Power Hawk Series 700
Closely-Coupled Programming Guide

CONCURRENT COMPUTER CORPORATION

0891087-000
June 2001

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent Computer Corporation makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2881 Gateway Drive Pompano Beach, FL 33069. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

UNIX is a registered trademark of The Open Group.
Ethernet is a trademark of Xerox Corporation.
PowerMAX OS is a registered trademark of Concurrent Computer Corporation.
Power Hawk and PowerStack II/III are trademarks of Concurrent Computer Corporation.

Other products mentioned in this document are trademarks, registered trademarks, or trade names of the manufactures or marketers of the product with which the marks or names are associated.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original	000	PowerMAX OS Release 5.1

Preface

Scope of Manual

This manual is intended for programmers that are writing applications which are distributed across multiple single board computers (SBCs) which either share the same VMEbus or which are connected via a Real-time Clock and Interrupt Module (RCIM). Programming interfaces which allow communication between processes resident on separate single board computers in such a configuration are discussed. For information on configuring and administering these configurations, see the *Power Hawk Series 700 Diskless Systems Administrator's Guide*.

Structure of Manual

This manual consists of a title page, this preface, a master table of contents, four chapters, local tables of contents for the chapters, one appendix, glossary of terms, and an index.

- Chapter 1, *Introduction*, contains an overview of closely-coupled systems (CCS) and the programming interfaces that are unique to closely-coupled single board computer (SBC) configurations.
- Chapter 2, *Reading and Writing Remote SBC Memory,* explains how to use shared memory to read and write remote SBC memory in a cluster configuration.
- Chapter 3, *Shared Memory,* explains how SBCs within the same cluster can be configured to share memory with each other.
- Chapter 4, *Inter-SBC Interrupt Generation and Notification,* describes how program interfaces are available via **`ioctl(2)`** commands to interrupt SBCs within the same cluster in an CCS system.
- Glossary explains the abbreviations, acronyms, and terms used throughout the manual.

The index contains an alphabetical list of all paragraph formats, character formats, cross reference formats, table formats, and variables.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.

`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man 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 appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments

Referenced Publications

Concurrent Computer Corporation Manuals:

0890429	*System Administration Manual (Volume 1)*
0890430	*System Administration Manual (Volume 2)*
0891084-reln	*Power Hawk Series 700 PowerMAX OS Release Notes* (reln = release number)
0890466	*PowerMAX OS Real-Time Guide*
0890479	*PowerMAX OS Guide to Real-Time Services*
0891086	*Power Hawk Series 700 Diskless System Administrator's Guide*
0891082	*Real-Time Clock & Interrupt Module (RCIM) User's Guide*
0890425	*Device Driver Programming Manual*

Contents

Tables

1
Introduction

Overview

This manual is a guide to the programming interfaces that are unique to closely-coupled single board computer (SBC) configurations. A closely-coupled configuration is one where there are multiple Series 700 SBCs in the same VME backplane that are also connected together with a common PCI-to-PCI (P0Bus) Bus. The programming interfaces described in this book allow inter-process communication between processes that are resident on separate SBCs in a closely-coupled configuration. Many of these interfaces are designed to be compatible with the interfaces available for interprocess communication on a symmetric multi-processor. The *Power Hawk Series 700 Diskless Systems Administrator's Guide* is a companion to this manual and contains information on configuring, booting and administering closely-coupled configurations as well as other diskless configurations.

The types of inter-process, inter-board communication mechanisms supported for transferring data include:

Shared memory

A shared memory region is located in the physical memory of one SBC that is located in the VME cluster. Other SBCs access that physical memory across the P0Bus, through configured P0Bus upstream and downstream windows. Once configured, access to shared memory is accomplished through either the **shmat(2)** family of system calls or via the **mmap(2)** system call in the same manner as access to shared memory regions which are strictly local to one SBC.

Posix message queues

These interfaces can be used to pass data across the P0Bus between processes that reside on different SBCs in the cluster. P0Bus messages are used to pass data to and from a message queue. Storage space for the messages in the message queue is user-defined to be resident on one SBC in the VME cluster.

P0Bus networking sockets

Standard network protocols can be configured to operate across the P0Bus. The P0Bus is then utilized like any other network fabric. The standard **socket(3)** interfaces can be used to establish P0Bus networking connections between processes that are running on different SBCs in the same cluster.

DMA to reserved memory on another board

Data cam be DMAed directly onto the memory of another SBC that is within the same cluster. Physical memory must be reserved on an SBC in order to use this DMA capability. Data can be transferred directly to and from this reserved memory via **`read(2)`**/**`write(2)`** calls. The data will be DMAed across either the VMEBus or the P0Bus, depending upon the device file that is being used.

The types of inter-process, inter-board communication mechanisms supported for synchronization and notification include:

Signals

It is possible to send a signal to a process on another SBC. The interface is not the standard signal interface, but rather an **`ioctl(2)`** to **`/dev/targetn`**. This system call causes a mail-box interrupt to be generated on another processor which results in a signal being delivered to the process that has registered for notification of the arrival of that interrupt.

Posix semaphores

These interfaces can be used to synchronize access to shared memory data structures or to asynchronously notify a process on another SBC that is in the same cluster. The semaphore is user-defined to be resident on a particular SBC. Messages are passed across the P0Bus to that SBC and local test and set operations guarantee that only one process can lock the semaphore at any given point in time.

VME interrupt generation

Using an **`ioctl(2)`** to **`/dev/vmebus/targetn`** it is possible to generate a VME interrupt. This interrupt can be caught on another processor using a user-level interrupt connection to the VME vector.

Mailbox interrupt generation

An inter-SBC mailbox interrupt may be used to remotely generate a mailbox interrupt on one SBC from a remote SBC that is located in the same cluster. The generation of this interrupt is accomplished by writing across the P0Bus to a specific memory location on the SBC that is receiving the mailbox interrupt. Mail-box interrupts are generated and caught via an **`ioctl(2)`** to **`/dev/targetn`**. Notification of the arrival of a mailbox interrupt can be either via a user-level interrupt or a signal.

RCIM interrupt generation

The RCIM is a Concurrent-developed PMC board, which provides additional connectivity between SBCs. It is possible to generate an interrupt on another SBC when both boards share an RCIM connection. The advantage of RCIM connected boards is that there is no latency in sending an interrupt, because there is no

need to gain access to the P0Bus for passing interrupt notification messages.

Frequency-based scheduling

A frequency-based scheduler (hereinafter also referred to as FBS) is a task synchronization mechanism that enables you to run processes at fixed frequencies in a cyclical pattern. Processes are awakened and scheduled for execution based on the elapsed time as measured by a real-time clock, or when an external interrupt becomes active (used for synchronization with an external device).

While the standard FBS support may be used to schedule processes within a single SBC, there are also Coupled FBS extensions to the FBS support which may be used to provide cluster-wide synchronization of processes by using frequency-based schedulers that are running off of the same Coupled FBS timing device. In this case, each SBC in the cluster may have its own local scheduler attached to the same Coupled FBS timing device that other schedulers residing on other SBCs within the same cluster are also using. It should also be mentioned that there are two types of Coupled FBS timings devices: Closely Coupled and RCIM Coupled timing devices. While Closely Coupled timing devices may be used by each SBC in within the same cluster, RCIM Coupled timing devices may be used by any mix of stand lone SBCs, netbooted SBCs, and SBCs within a closely-coupled cluster, as long as certain configuration requirements are met. See the *PowerMAX OS Guide to Real-Time Service* manual for more information about using these two types of Coupled FBS timing devices.

Both the standard and the Coupled FBS timing devices allow for the use of the integral real-time clocks and the RCIM real-time clocks and edge-triggered interrupts as the timing devices for FBS schedulers.

Except for RCIM-based operations and for **`read(2)`** and **`write(2)`** operations explicitly issued on **`/dev/vmebus`** device files, the communication mechanisms previously mentioned in this chapter all utilize the P0Bus for communicating between processes that are running on separate SBCs. Because of the need to arbitrate for the P0Bus and because of the indeterminism of gaining this access in the presence of other P0Bus block transfers, these operations can be significantly slower than similar inter-process operations on a symmetric multiprocessor. For this reason, care must be taken in deciding processor assignments for the tasks that comprise a distributed application on a closely-coupled system.

The most efficient means of transferring large amounts of data between SBCs is to use the DMA capability for transferring data directly into or out of the memory of another SBC. This technique requires only a single arbitration of the P0Bus for transferring each block of DMA data. P0Bus networking sockets are efficient in terms of their access usage on the P0Bus (that is, they use the DMA capability in the same way as described above), but there is additional overhead in transferring data because of the network protocols used in this style of communication. For some applications, TCP/IP sockets would be the communication mechanism of choice because: 1) a TCP/IP socket provides a reliable

connection between the two processes, and 2) sockets across the P0Bus have exactly the same user interface as sockets across any other network fabric and are thus a more portable interface.

2
Reading and Writing Remote SBC Memory

Overview

In addition to using shared memory to read and write remote SBC memory in a cluster, the **read(2)** and **write(2)** system services calls are available to examine or modify another SBC's local DRAM memory. Read and write act on an SBC's physical memory. Therefore, **read(2)** and **write(2)** operations to remote SBC memory should usually be done to physical memory that is either reserved, or is part of the Slave Mmap memory area.

While the **read(2)** and **write(2)** system calls require the caller to enter the kernel in order to access the remote memory, this method is still more efficient than the shared memory method for transferring larger amounts of data between SBCs. This is because read and write use DMA transfers which make more efficient use of the P0Bus or VMEbus than the single word transfers performed when using shared memory CPU accesses. Unlike the shared memory method, the **read** and **write** method places no restrictions on the number of other SBCs that may be accessed from one SBC, and also, requires less kernel configuration setup.

Note that the read/write interface is only available between SBC's in the same cluster (i.e., SBC's residing in same VME chassis and therefore, sharing the same VME and PCI-to-PCI (P0) buses). In this chapter, the term "remote SBC" refers to another SBC and/or its memory in the same cluster as the SBC (sometimes referred to as the "local SBC") doing the read/write operation.

User Interface

Device Files

Reading and writing from or to a remote SBC's memory is accomplished by opening the appropriate SBC device file, followed by issuing the appropriate sequence of **lseek(2)**, **read(2)**, **readv(2)**, **write(2)**, **writev(2)** and **close(2)** system service calls.

On Series 700 closely-coupled systems, the set of SBC device files that may be used for reading and writing are:

/dev/host, **/dev/target[n]**
/dev/p0bus/host, **/dev/p0bus/target[n]**
/dev/vmebus/host, **/dev/vmebus/target[n]**

On Series 700 closely-coupled systems, the host and **target[n]** device files located in **/dev** and **/dev/p0bus** are functionally equivalent; they will both result in **read(2)** and **write(2)** DMA transfers across the P0Bus. Use of the higher-speed P0Bus is generally recommended over the slower VMEBus, especially if there are I/O devices located on the VME Bus that could cause contention for use of the VMEBus. However, the **/dev/vmebus** device files may also be used for issuing **read(2)** and **write(2)** DMA transfers across the VMEBus, if the user so chooses.

The file server/host SBC's device files are the **/dev/host**, **/dev/p0bus/host** and **/dev/vmebus/host** files, where the file server SBC always has a board id of 0. It should also be mentioned that the **/dev/target0**, **/dev/p0bus/target0** and **/dev/vmebus/target0** files also correspond to the file server/host SBC.

The other SBCs in the cluster have target device file names, where a SBC with a board id of 2, for example, would correspond to the device files **/dev/target2**, **/dev/p0bus/target2**, or **/dev/vmebus/target2**.

NOTE

Applications that execute on both Series 600 and Series 700 closely-coupled systems should usually make use of the **/dev/host** and **/dev/target[n]** device files whenever possible, instead of the **/dev/p0bus** device files, which are not available on Series 600 systems. On both Series 600 and 700 closely-coupled systems, the **/dev/host** and **/dev/target[n]** device files correspond to the default I/O bus. On Series 600 systems, the default I/O bus is the VMEBus, and on Series 700 systems, the default I/O bus is the P0Bus. Therefore, by using the **/dev/host** and **/dev/target[n]** device files, the **read(2)/write(2)** application will run on either type of closely-coupled system without the need for any source code changes to the name of the SBC device file that is **open(2)ed** for the **read(2)** and **write(2)** operations.

Using lseek, read and write Calls

The **read** and **write** data transfers are accomplished through use of an on-board DMA controller for transferring the data to and from a remote SBC's DRAM memory across either the P0Bus or the VMEbus.

More than one process may open a SBC's device file at the same time; the coordination between use of these device files is entirely up to the user.

It is not possible to **read** or **write** the physical memory on the local SBC; either **shmbind(2)** or **mmap(2)** of **/dev/mem** or the user accessible slave shared memory (SMAP) may be used to access locally reserved physical memory.

Usually **lseek(2)** is used first to set the starting physical address location on the remote SBC. The physical address offsets specified on **lseek(2)** calls should be as though the

memory was being accessed locally on that SBC, starting with physical address 0. No checking of the specified offset is made during the **`lseek(2)`** call; if the offset specified is past the end of the remote SBC's memory, then any error notification will not occur until the subsequent **`read(2)`** or **`write(2)`** call is issued.

CAUTION

The read/write interface allows writing data to any memory location on every other SBC in the same cluster. Writing to an incorrect address can have severe effects on the remote SBC; crashes and data corruption may occur.

Following the **`lseek(2)`** call, the **`read(2)`** or **`write(2)`** commands may be used to read or write the data from or to the remote SBC physical memory locations. When successful, the **`read(2)`** or **`write(2)`** call will return the number of bytes read or written. When the current offset to read or write is beyond the end of the remote SBC memory, zero will be returned as the byte count. When the entire number of bytes cannot be read or written due to reaching the end of remote SBC memory, then as many bytes as possible will be read or written, and this amount will be returned to the caller as the byte count.

Although any source and target address alignments and any size byte counts may be used to read and write the remote memory locations, for best performance, double-word aligned source and target addresses should be used, along with double-word multiple byte counts. Following these restrictions allows the 64 bit DMA transfer mode to be used instead of the slower 32 bit transfer mode.

When the byte count of a **`read(2)`** or **`write(2)`** call is greater than the value of the tunable **P0BUS_DIRECT_BC** for P0Bus transfers, or is greater than the value of the tunable **DMAC_DIRECT_BC** for VMEBus transfer, then the user's data will be directly DMA'd into or out of the user's buffer. In this case, the user must have the **P_PLOCK** privilege. To further improve performance, the application writer may want to also lock down the pages where the buffer resides before making the subsequent **`read(2)`** and **`write(2)`** calls, in order to lower the amount of page locking processing done by the kernel during the **`read(2)`** or **`write(2)`** calls, although this is not required.

When the byte count is less than or equal to **P0BUS_DIRECT_BC** for P0Bus transfers, or **DMAC_DIRECT_BC** for VMEBus transfers, the user's data is copied in or out of a kernel buffer, where the kernel buffer becomes the source or target of the DMA operation.

The system administrator may use the **`config(1M)`** utility to examine or modify the **P0BUS_DIRECT_BC** or **DMAC_DIRECT_BC** tunables. Note that in order to modify or examine these tunables for a SBC other than the host SBC, the **`-r`** option must be used to specify the virtual root directory of the client SBC.

Using ioctl Commands

There are also several **ioctl(2)** commands that may be helpful for supplementing the application's **read(2)** and **write(2)** system service calls. Applications that use **read(2)** and **write(2)** can determine their own board id with the **SBCIOC_GET_BOARDID ioctl(2)** command:

```
#include <sys/sbc.h>
int fd, board_id;
ioctl(fd, SBCIOC_GET_BOARDID,& board_id);
```

where the local SBC's board id is returned at the location **board_id**, and the value in **board_id** will contain a value from 0 to *n*.

Since, presumably, the local board id is not known at the time that this **ioctl(2)** call is made, the '**fd**' would normally be the file descriptor of an **open(2)** call that was made by opening the **/dev/host** device file, since the **/dev/host** file will always be present on all SBCs in a cluster configuration.

Once the local SBC board id is known, it may also be useful to know what other SBCs are present within the cluster. This information may be obtained with the **SBCIOC_GET_REMOTE_MASK ioctl(2)** command:

```
#include <sys/sbc.h>
int fd;
u_int board_mask;
ioctl(fd, SBCIOC_GET_REMOTE_MASK, & board_mask};
```

Upon return from this call, a bit mask of all the remote SBC board ids that are present in the cluster will be returned at the location **board_mask** (SBC0 is the least significant bit). The **fd** file descriptor may be obtained by opening any **/dev/host** or **/dev/target[-*n*]** file, as long as that device file corresponds to a SBC that is actually present within the cluster.

To transfer data from local memory to (or from) a remote SBC's slave shared memory segment, information about a given SBC's Slave Mmap shared memory area may be obtained with the **SBCIOC_GET_SWIN_INFO ioctl(2)** command:

```
#include <sys/sbc.h>
int fd;
swinfo_t si;
ioctl(fd, SBCIOC_GET_SWIN_INFO, &si);
```

Upon return from this call, information on the remote client's slave shared memory area is returned in the **swinfo_t** structure. The **swinfo_t** structure contains various information about the slave window configuration.

The **swinfo_t data** structure

```
typedef struct sbc_swin_info {
ulong_t flags;      /* flags defined below */
ulong_t win_size;   /* size of the slave window */
paddr_t win_base;   /* physical I/O address where window exists */
ulong_t dmap_size;  /* size of the DMAP area */
```

```
paddr_t dmap_addr;   /* physical address of DMAP area */
ulong_t mmap_size;   /* size of the MMAP area */
paddr_t mmap_addr;   /* physical address of MMAP area */
paddr_t dmac_addr;   /* lseek(2) offset for DMA read/write */
paddr_t bind_addr;   /* shmbind(2) address for mmap area */
} swinfo_t;
```

flags — This field describes information about the window. The following flags bits are currently defined:

SWIN_DYNAMIC_MEMORY indicates that the slave DRAM used by the local SBC was dynamically allocated during the system initialization process.

SWIN_RESERVED_MEMORY indicates that the slave DRAM used by the local SBC uses system reserved memory as defined by the **res_sects[]** array in the MM device driver (**../pack.d/io/mm/space.c**) and the SBC device driver's **SBC_SLAVE_MMAP_START** tunable.

SWIN_P0_BUS indicates that this Slave Mmap shared memory will be remotely accessed fromn across the P0Bus. This flag will always be set on Power Hawk Series 700 systems.

SWIN_VME_BUS indicates that this Slave Mmap shared memory area will be remotely accessed from across the VME Bus. This flag will never be set on Power Hawk Series 700 systems. (Only the Power Hawk Series 600 closely coupled systems access the Slave Mmap shared memory area from across the VME Bus.)

win_size — This field reports the P0Bus slave window size, in bytes, that is configured for each SBC in the cluster. This size was defined with the **SBC_SLAVE_MMAP_MAXSZ** tunable.

win_base — This field reports the physical P0Bus address where the start of this SBC's slave window resides, out on the P0Bus.

dmap_size — This field reports the size, in bytes, of the kernel portion of the Slave Window. Closely-coupled system drivers use this area to report information about each other (such as the `swinfo_t` data), as well as for passing messages between SBCs.

dmap_addr — This field reports the local processor relative physical address used to access the DMAP portion of the slave shared memory area. This area is reserved for kernel use.

mmap_size* — This field reports the actual size of the user accessible slave shared memory area. If this value is zero, then the remote SBC has not been configured with a slave shared memory area.

mmap_addr — This field reports the local processor relative physical address used to access the user accessible MMAP portion of the slave shared memory area.

dmac_addr* — The field reports the offset into the remote SBC's memory used to access the slave shared memory area. Using **si.dmac_addr** in

the **lseek(2)** "offset" argument (assuming the "whence" field is set to **SEEK_SET**) points to the first byte of user accessible slave shared memory (SMAP) area. If this value is zero, then the **read(2)** and **write(2)** interface cannot be used to perform a data transfer.

bind_addr* — The field reports the address to be used in **shmbind(2)** shared memory accesses. If this value is zero, then **shmbind(2)** cannot be used to access the slave shared memory area.

NOTE

User level processes which need to access slave shared memory will normally only need to reference "**mmap_size**" to see if the client has defined a shared memory area, "**dmac_addr**" if the application is going to use the **read(2)**/**write(2)** interface, and "**bind_addr**" if the application is going to use **shmbind(2)** to access shared memory.

In addition to the **ioctl(2)** commands that return information about SBC board ids and slave shared memory information, there is another **ioctl(2)** command that can be used to send a VME interrupt to another SBC within the cluster. This **ioctl(2)** could be used, for example, to notify a remote SBC that new data has been placed into its memory. The interface to this command is:

```
#include <sys/sbc.h>
int fd;
u_short irq_vector;
ioctl(fd, SBCIOC_GEN_VME_INTR, irq_vector};
```

Where **'fd'** is a file descriptor of a **/dev/vmebus/host** or **/dev/vmebus/target[n]** SBC device file where the VME interrupt is to be sent. Note that a **/dev/vmebus** device file MUST be used for this **ioctl(2)** command. The `irq_vector` contains the VME interrupt request level (`irq`) in the most significant byte and the VME vector number in the least significant byte.

This command will broadcast a VME interrupt on the VME backplane at the interrupt request level specified. The SBC that receives this VME interrupt will process the interrupt using the interrupt vector routine that corresponds to the VME vector number that was specified in `irq_vector`.

The VME interrupt request level in `irq_vector` should be in the range of 1 to 7, and the VME vector number in `irq_vector` should be in the range of 0 to 255. This vector number must be a vector that the receiving SBC is specifically set up to process, either with a kernel interrupt handling routine, or a user-level interrupt routine (see below).

There are several configuration requirements and restrictions to be followed in order to properly use this **ioctl(2)** to send an interrupt to another SBC in the cluster.

The sending SBC (the SBC making the **ioctl(2)** call) must not be enabled to receive the VME level interrupt specified in `irq_vector`. The kernel may be disabled for receiving this VME level by using **config(1M)** to set the appropriate **VME_IRQ[1-7]_ENABLE** tunable to 0. Note that in order to modify or examine the

VME_IRQ[1-7]_ENABLE tunable for a SBC other than the host SBC, the **-r** option must be used to specify the virtual root directory of the client SBC.

The SBC that is to receive the VME interrupt should have its kernel enabled for receiving the VME level interrupt. The **config(1M)** utility should be used to set the appropriate **VME_IRQ[1-7]_ENABLE** tunable to 1. Only one SBC kernel in the cluster should be enabled to receive the VME interrupt. Note that in order to modify or examine the **VME_IRQ[1-7]_ENABLE** tunable for a SBC other than the host SBC, the **-r** option must be used to specify the virtual root directory of the client SBC.

The SBC that is to receive the VME interrupt should also have either a kernel or user-level interrupt handler for processing the VME interrupt vector. On the receiving SBC, a specific interrupt vector should usually be be allocated by using the **iconnect(3c)** **ICON_IVEC** command, with the **II_ALLOCATE** and **II_VECSPEC** **ii_flags** specified in the **icon_ivec** structure. By allocating a specific interrupt vector, the sending SBC will know which interrupt vector to use in its `irq_vector` parameter.

Refer to the *Device Driver Programming* manual for details on writing a kernel interrupt routine and refer to the "User-Level Interrupt Routines" chapter in the *PowerMAX OS Real-Time Guide* for details on how to allocate an interrupt vector and on how to set up a user-level interrupt routine.

Reserving Memory

Note that it is entirely up to the application to properly reserve those portions of physical memory on each SBC that will be the source or target of **read(2)** or **write(2)** operations.

The reservation of memory is accomplished by modifying the **res_sects[]** array in the **/etc/conf/pack.d/mm/space.c** file for the host SBC, and/or the **<virtual_rootpath>/etc/conf/pack.d/mm/space.c** file for a client SBC. For example, to reserve 16384 bytes of memory, starting at the 24MB physical memory location, the following entry would be added:

```
struct res_sect res_sects[] = {
        /* r_start, r_len, r_flags */
        {0x1800000, 0x4000, 0};
        {0, 0, 0}
};
```

Note that the last entry should always be the NULL (zero) entry, for the purpose of terminating the list.

Sample Application Code

The sample code shown below illustrates some basic examples of how to use the **read(2)** and **write(2)** system services, along with the **ioctl(2)** calls previously mentioned.

This sample code accomplishes the following:

- creates a I/O buffer space using **`mmap(2)`**,
- locks down the buffer space,
- determines the local SBC id,
- gets the mask of all remote SBC ids,
- chooses one remote SBC to read/write to,
- fills the write buffer with a data pattern,
- **`lseek(2)`**s to set the remote physical memory address start location,
- **`write(2)`**s the data to the remote SBC memory via the P0Bus,
- **`lseek(2)`**s to reset the remote physical memory address start location,
- **`read(2)`**s the data back from remote SBC memory via the P0Bus,
- verifies that the data is valid,
- sends a VME interrupt to the remote SBC, using the appropriate **`/dev/vmebus`** SBC device file,
- closes file descriptors and exits.

Note that the following assumptions are made in this sample code:

- The physical address range from **`0x1800000`** to **`0x1803fff`** has been reserved on the remote SBC in the **res_sect[]** array.
- The remote SBC's **VME_IRQ3_ENABLE** tunable has been set to 1, and all other SBCs in the cluster, including the local SBC, has set this tunable to 0.
- The remote SBC has either a kernel interrupt routine or user-level interrupt routine set up to handle interrupt vector 252.
- The remote SBC memory has not been modified by another SBC between the time that the **`write(2)`** and **`read(2)`** calls are made by this local SBC; otherwise, the data pattern comparison would fail.

Begin Sample Application Code --

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/mman.h>
#include <sys/sbc.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

/*
 * File descriptors.
 */
int local_fd;              /* The local SBC */
int remote_fd;             /* The remote SBC */

/*
 * File name buffer.
 */
char filename[MAXPATHLEN];

/*
 * Physical address range starts at 24mb for 4 pages.
 */
#define PHYS_START_ADDR    0x1800000
#define BUFFER_SIZE        0x4000

/*
 * Starting value for the write buffer.
 */
#define PATTERN_SEED       0x10203040

/*
 * The VME level and vector for sending a VME interrupt to the remote SBC.
 */
#define INT_VECTOR         252  /* interrupt vector (0xfc) */
#define VME_LEVEL          3    /* VME level 3 */

/*
 * Construct the irq_vector parameter for the SBCIOC_GEN_VME_INTR ioctl(2).
 */
u_short irq_vector = ((VME_LEVEL << 8) | INT_VECTOR);


main(argc, argv)
int argc;
char **argv;
{
     int i, status, value, fd;
     int local_board_id;        /* local SBC id */
     int *bufferp;              /* mmap(2)ed buffer */
     int *bp;                   /* pointer to walk through the buffer */
     int remote_board_id;       /* remote SBC id to read/write */
     u_int remote_board_id_mask;
                                /* mask of all remote SBCs in the cluster */
```

```
/*
 * mmap(2) a zero-filled buffer into the address space.
 */
fd = open("/dev/zero", O_RDWR);
if (fd == -1) {
     printf("ERROR: open(2) of /dev/zero failed, errno %d\n",
          errno);
     exit(1);
}

bufferp = (int *)mmap((void *)NULL, (size_t)BUFFER_SIZE,
     PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE, fd, 0);
if (bufferp == (int *)-1) {
     printf("ERROR: mmap(2) failed, errno %d\n", errno);
     exit(1);
}
close(fd);

/*
 * Lock down I/O buffer to improve performance.
 */
status = memcntl((caddr_t)bufferp, (size_t)BUFFER_SIZE,
     MC_LOCK, 0, 0, 0);
if (status == -1) {
     printf("ERROR: memcntl(2) failed, errno = %d\n", errno);
     exit(1);
}

/*
 * Open(2) the host device file, since it is known to exist.
 */
fd = open("/dev/host", O_RDWR);
if (fd == -1) {
     printf("ERROR: open(2) of /dev/host failed, errno %d\n",
          errno);
     exit(1);
}

/*
 * Get our local SBC board id.
 */
status = ioctl(fd, SBCIOC_GET_BOARDID, &local_board_id);
if (status == -1) {
     printf("ERROR: SBCIOC_GET_BOARDID ioctl(2) failed, errno %d\n",
          errno);
     exit(1);
}

/*
 * Open our local SBC board id.
 */
if (local_board_id) {
     close(fd);
     sprintf(filename, "/dev/target%d", local_board_id);
     local_fd = open(filename, O_RDWR);
     if (local_fd == -1) {
          printf("ERROR: open(2) of %s failed, errno %d\n",
               filename, errno);
```

```
          exit(1);
     }
}
else {
     /*
      * Local SBC is host, just use existing fd.
      */
     local_fd = fd;
}

/*
 * Get the mask of remote SBC board ids.
 */
status = ioctl(local_fd, SBCIOC_GET_REMOTE_MASK, &remote_board_id_mask);
if (status == -1) {
     printf( “ERROR: SBCIOC_GET_REMOTE_MASK ioctl(2) failed, errno %d\n”,
          errno);
     exit(1);

}
if (!remote_board_id_mask) {
     printf(“ERROR: no remote SBCs found.\n”);
     exit(1);
}

/*
 * Use the first remote SBC id in the returned id mask.
 */
for (i = 0; remote_board_id_mask; i++) {
     if (remote_board_id_mask & 1)
          break;
     remote_board_id_mask >>= 1;
}
remote_board_id = i;

/*
 * Open(2) the remote SBC device file.
 */
if (remote_board_id)
     sprintf(filename, “/dev/target%d”, remote_board_id);
else
     strcpy(filename, “/dev/host”);

remote_fd = open(filename, O_RDWR);
if (remote_fd == -1) {
     printf(“ERROR: remote open(2) of %s failure, errno = %d\n”,
          filename, errno);
     exit(1);
}

/*
 * Fill the write buffer with some known pattern.
 */
for (value = PATTERN_SEED, i = 0, bp = bufferp;
     i < (BUFFER_SIZE/4); i += 4, value += 1, bp++)
{
     *bp = value;
}
/*
```

```
 * Seek up to the specified starting location.
 */
status = lseek(remote_fd, PHYS_START_ADDR, SEEK_SET);
if (status == -1) {
     printf("ERROR: lseek() for write failure, errno = %d\n",
          errno);
     exit(1);
}

/*
 * Write the data to the remote SBC's memory.
 */
status = write(remote_fd, bufferp, BUFFER_SIZE);
if (status == -1) {
     printf("ERROR: write(2) failure, errno = %d\n", errno);
     exit(1);
}
if (status == 0) {
     printf("ERROR: write(2) returned EOF.\n");
     exit(1);
}
if (status < BUFFER_SIZE) {
     printf("ERROR: write returned only %d bytes\n", status);
     exit(1);
}

/*
 * Set the file position back to where we started.
 */
status = lseek(remote_fd, PHYS_START_ADDR, SEEK_SET);
if (status == -1) {
     printf("ERROR: lseek() for read failure, errno = %d\n", errno);
     exit(1);
}

/*
 * Now read the data that we just wrote to see that it matches.
 */
status = read(remote_fd, bufferp, BUFFER_SIZE);
if (status == -1) {
     printf("ERROR: read(2) failure, errno = %d\n", errno);
     exit(1);
}
if (status == 0) {
     printf("ERROR: read(2) returned EOF.\n");
     exit(1);
}
if (status < BUFFER_SIZE) {
     printf("ERROR: read returned only %d bytes\n", status);
     exit(1);
}

/*
 * Check the data in the read buffer against the values expected.
 */
for (value = PATTERN_SEED, i = 0, bp = (int *)bufferp;
     i < (BUFFER_SIZE/4); i += 4, value += 1, bp++)
{
     if (*bp != value) {
```

```
            printf(“ERROR: data mismatch at offset 0x%x.\n”, i);
            printf(“   Expected 0x%x, read 0x%x\n”,
                    value, *bp);
            exit(1);
        }
    }

    /* Open the local SBC's VMEBus device file for sending
     * the VME interrupt to the remote SBC.
     */
    close(local_fd);

    sprintf(filename, "/dev/vmebus/target%d", local_board_id);
    local_fd = open(filename, O_RDWR);
    if (local_fd == -1) {
        fprintf(stderr,
            "ERROR: open(2) of %s failed, errno %d\n",
            filename, errno);
        exit(1);
    }

    /*
     * Send an interrupt to the remote SBC to let
     * it know that new data is available.
     */
    status = ioctl(local_fd, SBCIOC_GEN_VME_INTR, irq_vector);
    if (status == -1) {
        printf(“ERROR: SBCIOC_GEN_VME_INTR ioctl(2) failed, errno %d\n”,
            errno);
        exit(1);
    }

    /*
     * All done. Close the files.
     */
    close(local_fd);
    close(remote_fd);
}
```

End Sample Application Code.

3
Shared Memory

Overview

SBCs in the same cluster can be configured to share memory with each other. Accesses to shared memory on a different SBC in the cluster is done via CPU read and write accesses across the P0Bus. The P0Bus is a PCI-to-PCI bus, where each SBC contains a P0Bus bridge that connects its own local PCI bus to a common P0Bus bus.

The method that is used for configuring and accessing shared memory in a closely-coupled system (CCS) is called Slave MMap. The name comes from the fact that each SBC may optionally place a P0Bus downstream (Slave) window out on the P0Bus that provides other SBCs with remote access to a section of that SBC's local DRAM memory from across the P0Bus. This downstream P0Bus window thus provides a “Slave” “M”emory “Map” ping (Slave MMap) method for accessing a remote SBC's memory.

Slave MMap memory accesses provide the fastest and most efficient method of reading and writing small amounts of data between two SBCs.

Slave MMAP Shared Memory Overview

The Slave MMAP interface allows simultaneous access to physical memory (DRAM) on every SBC in a cluster. The local processor defines a shared memory segment which is then mapped into a P0Bus downstream window at a well known P0Bus physical address location.

Implementation:

- Each SBC driver sets up a downstream P0Bus window at a “well known” P0Bus physical I/O address into which it maps its own local shared DRAM memory.
- Each SBC driver sets up an upstream P0Bus window (which resides on the local SBC's PCI bus) that maps onto the entire range of the “well known” Slave MMap memory areas out on the P0Bus bus. This P0Bus address area consists of a range of physically contiguous P0Bus addresses, where each remote SBC's Slave MMap memory area may be accessed by using its SBC board id value as an index into this space.
- SBCs access any other SBC's Slave MMap memory area by reading and writing to the correct local PCI upstream P0Bus window on their own SBC. These accesses move through the local PCI bus upstream P0Bus window,

out onto the P0Bus, and down into the remote SBC via the remote SBC's downstream P0Bus window.

Advantages:

- Simultaneous access to all other SBCs which define a Slave MMAP memory region.
- No need to know the physical DRAM address of any of the remote memory regions in order to access them.
- No need to reconfigure existing clients when adding additional SBCs to an existing configuration.
- Supports both **`mmap(2)`** and **`shmbind(2)`** shared memory interfaces.
- Shared memory may be either dynamically allocated or statically allocated (by defining a **`res_sects[]`** in the MM driver's space.c).

Disadvantages:

- Only one contiguous shared memory region is configurable per SBC.
- The amount of physical memory that can be mapped on any one SBC is limited to one fourth of the amount of memory defined by the **VME_DRAM_WINDOW** tunable.

Accessing Shared SBC Memory

Using read(2) and write(2) to Access Shared SBC Memory

The **`read(2)`** and **`write(2)`** system service calls are available for examining and modifying another SBC's Slave MMap memory area. A highly efficient DMA block transfer mode that utilizes the embedded Symbios SCSI **`Move Memory`** command to move blocks of data across the P0Bus is used for satisfying these **`read(2)`** and **`write(2)`** requests.

The interface for using **`read(2)`** and **`write(2)`** to a Slave MMap area was previously discussed in Chapter 2, “Reading and Writing Remote SBC Memory”, under the “Using ioctl Commands” section that describes the SBCIOC_GET_SWIN_INFO **`ioctl(2)`** command. There is also an example of using this **`ioctl(2)`** command to **`read(2)`** and **`write(2)`** to the Slave MMap memory area in the diskless pkg, which is located in **`/usr/etc/diskless.d/ccs.program.examples/shared_memory/shm.c`**.

Using mmap(2) To Access Shared SBC Memory

The **`mmap(2)`** system service call can be used to access Slave MMap shared memory. The **`mmap(2)`** interface allows processes to directly map both local and remote closely-

coupled shared memory into it's own address space for normal load and store operations.

Using shmbind(2) To Access Shared SBC Memory

The **shmbind(2)** system service call can be used to access Slave MMap shared memory. The **shmbind(2)** system service call can always be used to access a remote SBC's Slave MMap memory area, regardless of whether or not the remote Slave MMap memory was dynamically or statically allocated. However, in order to successfully use **shmbind(2)** to bind to the local SBC's Slave MMap memory area, the memory must be statically allocated.

Closely-Coupled Shared Memory Limitations

The following are the limitations for using Slave MMap memory:

- The **test** and **set** type of instructions are not supported on remote shared memory through either the **mmap(2)** or **shmbind(2)** memory. (However, any such mapping to a processor's local memory may use the following system calls.) The following features make use of **test** and **set** functionality and therefore, cannot be used in remote SBC memory:
  - **_Test_and_Set(3C)** - the test and set intrinsic
  - **sem_init(3)** - the family of POSIX counting semaphore primitives
  - **synch(3synch)** - the families of Threads Library synchronization primitives including **_spin_init, mutex_init, rmutex_init, rwlock_init, sema_init, barrier_init** and **cond_init**
  - **spin_init(2)** - the family of spin lock macros
- The Slave MMap shared memory P0Bus downstream window is no longer present after a reset is issued to an SBC. This downstream window is initialized by PowerMAX OS, and is thus not available until a new kernel is downloaded and started up on the board which was reset.

  During this time interval, memory accesses from applications (local processes) accessing remote **mmap(2)** memory (or remote **shmbind(2)** accesses to Slave MMAP memory) cannot be resolved.

  The IGNORE_BUS_TIMEOUTS tunable (enabled by default in closely-coupled configurations) should be kept enabled in order to prevent a machine check panic or a system fault panic from occurring on the system that is issuing the remote memory request.

With the IGNORE_BUS_TIMEOUTS tunable enabled, the application will not receive any notification that these reads and/or writes are not completing successfully. However, writes to the remote DRAM memory will not actually take place, and the reads from the remote DRAM memory will return values of all ones. For example, word reads will return values of 0xFFFFFFFF. Once the remote SBC's Slave MMap downstream P0Bus window has been re-initialized by PowerMAX OS, the remote DRAM memory reads and writes will once again operate normally.

If the IGNORE_BUS_TIMEOUTS tunable is not enabled, a system panic will then occur. Therefore, it is recommended that the tunable IGNORE_BUS_TIMEOUTS be enabled; otherwise, applications that are known to be actively accessing the memory on a remote SBC should be stopped before that remote SBC is reset, or rebooted via **`sbcboot(1M)`**.

Slave Shared Memory (SMAP)

The Slave MMap shared memory interface (hereafter referred to as SMAP) provides an interface which supports simultaneous access to physical memory (DRAM) on every SBC in the cluster.

The SMAP interface provides shared access to it's local DRAM by creating a downstream P0Bus window out on the P0Bus that maps onto the local SBC's SMAP memory. The downstream P0Bus window is placed out on the P0Bus on a pre-configured P0Bus address range.

In order to access other remote SBC SMAP areas, each SBC additionally creates an upstream P0Bus window out on their own local PCI bus. This upstream window provides access from the local SBC out onto the P0Bus in the address ranges on the P0Bus where the remote SMAP area downstream P0Bus windows reside.

Thus, SBCs in the cluster access remote SMAP memory by attaching to the local PCI upstream P0Bus window addresses that map onto the P0Bus at the appropriate address ranges, using either the **`mmap(2)`** or **`shmbind(2)`** system call interfaces.

SMAP User Interface

SMAP mmap(2) system call interface

Access to SMAP shared memory is obtained by opening the **`/dev/host`** and/or **`/dev/target[n]`** device files, followed by an **`mmap(2)`** call, using the file descriptor that was returned from the **`open(2)`** call.

When accessing SMAP shared memory, opening the device file associated with the local SBC results in **`mmap(2)`** access to local DRAM. If the device file opened and subsequently mmap'ed refers to a remote SBC, then access to the remote SBC's DRAM will be performed over the P0Bus. For example, if SBC1 opens **`/dev/target1`**, and mmaps memory using the **`/dev/target1`** file descriptor, the mapped memory directly accesses local DRAM.

This DRAM is visible to any other SBC in the cluster when they open **/dev/target1**. The only difference is that the remote SBC accesses to this DRAM will be made over the P0Bus.

If SBC1 now opens **/dev/host**, SBC1 will be able to access memory on SBC0 (the host) over the P0Bus. Additionally, SBC1 could also open **/dev/target2** and gain access to SBC2's shared memory area.

All three memory area's in this example can be accessed at the same time.

When issuing a **mmap(2)** system call, the “off” parameter that is specified is relative to the starting physical address that is mapped by the local or remote client.

SMAP shmbind(2) system call interface

The **shmbind(2)** system call interface can be used to access all remote SBCs slave shared memory. However, if it becomes necessary to **shmbind(2)** to on-board DRAM, you must allocate the slave shared memory using the **res_sects[]** array (memory cannot be dynamically allocated). Furthermore, when **shmbind(2)**ing to this memory, the DRAM address must be used (as defined in **res_sects[]**). Do not attempt to **shmbind(2)** to local memory through the P0Bus window address. This may result in a P0Bus bus error or P0Bus hang.

SMAP Limitations and Considerations

- The upstream and downstream P0Bus SMAP shared memory windows are setup by the kernel during system initialization. This means that this memory area should only be accessed while the remote SBC is up.
- The maximum amount of SMAP shared memory that can be configured is equal to one fourth the size of the VME_DRAM_WINDOW tunable, minus 4KB. For example, in a cluster where the SBC with the largest sized DRAM is 256MB and the VME_DRAM_WINDOW is therefore set to 3 (256MB), then the largest SMAP shared memory area that may be allocated on any one SBC would be:

  (256MB / 4) - 4KB = 64MB - 4KB = 67104768 (0x3fff000) bytes

Note

When a Backplane P0 (BPP0) Bridge board is installed in the cluster, then under certain circumstances it may be necessary to further limit the maximum SMAP shared memory size to one eighth the size of the VME_DRAM_WINDOW minus 4KB. See the *Power Hawk Series 700 Diskless System Administrator's Guide*, section 4.4 “Cluster Configuration" for more details on this additional size limitation.

SMAP Kernel Configuration

SMAP Kernel Tunables

The SMAP shared memory interface is implemented using upstream and downstream windows on the P0Bus. These upstream and downstream windows are created at system initialization time, based upon certain kernel tunables (see Table 3-2). Typically, the SMAP shared memory area tunables are configured and modified by using the **`vmebootconfig(1M)`** and **`mkvmebstrap(1M)`** diskless utility commands.

Table 3-1. SMAP Kernel Tunables

Kernel Tunable	Module	Default	Min.	Max.	Unit
VME_DRAM_WINDOW	vme	2	1	5	1 = 64MB 2 = 128MB 3 = 256MB 4 = 512MB 5 = 1GB
SBC_SLAVE_MMAP_MAXSZ	sbc	1	1	17	Power of 2 index value. (See Table 3-2, “SMAP Size Index Values” for more information.)
SBC_SLAVE_MMAP_START	sbc	0	0	0x3ffff000	Physical DRAM Address
SBC_SLAVE_MMAP_SIZE	sbc	1	1	17	Power of 2 index value. (See Table 3-2, “SMAP Size Index Values” for more information.)

Table 3-2. SMAP Size Index Values

Value	Size	Value	Size	Value	Size	Value	Size	Value	Size
1	4KB	5	64KB	9	1MB	13	16MB	17	256MB
2	8KB	6	128KB	10	2MB	14	32MB		
3	16KB	7	256KB	11	4MB	15	64MB		
4	32KB	8	512KB	12	8MB	16	128MB		

The VME_DRAM_WINDOW and SBC_SLAVE_MMAP_MAXSZ tunable values are cluster-wide values (they apply to all SBCs in the cluster) that are defined in the **`/usr/etc/diskless.d/profiles.conf/cluster.profile`** file.

The VME_DRAM_WINDOW tunable should be set to a value that reflects the largest DRAM that is located on any SBC in the cluster.

The SBC_SLAVE_MMAP_MAXSZ tunable defines the largest possible Slave MMAP area of any SBC in the cluster. The SBC_SLAVE_MMAP_MAXSZ may be set to a value that is no larger than one fourth the size of the VME_DRAM_WINDOW size. Note that the first 4KB of the Slave MMap area is always set aside for kernel use. The remaining area is used as the user-accessible SMAP shared memory area. The Slave MMap area size must be a power-of-2, and the index values as defined in Table 3-2 “SMAP Size Index Values” show the valid power-of-2 sizes for this tunable.

NOTE

The SBC_SLAVE_MMAP_MAXSZ tunable is a maximum value; not all SBCs have to actually dedicate DRAM for SMAP shared memory use. The actual amount of DRAM that is used by each SBC for Slave MMap memory is defined by the per-SBC SBC_SLAVE_MMAP_SIZE tunable.

Even though a SBC may not use or access the SMAP shared memory areas, all SBCs in the cluster must be configured with the same VME_DRAM_WINDOW and SLAVE_MMAP_MAXSZ values in order to ensure proper cluster system operation.

The SBC_SLAVE_MMAP_SIZE and SBC_SLAVE_MMAP_START tunables are per-SBC tunables that are defined for each SBC in the cluster in the **`/usr/etc/diskless.d/profile.conf`** directory, within each client profile file. The file server SBC values for these tunables are defined in the **`/usr/etc/diskless.d/profile.conf/cluster.profile`** file.

The SBC_SLAVE_MMAP_SIZE tunable defines the actual Slave MMap area size for a given SBC. This tunable is also a power-of-2 tunable that is defined in an index value as defined in Table 3-2 “SMAP Size Index Values”. This tunable may be equal to or less than the SBC_SLAVE_MMAP_MAXSZ tunable value.

The SBC_SLAVE_MMAP_START tunable determines whether the Slave MMap area is statically or dynamically allocated. When SBC_SLAVE_MMAP_START is set to zero, then the Slave MMap memory area is dynamically allocated during system initialization. This is the preferred allocation setting, unless a particular application requires **`shmbind(2)`** support for accessing this Slave MMap area on the local SBC. When SBC_SLAVE_MMAP_START is non-zero, then this indicates that the SMAP area is statically allocated. In this case, SBC_SLAVE_MMAP_START must be set to a physical DRAM address value that is aligned on a boundary that is a multiple of the tunable value SBC_SLAVE_MMAP_SIZE.

When SBC_SLAVE_MMAP_START is non-zero, then the SBC driver will attempt to use the reserved memory area that must be defined in the res_sects[] array of that SBC. The SBC driver will search the res_sects[] array and try to locate an entry that starts at the SBC_SLAVE_MMAP_START value, with a length equal to the SBC_SLAVE_MMAP_SIZE tunable value.

For example, if SBC_SLAVE_MMAP_START is set to 0x1400000 and SBC_SLAVE_MMAP_SIZE is set to a value of 9 (for a 1MB size), then the following res_sects[] entry would reserve that range of physical DRAM memory:

```
  struct res_sect res_sects[] = {
  /* r_start, r_len, r_flags */
  { 0x1400000, 0x100000, 0 },  /* Slave MMap area */
  { 0, 0, 0 }  /* This must be the last line, DO NOT change
it. */
  };
```

4
Inter-SBC Synchronization and Coordination

Overview

Several mechanisms exist for processes on different SBCs to synchronize and coordinate their activities. This chapter will discuss five:

- Interrupt generation and notification. This method would primarily be used by a process to signal another process on a specific SBC.
- Remote message queues. Used to transfer data between processes located on any SBC within the cluster. The full functionality of POSIX message queues is provided.
- Remote semaphores. Used to synchronize the activities of processes located on any SBC within the cluster. The full functionality of POSIX semaphores is provided.
- CCS_FBS. Provides cluster-wide synchronization for all FBS schedulers that are attached to the same Closely Coupled timing device.
- RCIM Coupled FBS. While not specific to closely-coupled systems, RCIM Coupled FBS timing devices may used by SBCs within a single cluster for achieving cluster-wide synchronization for all FBS schedulers that are attached to the same RCIM Coupled timing device. Additionally, RCIM Coupled timing devices may also be attached to by FBS schedulers on SBCs where one or more of those SBCs may reside outside of the cluster.

Inter-SBC Interrupt Generation and Notification

Program interfaces are available via **`ioctl(2)`** commands to interrupt SBCs within the same cluster in a closely-coupled system (CCS).

Processes with the appropriate privilege (P_USERINT) may interrupt an arbitrary SBC and/or receive a notification when an interrupt is received. (For information on privileges, refer to the **`intro(2)`** and **`privilege(5)`** system manual pages, and the “Administering Privilege” Chapter in the *System Administration (Volume 1)* manual).

Associated with each inter-SBC interrupt is a “virtual interrupt” id, which ranges from 0 to 127. The effect is that there are 128 virtual interrupts available to these **`ioctl`** commands.

The **`ioctl`** commands are applied to the SBC device files, i.e. **`/dev/host`** or **`/dev/targetn`** (where n = 0 to 7).

Interrupt generation is done by specifying an SBC that will receive the interrupt in addition to a virtual interrupt id. The SBC that will receive the interrupt must be within the same cluster as the sending SBC. An interrupt may also be sent to the local SBC.

Interrupt notification is done by specifying the virtual interrupt id as well as the notification type. Notification will occur whenever the indicated virtual interrupt is received on the local SBC (regardless of originating SBC).

Interrupt notification may be done by either signal or user-level interrupt. The signal number or interrupt vector number must be specified. The caller is responsible for establishing the disposition of the signal handler or user-level interrupt routine.

Interrupt notification may be either permanent or temporary. A permanent notification remains until explicitly removed. A temporary notification is removed when a virtual interrupt (indicated id) is received.

Only one notification type is allowed per virtual interrupt.

Calling Syntax

The calling syntaxes for interrupt generation, signal notification and interrupt notification are shown below.

Interrupt Generation

```
#include <sys/sbc.h>
ioctl(fildes, SBCIOC_MBINTR_GEN, parms);
int fildes, command, *parms;
struct parms {
    int     virtual_interrupt_id;
};
```

Generates a virtual interrupt on the SBC specified by fildes. fildes is a file descriptor obtained by having previously opened **`/dev/host`** or **`/dev/targetn.`** f*ildes* determines which SBC a generated interrupt will be directed to. The receiving SBC must be within the same cluster as the sending SBC. The virtual interrupt number is specified by *virtual_interrupt_id.*

virtual_interrupt_id is a number between 0-127.

Returns:

0 : successful
ENXIO: sbc module not configured.
ENODEV: sbc device not present or not CCS system.
EINVAL: *virtual_interrupt_id* is out of range (0-127).
EHOSTDOWN: specified sbc is not available.
ENOLINK: communication failure.
EPERM: caller does not have P_USERINT privilege.
EFAULT: illegal address for *parms.*

Signal Notification

```
#include <sys/sbc.h>
ioctl(fildes, SBCIOC_MBINTR_SIGNAL, parms);
int fildes, command, *parms;
struct parms {
    int     virtual_interrupt_id;
    int     signo;
    int     op;
};
```

Attaches (or detaches) signal notification for the calling process when a virtual interrupt with the specified id is delivered to local SBC.

f*ildes* is a file descriptor obtained by opening **`/dev/host`** or **`/dev/targetn`**. The specific SBC associated with this file descriptor is not significant as notification is always delivered to the calling process.

virtual_interrupt_id is a number between 0-127. *signo* is the signal number to be used for process notification.

op is one of:

```
    SBCIOC_MBINTR_ATTACH
        -or-
    SBCIOC_MBINTR_DETACH

plus the following flag may also be Or'd in:

    SBCIOC_MBINTR_PERM
```

The operation SBCIOC_MBINTR_ATTACH attaches signal notification to the specified *virtual_ interrupt_ id*. If the flag SBCIOC_MBINTR_PERM is set, then the attachment is permanent and is only removed by an explicit SBCIOC_MBINTR_DETACH. If the flag SBCIOC_MBINTR_PERM is NOT set, then the attachment is temporary and is removed when a virtual interrupt at this id occurs (or is explicitly detached).

The operation SBCIOC_MBINTR_DETACH removes a signal notification.

`Returns:`

0 : successful
ENXIO: sbc module not configured.
ENODEV: sbc device not present or not CCS system.
EINVAL: *virtual_interrupt_id* is out of range (0-127).
EINVAL: illegal signal number.
EPERM: caller does not have P_USERINT privilege.
EBUSY: interrupt notification already present for this virtual interrupt (SBCIOC_MBINTR_ATTACH)
ESRCH: no interrupt notification for this virtual interrupt (SBCIOC_MBINTR_DETACH).
EFAULT: illegal address for *parms.*

Interrupt Notification

```
#include <sys/sbc.h>
ioctl(fildes, SBCIOC_MBINTR_UI, parms);
int fildes, command, *parms;
struct parms {
    int     virtual_interrupt_id;
    int     vector;
    int     op;
};
```

Attaches (or detaches) user-level interrupt notification when a virtual interrupt with the specified id is delivered to local SBC.

f*ildes* is a file descriptor obtained by opening **`/dev/host`** or **`/dev/targetn`**. The specific SBC associated with this file descriptor is not significant as notification is always delivered to the SBC on which the calling process is executing.

virtual_interrupt_id is a number between 0-127. *vector* is the interrupt vector number of the user-level interrupt to invoke.

op is one of:

```
    SBCIOC_MBINTR_ATTACH
        -or-
    SBCIOC_MBINTR_DETACH

plus the following flag may also be Or'd in:

    SBCIOC_MBINTR_PERM
```

The operation SBCIOC_MBINTR_ATTACH attaches user-level interrupt notification to the specified *virtual_interrupt_id*. If the flag SBCIOC_MBINTR_PERM is set, then the attachment is permanent and is only removed by an explicit. SBCIOC_MBINTR_DETACH. If the flag SBCIOC_MBINTR_PERM is NOT set, then the attachment is temporary and is removed when a virtual interrupt at this id occurs (or is explicitly detached).

The standard initialization required for user-level interrupts (i.e. **`iconnect(3C)`** and **`ienable(3C))`**, must still be done.

Note that the calling process may not necessarily be the same process that will receive the user-level interrupt. The user-level interrupt is delivered to the process which is connected to the interrupt *vector.*

For more information on user-level interrupts, refer to the "User-Level Interrupt Routines" Chapter in the *PowerMAX OS Real-Time Guide*.

The operation SBCIOC_MBINTR_DETACH removes a user-level interrupt notification.

Returns:

0 : successful
ENXIO: sbc module not configured.

ENODEV: sbc device not present or not CCS system.
EINVAL: *virtual_interrupt_id* is out of range (0-127).
EPERM: caller does not have P_USERINT privilege.
EBUSY: interrupt notification already present for this virtual interrupt (SBCIOC_MBINTR_ATTACH)
ESRCH: no interrupt notification for this virtual interrupt (SBCIOC_MBINTR_DETACH).
EFAULT: illegal address for *parms.*

Example Send/Receive Inter-SBC interrupts Programs:

The following are two simple programs that demonstrate how to send and receive inter SBC interrupts.

The first program sends the interrupt:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/iconnect.h>
#include <sys/mman.h>
#include <sys/sbc.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
** send virtual interrupt to an SBC
*/
main()
{
    int ret;        /* return values */
    charfname[100];/* device file name */
    int fileds;     /* device file descriptor */
    int vid;        /* virtual interrupt number */
    int sbcid;      /* SBC number */

    printf (“sbc id:”);/* ask operator for SBC number */
    scanf (“%d”, &sbcid);/* read in SBC number */
    sprintf (fname, “/dev/target%1d”, sbcid); /* build device file name ... */
                    /* format: /dev/targetn, n = SBC id */

    printf (“vid: “);/* ask operator for virtual interrupt ... */
                    /* number.  vids are between 0..127 */
    scanf (“%d”, &vid);/* read in virtual interrupt number */

    fileds = open (fname, O_RDWR);/* open device file of SBC to direct ... */
                    /* virtual interrupt to */
    if (fileds == -1) {
        perror (“open”);/* open failed */
        exit (1);
    }
```

```
    ret = ioctl (fileds, SBCIOC_MBINTR_GEN, &vid);  /* send virtual interrupt
*/
    if (ret == -1) {
        perror (“ioctl”);/* ioctl failed */
        exit (1);
    }
}
```

The second program receives the interrupt:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/iconnect.h>
#include <sys/mman.h>
#include <sys/sbc.h>
#include <sys/stat.h>
#include <signal.h>
#include <fcntl.h>

/*
** attach signal notification to virtual interrupt
*/
main()
{
    int ret;        /* return values */
    charfname[100];/* device file name */
    int fileds;     /* device file descriptor */
    int vid;        /* virtual interrupt number */
    int parms[3];/* parameter list */
    externvoid sig_handler();

    sprintf (fname, “/dev/host”);/* build device file name ... */
                    /* can use any SBC device file: */
                    /* /dev/host or /dev/targetn */

    printf (“vid: “);/* ask operator for virtual interrupt ... */
                    /* number.  vids are between 0..127 */
    scanf (“%d”, &vid);/* read in virtual interrupt number */

    fileds = open (fname, O_RDWR);/* open device file */
    if (fileds == -1) {
        perror (“open”);/* open failed */
        exit (1);
    }

    signal (SIGUSR1, sig_handler);/* establish signal handler */

    parms[0] = vid;/* parms[0] = virtual interrupt number */
    parms[1] = SIGUSR1;/* parms[1] = signal number to receive */
    parms[2] = SBCIOC_MBINTR_ATTACH; /* parms[2] = cmd, attach signal ... */
                    /* notification to virtual interrupt */
```

```
    parms[2] |= SBCIOC_MBINTR_PERM;/* parms[2] or-in additional flag ... */
                    /* make attachment permanent, ... */
                    /* otherwise it is removed when ... */
                    /* notification arrives */

    ret = ioctl (fileds, SBCIOC_MBINTR_SIGNAL, &parms); /* attach notification
*/
    if (ret == -1) {
        perror ("ioctl");/* ioctl failed */
        exit (1);
    }

    pause ();   /* wait for a signal */

    parms[0] = vid;/* parms[0] = virtual interrupt number */
    parms[1] = SIGUSR1;/* parms[1] = signal number to receive */
    parms[2] = SBCIOC_MBINTR_DETACH; /* parms[2] = cmd, detach signal ... */
                    /* notification to virtual interrupt */

    ret = ioctl (fileds, SBCIOC_MBINTR_SIGNAL, &parms); /* detach notification
*/
    if (ret == -1) {
        perror ("ioctl");/* ioctl failed */
        exit (1);
    }
}

void
sig_handler ()
{
    printf ("got a signal\n");
}
```

Remote Message Queues and Remote Semaphores

A remote message queue or semaphore is one that is located on a remote SBC and is accessed using an RPC-like protocol. The full functionality of message queues and semaphores is available when accessed remotely.

Remote message queues and semaphores are named by pre-pending the host name to the name of the message queue or semaphore. The host name can be any SBC within the cluster.

Remote message queues and semaphores are implemented by having a unique connection with a file server process located on the SBC where the message queue is at. Requests are sent to the file server process which executes the operation and replies with the result.

More information on message queues can be found in the chapter “Real-Time Interprocess Communication” in the *PowerMAX OS Real-Time Guide*. Semaphores are described in the chapter “Interprocess Synchronization” in the *PowerMAX OS Real-Time Guide*.

The daemon **sbc_msgd(3)** is responsible for performing remote file server operations. While this daemon will automatically start on the file server SBC without any system configuration changes, this daemon must be configured to be automatically started on each client SBC within a closely-coupled cluster by enabling the CCS_IPC subsystem. Refer to the **sbc_msgd(3)** man page for more information on **sbc_msgd**, and see the **vmebootconfig(1M)** man page for more information on enabling the CCS_IPC subsystem.

Coupled Frequency-Based Schedulers

The Coupled Frequency Based Scheduler (FBS) support provides two types of timing devices: Closely Coupled and RCIM Coupled timing devices. Both of these timing devices may be used to provide cluster-wide synchronization for all FBS schedulers that are attached to the same Coupled FBS timing device. In addition, RCIM Coupled timing devices may be used to coupled together FBS schedulers on SBCS that may reside both within and outside a given closely-coupled cluster.

A Coupled FBS timing device may be attached to a scheduler by making use of the same library function calls or **rtcp(1)** commands that are used to attach other types of timing devices. However, a Coupled FBS timing device must first be "registered" as a Coupled FBS timing device on the host/SBC where the device interrupt originates, before it may be attached to FBS schedulers on the local and/or remote hosts/SBCs. Note that only one FBS scheduler on each host/SBC may be attached to the same Coupled FBS timing device.

More information about the Coupled FBS support can be found in the *PowerMAX OS Guide to Real-Time Services* manual.

In order to make configuration of client SBCs easier, the CCS_FBS subsystem support may be used to properly configure SBC clients so that they may make use of Closely Coupled timing devices. Additionally, the RCFBS subsystem support may be used to properly configure SBC clients with support for RCIM Coupled timing devices.

For more information on this topic, see the "Subsystem Support" section in the "VME Boot System Administration" chapter of the *Power Hawk Series 700 Diskless Systems Administrator's Guide.*

Closely Coupled Timing Devices

A requirement and restriction for Closely Coupled timing devices is that all SBCs must be located within the same cluster of a closely-coupled system. This is due to the fact that SBC messaging is relied upon for the inter-host/SBC message passing mechanism.

For some Closely Coupled timing devices such as the integral real-time clocks, the SBC message mechanism is also used to propagate the timing device interrupts to all attached

schedulers on the various SBCs within the cluster. However, the Real-Time Clocks and Interrupts Module (RCIM) devices may also be used as Closely Coupled timing devices; and in this case, the device interrupts may be optionally distributed by hardware through the RCIM cable directly to each receiving SBC, for faster and more deterministic interrupt response times than the P0Bus SBC messaging mechanism can provide.

RCIM Coupled Timing Devices

When a RCIM Coupled timing device is used to coupled together FBS schedulers residing on different SBCs, then any set of standalone SBCs, SBCs within a closely-coupled cluster, and/or netbooted SBCs may be used, as long as the RCIM Coupled configuration requirements are met.

The requirements for making use of a RCIM Coupled timing device are:

- the device must be a real-time clock or edge-triggered RCIM device that is configured to distribute its interrupts through the RCIM cable,
- all hosts/SBCs making use of the RCIM Coupled timing device must be connected to the same RCIM cable,
- all remote hosts must be configured to receive this specific RCIM interrupt through the RCIM cable, and
- all hosts that make use of this RCIM Coupled timing device must be able to communicate between each other using TCP/IP sockets as the method of inter-host communication.

In all cases, the distributed device interrupt that is sent through the RCIM cable is used to directly interrupt each host/SBC that has a FBS scheduler attached to the RCIM Coupled timing device.

Note that due to the above networking requirement, embedded clients in a closely-coupled cluster may not make use of RCIM Coupled timing devices; however, embedded clients may make use of Closely Coupled timing devices.

Glossary

Abbreviations, Acronyms, and Terms to Know

10base-T

See twisted-pair Ethernet (10base-T).

100base-T

See twisted-pair Ethernet (100base-T).

ARP

Address Resolution Protocol as defined in RFC 826. ARP software maintains a table of translation between IP addresses and Ethernet addresses.

AUI

Attachment Unit Interface (available as special order only)

asynchronous

An event occurring in an unpredictable fashion. A signal is an example of an asynchronous event. A signal can occur when something in the system fails, but it is not known when the failure will occur.

asynchronous I/O operation

An I/O operation that does not of itself cause the caller to be blocked from further use of the CPU. This implies that the caller and the I/O operation may be running concurrently.

asynchronous I/O completion

An asynchronous read or write operation is completed when a corresponding synchronous read or write would have completed and any associated status fields have been updated.

Backplane P0 Bridge Board (BPP0)

A P0*PCI bridge board, which may be used to connect two P0Bus Overlay boards together in order to create a larger common P0Bus in a closely-coupled system configuration. See definitions for **P0Bus Overlay** and **P0*PCI (P0Bus)**.

block data transfer

The method of transferring data in units (blocks) between a block device such as a magnetic tape drive or disk drive and a user program.

block device

A device, such as a magnetic tape drive or disk drive, that conveys data in blocks through the buffer management code. Compare `character device`.

block driver

A device driver, such as for a magnetic tape device or disk drive, that conveys data in blocks through the buffer management code (for example, the `buf` structure). One driver is written for each major number employed by block devices.

block I/O

A data transfer method used by drivers for block access devices. Block I/O uses the system buffer cache as an intermediate data storage area between user memory and the device.

block

The basic unit of data for I/O access. A block is measured in bytes. The size of a block differs between computers, file system sizes, or devices.

boot

The process of starting the operating system. The boot process consists of self-configuration and system initialization.

boot device

The device that stores the self-configuration and system initialization code and necessary file systems to start the operating system.

boot image file

A file that can be downloaded to and executed on a client SBC. Usually contains an operating system and root filesystem contents, plus all bootstrap code necessary to start it.

bootstrap

The process of bringing up the operating system by its own action. The first few instructions load the rest of the operating system into the computer.

buffer

A staging area for input-output (I/O) processes where arbitrary-length transactions are collected into convenient units for system operations. A buffer consists of two parts: a memory array that contains data from the disk and a buffer header that identifies the buffer.

cache

A section of computer memory where the most recently used buffers, i-nodes, pages, and so on are stored for quick access.

character device

A device, such as a terminal or printer, that conveys data character by character.

character driver

The driver that conveys data character by character between the device and the user program. Character drivers are usually written for use with terminals, printers, and network devices, although block devices, such as tapes and disks, also support character access.

character I/O

The process of reading and writing to/from a terminal.

client

A SBC board, usually without a disk, running a stripped down version of PowerMAX OS and dedicated to running a single set of applications. Called a client since if the client maintains a P0Bus or Ethernet connection to its File Server, it may use that File Server as a kind of remote disk device, utilizing it to fetch applications, data, and to swap unused pages to.

controller

The circuit board that connects a device, such as a terminal or disk drive, to a computer. A controller converts software commands from a driver into hardware commands that the device understands. For example, on a disk drive, the controller accepts a request to read a file and converts the request into hardware commands to have the reading apparatus move to the precise location and send the information until a delimiter is reached.

cyclic redundandancy check (CRC)

A way to check the transfer of information over a channel. When the message is received, the computer calculates the remainder and checks it against the transmitted remainder.

datagram

Transmission unit at the IP level.

data structure

The memory storage area that holds data types, such as integers and strings, or an array of integers. The data structures associated with drivers are used as buffers for holding data being moved between user data space and the device, as flags for indicating error device status, as pointers to link buffers together, and so on.

data terminal ready (DTR)

The signal that a terminal device sends to a host computer to indicate that a terminal is ready to receive data.

data transfer

The phase in connection and connection-less modes that supports the transfer of data between two DLS users.

device number

The value used by the operating system to name a device. The device number contains the major number and the minor number.

diagnostic

A software routine for testing, identifying, and isolating a hardware error. A message is generated to notify the tester of the results.

DLM

Dynamically Loadable Modules.

DRAM

Dynamic Random Access Memory.

driver entry points

Driver routines that provide an interface between the kernel and the device driver.

driver

The set of routines and data structures installed in the kernel that provide an interface between the kernel and a device.

embedded

The host system provides a boot image for the client system. The boot image contains a UNIX kernel and a file system image which is configured with one or more embedded applications. The embedded applications execute at the end of the boot sequence.

error correction code (ECC)

A generic term applied to coding schemes that allow for the correction of errors in one or more bits of a word of data.

FDDI

Fiber Distributed Data Interface.

flash autobooting

The process of booting a target from an image in its Flash memory rather than from an image downloaded from a host. Flash booting makes it possible to design targets that can be separated from their hosts when moved from a development to a production environment.

flash booting

See definition for **flash autobooting**.

flash burning

The process of writing a boot or other image into a Flash memory device. On SBC boards, this is usually accomplished with **SMon fp uf** command.

flash memory

A memory device capable of being occasionally rewritten in its entirety, usually by a special programming sequence. Like ROM, Flash memories do not lose their contents upon power down.

FTP (ftp)

The File Transfer Protocol is used for interactive file transfer.

File Server

The File Server has special significance in that it is the only system with a physically attached disk(s) that contain file systems and directories essential to running the PowerMAX OS. The File Server boots from a locally attached SCSI disk and provides disk storage space for configuration and system files for all clients. All clients depend on the File Server since all the boot images and the system files are stored on the File Server's disk.

function

A kernel utility used in a driver. The term function is used interchangeably with the term kernel function. The use of functions in a driver is analogous to the use of system calls and library routines in a user-level program.

host

A SBC running a full fledged PowerMAX OS system containing disks, networking, and the netboot development environment. Called a File Server since it serves clients with boot images, filesystems, or whatever else they need when they are running.

host board

The single board computer of the File Server.

host name

A name that is assigned to any device that has an IP address.

host system

A term used for the File Server. It refers to the prerequisite Power Hawk system.

interprocess communication (IPC)

A set of software-supported facilities that enable independent processes, running at the same time, to share information through messages, semaphores, or shared memory.

interrupt level

Driver interrupt routines that are started when an interrupt is received from a hardware device. The system accesses the interrupt vector table, determines the major number of the device, and passes control to the appropriate interrupt routine.

interrupt vector

Interrupts from a device are sent to the device's interrupt vector, activating the interrupt entry point for the device.

ICMP

Internet Control Message Protocol, an integral part of IP as defined in RFC 792. This protocol is part of the Internet Layer and uses the IP datagram delivery facility to send its messages.

IP

The Internet Protocol, RFC 791, is the heart of the TCP/IP. IP provides the basic packet delivery service on which TCP/IP networks are built.

ISO

International Organization for Standardization

kernel buffer cache

A set of buffers used to minimize the number of times a block-type device must be accessed.

kdb

Kernel debugger.

loadable module

A kernel module (such as a device driver) that can be added to a running system without rebooting the system or rebuilding the kernel.

MTU

Maximum Transmission Units - the largest packet that a network can transfer.

memory file system image

A cpio archive containing the files which will exist in the root file system of a client system. This file system is memory resident. It is implemented via the existing *memfs* file system kernel module. The kernel unpacks the cpio archive at boot time and populates the root memory file system with the files supplied in the archive.

memory management

The memory management scheme of the UNIX operating system imposes certain restrictions on drivers that transfer data between devices.

modem

A contraction of modulator-demodulator. A modulator converts digital signals from the computer into tones that can be transmitted across phone lines. A demodulator converts the tones received from the phone lines into digital signals so that the computer can process the data.

netboot

The process of a client SBC downloading into its own memory and then executing a boot image file that is retrieved from a File Server SBC by using the TFTP network protocol. On client SBC boards, networking is configured with the **`SMon smonconfig`** command, and a **`SMon`** startup script may be created and configured to automatically execute after a reset, in order to download and execute a boot image via TFTP with the **`SMon tftpboot`** command.

netload

The process of a target loading a boot image as discussed under netboot, but without subsequently executing it. On SBC boards, netloading is invoked with the **`Smon load`** command.

network boot

See definition for **netboot**.

network load

See definition for **netload**.

netstat

The **`netstat`** command displays the contents of various network-related data structures in various formats, depending on the options selected.

NFS

Network File System. This protocol allows files to be shared by various hosts on the network.

NFS client

In a NFS client configuration, the host system provides UNIX file systems for the client system. A client system operates as a diskless NFS client of a host system.

NIS

Network Information Service (formerly called yellow pages or yp). NIS is an administrative system. It provides central control and automatic dissemination of important administrative files.

NVRAM

Non-**V**olatile **R**andom **A**ccess **M**emory. This type of memory retains its state even after power is removed.

P0*PCI (P0Bus)

The PCI-to-PCI (P0) hardware bus interface that provides improved SBC board-to-board performance. The P0Bus is 64 bits wide and operates at 33 MHz, for a theoretical maximum of 264 MB/sec. This is more than three times the theoretical maximum of the standard VME64 bus of 80 MB/sec. The Power Hawk Series 700 P0Bus interface is based on the Intel 21554 64-bit PCI-to-PCI bridge chip.

The P0Bus hardware is required for Power Hawk Series 720/740 Closely Coupled configurations, where the P0Bus is used for Closely Coupled inter-SBC communications.

P0*PCI (P0Bus) Overlay

A P0*PCI connector board, which is used to connect multiple SBCs in the same cardcage (cluster) to a common P0Bus.

panic

The state where an unrecoverable error has occurred. Usually, when a panic occurs, a message is displayed on the console to indicate the cause of the problem.

PDU

Protocol Data Unit

PowerPC G4

The PowerPC G4 (7400) microprocessor. Part of the PowerPC family of microprocessors; an architecture based on Motorola/IBM's 32-bit RISC design CPU core.

PPP

Point-to-Point protocol is a method for transmitting datagrams over point-to-point serial links

prefix

A character name that uniquely identifies a driver's routines to the kernel. The prefix name starts each routine in a driver. For example, a RAM disk might be given the **`ramd`** prefix. If it is a block driver, the routines are **`ramdopen`**, **`ramdclose`**, **`ramdsize`**, **`ramdstrategy`**, and **`ramdprint.`**

protocol

Rules as they pertain to data communications.

RFS

Remote File Sharing.

random I/O

I/O operations to the same file that specify absolute file offsets.

raw I/O

Movement of data directly between user address spaces and the device. Raw I/O is used primarily for administrative functions where the speed of a specific operation is more important than overall system performance.

raw mode

The method of transmitting data from a terminal to a user without processing. This mode is defined in the line discipline modules.

rcp

Remote copy allows files to be copied from or to remote systems. rcp is often compared to ftp.

read queue

The half of a STREAMS module or driver that passes messages upstream.

rlogin

Remote login provides interactive access to remote hosts. Its function is similar to telnet.

routines

A set of instructions that perform a specific task for a program. Driver code consists of entry-point routines and subordinate routines. Subordinate routines are called by driver entry-point routines. The entry-point routines are accessed through system tables.

rsh

Remote shell passes a command to a remote host for execution.

SBC

Single Board Computer

SCSI driver interface (SDI)

A collection of machine-independent input/output controls, functions, and data structures, that provide a standard interface for writing Small Computer System Interface (SCSI) drivers.

sequential I/O

I/O operations to the same file descriptor that specify that the I/O should begin at the "current" file offset.

SLIP

Serial Line IP. The SLIP protocol defines a simple mechanism for "framing" datagrams for transmission across serial line.

server

See definition for **File Server** and **host**.

SMon

A board-resident ROM monitor utility that provides a basic I/O system (BIOS), a boot ROM, and system diagnostics for Power Hawk Series 700 single board computers (SBCs).

SMon startup script

As part of the boot process, **SMon** can automatically perform **SMon** commands and/or user defined functions written in a startup script that is stored in NVRAM (nonvolatile RAM). Special startup scripts are used for booting client SBCs in closely-coupled configurations, and also for netbooting client SBCs in loosely-coupled configurations.

SMTP

The Simple Mail Transfer Protocol, delivers electronic mail.

small computer system interface (SCSI)

The American National Standards Institute (ANSI) approved interface for supporting specific peripheral devices.

SNMP

Simple Network Management Protocol

Source Code Control System (SCCS)

A utility for tracking, maintaining, and controlling access to source code files.

special device file

The file that identifies the device's access type (block or character), the external major and minor numbers of the device, the device name used by user-level programs, and security control (owner, group, and access permissions) for the device.

SYM (sym)

Internal Symbios Logic SYM53C885 PCI-SCSI/Fast Ethernet Multifunction Controller.

synchronous data link interface (SDLI)

A UN-type circuit board that works subordinately to the input/output accelerator (IOA). The SDLI provides up to eight ports for full-duplex synchronous data communication.

system

A single board computer running its own copy of the operating system, including all resources directly controlled by the operating system (for example, I/O boards, SCSI devices).

system disk

The PowerMAX OS requires a number of "system" directories to be available in order for the operation system to function properly. In a closely-coupled cluster, these directories include: **`/etc`**, **`/sbin`**, **`/dev`**, **`/usr`** and **`/var`**.

system initialization

The routines from the driver code and the information from the configuration files that initialize the system (including device drivers).

System Run Level

A netboot system is not fully functional until the files residing on the File Server are accessible. **`init(1M)`** '**init state 3**' is the initdefault and the only run level supported for netboot systems. In **init state 3**, remote file sharing processes and daemons are started. Setting initdefault to any other state or changing the run level after the system is up and running, is not supported.

swap space

Swap reservation space, referred to as 'virtual swap' space, is made up of the number of real memory pages that may be used for user space translations, plus the amount of secondary storage (disk) swap space available.

target

See definition for **client**.

TELNET

The Network Terminal Protocol, provides remote login over the network.

TCP

Transmission Control Protocol, provides reliable data delivery service with end-to-end error detection and correction.

Trivial File Transfer Protocol(TFTP)

Internet standard protocol for file transfer with minimal capability and minimal overhead. TFTP depends on the connection-less datagram delivery service (UDP).

twisted-pair Ethernet (10base-T)

An Ethernet implementation in which the physical medium is an unshielded pair of entwined wires capable of carrying data at 10 Mbps for a maximum distance of 185 meters.

twisted-pair Ethernet (100base-T)

An Ethernet implementation in which the physical medium is an unshielded pair of entwined wires capable of carrying data at 100 Mbps for a maximum distance of 185 meters.

UDP

User Datagram Protocol, provides low-overhead, connection-less datagram delivery service.

unbuffered I/O

I/O that bypasses the file system cache for the purpose of increasing I/O performance for some applications.

upstream

The direction of STREAMS messages flowing through a read queue from the driver to the user process.

user space

The part of the operating system where programs that do not have direct access to the kernel structures and services execute. The UNIX operating system is divided into two major areas: the user programs and the kernel. Drivers execute in the kernel, and the user programs that interact with drivers execute in the user program area. This space is also referred to as user data area.

yellow pages

See definition for **NIS** (Network Information Services).

Index

V

W

Y

Spine for 1/2” 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