_iocb.i_opcode = HSDE_OP_COMMAND;
        hsde_iocb.i_info = 0;
        hsde_iocb.i_command = HSDE_PUT;
        if (ioctl(hsde_fd, HSDE_COMMAND, &hsde_iocb) == -1)
        {
                perror("ioctl HSDE_COMMAND");
                exit(1);
        }

        /*
         * Recall it is assumed a longword transfer is taking place; therefore,
         * strlen(hsde_buf)+1 (+1 for the string-termination character, '\0')
         * must be a multiple of a longword size
         */
        hsde_bc = ROUNDUP((strlen(hsde_buf)+1), HSDE_LONG);

        /* Send the file name to the slave hsde */
        if (write(hsde_fd, hsde_buf, hsde_bc) == -1)
        {
                fprintf(stderr, "write %d bytes failed (errno %d)\n", hsde_bc);
                perror("hsde file name write");
                exit(1);
        }

        /* Send file data */
        while ((bc = read(file_fd, hsde_buf, HSDE_MAXBC(&hsde_modes))) > 0) {

                /* Recall it is assumed a longword transfer is taking place;
                 * therefore, the writes must be a multiple of a longword.
                 * For this example, if the local file is not a multiple of
                 * four (4) bytes, the remote file size will be rounded to
                 * the a multiple of  4 bytes
                 */
                hsde_bc = ROUNDUP(bc, HSDE_LONG);
                printf("read %d bytes, write %d bytes\n", bc, hsde_bc);
                fflush(stdout);

                if (write(hsde_fd, hsde_buf, hsde_bc) == -1)
                {
                        fprintf(stderr,
                                "write %d bytes failed (errno %d)\n",
                                 hsde_bc, errno);
                        perror("hsde file write");
                        exit(1);
                }
        }

        /* File transfer complete -- sync up with slave hsde and send */
        /*                           HSDE_EOF command                 */
        if (ioctl(hsde_fd, HSDE_STATUS, &hsde_status) == -1)
        {
                perror("ioctl HSDE_STATUS");
                exit(1);
        }
        hsde_iocb.i_opcode = HSDE_OP_COMMAND;
        hsde_iocb.i_info = 0;
        hsde_iocb.i_command = HSDE_EOF;
        if (ioctl(hsde_fd, HSDE_COMMAND, &hsde_iocb) == -1)
        {
                perror("ioctl HSDE_COMMAND");
                exit(1);
        }

        /* See if user wants to send another file */
        goto main_loop;
}
```

Slave HSDE Control Program

```
/*
 * hsdeslaved -- hsde daemon process.
 *
 *      Put hsde in slave mode and listen for HSDE_PUT requests.
 *      NOTE:  This program assumes that all operations will use
 *             longword transfers.
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/hsde.h>

#include "hsdeput_ex.h"

int     hsde_fd;                    /* hsde file descriptor */
char    *hsde_buf;                  /* pointer to hsde i/o buffer */
char    *hsde_name = "/dev/hsde1"; /* hsde special file name */

extern int errno;

main()
{
        int         hsde_bc;        /* read/write byte count */
        configuration_block_t  hsde_modes;    /* hsde mode structure */
        hsde_iocb_t  hsde_iocb;     /* hsde i/o command block structure */
        u_long      hsde_status;    /* hsde device status register value */

        /* Open the hsde */
        hsde_fd = open(hsde_name, O_RDWR);
        if (hsde_fd == -1)
        {
                printf("hsdeslaved: can not open %s.\n", hsde_name);
                exit(1);
        }

        /* Enable slave mode and set the transfer size to HSDE_LONG */
        if (ioctl(hsde_fd, HSDE_GET_MODE, &hsde_modes) < 0)
        {
                perror("ioctl HSDE_GET_MODE");
                exit(1);
        }
        hsde_modes.hsde_operation_mode = HSDE_SLAVE;
        hsde_modes.hsde_data_path_width = HSDE_LONG;
        if (ioctl(hsde_fd, HSDE_SET_MODE, &hsde_modes) < 0)
        {
                perror("ioctl HSDE_SET_MODE");
                exit(1);
        }

        /* Attach to a physically contiguous hsde i/o buffer
```

```
	 * using the HSDE common routine hsde_attach().
	 * This also assumes that the /etc/rc hsde shmconfig
	 * entries have been established.
	 */
	hsde_buf = (char *)hsde_attach(hsde_name, 0x40000);

	/* loop forever, as daemons like to do */
	while (1) {

		/* Get the next command */
		if (ioctl(hsde_fd, HSDE_GET_CMD, &hsde_iocb) == -1)
		{
			perror("ioctl HSDE_GET_CMD");
			exit(1);
		}

		/* Process the command */
		switch(hsde_iocb.i_opcode) {

		case HSDE_OP_COMMAND:
			/* Process application specific command */
			if (hsde_iocb.i_command == HSDE_PUT) {
				hsde_put();
				break;
			}
			else
				/* fall through */

		default:
			printf("hsdeslaved: invalid command received.\n");
			break;
		}
	}
}

/*
 * hsde_put -- receive a file.
 */
hsde_put()
{
	hsde_iocb_t	hsde_iocb;
	int		bc;
	int		file_fd;
	int		eof;

	/* Get next command, should be a write command */
	if (ioctl(hsde_fd, HSDE_GET_CMD, &hsde_iocb) == -1)
	{
		perror("hsde_put: ioctl HSDE_GET_CMD");
		exit(1);
	}
	if (hsde_iocb.i_opcode != HSDE_OP_WRITE) {
		printf("hsde_put: write command expected.\n");
		return;
```

```
        }

        /* Read target file name which has been placed into the hsde
         * reserved memory segment.
         * NOTE:  Transfer count (i_tc) must be converted
         *        to bytes for read() by multiplying
         *        it by the transfer path width (i.e.,
         *        in this case, longword).
         */
        if (read(hsde_fd, hsde_buf, (hsde_iocb.i_tc*HSDE_LONG)) < 0)
        {
                perror("hsde_put: read error");
                exit(1);
        }

        printf("hsde_put: create %s\n", hsde_buf);
        fflush(stdout);

        /* Create target file */
        file_fd = creat(hsde_buf, 0777);
        if (file_fd < 0)
        {
                perror("hsde_put: create");
                exit(1);
        }

        /* Receive file data */
        eof = 0;
        while (! eof) {
                /* Get next command, should be a write, or
                 * HSDE_EOF device command
                 */
                if (ioctl(hsde_fd, HSDE_GET_CMD, &hsde_iocb) == -1 )
                {
                        perror("hsde_put: ioctl HSDE_GET_CMD");
                        exit(1);
                }

                switch (hsde_iocb.i_opcode) {
                case HSDE_OP_WRITE:
                        /* read file data, and write it to target file */
                        /* NOTE:  Transfer count (i_tc) must be converted
                         *        to bytes for read/write by multiplying
                         *        it by the transfer path width (i.e.,
                         *        in this case, longword).
                         */
                        if (read(hsde_fd, hsde_buf,
                                          (hsde_iocb.i_tc * HSDE_LONG)) == -1)
                        {
                                perror("hsde_put: read");
                                exit(1);
                        }
                        if (write(file_fd, hsde_buf,
                                           (hsde_iocb.i_tc*HSDE_LONG)) == -1)
                        {
                                perror("hsde_put: write");
                                exit(1);
                        }
                        break;

                case HSDE_OP_COMMAND:
                        if (hsde_iocb.i_command == HSDE_EOF) {
                                /* end of file */
                                printf("hsde_put: EOF\n");
                                fflush(stdout);
                                eof = 1;
                                break;
                        }
                        else
                                /* fall through */

                default:
                        /* invalid command received, abort transfer */
                        printf("hsde_put: put protocol violation.\n");
                        eof = 1;
                }
        }

        /* close target file */
        close(file_fd);
}
```

Master HSDE Data Chain Program

```
/*
 *  mdc - Master HSDE data chain test program.
 *        Simple test program in which the master HSDE requests a block of
 *        data from a slave HSD device.  The slave will send the data as
 *        a data chain of 3 buffers of 16 bytes each.  Each buffer will
 *        be filled with the letter 'A', 'B', and 'C' respectively.
 *        This program operates in conjunction with the slave HSDE data
 *        chain program "sdc.c".
 *
 *  Execute:  "mdc /dev/hsdeX"  where X is the HSDE minor device number.
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/hsde.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int  hsde_fd;
```

```
char *hsde_buf;
char *hsde_attach();

main(int  argc, char **argv)
{
        configuration_block_t   hsde_modes;  /* HSDE configuration struct. */

        /* Open the HSDE device. */
        hsde_fd = open(argv[1], O_RDWR);
        if (hsde_fd == -1) {
                perror("open");
                exit(1);
        }

        /* Get and set the HSDE configuration. */
        if (ioctl(hsde_fd, HSDE_GET_MODE, &hsde_modes) < 0) {
                perror("ioctl HSDE_GET_MODE");
                exit(1);
        }
        hsde_modes.hsde_operation_mode = HSDE_MASTER;
        hsde_modes.hsde_data_path_width = HSDE_LONG;
        if (ioctl(hsde_fd, HSDE_SET_MODE, &hsde_modes) < 0) {
                perror("ioctl HSDE_SET_MODE");
                exit(1);
        }

        /* Attach to the HSDE shared memory segment using the
         * HSDE common hsde_attach() routine. This also assumes
         * that the /etc/rc.d/hsde shmconfig entries have been
         * established.
         */
        hsde_buf = (char *) hsde_attach(argv[1], 0x40000);

        /* Clear out the shared memory segment. */
        bzero(hsde_buf, 64);

        /* Read from the slave HSD device. */
        if (read(hsde_fd, hsde_buf, 48) < 0) {
                perror("read");
                exit(1);
        }

        /* Display the data. */
        printf("Contents of hsde_buf:\n");
        printf("%s\n", hsde_buf);

        /* Close the HSDE device. */
        close(hsde_fd);
}
```

Slave HSDE Data Chain Program

```
/*
 *  sdc - Slave HSDE data chain test program.
 *        Slave will write a data chain to the remote HSDE master device.
 *        This simple test program builds and initializes three buffers
 *        in the slave HSDE's reserved shared memory segment.  The buffers
 *        are then pointed to by iovec structures within an array.  This
 *        array of iovec structures is then written to the HSDE slave.
 *
 *  Execute:  "sdc /dev/hsdeX"  where X is the HSDE minor device number.
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/hsde.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/uio.h>

char *hsde_attach();
int  hsde_fd;            /* HSDE file descriptor. */
char *hsde_buf;          /* Pointer to the HSDE shared memory segment. (Also
                            pointer to the first buffer in the segment.) */
char *hsde_buf2;         /* Pointer to second buffer in shared memory segment.*\
/
char *hsde_buf3;         /* Pointer to third buffer in shared memory segment. *\
/

/* Iovec array which constitutes the HSDE data chain. */
#define IOVEC_CNT       3
struct iovec    iov[IOVEC_CNT];

main(int  argc, char **argv)
{
        configuration_block_t hsde_config; /* HSDE configuration struct. */
        int             err;               /* System call return value. */
        int             i;                 /* For loop counter variable. */
        char            *bufptr;           /* Pointer to data buffers. */

        /* Open the HSDE device. */
        hsde_fd = open(argv[1], O_RDWR);
        if (hsde_fd == -1) {
                perror("open");
                exit(1);
        }

        /* Get and set the HSDE configuration. */
        err = ioctl(hsde_fd, HSDE_GET_MODE, &hsde_config);
        if (err == -1)
        {
                perror("ioctl HSDE_GET_MODE");
                exit(1);
        }
        hsde_config.hsde_operation_mode = HSDE_SLAVE;
        hsde_config.hsde_data_path_width = HSDE_LONG;
        err = ioctl(hsde_fd, HSDE_SET_MODE, &hsde_config);
        if (err == -1)
        {
                perror("ioctl HSDE_SET_MODE");
                exit(1);
        }

        /* Attach to HSDE shared memory segment using the
         * HSDE common hsde_attach() routine.
         * This also assumes that the /etc/rc hsde shmconfig
         * entries have been established.
         */
        hsde_buf = (char *) hsde_attach(argv[1], 0x40000);

        /* Allocate data buffers in the shared memory segment. */
        hsde_buf2 = hsde_buf + 16;
        hsde_buf3 = hsde_buf2 + 16;

        /* Zero out a portion of the shared memory segment. */
        bzero(hsde_buf, 64);

        /* Initialize data buffers in the shared memory segment. */
        for (i = 0, bufptr = hsde_buf; i < 16; i++, bufptr++)
                *bufptr = 'A';
        for (i = 0, bufptr = hsde_buf2; i < 16; i++, bufptr++)
                *bufptr = 'B';
        for (i = 0, bufptr = hsde_buf3; i < 16; i++, bufptr++)
                *bufptr = 'C';

        /* Initialize the data chain. */
        iov[0].iov_base = (caddr_t)hsde_buf;
        iov[0].iov_len = 16;

        iov[1].iov_base = (caddr_t)hsde_buf2;
        iov[1].iov_len = 16;

        iov[2].iov_base = (caddr_t)hsde_buf3;
        iov[2].iov_len = 16;

        /* Send the data chain to the master. */
        hsde_send();
}

/*
 *  hsde_send - Send the data chain to the master upon request.
 */
hsde_send()
{
        hsde_iocb_t     hsde_iocb;       /* IOCB received from master. */
        int             err = 0;         /* System call return value. */
        /* Await request from the master HSDE device. */
        err = ioctl(hsde_fd, HSDE_GET_CMD, &hsde_iocb);
        if (err == -1)
        {
                perror("ioctl HSDE_GET_CMD");
                exit(1);
        }

        /* Inspect IOCB received.  Make sure it is a READ operation. */
        if ((hsde_iocb.i_opcode & HSDE_OP_MASK) != HSDE_OP_READ)
        {
                printf("slave: Received invalid request: %x\n",
                                                    hsde_iocb.i_opcode);
                exit(1);
        }

        /* Send the iovec array containing the data chain. */
        err = writev(hsde_fd, iov, IOVEC_CNT);
        if (err < 0)
        {
                perror("writev");
                exit(1);
        }

}
```

Master HSDE Command Chain Program

```
/*
 *  mcc.c - HSDE master mode command chain test.
 *          This is a very simple command chain test.  A command
 *          chain is built in which an 8-byte file name is transferred to
 *          the slave device.  Within the chain, device status and commands
 *          are added.  A reserved shared memory segment is utilized by
 *          the HSDE in which to place and transfer data.  After one
 *          execution of the command chain, the chain is reexecuted or
 *          repeated 6 times.  NOTE:  This transfer utilizes a data
 *          path width of LONG.
 *
 *  Execute:  "mcc /dev/hsdeX"  where X is the HSDE minor device number.
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/hsde.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "hsdeput_ex.h"
```

```
#define COUNT   6       /* Number of times to repeat command chain.
                        * NOTE:  See hsde(7) man page concerning the
                        *        repetition of command chains.
                        */
int  hsde_fd;
char *hsde_buf;
char *hsde_attach();

main(int argc, char **argv)
{
        configuration_block_t   hsde_modes;  /* HSDE configuration struct. */
        hsde_iocb_t     hsde_iocb[5];        /* Command chain */
        int             xfer_count = 0;      /* # times chain is executed */
        int             i;                   /* For loop counter variable */

        /* Open the HSDE device. */
        hsde_fd = open(argv[1], O_RDWR);
        if (hsde_fd < 0) {
                perror("open");
                exit(1);
        }

        /* Get and set the HSDE configuration. */
        if (ioctl(hsde_fd, HSDE_GET_MODE, &hsde_modes) < 0)
        {
                perror("ioctl HSDE_GET_MODE");
                exit(1);
        }

        hsde_modes.hsde_operation_mode = HSDE_MASTER;
        hsde_modes.hsde_data_path_width = HSDE_LONG;
        hsde_modes.enable_cmd_chain_mode = 1;
        if (ioctl(hsde_fd, HSDE_SET_MODE, &hsde_modes) < 0)
        {
                perror("ioctl HSDE_SET_MODE");
                exit(1);
        }

        /* Attach to the reserved shared memory segment.
         * This also assumes that the /etc/rc hsde shmconfig
         * entries have been established.
         */
        hsde_buf = (char *) hsde_attach(argv[1], 0x40000);

        /* Place the file name in the shared memory segment. */
        strcpy(hsde_buf, "testfile");

        /* Initialize the command chain.
         * Note the HSDE_OP_CCHAIN usage.
         */
        hsde_iocb[0].i_opcode = HSDE_OP_STATUS | HSDE_OP_CCHAIN;
        hsde_iocb[0].i_info = 0;
        hsde_iocb[0].i_command = 0L;
        hsde_iocb[0].i_la = 0L;

        hsde_iocb[1].i_opcode = HSDE_OP_COMMAND | HSDE_OP_CCHAIN;
        hsde_iocb[1].i_info = 0;
        hsde_iocb[1].i_tc = 0;
        hsde_iocb[1].i_command = HSDE_PUT;
        hsde_iocb[1].i_la = 0L;

        hsde_iocb[2].i_opcode = HSDE_OP_WRITE | HSDE_OP_CCHAIN;
        hsde_iocb[2].i_info = 0;
        hsde_iocb[2].i_tc = 2;
        hsde_iocb[2].i_ta = (int)hsde_buf;
        hsde_iocb[2].i_la = 0L;

        hsde_iocb[3].i_opcode = HSDE_OP_STATUS | HSDE_OP_CCHAIN;
        hsde_iocb[3].i_info = 0;
        hsde_iocb[3].i_tc = 0;
        hsde_iocb[3].i_command = 0L;

        /* Note last IOCB in chain does not contain HSDE_OP_CCHAIN bit. */
        hsde_iocb[4].i_opcode = HSDE_OP_COMMAND;
        hsde_iocb[4].i_info = 0;
        hsde_iocb[4].i_tc = 0;
        hsde_iocb[4].i_command = HSDE_EOF;
        hsde_iocb[4].i_la = 0L;

loop:
        /* Write the chain to the HSDE device. */
        if ((xfer_count = write(hsde_fd, hsde_iocb, 5)) < 0) {
                perror("write");
                exit(1);
        } else
                printf("hsde performed %d commands in the chain.\n",
                                xfer_count);

        /* Cycle or reexecute the chain COUNT times. */
        for( i = 0; i < COUNT; i++) {
                if (ioctl(hsde_fd, HSDE_CYCLE_CHAIN, 0) < 0) {
                        perror("ioctl HSDE_CYCLE_CHAIN:\0");
                        exit(1);
                } else
                        printf("hsde performed cmd chain cycling.\n");
        }
        /* Continuously cycle the command chain. */
        printf("loop ? ");
        if ((i = getchar()) == 'y')
                goto loop;
}
```

Slave HSDE Command Chain Program

```
/*
 *  scc.c - Slave HSDE command chain test program.
 *
 *  Execute:  "scc /dev/hsdeX"  where X is the HSDE minor device number.
 *            This program runs in conjunction with the "mcc" or master
 *            mode command chain program.
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/hsde.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "hsdeput_ex.h"

char *hsde_attach();
int  hsde_fd;    /* File descriptor for HSDE device. */
char *hsde_buf;  /* Pointer to physically contiguous shared memory segment. */

main(int argc, char **argv)
{
        configuration_block_t   hsde_config; /* HSDE configuration struct. */
        hsde_iocb_t             hsde_iocb;   /* IOCB received from master. */
        int                     err;         /* Ioctl return value. */

        /* Open the HSDE device. */
        hsde_fd = open(argv[1], O_RDWR);
        if (hsde_fd == -1) {
                perror("open");
                exit(1);
        }

        /* Get and set the HSDE configuration. */
        err = ioctl(hsde_fd, HSDE_GET_MODE, &hsde_config);
        if (err == -1)
        {
                perror("ioctl HSDE_GET_MODE");
                exit(1);
        }
        hsde_config.hsde_operation_mode = HSDE_SLAVE;
        hsde_config.hsde_data_path_width = HSDE_LONG;
        err = ioctl(hsde_fd, HSDE_SET_MODE, &hsde_config);
        if (err == -1)
        {
                perror("ioctl HSDE_SET_MODE");
                exit(1);
        }

        /* attach to shared memory.
```

```
         * This also assumes that the /etc/rc hsde shmconfig
         * entries have been established.
         */
        hsde_buf = (char *) hsde_attach(argv[1], 0x40000);

        while(1)
        {
                err = ioctl(hsde_fd, HSDE_GET_CMD, &hsde_iocb);
                if (err == -1)
                {
                        perror("ioctl HSDE_GET_CMD");
                        exit(1);
                }

                /* Remember to mask out the command chain bit when
                 * reading the IOCB opcode.
                 */
                switch(hsde_iocb.i_opcode & ~HSDE_OP_CCHAIN)
                {
                case HSDE_OP_COMMAND:
                        if (hsde_iocb.i_command == HSDE_PUT)
                                printf("HSDE_PUT command received.\n");
                        else if (hsde_iocb.i_command == HSDE_EOF)
                                printf(" HSDE_EOF command received.\n");
                        else
                                printf("Invalid command received: %x\n",
                                                hsde_iocb.i_command);
                        break;

                case HSDE_OP_STATUS:
                        printf("Status request received.\n");
                        break;

                case HSDE_OP_WRITE:
                        printf(" Write command received.\n");
                        hsdeput(&hsde_iocb);
                        break;

                default:
                        printf("Invalid IOCB opcode received: %x\n",
                                                hsde_iocb.i_opcode);
                        break;
                }
        }
}

/*
 * hsdeput - Receive a file name from the master HSDE.
 * Argument is an IOCB received from the master HSDE.
 */
hsdeput(hsde_iocb_t *iocbp)
{
        int             err = 0;        /* System call return value. */
        int             i;              /* For loop counter variable. */
        int             hsde_cnt;       /* Byte count to be transferred. */
        char *          hsde_buf_pt;    /* Pointer to the HSDE shared
                                           memory. */

        /* Calculate byte count to be transferred from the master. */
        hsde_cnt = (iocbp->i_tc * HSDE_LONG);

        /* Zero out the HSDE shared memory buffer. (Optional) */
        bzero(hsde_buf, 1024);

        /* Initiate read operation with master side to receive data
         * into the shared memory buffer.
         */
        err = read(hsde_fd, hsde_buf, hsde_cnt);
        if (err < 0)
        {
                perror("read");
                exit(1);
        }

        /* Display name placed into shared memory buffer. */
        printf("hsde slave: File name received:\n");
        hsde_buf_pt = hsde_buf;
        for (i=0; i < hsde_cnt; i++)
        {
                printf("%c", *hsde_buf_pt);
                hsde_buf_pt++;
        }
        printf("\n");
}
```

Index

Spine for 1” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

Progr

Real-Time Guide

0890466