# VME Services for HP-UX 10 and 11

**HP Part No. A4412-90022**
**Special Online Edition E0998**
**Printed in U.S.A.**

© Hewlett-Packard Co. 1997

Printing History

First Printing: August, 1996
Latest Printing: September, 1998

UNIX is a registered trademark of the Open Group.

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information that is protected by copyright. All rights reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227.7013. Hewlett-Packard Co., 3000 Hanover St., Palo Alto, CA 94304.

**10 9 8 7 6 5 4 3 2 1**

# Contents


**Preface**

Audience xiv

Organization xv

- Device Driver Writers xv
- BPN Interprocess Communication Programmers xv
- System Administrators xvi

Support and Compatibility Disclaimers xvi

- Documenting Your Driver's Configuration Needs xvi

Related Documentation xvii

- Release Document(s) xvii
- Manual Pages xvii
- HP-UX Manuals xvii
- HP-RT Manuals xviii
- Obtaining Additional Material xviii

Revision History xix

Document Conventions xx

Problems and Questions xx

**1 Introducing VME Driver Concepts**

HP VME-Class Workstations 1-3

- HP VME-Class Buses 1-3
- Local Address Space Mapping 1-4
- I/O Space 1-5
- EEPROM and VME Configuration 1-5
- DMA and Cache Memory 1-6

The VME Bus 1-6

- Types of VME Address Space 1-7
- Allocation of VME Address Space 1-7

VME Interrupt Handling 1-8
VME Bus Arbitration 1-8
Releasing the VME Bus 1-8
Location Monitor and FIFO 1-9

VME Data Transfers 1-9
Local Master Transfers 1-9
Host DMA Transfers 1-10
Card Bus Master Transfers 1-10

Writing a Device Driver 1-11
User-Level VME Access 1-11
Block and Character Drivers 1-12
I/O System Calls 1-12
Functions that You Write 1-13
Example Device Driver Code 1-14
Integrating a Device and Its Driver 1-15
How HP-UX Finds Your Functions 1-16
Driver Development Environments 1-16

**2 Memory Management**

VME Memory Mapping for the VME-Class 2-2
Mapping A16 Memory 2-3
Master Mapping 2-3
Slave Mapping 2-4
The Safe Page 2-5
Direct Mapping 2-6
Anatomy of a Memory-Mapped Read Request 2-7

Memory Mapping Routines 2-9
The isc Structure 2-9
Setting the Address Modifier 2-9
Setting up the Master Mapper 2-10
Unmapping Master Mapping 2-11
Mapping Slave Memory 2-11

Obtaining Current System Parameters 2-12
The io_parms Structure 2-12
Making Your Memory Available to the Bus 2-13
Mapping Your Memory to a Specific VME Address 2-13
Mapping Pages of Memory to the VME Bus 2-13
Mapping a Variable Size Buffer to the VME Bus 2-14
Mapping Multiple Regions to the VME Bus 2-14
Unmapping Slave Memory 2-15

## 3 Interrupts

Series 700 I/O Interrupt Handling 3-2
Interrupt Service Routines 3-3
Recognizing which Device Interrupted 3-3

Your Device Driver's Interrupt Handler 3-4
Two Halves of a Driver 3-4
spl* Routines 3-5
A Skeletal spl* Routine 3-6
Using Interrupts 3-6
Finding Available Interrupts 3-6
Setting Up a VME Card Interrupt Vector 3-7
Writing the Interrupt Service Routine 3-9
An Example driver_isr Routine 3-9

VME Bus Errors and Panics 3-10
Setting an Error Handler 3-10
Bus Error Panic Hook 3-11
ISR Panic Hook 3-12

Software Triggers 3-13
Software Trigger Mechanism 3-13
A Skeletal Driver Fragment 3-15
Selecting the Software Trigger Level 3-16

Timeout Mechanisms 3-17

Location Monitor and FIFO 3-19
Location Monitor Functions 3-19
FIFO Functions 3-19

## 4 Direct Memory Access (DMA)

VME-Class DMA Hardware 4-3
Model 743/744 DMA Cycle and Transfer Types 4-3
Asynchronous DMA on Model 743/744 Systems 4-4
Synchronous Host DMA with Remote VME Cards 4-4

Synchronous DMA the Easy Way with vme_copy 4-5

Synchronous Local Host DMA 4-8
Setting up a Chain for DMA Transfers 4-8
Synchronous VME to Kernel RAM DMA with vme_dmacopy 4-9
Cleaning up after DMA 4-11

Asynchronous Local Host DMA 4-12
Queuing Transfer Buffers 4-12
Providing a Callback Function 4-13
Checking and Abnormally Terminating the Transfer 4-14
Sleeping until the Queue Empties 4-14
Checking DMA Status 4-15
Terminating DMA 4-15

Remote DMA Controllers 4-16
Remote DMA Setup Routine 4-18
Remote ISR Routine 4-18
Remote DMA without vme_dma_setup 4-19

Cache Coherency Issues 4-21
Flushing Cache 4-21
Purging Cache 4-22
Shared Cache Line Problems 4-22

## 5 User-Level VME Access

User-Level Access Advantages 5-2
User-Level Disadvantages 5-2

The ioctl Routine 5-2

Probe Commands 5-3

Register Access Commands 5-6

Map and Unmap Commands 5-8

The User Copy Command 5-10

The Enable IRQ Command 5-11

Location Monitor and FIFO Commands 5-11

Location Monitor Commands 5-12
FIFO Functions 5-12

The CPU Number Command 5-13

## 6 Writing Driver Entry Points

Attach Routines 6-2

wsio_drv_info Structure 6-4
drv_ops Struct 6-4
drv_info Struct 6-4
drv_data Struct 6-5

Open Routines 6-6

Opening Devices 6-7

Character Device Read and Write Routines 6-9

Performing Input from a Device 6-9
Performing Output to a Device 6-10
Using physio 6-11

Using uiomove 6-14
Data Transfer with vme_mod_copy 6-16

Character Device ioctl Routines 6-18
Using ioctl 6-18
Defining the Command Parameter 6-19
driver_ioctl 6-21

Character Device Select Routines 6-22

Block Device Strategy Routines 6-26
Overview of Block I/O 6-26
The driver_strategy Routine 6-27
driver_strategy for write() 6-28
driver_strategy for read() 6-29

Writing Synchronous Host DMA Strategy Routines 6-31

Strategy Routines for a Remote Master's DMA 6-33

Writing Asynchronous Host DMA Strategy Routines 6-35

Transfer Routines 6-37

Close Routines 6-38
driver_close 6-38

**7 Installing VME Devices**
Overview of Installing VME Devices 7-2

Installing the VME Product 7-2
Verifying the vme2 Driver 7-3
Choosing a Kernel Configuration File 7-4

Installing a Card in the VME Backplane 7-4

Making Device Files for Your Drivers 7-5

Editing System Files for the Driver 7-6
Creating a Master File for Your Driver 7-6
The Install Section 7-7
The Driver Dependency Section 7-7
The Driver Library Section 7-7
Editing the Build File 7-7

Compiling the Driver 7-8
Declaring Variables and Functions 7-8
Compiling the driver 7-9

Building the Driver into the Kernel 7-9

**8 Configuring VME Devices**

vme.CFG, the VME Configuration File 8-2

General Format for Configuration Records 8-3
Common Fields for Configuration Records 8-3

Card Records 8-4

Memory Records 8-5

Configuring Processor-Related Records 8-7
Processor Records 8-7
Interrupt Records 8-9
Slot 1 Function Records 8-9
Requesting the VME Bus 8-10
Releasing the VME Bus 8-10
Combining Levels and Modes 8-11

DMA Parameter Records 8-11

Updating the Series 700’s EEPROM 8-13

vme_config Options 8-15

## 9 Backplane Networking at HP-UX 10.20 and later releases

To Install BPN: 9-2
To Use TCP/IP: 9-2
To Use VME BPN Pipes: 9-2

Installing the HP-UX BPN Fileset 9-3

Configuring VME Memory for BPN Communication 9-3

Configuring the Shared Memory Area 9-4
Specifying Address Modifiers and Memory Locations 9-4
Configuring Shared Memory for TCP/IP Communications 9-4
Configuring Shared Memory for VME BPN Pipes 9-6
Tuning VME BPN Pipes 9-6

Establishing TCP/IP Communications 9-7

Specifying TCP/IP Addresses 9-8
Running the bp_config Configuration Utility 9-10
Bringing up the Backplane Network for TCP/IP 9-11

Using Berkeley Socket Communications 9-12

Client/Server Connection-Oriented Protocol 9-13
Socket-Related Structures 9-14
The socket() Call 9-15
The bind() Call 9-15
The listen() Call 9-16
The accept() Call 9-17
The connect() Call 9-17
The recv() and recvmsg() Calls 9-18
The send() and sendmsg() Calls 9-19
The shutdown() Call 9-19

## A Structures

bdevsw A-2

buf A-3

cdevsw A-6

dma_parms A-7

drv_info A-9

drv_ops A-9

iobuf A-10

io_parms A-12

iovec A-13

isc A-13

proc A-15

sw_intloc A-16

uio A-16

user A-17

vme_hardware_map_type A-18

vme_hardware_type A-19

vme_polybuf A-20

vme2_copy_addr A-21

vme2_int_control A-22

vme2_io_regx A-22

vme2_map_addr A-23

vme2_io_testx A-24

vme2_lm_fifo_setup A-25

wsio_drv_data A-25

wsio_drv_info A-26

**B Kernel and Driver Routine Summaries**

acquire_buf(), release_buf() B-2

bcopy() B-3

brelse() B-3

bzero() B-3

copyin() B-4

copyout() B-4

dma_sync() B-4

free_isc() B-5

geteblk() B-5

geterror() B-6

ioctl() B-6

iodone() B-6

io_malloc(), io_free() B-7

iowait() B-7

isc_claim() B-8

isrlink() B-8

issig() B-9

kvtophys() B-9

major(), minor() B-9

map_mem_to_bus() B-10

map_mem_to_host() B-10

minphys() B-11

msg_printf() B-11

panic() B-11

physio() B-12

printf() B-14

release_buf() B-14

selwakeup() B-14

sleep(), wakeup() B-15

snooze() B-16

spl*() B-16

suser() B-16

sw_trigger() B-17

timeout() B-17

uiomove() B-18

unmap_mem_from_bus() B-19

unmap_mem_from_host() B-19

untimeout() B-20

vme_change_adm() B-20

vme_clr() B-21

vme_copy() B-21

vme_create_isc() B-22

vme_dma_cleanup() B-22

vme_dmacopy() B-22

vme_dma_nevermind() B-23

vme_dma_queue(), vme_dma_queue_polybuf() B-23

vme_dma_setup() B-25

vme_dma_status() B-25

vme_dma_wait_done() B-26

vme_fifo_copy() B-26

vme_fifo_grab() B-27

vme_fifo_poll() B-28

vme_fifo_read() B-28

vme_fifo_release() B-29

vme_generate_interrupt() B-29

vme_get_address_space() B-30

vme_get_cpu_number() B-30

vme_get_dirwin_host_address() B-30

vme_get_iomap_host_address() B-31

vme_get_status_id_type() B-31

vme_hardware_info() B-31

vme_hardware_map_info() B-32

vme_isrlink() B-32

vme_isrunlink() B-33

vme_locmon_grab() B-33

vme_locmon_poll() B-34

vme_locmon_release() B-34

vme_map_largest_to_bus () B-34

vme_map_mem_to_bus2() B-35

vme_map_pages_to_bus() B-35

vme_map_polybuf_to_bus() B-36

vme_mod_copy() B-36

vme_reg_read() B-37

vme_reg_write() B-37

vme_remap_mem_to_host() B-38

vme_rmw() B-38

vme_set_address_space() B-39

vme_set_attach_function() B-39

vme_set_mem_error_handler() B-40

vme_set_status_id_type() B-40

vme_test_and_set() B-41

vme_testr() B-41

vme_testw() B-42

wakeup() B-42

**C Porting Device Drivers**

Porting HP-UX 9.x Drivers C-2

New Attach Function and Process C-2
Other New Functions C-3
New Access Functions C-3
New Create ISC Function C-3
New 10.0 vme_ Calls that Replace io_ Calls C-3
New vme_hardware_info() Function C-3
New 10.1 Functions C-3
New Include File Pathnames C-4
Building and Installing Your Driver C-4

Porting Third-Party Drivers C-4

Modifying Entry Point Routine Names C-4
Modifying Header File Inclusion C-5
Checking Driver Entry-Point Routines C-5
Checking Return Values C-5
Checking Entry Point Routine Parameters C-5
Checking Kernel Routine Parameters C-6
Checking the ioctl Command Format C-6
Checking Timeout Parameters C-6
Checking Interrupt Service Routine Parameters C-7

Miscellaneous Suggestions C-7
Kernel Mapping Routines C-7
Kernel Address Conversion and Buffer Alignment C-7
Unique Routines C-8

## D Skel Device Drivers

Non-DMA Skeleton Driver D-2
skel_isr() D-3
skel_open() and skel_close() D-3
skel_strategy() D-4
skel_read() D-4
skel_write D-5
skel_ioctl() D-5
Structs D-6
skel_attach() D-7
skel_install() D-8

VME DMA Driver Skeleton D-8
skel_isr() D-9
skel_open() and skel_close() D-10
skel_strategy() D-10
skel_read() and skel_write() D-11
skel_dma_setup() and skel_dma_start() D-12
skel_ioctl() D-13
Structures D-13
skel_attach() D-14
skel_install() D-15

# Contents

## Figures

VME-Class Workstation Board and Card Cage 1-2
VME-Class Buses: Cache, Memory, and System 1-3
Virtual-to-Physical Memory Translation 1-4
A Device Driver’s Appearance in Directory Listing 1-16
Development Target and Host System 1-17
Master Mapper Operation 2-4
VME Bus to HP Memory via Slave Mapper 2-5
Direct Mapping 2-6
Memory-Mapped Read Request 2-8
Example VME Backplane Network 9-8
Client/Server, Connection-Oriented Protocol 9-13

## Tables

Symbolic Names for Memory Address Modifiers 2-10
ioctl Commands with Structs 5-3
Names for Memory Address Modifier Declarations 8-6

# Preface

The VME Services product provides configuration tools, a kernel-level driver, and user- and kernel-level VME bus access functions to install, administer, and access VME bus functionality for application-level programs and application-level and kernel-level drivers.

Generally, the **vme2** kernel driver supports programming of the VME bus adapter to the VME bus on behalf of the caller, arranging for bus transfers; mapping memory between the HP-UX host and VME bus address space (and vice versa); interrupt handling; reading and writing of EEPROM VME data; and so on.

This manual describes how to:

- Write VME device drivers and access VME devices with user-level functions
- Configure VME devices and resources
- Use the Backplane Networking (BPN) services to communicate between processes running on different HP VME computers (with HP-UX 10.20 and later releases)
- Use BPN as the physical medium for TCP/IP networking
- Configure BPN

If you are configuring VME devices, we assume that you have received sufficient documentation from their manufacturer.

## Audience

This manual is primarily intended for C programmers who:

- Understand software development in an HP-UX environment
- Need to write kernel- and/or user-level VME I/O drivers
- Configure or provide configuration instructions for the installation of the driver(s) and associated devices
- Use Berkeley socket programming to provide client-server, interprocess communication

Parts of this manual are also intended to be used by HP-UX system administrators who are installing and configuring VME resources.

# Organization

This manual has been organized with the purposes of several types of readers in mind.

Every reader should read this Preface and Chapter 1, which provides a general overview of VME on the HP VME-Class systems. Likewise, there is a Glossary and Index that should prove useful for any reader.

## Device Driver Writers

If you are writing a VME device driver, follow this reading path:

- Chapters 2 through 4 provide a conceptual overview and introduction to programming with the functions that support memory allocation, interrupt processing, direct memory access, and the location monitor and FIFO.
- Chapter 5 is a brief introduction to user-level access to the VME functions; this type of access may be used to write a non-kernel-level driver, or just access the functions of a VME card without writing a driver.
- Chapter 6 describes how to write the entry points for a device driver.
- Chapter 7 provides help in compiling your driver and building it into the kernel.
- Appendices A and B provide quick-reference summaries of the major structures and routines that are pertinent to writing a device driver.
- Appendix C gives help on porting drivers, and Appendix D provides a skeletal driver that you can use as a start to writing your own device driver.

## BPN Interprocess Communication Programmers

If you have HP-UX 10.20 (or later) and are going to use the VME backplane as the physical medium for TCP/IP networking; or if you are using Berkeley socket programming to communicate between HP VME systems, follow these reading suggestions:

- Chapter 9 introduces the BPN configuration utility **bp_config** that works in conjunction with **vme_config** to allow you to define the configuration of the backplane network for TCP/IP.
- Chapter 9 also includes sections on BPN programming.

### System Administrators

If you are going to configure a VME system, follow these reading suggestions:

- Chapter 7 provides an overview of installing VME Services.
- Chapter 8 introduces the VME configuration file and **vme_config** utility that together allow you to define the configuration in an ASCII file, correct syntax errors in it, and compile the information into the VME-Class's EEPROM.
- Chapter 9 describes BPN configuration and the use of **bp_config** and **ifconfig**.

## Support and Compatibility Disclaimers

Since drivers function at the level of the kernel, Hewlett-Packard Company (HP) reminds you of the following:

- Adding your own driver to HP-UX requires re-linking of the driver into HP-UX. With each new release of HP-UX, you should plan on re-compiling your driver and re-installing it. Drivers typically use some header files that may change across releases (that is, you can have some system dependencies).
- To HP's knowledge, the information in this manual is correct when it is released, but some information such as kernel routines and header files may change.
- HP provides support services for HP products, including HP-UX. Products such as drivers from non-HP parties receive no support other than the support of those parts of a driver that rely on the documented behavior of supported HP products.
- Should difficulties arise during the development and test phases of writing a driver, you may receive HP's support in isolating problems provided:
  - HP determines that HP hardware is not at fault.
  - HP determines that HP software (and/or firmware) is not at fault by removing user-written kernel drivers.

### Documenting Your Driver's Configuration Needs

If you are providing VME cards and HP-UX kernel level drivers, you need to document the VME configuration steps required for a conflict-free configuration. The goal is to provide a system where memory can be shared with-

out collisions (for example, only one card responds to a given bus address and area modifier), and where interrupts and bus ownership are managed fairly and appropriately.

## Related Documentation

### Release Document(s)

Please refer to the *Release Document(s)* you received with your system or system software for additional information that we may not have been able to include in this guide at the time of its publication.

### Manual Pages

The **man**(1) pages for VME, BPN, and kernel-related functions should be installed and online in their appropriate sections (1, 1m, 2, and 4); but also see “Obtaining Additional Material” on page xviii.

When you see a utility or function name followed by a number in parentheses, that number indicates the section in the online man pages. In some cases, you need to specify this section number—generally if there are two or more man pages of the same name. To display a man page, for instance, when you are told to “see *function*(2)” or “see *utility*(1M),” use one of the following commands:

**man** *name*
**man** *section_number name*

### HP-UX Manuals

The following manuals may be useful in installing and configuring your HP-UX systems:

- *Installing and Updating HP-UX* (B2355-90039)
- *HP-UX System Administration Tasks* (B2355-90079)
- *Configuring HP-UX for Peripherals* (B2355-90053)

## HP-RT Manuals

For information on configuring VME Services on HP-RT systems, refer to:

- *HP-RT System Administration Tasks*
- *VME Backplane Networking Guide*

To order manuals, please contact your local sales office.

## Obtaining Additional Material

The 10.20 and later releases of VME and BPN Services on HP-UX includes only the following man pages in the distribution:

- **vme_config**(1M)
- **bp_config**(1M)
- **vme.CFG**(4)

To obtain the many additional man pages for VME and BPN, as well as the kernel reference man pages in the PostScript document *HP-UX Driver Development Reference* (98577-90602), visit our World Wide Web homepage at:

http://www.hp.com/es

(You can also obtain the patches to VME Services via the WWW, along with installation instructions.)

# Revision History

The following table displays the part numbers for manuals that support VME on different releases of HP-UX.

**Ordering Information for Manuals for VME Services**

| Title | HP Part No. | Edition | Edition Notes |
|---|---|---|---|
| *HP-UX 9.05 VME Device Driver Guide* | A2636-90020 | E0594 | Incorporates the *VME Configuration Guide* |
| *HP-UX 10.0 VME Device Driver Guide* | A4412-90020 | E0296 | For HP-UX 10.01 and 10.1 |
| *VME Services for HP-UX 10* | A4412-90021 | E0896 | HP-UX 10.1, 10.2 |
| *VME Services for HP-UX 10* | A4412-90022 | E0797 | HP-UX 10.1, 10.2, and fixes |
| *VME Services for HP-UX 10 and 11* | A4412-90022 | Special Online Edition E0998 | HP-UX 10.1, 10.20, 11.x, and fixes |

This manual contains information to support HP-UX VME Services as follows:

- BPN in HP-UX 10.20 and later environments
- VME devices and drivers in HP-UX 10.01, 10.10, 10.20, and later environments

For 10.1 and later,VME Services has been updated to provide additional function calls for asynchronous DMA purposes relating to queues, status, and interrupts; for mapping pages and buffers to the VME bus, and remapping memory; and for protecting access to the data and functions involved. (To receive this updated functionality for 10.1, obtain the appropriate patch from the WWW source, as explained on the previous page.)

## Document Conventions

The following table displays the typographic conventions this manual uses to represent various entities.

**Typographic Conventions for this Manual**

| Convention | Explanation |
|---|---|
| `output` | Mono-spaced type indicates program code that you enter using a text editor, or displayed results from the HP-UX system. |
| **literal values** | Bold font indicates a function name, a word that is being defined, or, more often, information that you supply literally for pathnames, commands, keywords, field names in structs, etc. |
| *example values* | Italics are used for values that you need to supply for pathnames, command parameters, arguments to functions, variables, and struct names. |
| SMALL CAPS | Small caps indicate flags, macros, and constants that have been defined in one of the VME or kernel include files. |

## Problems and Questions

If you have any questions or problems with our hardware, software, or documentation, please contact either your HP Response Center or your local HP representative.

# 1

# Introducing VME Driver Concepts

Since VME's introduction in 1981, thousands of **Versa Module Eurocard** products have been introduced.

The VME bus (backplane) is microprocessor independent, implements a reliable mechanical standard, and is compatible with products from many independent vendors, including HP.

The HP VME computer systems use the VME backplane (or bus) as well, thus helping to support the wide variety of existing hardware and software that is an attractive feature of VME.

This versatility comes at the price of demanding care and skill in writing the device drivers that let applications talk to the VME devices. The driver itself must be installed in the kernel of the HP-UX operating system. The hardware must be installed in a **card cage** or an HP 748 workstation. And the way in which the new hardware provides services such as memory or processing needs to be orchestrated with all other boards in the cage so that one board doesn't step on another, and they can all take their turns in an orderly fashion.

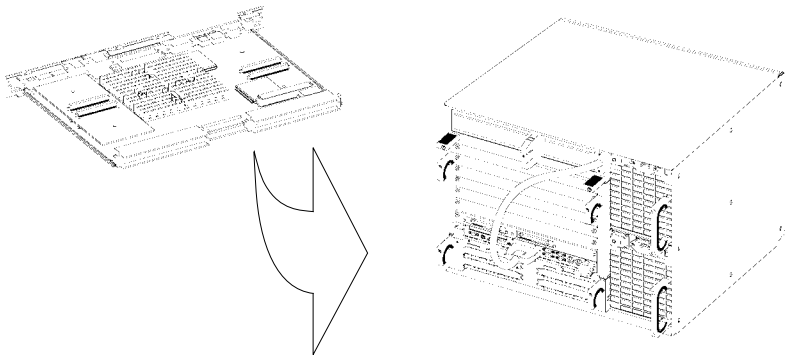


**Figure 1-1** **VME-Class Workstation Board and Card Cage**

This chapter introduces the major concepts that you need to know before you can effectively do even a simple board installation and configuration. It continues by explaining some of the basics of using user-level **ioctl** (I/O Control) commands in an application program to talk to a device, and it concludes by introducing some of the basic concepts involved in writing device drivers.

# HP VME-Class Workstations

This section describes the basic architecture of the HP VME-Class workstations and how their processor (the HP PA-RISC CPU) communicates with its **local bus** memory, I/O space, and the ASIC (Application-Specific Integrated Circuit) VME bus adapter that interfaces between the VME bus and the VME-Class's memory I/O controller.

## HP VME-Class Buses

The HP VME-Class has three local buses, as shown in Figure 1-2:

- The cache bus, connecting the CPU to the instruction/data cache
- The memory bus, connecting the memory I/O controller to external DRAM
- The system bus, which connects the memory I/O controller to the interface controllers

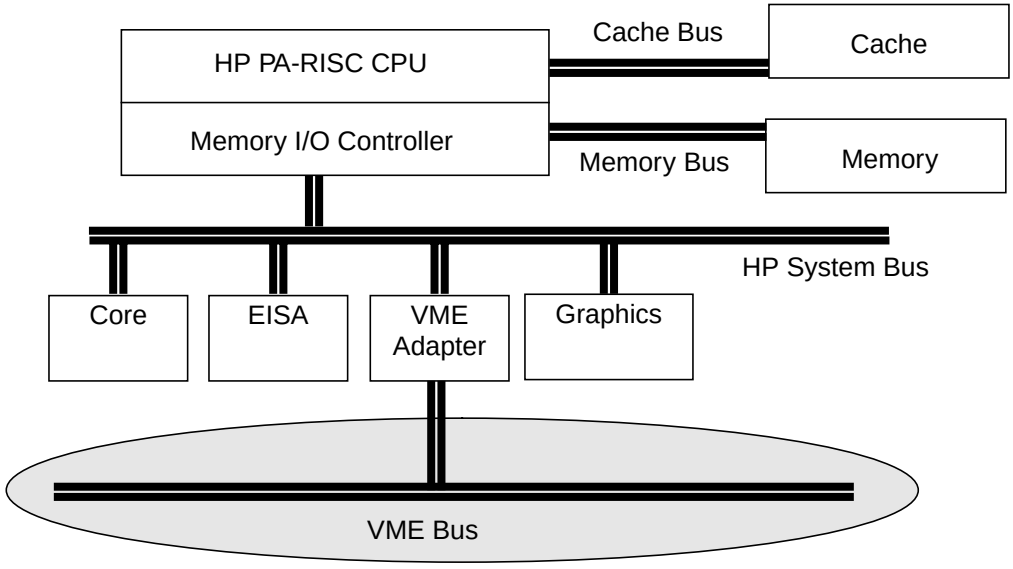



**Figure 1-2** **VME-Class Buses: Cache, Memory, and System**

## Local Address Space Mapping

The HP-UX operating system uses 32-bit virtual addresses to access memory, thus providing 4 GB of address space per process.

When the CPU addresses its local memory, built-in virtual paging hardware translates the virtual page addresses into physical page addresses. The physical page addresses need not be sequential. When the operating system allocates a shared memory buffer, for instance, the physical pages may be scattered throughout the system's main memory, but the virtual translations allow the CPU to access the buffer as contiguous virtual pages.

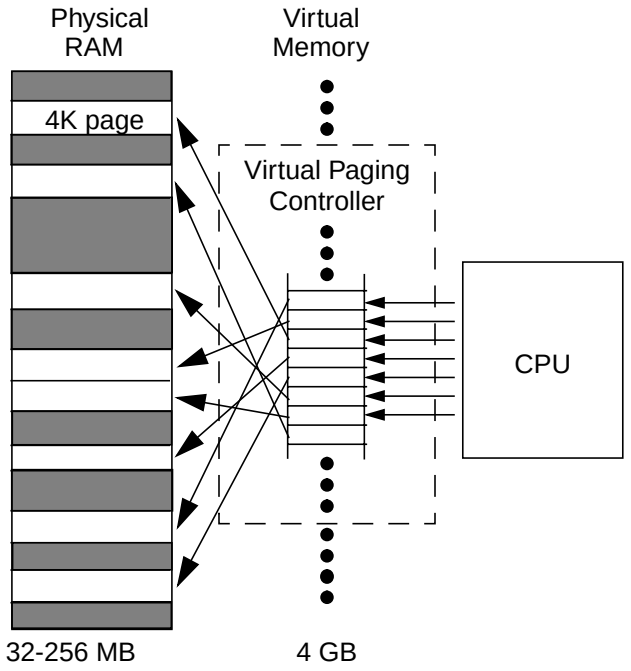



**Figure 1-3** **Virtual-to-Physical Memory Translation**

## I/O Space

The VME-Class board can access memory on other VME boards on the VME bus, because the VME bus adapter provides for **mapping** the other VME devices' address space to the memory on the local bus seen by the PA-RISC CPU.

**Master mapping** lets your CPU talk to the other boards by putting your address request onto the VME bus; **slave mapping** lets your board respond to requests for memory from other boards. In technical terms, masters are the ones that initiate bus read or write cycles to which the slaves respond.

The PA-RISC virtual memory for all processes contains a special 256 MB dedicated to various I/O types of functions. This is **I/O space**, and it's special because it will not be paged out.

The kernel allocates some of the I/O space for VME at power-up time, based on the needs of graphics boards and EISA bus adapters, each of which takes approximately 64 Mbytes of the total 256 Mbytes of the I/O space. This allocation directly determines how much this VME board can see of the "outside world," that is, memory provided by the other boards on the VME bus.

## EEPROM and VME Configuration

VME devices require system administration attention to configure properly so that a particular HP-UX system does not try to configure its slave-mapped VME memory on top of that provided by another card. The utility **vme_config**(1M) needs to be run against an ASCII configuration file that contains information about all of the VME boards in the cage. When run successfully, **vme_config** parses that information and determines what memory the HP-UX system should provide to the VME bus and where to place it to avoid memory conflicts with other VME boards. This information is then stored in a nonvolatile EEPROM.

When the HP-UX system boots, it reads the EEPROM and uses that information to program the VME bus adapter's access to the VME bus.

### DMA and Cache Memory

HP makes several VME board computers, as well as a Model 748 workstation that serves as a card cage for them. They can run either HP-UX or HP-RT (a real-time operating system available as a separate product) and are distinguished by an *i* for those running HP-UX, and *rt* for those running HP-RT (unless the particular operating system is not relevant).

The 743 and 744 processors have a VME DMA engine capability in hardware. (This capability is not present in the 742 or 747.) The DMA controller runs independently of the VME-Class processor to transfer data between local memory and other VME boards, and independently of other VME boards transferring their local memory to and from the VME-Class's RAM.

## The VME Bus

The VME bus originally was the single local bus in a system that used a Motorola processor motherboard connected through the local bus backplane to other boards such as memory and I/O. Currently, the VME boards that connect to it most often have their own local buses (as we saw above with the PA-RISC architecture).

The VME bus provides a form of virtual memory that is similar to the virtual memory provided on the VME-Class systems. VME address space is actually composed of physical memory on the VME boards. Each board has the facilities to put the contents of a requested memory location onto the VME bus, according to whether the board recognizes the combination of the VME address and address modifiers. Note that the VME boards have the ability to see VME physical memory, not virtual memory or the processor's cache memory.

VME also has its peculiarities in the types of interrupt handling that it recognizes and expects, as briefly described on page 1-8.

## Types of VME Address Space

The VME backplane has 196 lines that communicate with each VME card through two connections known as **pins** or **slots** on the card. Some of these lines carry data; others specify VME address space.

The VME backplane can read or write data at an address by using 8, 16, or 32 lines to carry the data, depending on the width of the data. It uses 16, 24, or 32 additional lines to specify the address, and six more lines to specify the type of address space.

Boards occupy different portions and amounts of VME address space known as **A16 space**, **A24 space**, and **A32 space**. Different types of address space exist because of historical limitations or different functionality as the VME bus specification and associated hardware have developed. Because some early VME boards used Motorola 68000 microprocessors that could present 24 address bits through their pins, memory boards (for example) became available that had address decoding for only 24 bits. These are known as A24 boards. Early peripheral I/O boards, on the other hand, used only 16-bit architectures because they had 16-bit CPUs, so when they were incorporated into VME architectures, they used A16 space.

The VME bus adapter on the 742/747 Series machines can distinguish only A16, A24, and A32 space, while the VME bus adapter on 743/744 Series machines can distinguish 64 distinct spaces for each of these three types of space, as indicated by an additional group of six VME signal lines known as **address modifiers**. (Most possible modifiers are actually reserved.) Many cards accept several different modifiers for the address range they support.

The 742/747 systems can generate any address modifier. Only when they slave memory are they restricted to A24 or A32. (No 742/747 or 743/744 architecture slaves A16 space.)

## Allocation of VME Address Space

You need to specify how VME address space is proportioned to the various VME boards, by editing a configuration file and running the **vme_config** program.

There may be more than one card with the same address on the VME bus, provided each card responds to different address modifiers. Some cards are A24 cards, others are A32 cards, and others respond to both A24 and A32 requests.

VME address space therefore is not a single continuum of addresses that are apportioned to the various boards, but rather a “family” of continua of memory provided by the boards as they answer to bus requests that specify some range of addresses and some set of address modifiers.

## VME Interrupt Handling

VME boards can generate interrupts on seven hardware lines, which have increasing priority from 1 to 7. When a device interrupts, it places a hardware status ID on the VME bus to be read by the device that is handling the interrupt. The operating system that handles the interrupt uses the status ID to determine which interrupt service routine to call.

### VME Bus Arbitration

The VME bus requires an arbiter to mediate bus requests. Even if each card has an arbiter, only the arbiter in Slot 1 of the VME card cage is used. For the VME-Class systems, you define the arbiter in the VME configuration file.

A **request level** can also be specified in the configuration file to define the level at which a VME board or an interrupt handler requests the bus. An **arbitration mode** (priority or round-robin) likewise can be specified in the configuration file to define how the system determines which request level to handle first.

### Releasing the VME Bus

You can also specify in the VME configuration file a request mode to indicate when the VME bus is to be released after an interrupt has been placed on it. You can specify, for example, that the interrupt handler should finish its task before releasing the bus; or that it should wait until another VME board requests it; or that it will handle only a small part of the interrupt, allow other VME boards to request the bus, and wait for them to finish before re-requesting it.

### Location Monitor and FIFO

The Location Monitor and FIFO mechanisms are implemented in the VME bus adapter hardware. They report when a specified area of memory has been accessed.

The Location Monitor provides an interrupt mechanism that responds when memory in a particular range of addresses has been read from or written into by a VME bus master.

The VME FIFO provides a method of ordering asynchronous communications received by a VME processor. Whenever its FIFO is written into, an interrupt occurs, and a user-specified function can be executed to read and react to the FIFO's contents.

## VME Data Transfers

All VME transfers involve a master and a slave. The VME master controls the transfer by establishing properties such as address, direction, and type of transfer. Your driver, running in the context of the HP's processor and RAM, needs to direct a VME master to accomplish transfers in the VME context.

The types of available transfers are introduced in the following subsections:

- Local Master Transfers (742/747 and 743/744)
- Host DMA Transfers (743/744)
- Card Bus Master Transfers (742/747 and 743/744)

### Local Master Transfers

You can use the built-in VME master under direct control of the HP PA-RISC CPU in a fashion that is generally transparent to you as a driver writer. The CPU's transactions involving I/O space controlled by the VME bus adapter are converted in the adapter from local bus to VME bus addresses.

Essentially, you map the VME space to local bus memory with a call to **map_mem_to_host**(), and choose one of the following approaches to transfer the data:

- Use C constructs such as:

```
*dest = *source;
```

- Call **vme_copy**(), **vme_mod_copy**(), or **vme_fifo_copy**()

You can find an example of code that calls **vme_copy** in “Synchronous DMA the Easy Way with vme_copy” on page 4-5. For **vme_mod_copy**, see “Data Transfer with vme_mod_copy” on page 6-16.

## Host DMA Transfers

The VME bus adapter used on the 743/744 systems contains two VME master finite state machines, one for DMA with another VME card that masters the bus (see “Card Bus Master Transfers” below), and one for DMA with VME space in which the 743/744 system masters the bus, i.e., **host DMA transfers**.

One can run DMA cycles between VME space and the PA-RISC CPU’s local RAM without that CPU’s involvement. The PA architecture does not interleave DMA in the classic sense (i.e., during wait states), because only one PA master can access RAM at a time, whether through the PA processor or the VME bus adapter.

The adapter’s DMA engine runs transactions between VME space and its on-board FIFO, and then obtains the internal, local system bus to transfer the entire FIFO at one time. The relative speed of the two buses allows this FIFO-size interleaving to occur without noticeable slow-down on either side. Access to the VME bus adapter’s DMA master is provided by **vme_dmacopy** and, with certain options, **vme_copy**. Direct driver access to the VME bus adapter’s DMA master is not allowed.

## Card Bus Master Transfers

If your VME card has VME bus master capability, you can program it to run the transfer. You need to get the physical address of the VME-Class’s RAM buffer, handle any cache issues, program your card to run the transfer (interrupting upon completion), and then handle any cache issues yourself. Since

the physical addresses of the PA buffer are generally not contiguous, you may need to interrupt for every page transferred (i.e., every 4 KB) unless your card has “scatter-gather” capability.

## Writing a Device Driver

A device driver intermediates between an operating system and its I/O devices in the following sequence:

1. User-level application programs make system calls to the operating system to open and read devices (as well as files).
2. The operating system then issues requests to the appropriate device driver.
3. The device driver sends commands to the hardware interface to perform the operations needed to service the request.

### User-Level VME Access

HP-UX provides the capability to access VME address space without having to write a driver. This is very convenient for one-time-only access to VME, or to a device that requires only an occasional read or write.

This **ioctl**() access has the effect of allowing an application to call routines that a device driver such as **vme2** has defined and implemented to open, read, write, and otherwise access a VME device. This type of access may be called **user-level VME access**.

This access uses **ioctl commands** that have been pre-defined to call the device driver entry points provided by the **vme2** device driver. These commands include VME2_REG_READ, VME2_REG_WRITE, VME2_MAP_ADDR, VME2_UNMAP_ADDR, VME2_USER_COPY, and so on.

If you write your own device driver, you can specify **ioctl** commands that can be used by user-level applications to access similar functions.

## Block and Character Drivers

**Character drivers** are used for many kinds of devices, usually such as line printers that can read or write in byte-sized pieces, but they can be any device whose chunks of data are smaller or larger than the standard, fixed-sized buffers used by block drivers.

**Block drivers** communicate with devices such as disk drives. For these drivers, HP-UX uses a **cache** of fixed-sized buffers. If the user application asks for data already in the cache, HP-UX doesn't need to access a device driver's **read** or **write** functions to supply the data, but when the data isn't already in the cache, or the cache has changed and must be written out, HP-UX accesses the driver's **strategy** function.

A block driver differs from a character driver in the structure of the requests sent to it, and in their origination. Character drivers receive requests directly from user processes making system calls, e.g., **read**(), **write**(), etc. Block drivers receive requests from the "buffer cache."

For example, consider a request such as the following to open an ordinary file:

```
fd = open ("/tmp/ordinary", O_RDWR | O_CREAT)
```

This request becomes a request to the file system. The file system then makes a request to the buffer cache, and the buffer cache *may* call a device driver if the request cannot be satisfied from the cache.

Another major difference between block and character drivers is that user processes interact directly with character drivers. The I/O request is passed essentially unchanged to the driver to process, and the character driver is responsible for transferring the data directly to and from the user process. The block driver, on the other hand, exchanges data with a kernel buffer that fills and empties essentially independently from the user process.

## I/O System Calls

The routines, functions, or **entry points** of a device driver, such as the **mydevice_open**() and **mydevice_read**() functions shown in "Example Device Driver Code" on page 1-14, are called by HP-UX using the standard C function-calling mechanism. You compile these routines for the device

driver and link them with the object code for the operating system itself, resulting in a new file that contains a bootable operating system with all the device drivers.

The system calls that let an application perform I/O are called **I/O system calls**. They include **open**(2), **close**(2), **read**(2), **write**(2), **select**(2), and **ioctl**(2). In using these calls, I/O goes through the system-call interface, the kernel, and a device driver. Since devices are accessed (represented) as files, a user process performs I/O by making system calls on device files.

When a user issues an I/O system call on a device file, the kernel takes actions that include, for example:

- Checking that the user has access permissions to the device file
- Obtaining any required system buffers
- Using the major number to index into a device switch table
- Calling the driver associated with the device file with the appropriate parameters

Then the kernel calls one of your device driver routines corresponding to the I/O system call made by the application.

## Functions that You Write

The driver functions that you write perform the functions expected by the user application's I/O system calls. They include the following (where *driver* substitutes for your driver's name).

*driver*_**install():** During system initialization, your *driver*_**install()** function is called. It is responsible for registering the driver's *driver*_**attach()** function and kernel structures that specify the functions and configuration data associated with the driver. This is the single routine in your driver's repertoire that you can rely on to be called.

*driver*_**attach():** When your system boots and your *driver*_**attach()** function is called, you can perform any initialization that is appropriate for your card.

*driver*_**open():** Used to enable a device for subsequent file system operations. If the device is off-line, does not exist, or cannot be accessed, an error should be returned. When appropriate, your open routine can do nothing. The corresponding application-level system call is **open**().

*driver*_**read():** An entry point for character devices only. It should return the requested data read from the device. The corresponding application-level system call is **read**().

*driver*_**write():** An entry point for character devices only. It should write the requested data to the device. The corresponding application-level system call is **write**().

*driver*_**close():** Used to close a device. Your **close** routine can disable interrupts, reset a device, free resources, or perform other tasks required when the device is closed. The corresponding application-level system call is **close**().

*driver*_**ioctl():** An entry point for character devices only. Used to read or write information from or to a device. Also, it can be used to provide a driver-dependent function not implemented by any other routine.The corresponding application-level system call is **ioctl**().

*driver*_**select():** An entry point for character devices only. Used to test whether I/O has completed or if some driver-dependent exception condition has occurred. If your device is always ready for reading or writing, you can set this to always return true. The corresponding application-level system call is **select**().

*driver*_**size():** An entry point for block devices only. It should return the size of the swap partition for swap devices. You should consider writing a *driver*_**size** routine only if your device is used for swapping.

*driver*_**strategy():** An entry point for block devices only. Used to queue the I/O request for either reading or writing. Character device drivers often have a strategy routine that is called by their read and write routines, but it is not an entry point into the driver.

## Example Device Driver Code

For example, the following code might represent the user-level application program's calls to open and read from a device:

```
fd = open ("/dev/mydevice", O_RDWR);
read (fd, buf, 13);
printf ("%s", buf);
```

The operating system code executed by HP-UX is known as **kernel code**. In the case of a device driver, the device driver code is compiled and linked with the operating system code, so even though you as a device driver writer have written it, it is not user-level code. The device-specific code that you might write to provide the **open** and **read** functions for your device could be:

```
mydevice_open (dev, flag)
   dev_t dev;
   int flag;
{
   return(0);
}

mydevice_read (dev, uio)
   dev_t dev;
   struct uio *uio;
{
   uiomove (“hello world\n”, 13, UIO_READ, uio);
   return(0);
}
```

## Integrating a Device and Its Driver

HP-UX can find your driver’s functions and communicate with the VME devices because of what you tell it when configuring the device and driver and building the driver’s functions into the kernel. Very briefly, you:

1. Edit a text file to specify the VME configuration (placement of memory, interrupt handling, etc.).
2. Run **vme_config** to store the configuration file’s data into EEPROM that can be read when the system boots.
3. Issue a **mknod** command to create a special device file.
4. Compile your driver into an object file.
5. Update various files to include information about your driver (its name, dependencies, etc.).
6. Generate a new kernel that contains your device driver.
7. Reboot with the new kernel.

## How HP-UX Finds Your Functions

When the new kernel is built, it contains two **switch tables** with entries for all of the entry points of all the device drivers (character or block) that it represents, including those that you define and build into the kernel.

For example, the following shell command might cause the file named *filename* to be printed on the line printer **lp**:

```
cat filename > /dev/lp
```

This asks HP-UX to open the file **/dev/lp** to receive the contents of *filename*. You can see what is reported about the file **/dev/lp** if you use the command:

```
ll -g /dev/lp
```

Look for the driver type and major device number, as shown in Figure 1-4.

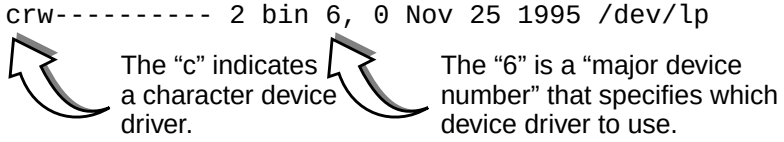



**Figure 1-4** **A Device Driver’s Appearance in Directory Listing**

When HP-UX sees that the file to be opened is a character special file, it uses the major device number to index into the switch table of all of the character drivers on the system. For example, here is the C struct definition of the character driver event switch table (**cdevsw**) for character drivers:

```
struct cdevsw cdevsw[] = {
   . . .
/*6*/ {lp_open, lp_close, . . . }
   . . .
/*59*/ {my_open, my_close, my_read, . . .}
}
```

## Driver Development Environments

HP-UX device drivers can be developed and run on a single system, or downloaded to a target system as shown in Figure 1-5.

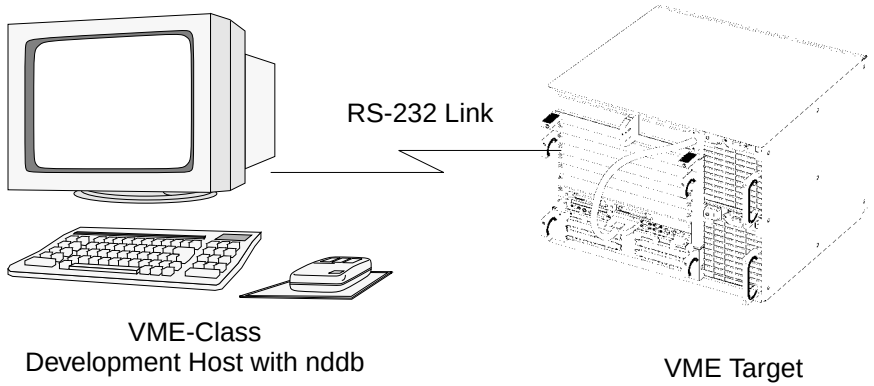



**Figure 1-5** **Development Target and Host System**

Some people find it easier to develop a driver and download it to a diskless system, which will reboot faster than waiting for their host HP-UX to do a file system check when it crashes. Others prefer the simpler directory structure of running the driver on their own system, and they do a **sync** command so that the inevitable **fsck** takes minimal time.

You need the ANSI C compiler, and the device driver debugger, **nddb**. The **nddb** debugger is very similar to **xdb** in its functionality, and for the most part you can use **xdb**'s man page for information on it. (See you local sales representative for help on obtaining **nddb**, which, although available on request, is not a supported product.)

# 2

# Memory Management

This chapter presents the basic concepts involving VME memory management and the interrelationship between master and slave VME boards. Its general structure is:

- Basic concepts of VME memory management
- Introduction to the memory-related VME functions and user-level **ioctl** commands

Although the Location Monitor and FIFO are related to memory management, we discuss them in later chapters.

## VME Memory Mapping for the VME-Class

Mapping came about because of the evolution of the VME bus standard. When there was originally only a 64-KB range of memory, there was no need for mapping. When the 24-bit A24 memory space became available, and later the 32-bit A32 space, the question arose as to where to allocate the different types of address space.

The address lines on the VME bus include the 32 lines that can directly specify a 4-GB address range. For A32 space, all these lines are used for 4 GB. For A24 and A16 space, the top 8 and 16 lines, respectively, are not considered, and the resulting address ranges are smaller (16 MB and 64 KB).

However, to specify what type of space is being requested takes at least an additional 2 lines (to represent the A16, A24, and A32 modifications). In fact, there are 6, not 2, additional address lines, because the VME bus standard was expanded to include the potential for distinguishing between the **sub-space modifiers** that might be used to specify varying types of supervisory and non-privileged access. Because the PA-RISC and local-bus memory architecture of the HP VME-Class doesn’t have these additional six lines, the VME bus adapter is used to provide the mapping between the 32-bit local I/O space address and the 38 bits provided by the VME bus.

## Mapping A16 Memory

The HP VME-Class systems automatically map 64 KB of local-bus, I/O memory to VME A16 memory. However, programs set up access to this A16 memory via the **map_mem_to_host**() function, just as they do to access A24 and A32 memory.

***NOTE:*** This is a one-way mapping: the VME bus adapter on the 700 Series machine does not respond to A16 cycles mastered by another VME board. Other VME boards can master (that is, request) addresses in this A16 space, but the local board can not make A16 memory accessible by providing a slave mapping for it.

## Master Mapping

**Master mapping** is the term used to indicate that we are trying to access VME address space that is not local to our CPU. As a bus master, the HP 743 system can map a portion of VME space into the PA-RISC I/O space. When it reads or writes to these addresses in I/O space, addresses and address modifier lines are raised on the VME bus and are recognized by other VME boards that provide the physical memory.

Just as you use virtual addresses to access a local memory area, you use a virtual address to access VME memory through a set of registers called **master mappers**. Each register maps 4 MB of VME space. Reads and writes in this range are translated into addresses in VME A24 or A32 space. (See Figure 2-1.) The actual VME address and specific sub-space depend on the programming of the selected mapper and the address modifier.

The kernel allocates mapper space at power-up time, based on the presence of graphics controllers and EISA bus adapters. This allocation directly determines the number of VME master mapper entries, each of which allows a 4-MB space in the I/O space to access 4 MB of VME space. The VME addresses are aligned to a 4-MB boundary.

.

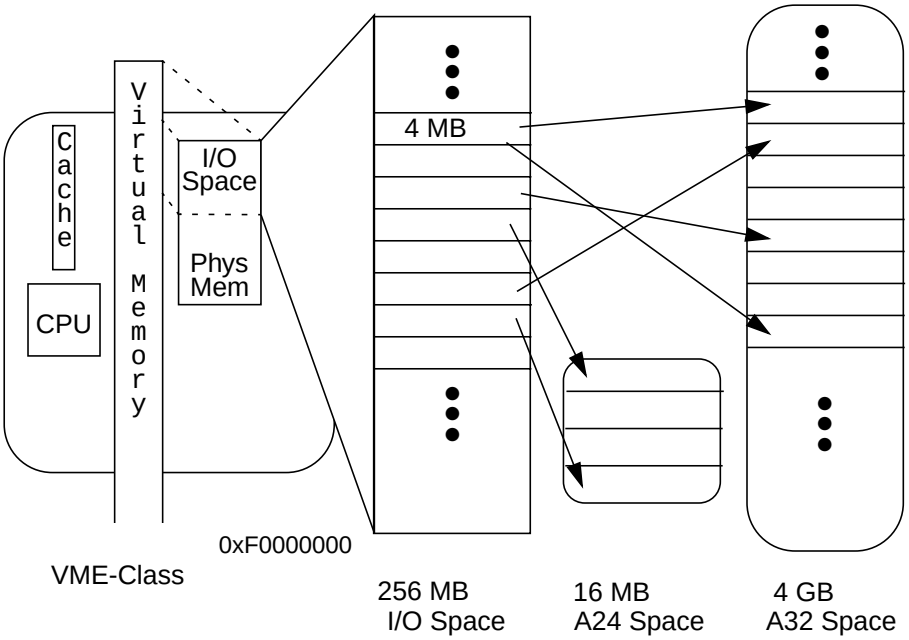



**Figure 2-1** **Master Mapper Operation**

The total available space for the master mapper entries varies from system to system, from less than 48 MB in a 747*i* system with EISA and two graphics controllers, to 240 MB in a 742*rt* system, which does not contain graphics controllers or EISA. Since each entry takes 4 MB, there can be from 12 to 60 usable master mapper entries.

## Slave Mapping

When a VME master initiates a bus read or write cycle, it can be thought of as having requested memory supplied on another VME board for its use. The other board, of which the 700 Series is also an example, is known in this context as a **VME bus slave**. It provides the memory by using **slave mapper** facilities to translate the VME bus address into requests for physical RAM memory, performing a function very similar to virtual paging hardware. (See Figure 2-2.)

.

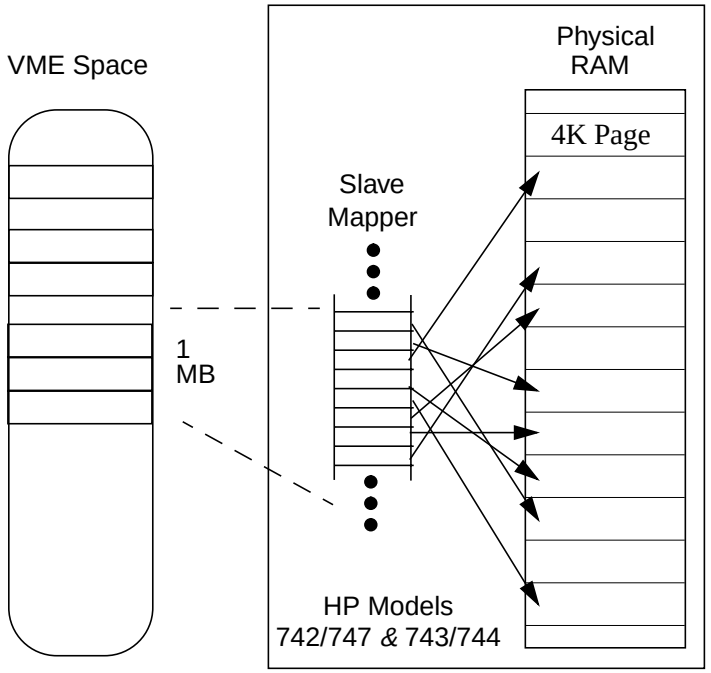



**Figure 4-2** **VME Bus to HP Memory via Slave Mapper**

Other VME masters view the 256-entry, 1-MB virtual-memory layout for the host slave mapper as a single, contiguous region, not the scattered, discontiguous 4-KB pages in physical memory. Within this 1-MB space, each 4-KB chunk may be mapped to any page of the HP physical memory.

### The Safe Page

You can see that the slave mapper points into physical memory. Until your program maps a particular physical page, all the pointers point to one single page, called the **safe page**. When pages are unmapped, the pointers return to pointing to this safe page. The safe page thus makes sure that any read or write to memory doesn't fly out of the mapper and into some page at random if that mapper register isn't initialized.

## Direct Mapping

The HP VME adapter also contains a second VME slave controller, the **direct mapper**. This controller can map more than a single MB of VME space to physical addresses, but it can't perform the page-by-page translations of the slave mapper. Therefore, to other VME masters, the pages appear in their physical order, making this mapper harder to use by local drivers, which don't see physically contiguous pages as virtually contiguous.

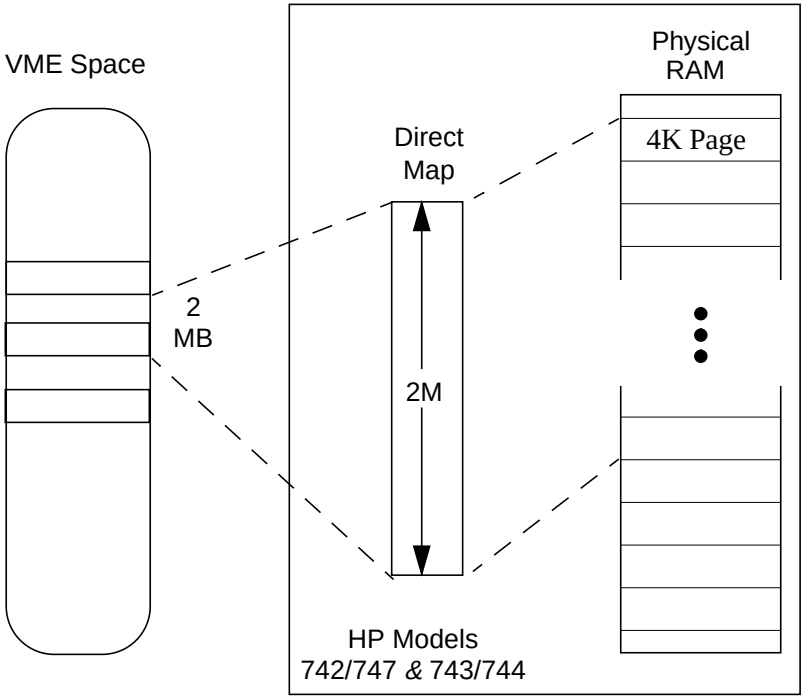



**Figure 4-3** **Direct Mapping**

There are actually two direct mappers, one for A24 space and one for A32.

If the A24 direct mapper is set to map, for example, 2 MB, then it maps 2 MB of VME A24 space to 2 MB of the HP CPU's physical address space. (See Figure 2-3.) The physical RAM can be made to appear at any 2-MB

boundary in VME A24 space, and other VME boards can reference it. The addresses in VME space and in the HP CPU's physical address space must both be aligned to the size of the chunk of memory being mapped.

The direct mappers on the 742/747 and the 743/744 systems differ as follows:

- The 742/747 A32 direct mapper maps 256 MB of HP-UX physical address space to the VME backplane, where the VME region is mapped on a 256-MB boundary. The first byte of the VME region maps to the first byte of physical memory on the HP-UX system. (Even though the HP system will not likely have 256 MB of physical memory, the entire region is assigned to it anyway, but an error is returned if another VME bus master tries to access nonexistent memory.)

  This A32 direct mapper supports only the A32_SUP address modifier (see "Setting the Address Modifier" on page 2-9), and it cannot be turned off.

- Similarly, the A24 direct mapper on the 742/747 maps the bottom 1 MB of HP-UX physical space to A24 space, on a 1-MB boundary. It only responds to the A24_SUP address modifier, and it also cannot be turned off.

- The two 743/744 direct mappers map a chunk of HP-UX physical address space to the VME backplane, as defined by the **vme_config** program and read at start-up. Each chunk's size is a power of 2 between 64 KB and 16 MB (for A24 space) or between 64 KB and 512 MB (for A32 space). Each can be configured to a starting point within the physical address space, on a boundary the size of the chunk, and each mapper can be turned off.

  Address modifiers can also be specified in the configuration utility. Each mapper can be programmed to respond to up to six modifiers in A24 space and A32 space, respectively.

## Anatomy of a Memory-Mapped Read Request

As an example of how the mapping works, consider Figure 2-4, below, which diagrams two HP VME-Class systems. The system on the right issues a read request, and the system on the left fulfills it. The following steps are taken, matching the numbered steps in the figure:

**1** The PA-RISC CPU on the right issues a read from an address in I/O space.

**2** The VME bus adapter's master mapper sees the request.

**3** The request is sent to its VME address mapping and the read request goes out onto the VME bus.

**4** The second CPU's slave mapper sees the request on the VME bus and recognizes it as one of its own addresses.

**5** This slave mapper maps it into this CPU's physical memory.

**6** The data gets put on the VME bus, taking the reverse path from **5** to **4.**

**7** The original CPU reads the data, taking the reverse path from **3** to **2** to **1**.

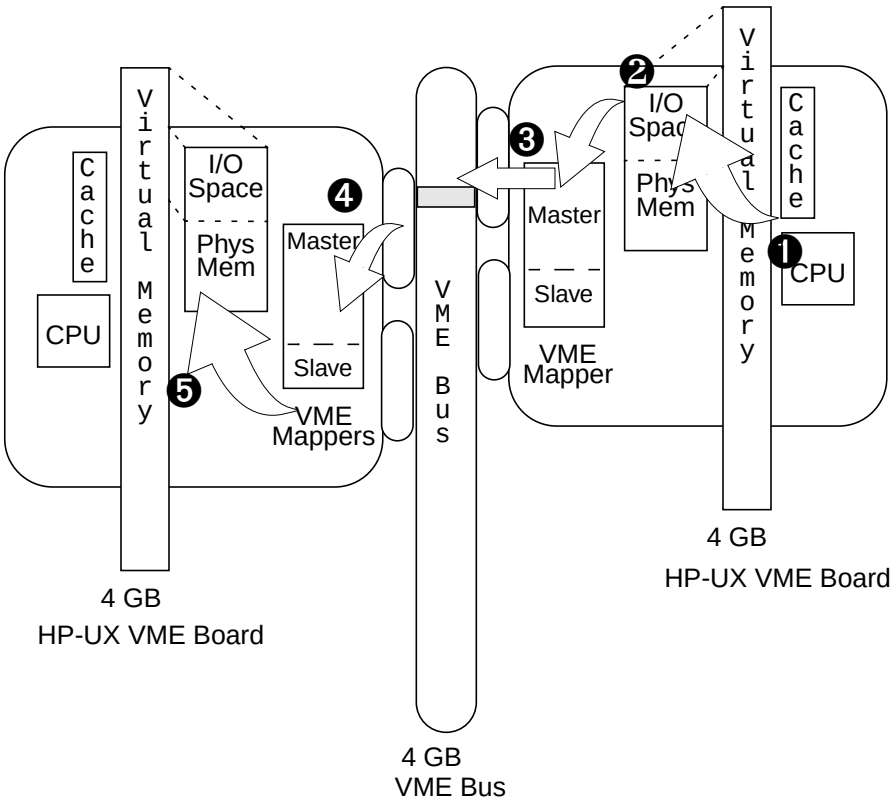



**Figure 4-4** **Memory-Mapped Read Request**

# Memory Mapping Routines

The mappers are set up and unmapped as follows:

- The master mapper—Set up from **map_mem_to_host**(2) and the **ioctl**() command VME2_MAP_ADDR; and unmapped by **unmap_mem_from_host** and VME2_UNMAP_ADDR.
- The slave mapper—Set up from **map_mem_to_bus** or a variant such as **vme_map_mem_to_bus2**. Unmapped by **unmap_mem_from_bus**.
- The direct mappers—Permanently allocated at boot time, and no map or unmap calls are provided.

## The *isc* Structure

The *isc* structure for a driver is returned from a series of driver-related functions that are executed during system start-up and returned to your *driver*_**attach** function. If you're using the **ioctl** mapping commands for user-level access, you don't have to worry about it.

*isc* is short for Interface Select Code, which is an operating system table that contains entries for each card (CORE I/O select code, EISA slot, or VME card) in the system. As used in this manual, an *isc* is a structure describing a VME card. For information about its fields, see "isc" on page A-13. See also "isc_claim()" on page B-8, and "Attach Routines" on page 6-2.

Much of the control over VME boards comes from modifying fields in structures such as *isc*, *dma_parms*, *io_parms*, and *buf*, the general buffer of information about a particular card. However, you can do mapping calls such as those described here without any change to the provided *isc*.

## Setting the Address Modifier

All of the mapping routines reference the addressing mode value that can be set in an *isc* structure by using **vme_set_address_space(**2**)**, which has the following form:

```
int vme_set_address_space (isc, address_modifier)
    struct isc_table_type *isc;
    int address_modifier;
```

Specify a modifier of VME_A16, VME_A24, or VME_A32 for the default modes (which assume supervisory data access) or one of the set of modifiers in the following table:

**Table 2-1 Symbolic Names for Memory Address Modifiers**

| Name | Explanation |
|---|---|
| SHORT_NP_ACCESS | A16 non-privileged access |
| SHORT_SUP_ACCESS | A16 supervisory access |
| STD_NP_DATA_ACCESS | A24 non-privileged data access |
| STD_NP_PRGM_ACCESS | A24 non-privileged program access |
| STD_SUP_DATA_ACCESS | A24 supervisory data access |
| STD_SUP_PRGM_ACCESS | A24 supervisory program access |
| EXT_NP_DATA_ACCESS | A32 non-privileged data access |
| EXT_NP_PRGM_ACCESS | A32 non-privileged program access |
| EXT_SUP_DATA_ACCESS | A32 supervisory data access |
| EXT_SUP_PRGM_ACCESS | A32 supervisory program access |
| STD_SUP_BLK_ACCESS | A24 supervisory block access |
| STD_NP_BLK_ACCESS | A24 non-privileged block access |
| STD_SUP_D64_ACCESS | A24 supervisory, 64-bit block transfer |
| STD_NP_D64_ACCESS | A24 non-privileged, 64-bit block transfer |
| EXT_SUP_BLK_ACCESS | A32 supervisory block access |
| EXT_NP_BLK_ACCESS | A32 non-privileged block access |
| EXT_SUP_D64_ACCESS | A32 supervisory, 64-bit block access |
| EXT_NP_D64_ACCESS | A32 non-privileged, 64-bit block access |

## Setting up the Master Mapper

**map_mem_to_host**, a kernel routine, sets up one or more master mapper entries to map a range of memory beginning at a VME address that you specify (*vme_addr*), and it returns your host's virtual I/O space—not physical—address for the beginning of the range. Later, reads and writes in this range access the specified VME addresses.

Its form is:

```
caddr_t map_mem_to_host (isc, vme_addr, num_bytes)
    struct isc_table_type *isc;
    caddr_t vme_addr;
    int num_bytes;
```

Once a successful call is made, you can use the returned I/O space pointer to read or write from the mapped space.

You can also set up error-handler functions to recover from attempted access to unmapped space. (See “Setting an Error Handler” on page 3-10.)

#### Unmapping Master Mapping

When you have finished with the mapping, you should unmap the VME area by using the unmap call whose form follows:

```
unmap_mem_from_host (isc, vme_addr, num_bytes)
    struct isc_table_type *isc;
    int vme_addr, num_bytes;
```

In general, call this only after a successful call to **map_mem_to_host** . And, because **unmap_mem_from_host** does not do any error checking, be sure that you always:

- Unmap both the original VME address
- Unmap the same number of bytes

## Mapping Slave Memory

The **map_mem_to_bus** and its kindred **vme2** functions (such as **vme_map_mem_to_bus2**, **vme_map_largest_to_bus, vme_map_pages_to_bus**, and **vme_map_polybuf_to_bus**) map local, physical memory to the VME bus so that other processors or smart I/O cards can access it. (This capability is not available for user-level VME access.)

For all of the mapping functions, the returned VME address takes into consideration the address that you send in and the address space modifier that you specify in the *isc* structure. If you ask for a VME address modifier in A24 space, the high byte of the returned address will be 0—that is, 0x00xxxxxx.

### Obtaining Current System Parameters

The **vme_hardware_map_info** function obtains the VME address range, base, and address modifiers pertinent to the direct and slave I/O maps on the current system. You can use this function, for instance, to

- Get the VME address range, so that you don't try to specify a VME address that isn't within the current I/O map when you call **vme_map_mem_to_bus2**
- Check if the address modifier that you want is supported and enabled
- Get the local, physical address range of direct-mapped memory

This function has the following form:

```
caddr_t vme_hardware_map_info(hw_map_type)
    struct vme_hardware_map_type *hw_map_type;
```

See "vme_hardware_map_type" on page A-18 for the structure.

### The *io_parms* Structure

The slave mapping routines use the *isc* structure as well as an *io_parms* structure in which the following fields are the only ones that you should change (see "io_parms" on page A-12):

```
struct io_parms
    . . . /* various fields */
    int (*drv_routine)();/* routine to be called */
    int drv_arg;/* passed to drv_routine */
    caddr_t host_addr;/* local-bus virtual addr */
    . . .
    u_int num_bytes;/* # total bytes to map */
};
```

If slave mapper pages cannot be immediately allocated (and you provide a routine to call), when the mapping becomes available as requested in either mapping function (below), the map function calls the **drv_routine** that you have specified, passing it the **drv_arg**.

Be sure to page-align the local-bus address (*io_parms*->**host_addr**) that you're requesting, if you use **vme_map_mem_to_bus2** or **vme_map_pages_to_bus**. (For **vme_map_pages_to_bus**, you must supply a list of pages, instead of setting the *io_parms*->**host_addr** field.)

#### Making Your Memory Available to the Bus

**map_mem_to_bus** returns a VME address that may be used by another VME card to access your local memory space. You don't need to specify the VME address, because this call selects one for you. This routine has the following form:

```
caddr_t map_mem_to_bus (isc, io_parms)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
```

It assumes that you have filled in the proper fields in the *io_parms* struct as explained above. If this call fails to acquire a mapping, your request is put in a queue, and your process continues until the resource becomes available, in which case **vme2** calls the driver routine that you specified in *io_parms*.

#### Mapping Your Memory to a Specific VME Address

**vme_map_mem_to_bus2** is the same as **map_mem_to_bus,** except that you can specify that your card's local-bus memory (*io_parms***.host_addr**) is visible at a specific VME address. This routine has the following form:

```
caddr_t vme_map_mem_to_bus2 (isc, io_parms, vme_addr)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
    caddr_t vme_addr;
```

This routine returns a page-aligned VME address that maps to the page-aligned virtual address that you request in *io_parms*. The routine gives you what you sent in, but it checks to see if you can have it.

#### Mapping Pages of Memory to the VME Bus

**vme_map_pages_to_bus** is also similar to **map_mem_to_bus,** except that it maps a specified number of pages, which need not be contiguous. As with **map_mem_to_bus**(), you don't specify the VME address—instead, this is returned to you as a pointer to the base of the slave area that is mapped. This routine can be used to scatter-gather a collection of non-contiguous regions or pages into a single, contiguous region in VME space (contiguous as seen by VME masters).

**vme_map_pages_to_bus** has the following form:

```
caddr_t vme_map_pages_to_bus (isc, io_parms, pages,
        num_pages_p, all_or_nothing)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
    char **pages;
    unsigned int *num_pages_p; /*how many to map */
    BOOLEAN all_or_nothing; /* map all pages or none */
```

When you set the *all_or_nothing* flag to FALSE, this function's behavior is similar to the next function, below.

#### Mapping a Variable Size Buffer to the VME Bus

The **vme_map_largest_to_bus** kernel routine interrogates the slave mapper for the number of pages required for the size of buffer that you specify, then calls **vme_map_pages_to_bus** to map the largest possible number of pages beginning at the host address that you specify.

This routine has the following form:

```
caddr_t vme_map_largest_to_bus(isc, io_parms)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
```

Before calling **vme_map_largest_to_bus**, first set *io_parms*->**host_addr** and *io_parms*->**size** to specify the host address and the size of the maximum buffer to be mapped. If the size can't be mapped, *io_parms*->**size** is adjusted to the largest buffer that can be mapped.

#### Mapping Multiple Regions to the VME Bus

The **vme_map_polybuf_to_bus** kernel routine lets you map an array of pointers to multiple buffers, to the beginning of the host address that you specify.

This routine has the following form:

```
caddr_t vme_map_polybuf_to_bus (isc, io_parms,
            max_pages)
struct isc_table_type *isc;
struct io_parms *io_parms;
uint max_pages;
```

Before calling this routine, set *io_parms*->**host_addr** to the *polybuf* struct containing pointers to the regions that you want to map (see “vme_polybuf” on page A-20); set *io_parms*->**drv_routine** to a callback function to be called when the mapping is complete, and *io_parms*->**arg** to a value to be sent to the callback.

The *max_pages* parameter lets you limit the number of 4-KB pages that will be used to map the buffers that you specify; a *max_pages* value of 0 lets the function map as many pages as needed, up to 256.

#### Unmapping Slave Memory

**unmap_mem_from_bus** returns slave mapper resources to the system. The I/O map has only 256 entries, so it should be treated as a scarce resource, and the entries returned to the system as soon as you are done with them. This routine has the following syntax:

```
unmap_mem_from_bus (isc, io_parms)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
```

This routine returns NULL on success.

***NOTE:*** You must pass in the ***same*** *io_parms* struct that you specified when you mapped the slave memory.

# 3

# Interrupts

This chapter presents the basic concepts involving VME interrupt handling by the Series 700 HP-UX systems. Its general structure is:

- The Series 700's I/O interrupt handling
- VME bus interrupt handling
- Interrupt-related VME functions
- Software triggers

Interrupts are generated by and handled differently for:

- VME cards or I/O devices, for which you use ISR interrupt service routines that you have registered with **vme_panic_isr_hook()**—see "ISR Panic Hook" on page 3-12.
- VME bus errors, for which you register interrupt functions with **vme_panic_berr_hook()**—see "Bus Error Panic Hook" on page 3-11.
- Location Monitor and FIFO—see "Location Monitor and FIFO" on page 3-19.

## Series 700 I/O Interrupt Handling

The VME bus has a 7-level interrupt structure for I/O interrupts. Level 7, or IRQ7, has the highest priority. Interrupts are generated by boards on the VME bus via a module called an **interrupter**. The interrupter asserts one of the seven IRQ lines. A board that services an interrupt must have a module called an **interrupt handler**.

HP-UX allows I/O interrupts a chance to interrupt the execution of the current HP-UX process. It then executes an interrupt handler until the handler completes or is itself interrupted by a higher priority interrupt. Thus there is an interrupter module that has placed the interrupt on the bus, and an interrupt handler, provided by the **vme2** driver, that eventually calls the interrupt service routine that you write.

The general steps in processing an interrupt from a VME I/O card are:

1. The I/O card generates an interrupt by placing a signal on the VME bus.
2. The Series 700's VME bus adapter recognizes the signal and gets the interrupt information from the bus (see "Recognizing which Device Interrupted" on page 3-3).

3 The Series 700 VME bus adapter signals the PA-RISC CPU and sets an appropriate EIR bit in the 700's External Interrupt Register (EIR) to indicate the interrupt level, then waits until the CPU makes the PA-RISC system bus available.

4 The **vme2** interrupt handler calls your driver's interrupt handler at level 5.

5 Your driver processes the interrupt as appropriate, then returns to the **vme2** interrupt handler.

6 The **vme2** interrupt handler re-enables the VME bus interrupts at this level, before searching through the remaining **IACK** (Interface Acknowledge) registers at this interrupt level for another status ID for another registered interrupt handler.

7 The HP-UX kernel handler re-enables the EIR bit.

## Interrupt Service Routines

When using interrupt-driven I/O, the device sends an interrupt when it has completed an I/O operation and is ready to send or receive more data. Your driver provides an **interrupt service routine** (ISR) that the HP-UX kernel calls when an interrupt comes in from the device.

The kernel knows to call your ISR because your driver will have called **isr_link**, on initialization, to link your ISR into the list of all of the interrupt service routines at a given interrupt level.

## Recognizing which Device Interrupted

When an interrupter asserts one of the seven IRQ lines, the bus adapter's interrupt handler responds with an interrupt-acknowledge cycle (IACK) if it is assigned to that interrupt level. The IACK acquires the bus in order to retrieve the status ID from the interrupter, and turns the status ID over to the HP-UX kernel to decide which driver's interrupt service routine to call to service the interrupt.

The status ID is typically programmed by the device's driver though a register. In the case of HP-UX VME Services, a call to **vme_isrlink**(2), with an argument of 0, assigns a status ID value, and it is up to the driver writer to tell the card what it is. If your card requires a specific value, then call **vme_isrlink** with this value.

# Your Device Driver's Interrupt Handler

Assume that your driver performs input and output using interrupt I/O.

## Two Halves of a Driver

At some point during the processing of an I/O request, control transfers to the interface driver responsible for direct communication with the interface card that initiates transfers (to or from the device) and handles interrupts.

The device driver entry-point routines constitute an **application-oriented half** of the driver and are the routines that you write, as described in Chapter 6. A system call from a user application program activates these. An **interrupt-oriented** half of the driver processes interrupts from the device. The halves work as follows:

1. The application-oriented half of the driver initiates activity on the device and waits.
2. The device completes the activity and interrupts, causing the interrupt-oriented half of the driver to tell the application-oriented half that it can continue.

Your ISR and supporting routines handle the interrupts.

The following steps illustrate the sequence of actions taken in an I/O request. The various *driver_routine*() calls are functions that you write.

1. A user executes a program that does a **read**(2) system call on a device file.
2. On entering the system call, the **read** function:
   - Performs necessary preprocessing of the request
   - Extracts the major number from the inode for the special file
   - Uses the number as an index into the **cdevsw** table that tells the system to call the *driver*_**read**() routine to complete the processing of the read request
3. **Read** invokes your *driver*_**read** routine, which calls **physio**(2) to invoke your *driver*_**strategy** routine.
4. Your *driver*_**strategy** routine queues a request on the interface driver for a read from the device. The interface driver is currently processing another request, so the *driver*_**strategy** routine returns, and **physio** sleeps, waiting for the I/O request

to complete.

5 The device interrupts when it completes the previous I/O request. The interface driver processes the interrupt and starts the next request waiting on the queue (this is our request). The interface driver tells the hardware to perform our read request.

6 The hardware completes our read request and interrupts the system. The interface driver processes the interrupt, waking up the device driver.

7 Control returns to the *driver*_**read** routine, which completes any final, device-specific processing of the request. It then returns to the **read** system call, which completes the request and returns control to the user process.

## spl* **Routines**

Interrupts cause the interrupt-oriented half of the driver to execute, and they can occur at any time. Since an interrupt could occur while the application-oriented half of your driver is executing, you must protect critical sections of code and the manipulation of shared data structures by your driver's application-oriented half from your device's interrupts.

The **spl***(2) routines are designed to let you protect critical sections of code to guarantee that an interrupt doesn't occur unexpectedly. **Spl1**, and so on through **spl7**, set the processor interrupt level to 1, and so on through 7, respectively, and return the current interrupt level before the routine was called. When the interrupt level is set, all interrupts at that level and below are turned off.

There are conventions in HP-UX regarding the types of devices that use each interrupt level. VME uses hardware interrupt level 6, which takes precedence over EISA, the system clock, timeouts, and core LAN, SCSI, RS-232, and parallel devices.

Uses for the **spl*** routines include making interrupts while:

- Manipulating common data
- Starting I/O so that a partially initiated request does not cause an interrupt at the wrong time
- Doing an operation that cannot be delayed, for instance, handshaking bytes from non-buffered, non-DMA devices

#### A Skeletal spl* Routine

Here is a skeleton showing the use of the **spl*** routines to protect a critical section of code:

```
/* code appearing in user context routine */
/* (driver_read, for instance)            */
    int s;

    s = spl6();

/* manipulate data structure that the     */
/* interrupt context (isr) also uses      */

    splx(s);
```

You must save the return value to use at the end of the critical section to call **splx(**2) to restore the previous interrupt level.

There are restrictions for using the **spl*** routines. During an interrupt, you can **spl** to a higher priority and back down again, but you cannot **spl** to a priority lower than the priority at the time of the interrupt. If you wish to do some processing at a lower priority level, use the software trigger mechanism described in "Software Triggers" on page 3-13.

## Using Interrupts

To use interrupts:

- Determine the interrupt level to use
- Set up a status ID value to represent the hardware vectoring
- Write and link the ISR with the status ID

#### Finding Available Interrupts

The **vme_config** program may split VME interrupt handling among multiple processors. To determine the available interrupts on your driver's processor, look at the **avail_interrupts** field of the *vme_hardware_type* struct, which is returned by **vme_hardware_info()** and contains other parameters of your driver's system.

```
#include "/usr/conf/wsio/vme2.h"
#define MYCARDINT1 0x2   /* interrupt mask for level 1 */
```

```
#define MYCARDINT2 0x4   /* interrupt mask for level 2 */
struct vme_hardware_type hw_type;

vme_hardware_info(&hw_type);
if ((hw_type->avail_interrupts) & MYCARDINT2)
        /* call vme_isrlink() to set up interrupts */
        /* code to set up card to interrupt on level 2 */
else if ((hw_type->avail_interrupts) & MYCARDINT1)
        /* call vme_isrlink() to set up interrupts */
        /* code to set up card to interrupt on level 1 */
else
        /* code to handle driver without interrupts */
```

You can also use the **vme2** macros VME_HIGHEST_PRI_AVAIL_IRQ and VME_LOWEST_PRI_AVAIL_IRQ to choose either the highest or lowest IRQ available on the processor.

#### Setting Up a VME Card Interrupt Vector

For your card to be serviced when it interrupts, it must send the kernel its hardware vector status ID so that the kernel **vme2** interrupt processor can determines which interrupt is yours.

When it determines this, **vme2** calls your ISR with a pointer to an *isc* and either (a) the *status_id* and the *interrupt_level*, or (b) a value that you have provided in a call to **vme_isrlink(**2), which has the following syntax:

```
int vme_isrlink (isc, isr_addr, status_id,
                 arg, sw_trig_lvl)
    struct isc_table_type *isc; /* to be updated */
    caddr_t isr_addr;           /* your isr */
    status_id;          /* status ID to try to obtain */
    int arg;            /*to be passed to isr */
    int sw_trig_lvl;    /* currently ignored */
```

**vme_isrlink**() links an interrupt service routine (*isr_addr*) to a VME-generated status ID.

If you can set the card status ID by setting a register, set *status_id* to 0x0 before calling **vme_isrlink** to let the kernel pick a vector.

If your VME card has restrictions on the value of its interrupt vector (perhaps the card sets it with DIP switches), set *status_id* to the vector that your card uses.

The *arg* parameter determines if the card's status ID will be passed into your ISR (if you set *arg* to NULL), or if your *arg* itself specifies the status ID value by containing it.

The following example illustrates ISR linkage and vector setup:

```
#include "/usr/conf/wsio/vme2.h"
#define MYCARDINT  0x2 /* interrupt mask for level 1 */
#define MYREALINT  0x1
#define DUMMYLEVEL  0x0
int my_isr();                    /* addr of isr   */
struct isc_table_type *my_isc;  /* set in attach */
struct my_cardregs *cardregs;   /* set in attach */
unsigned char my_vector;

int my_attach();
{
int use_vec = 0x0;
int my_lvl = DUMMYLEVEL;    /* parm not implemented */
int stat_id = 0x0;
struct vme_hardware_type hw_type;

vme_hardware_info (&hw_type);
if((hw_type.avail_interrupts)&MYCARDINT){
    VME_INIT_IF_INFO(D08_IRQ_TYPE,VME_A32);
    my_vector = vme_isrlink
                (my_isc,my_isr,use_vec,stat_id,my_lvl);
    if (my_vector == NULL)
        /* code to handle card no interrupts */
    }
    else{
        /* set up card */
        my_cardregs->int_vector = my_vector;
        my_cardregs->int_level = MYREALINT;
    }
} else
    /* code to handle card without interrupts */
}
```

Upon success, **vme_isrlink**() returns the status ID (vector number) obtained; on failure, it returns 0x0.

The **vme_get_status_id_type()** and **vme_set_status_id_type()** functions get and set VME interrupt handling types. These functions take an *isc* and a *value_type* as arguments, as seen in the following syntax:

```
vme_get_status_id_type(isc, value_type)
    struct isc_table_type *isc;
    int value_type;
```

where *value_type* is UNKNOWN_IRQ_TYPE, D08_IRQ_TYPE, D16_IRQ_TYPE, or D32_IRQ_TYPE. These routines return NULL on failure.

## Writing the Interrupt Service Routine

Your ISR is called by **vme2**'s interrupt routine and has the following signature:

```
int driver_isr (isc, arg, level)
    struct isc_table_type *isc;
    int arg, level;
```

Thus, your ISR is called with the following arguments:

- The *isc* for this driver.
- The meaning of the *arg* parameter depends on how you originally called **vme_isrlink()** to link your ISR with the VME-generated **status ID**. (See "vme_isrlink()" on page B-32.) Your call could:
    - Request a specific ID, which will later be passed to your ISR as *arg*—and if it failed to receive the ID, the value of *arg* would be zero
    - Accept an assigned status ID, and later pass it to your ISR as *arg*
- The third parameter is the VME IRQ *level*.

### An Example *driver*_isr Routine

The following code shows a *driver*_**isr()** routine for a non DMA transaction:

```
driver_isr (isc, arg, level)
struct isc_table_type *isc;
int arg, level;
{
    register struct isc_table_type *isc = info->temp;
    register char *card = (char *)isc->card_ptr;
```

```
/* Is this interrupt for us? */
    if ((card->status_reg & 0x40) == 0)
        return;
    /*  The reference to the control register
        clears the interrupt on that card. */
    parm1 = card->control_reg;
    wakeup(&skel_buf);
}
```

It is the responsibility of your driver's ISR to:

- Stop a card from interrupting
- Determine a reason for an interrupt (if appropriate)
- Take appropriate action, such as cleanup or retry
- Call **wakeup()** or **iodone(),** or initiate the next step in I/O

## VME Bus Errors and Panics

VME cards generate bus errors when access is attempted to memory that is not mapped. If these errors are not handled by a bus error handler, they will cause a system panic. To avoid this, you can:

- Probe first using **vme_testr**(2) or **vme_testw**(2); or **vme_reg_read**(2) or **vme_reg_write**(2)
- Set up bus-error handlers for master mapper errors returned from calls to **map_mem_to_host**(2), by calling **vme_set_mem_error_handler**()

### Setting an Error Handler

**vme_set_mem_error_handler**() assigns an error handler to each 4-MB entry of the master mapper for the specified A24 or A32 address space.

```
void vme_set_mem_error_handler (isc, pa_address, size,
                                handler, arg)
    struct isc_table_type *isc;
    unsigned int pa_address;
    int size;
```

```
ROUTINE handler;
caddr_t arg;
```

The *pa_address* and *size* of the buffer to associate with the error handler must be obtained from a prior call to **map_mem_to_host**. The *arg* is an argument to be passed to the error handler when a VME bus error occurs.

### Bus Error Panic Hook

The **vme2** driver has a pointer-to-a-routine named **vme_panic_berr_hook** whose default value is NULL. This allows a driver to link in a routine to handle bus errors caused by improper kernel driver accesses to VME space that are not handled by a per-master-mapper error handler.

The **vme2** interrupt handler is called whenever a VME bus error occurs. The interrupt handler first checks to see if the error occurred at a user-mapped address, and, if so, the interrupt handler signals the application that originally mapped the address via the VME2_MAP_ADDR user **ioctl**.

Otherwise, the interrupt handler checks if a driver has mapped in the master mapper involved in the bus error (if the master mapper is unmapped, the bus error is ignored). If the driver has registered an error handler for the master mapper, that error handler is called. Otherwise, as a catch-all for all bus errors caused by driver attempts to map non-master-mapped memory, the interrupt handler checks to see if the value of **vme_panic_berr_hook** is not NULL. If this the value is NULL, **panic**() is called; otherwise, the pointed-to handler is called, and, upon return, the **vme2** interrupt handler simply returns and the system continues running.

The following code shows how a bus error panic hook can be implemented. If the card that generates the bus error can keep generating errors, you will want to do more than print an error message such as shown below:

```
extern int (*vme_panic_berr_hook)();
void vpbushook();

vpisrhook_link(){
    vme_panic_berr_hook = vpbushook;
}

vpbushook(vme_addr)
unsigned int vme_addr;
{
    msg_printf("VME bus error at 0x%x\n",vme_addr);
}
```

### ISR Panic Hook

The **vme2** VME driver also has a routine pointer named **vme_panic_isr_hook**(), which is initially NULL. This allows a driver to link in a routine to handle status ID panics caused by an unidentified (i.e., unregistered) status ID that might occur during an IACK cycle. A return of 0 from the handler re-enables the IRQ level, while a non-zero return leaves that IRQ level disabled.

The following code shows an ISR handler for an unregistered status ID:

```
#define ENABLE_VME_INTS    0
#define DISABLE_VME_INTS   1
#define INT_HANDLED        4

extern int (*vme_panic_isr_hook)();
int vpisrhook();

vpisrhook_link(){
    vme_panic_isr_hook = vpisrhook;
}

vpisrhook(vmeisr)
struct vme_noisr_table_type *vmeisr;
{
    msg_printf(
    "Panic ISR called with level %d; status id 0x%x\n",
        vmeisr->irq_level.vmeisr->status_id);
    if(vmeisr->irq_level == INT_HANDLED){
```

```
        msg_printf("Panic ISR re-enabled IRQ %d\n",
                    vme->irq_level);
        return(ENABLE_INTS);
    }
    else{
        msg_printf("Panic ISR left IRQ %d disabled\n",
                    vmeisr->irq_level);
        return(DISABLE_VME_INTS);
    }
}

struct vme_int_control{
    int irq_level;
    int avail_interrupts;
    int error;
}
```

This function is particularly useful when developing a driver for new hardware that may not be behaving as expected.

## Software Triggers

Most of the PA-RISC interrupts are handled at a given level in hardware, and then a kernel routine is called at a software trigger level lower than the initial hardware interrupt level. Software triggering via calls to **sw_trigger**(2) adds a software trigger request to a list that is checked whenever an interrupt is processed. The appropriate use of software triggering allows higher priority interrupts (such as the system clock) to be handled prior to calling the rest of your driver's interrupt handler.

### Software Trigger Mechanism

The **sw_trigger** routine arranges the calling of an interrupt service routine in the interrupt context at a given priority level.

- You can provide a timeout routine that sets up a software trigger so it defers its timeout processing to a lower interrupt level.

Note that standard timeouts are no longer processed at interrupt level 5, but at level 2, beginning with HP-UX 10.0. However, if you use the **timeout.h** header file, the **timeout**() function is redefined to **Ktimeout**(), which interrupts at level 5, but is processed after all other level 5 interrupts are handled. Thus, you can still use **timeout**() from current code.

- Use software triggers when a device needs to have its interrupt acknowledged quickly, but there is additional processing to be done on an interrupt that is not so urgent. A device can be set up to interrupt at a high level, and then a software trigger can complete the interrupt processing at a lower interrupt level.
- Software triggers provide a way for the top half of a driver to trigger the lower half to perform a specific function.

The kernel implements software triggers by using a linked list of structures to be serviced. This list is checked during each processor bus interrupt. Elements of this list have the structure shown in “sw_intloc” on page A-16.

Drivers can add a software trigger request to this list by calling the **sw_trigger** routine, which has the following syntax:

```
sw_trigger (intloc, proc, arg, level, sublevel)
    static struct sw_intloc *intloc;/* struct to add */
    int (*proc)();          /* call proc on trigger */
    caddr_t arg;            /* argument to routine */
    int level;              /* priority of interrupt */
    int sublevel;           /* sub-priority */
```

where:

- *intloc*— A pointer to the *sw_intloc* structure to be added to the software trigger queue. Its fields are initialized by **sw_trigger**.
- *proc*—The address of the routine to be called when the software trigger is processed.
- *arg*—The argument to be passed to *proc*.
- *level*—The priority level of the software trigger.
- *sublevel*—The sub_priority of the software trigger. This field is not used by the Series 700*i* HP-UX kernel. Set it to 0.

At the time of an interrupt, any software trigger requests on the list with *level* greater than the current interrupt level are processed. To process a software trigger request, **proc** is called with *arg* as its argument.

The software trigger queue is checked again when the processing for each interrupt is being completed. Any pending software triggers are processed in decreasing order of priority.

The **sw_trigger** routine checks to see if the structure pointed to by the *intloc* parameter is already on the trigger queue. If it is, this request is thrown away, permitting only one pending request per *sw_intloc* structure. If your driver needs to have multiple software triggers pending, it must use separate *sw_intloc* structures.

*Level* has the following restrictions on values you can use:

- You cannot set a software trigger higher than your current processor priority level. For instance, if you are executing at processor priority level 5, you cannot call **sw_trigger** with level 6.
- You cannot call **sw_trigger** with level set to 7.

### A Skeletal Driver Fragment

The following code segment is a skeletal driver fragment that uses the software trigger mechanism to acknowledge an interrupt from a card at a high interrupt level and then defer the bulk of the interrupt processing to a lower priority.

```
#include <sys/types.h>
#include <sys/timeout.h>
struct sw_intloc *mycard_intloc;

mycard_isr (isc, arg)
struct isc_table_type *isc;
int arg;
{
int reason;

    /* stop card from interrupting */
    mycard->control = .....;

    /* determine reason for interrupt and do
    any immediate interrupt processing */

    reason = ...;   /* values from card regs */

    /* set up swtrigger request to perform   */
```

```
    /* remainder of interrupt processing at  */
    /* a lower level                         */

    sw_trigger(mycard_intloc,mycard_isrII,reason,1,0);
    return(0);
}

mycard_isrII (reason)
    int reason;  /* reason for interrupt,  */
        /* passed by mycard_isr() */
{
    /* complete secondary interrupt processing */
    switch(reason)
    {
    case IOCOMPLETE:
                /* process I/O complete condition */
    case IOERROR:
                /* processing for I/O error */
    . . .
    }
}
```

**Selecting the Software Trigger Level**

When selecting the software trigger level for your device, consider the following characteristics:

- If your device requires a lot of processing to complete an interrupt, it should not complete interrupt processing at a high interrupt level.
- You must not block out other devices for too long. You should probably not spend longer than 400 microseconds at a high interrupt level (this length of time is hardware-dependent).
- Some devices need to be acknowledged very quickly when they interrupt, so that they do not lose any data. These devices should be at a high interrupt level.
- If a device needs to be acknowledged quickly, but requires a lot of processing to complete the interrupt, you may need to use additional software triggering for further processing.

When deciding on the importance of your device, consider its relative priority in relation to the other devices in your system. The following general guidelines can help you determine the relative importance of your device:

- High Priority—The system clock is always at a high priority to ensure that timeouts are handled in a timely manner.
- Next Levels—The next level of devices include real time devices to ensure that they do not lose data. Examples of devices that should be in this priority range include serial devices; networking devices; and non-buffered, non-DMA devices.
- Lower Levels—The disk subsystem should be at the next level of priority so disk I/O bottlenecks can be avoided.
- Lowest Levels—The remaining priority levels should be used for other devices (e.g. the least-used devices).

## Timeout Mechanisms

A driver uses timeout mechanisms whenever it needs to wait for a response from a device. Timeouts ensure the return of control to the driver in the event a device fails to respond within an allotted time. By setting up a call to a timeout handling routine, a driver has an opportunity to recover even if the device hangs. Call **timeout**() with the following syntax:

```
timeout (function, arg, t, timeout)
    int (*function)();          /* routine to call */
    caddr_t arg;                /* arg for routine */
    int t;                      /* count to wait  */
    struct timeout *timeout;    /* NULL */
```

**Timeout** works as follows:

1. Your driver calls it to set a timeout on a request to the device.
2. After the specified number of clock ticks and no reply, the *function* specified as the first parameter is called to abort the request, depending on how you code it.

   This routine is called at interrupt level 5 and should be treated like an ISR in terms of data structure protection by using **spl*** () and **splx**() appropriately.
3. To remove the timeout, call **untimeout** (*function*, *arg*).

The following skeleton driver fragment uses the **timeout** and **untimeout** routines:

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

mycard_timeout()        /* call this on timeout */

{
    noerror = 0; /* cause polling loop in open to exit */
    return(0);
}

mydriver_open (dev, flag)
    dev_t dev;
    int flag;
{
    . . .
    noerror = 1; /* set up timeout for 1 second     */
    timeout(mycard_timeout,0,1*HZ,NULL);

    /* poll card - wait for bit set    */
    if (noerror){
        /* if here, card responded */
        untimeout(mycard_timeout,0);
    }
    else {
        /* perform error recovery, retry,  */
        /* or report error, as appropriate */
        return(ENXI0);
    }
    . . .
}
```

# Location Monitor and FIFO

The Location Monitor and FIFO stack are primarily designed for interprocessor communication. VME Services provides simple calls to access a basic functionality that you can use to build your own system.

## Location Monitor Functions

The Location Monitor provides an interrupt mechanism that can alert a user process when a particular range of addresses has been read from or written into. It passes the user process a pointer to a function that can be executed to react to this information.

See Appendix B for more information on the functions that are related to the Location Monitor:

- **vme_locmon_grab**—Locks the Location Monitor for exclusive use, thereby avoiding interference from other processes. You specify the address, address modifier, the size, and a routine to be called when an interrupt is generated because the monitored area has been written into or read from.
- **vme_locmon_poll**—Polls the Location Monitor to find out if any interrupts have occurred, and if so, how many.
- **vme_locmon_release**—Unlocks the exclusive grab that this process has on the Location Monitor.

## FIFO Functions

The FIFO provides a method of ordering asynchronous communications received by a VME processor. Whenever a FIFO is written into, an interrupt occurs on the HP's VME processor board, and a user-specified function can be executed to read and react to the FIFO's contents.

The primary differences between the FIFO and the Location Monitor are:

- The Location Monitor reports both reads and writes.
- The FIFO's address/register area contains information in the first two bytes.

See Appendix B for more information on the functions that are related to the FIFO. Essentially, they parallel the Location Monitor functions and are as follows:

- **vme_fifo_copy**—Transfers data to or from the FIFO. This is a clone of **vme_mod_copy**, except that the VME side of the transfer is expected to be a FIFO. In other words, only the PA-RISC CPU's side of the transfer's addresses will be auto-incremented.
- **vme_fifo_grab**—Locks the FIFO for exclusive use, thereby avoiding interference from other processes. You specify the address, address modifier, the size, and a routine to be called when an interrupt is generated because the FIFO has been written into.
- **vme_fifo_poll**—Polls the FIFO to find out if any interrupts have occurred, and if so, how many.
- **vme_fifo_read**—Reads the contents of the FIFO register.
- **vme_fifo_release**—Unlocks the exclusive grab that this process has on the FIFO.

# 4

# Direct Memory Access (DMA)

Direct Memory Access (DMA) uses separate hardware to transfer data to or from system memory and VME address space, thereby freeing the CPU for other computation.

The **vme2** driver provides:

- The queuing of buffers for asynchronous DMA transfer via the functions **vme_dma_queue**() and **vme_dma_queue_polybuf**(), with specification of a callback function to be called when the transfer completes
- User- and kernel-level, synchronous access to the DMA controller with calls to **vme_dmacopy**(), **vme_copy**(), and the VME2_USER_COPY **ioctl**() command
- Initialization of data structures and hardware for synchronous DMA hosted by either the Model 743/744 hardware or by another card's DMA master, via **vme_dma_setup**()
- The return of those data structures and hardware to their original, pre-**vme_setup** state, via **vme_dma_cleanup()**

# VME-Class DMA Hardware

The Model 743 and 744 processors have a VME DMA engine, but the Model 742 and 747 systems do not. This DMA hardware can be used only for transfers between VME space and the VME-Class's physical memory.

The DMA controller arbitrates for the VME-Class system bus and acts as a master on that bus when running transfers to and from VME space. (The DMA controller's 64-byte FIFO keeps the system bus from locking up.)

The DMA software functions provided by **vme2** support two types of DMA, synchronous and asynchronous:

- **Synchronous**—Each DMA transfer request consists of a list of physically contiguous transfer addresses and counts, along with data concerning the type of transfer. After providing the DMA controller with this list, the **vme2** driver calls **sleep**() and waits for completion of the transfer. The DMA controller fetches the list, processes each list item by obtaining the VME bus and the internal VME-Class system bus, and then mastering the individual transfers. At the end of the list or upon failure (such as a VME bus error), the DMA controller interrupts, waking up **vme2**, which returns the results to the calling process.
- **Asynchronous**—One or more buffers may be queued for DMA transfer, but the process requesting the DMA is not put to sleep. Instead, it provides a callback function that is called at **spl** level 2 when the DMA completes. In the meantime, the process may continue, but must not disturb the contents of the buffer(s) queued for DMA.

## Model 743/744 DMA Cycle and Transfer Types

The host DMA controller can run D16 (2-byte), D32 (4-byte), and D64 (8-byte) data bus block transfer cycles in A32 and A24 space. It can also run most other non-block VME cycle types, such as D16 or D32 transfers, in A16, A24, and A32 space.

The 743/744 DMA controller does not support a mode for writing to or reading from a fixed address. Rather, each data transfer goes to or from a new, sequential address—i.e., the *from* and *to* addresses are auto-incremented.

### Asynchronous DMA on Model 743/744 Systems

The DMA controller runs each cycle with the requested address modifier and data transfer size. The above cycle restrictions apply.

Asynchronous DMA also requires:

- A driver-provided callback function to be called when the DMA completes, or repeated calls to **vme_dma_queue_status** to poll for completion, or conversion to synchronous behavior by a call to **vme_dma_wait_done**
- Cache-alignment (see “Cache Coherency Issues” on page 4-21)

### Synchronous Host DMA with Remote VME Cards

Normally, the DMA controller runs each cycle with the requested address modifier and data transfer size. However, VME cards that slave the host DMA controller are expected to be capable of slaving varying data transfer sizes and possibly non-block-transfer address modifiers (because such transfers can occur even if the DMA controller is set up to do block transfers).

Not only can non-block transfers occur, but smaller data widths (D08, D16) may be used, even when a larger data width is specified (D16, D32), when:

- The DMA controller does beginning alignment to get to a legal block address boundary or D16/D32 address alignment.
- The DMA controller does the final few bytes of data that are less than the block size or data width.
- A DMA transfer is set up with a size so small that doing a block transfer is of no advantage, (e.g., 4-byte D32 and 8-byte D64 blocks will run non-block) or the transfer size is smaller than the data width.

To handle a possible non-block transfer, a VME slave should be set up to respond to both the block and the equivalent, non-block, data address modifier. To handle the shift in data width, a slave should be able to handle D08 and D16 data widths, or the driver should ensure that buffer addresses and transfer sizes are compatible with the data width restrictions of the card.

The cycle restrictions on the host DMA controller apply as follows:

- Calls to **vme_dmacopy()** always use the DMA controller, and the cycle restrictions on the DMA controller apply.
- Calls to **vme_copy**() or the VME2_USER_COPY **ioctl** use the DMA controller if it can master the transfer, otherwise they handle the transfer under CPU control.

## Synchronous DMA the Easy Way with vme_copy

The **vme_copy** routine moves data between VME and kernel addresses, and between kernel addresses. **vme_copy** will fail if either or both addresses are in user space.

**vme_copy** is the easiest way to automatically use host DMA hardware if it's present, and you don't need to set up the address and data chains, and so on, as described in later sections in this chapter. (If DMA hardware is unavailable or inappropriate, **vme_copy** calls **vme_mod_copy** to do the actual transfer via a software copy.)

The **vme_copy** routine is defined as follows:

```
vme_copy (to_va, from_va, count, options)
    caddr_t to_va;
    caddr_t from_va;
    unsigned int count;
    int options;
```

The arguments to this function have the following significance:

- *from_va* and *to_va*—Virtual addresses representing the transfer source and destination, respectively. At least one must be a kernel RAM address and the other a kernel address or a VME address.
- *count*—The number of bytes to be transferred, regardless of data width.
- *options*—Flags used to:
  - select the data width of the transfer:
    - Bits 0-5 of the options parameter can specify an address modifier, but this will be ignored if VME_IGNORE_ADM specifies the use of the currently mapped address modifier.
    - If you specify VME_OPTIMAL, the *from_va* and *to_va* addresses do not have to be aligned. If you *also* set one of the width flags, transfers of that width will occur, except that shorter width transfers might occur for unaligned leftovers.

    - If you don't use VME_OPTIMAL, be sure to use VME_D16 and VME_D32 only when *from_va* and *to_va* are aligned to 16-bit or 32-bit boundaries and *count* is a multiple of 2 or 4, respectively.
    - VME_D08 imposes no alignment restrictions.
  - whether to try to use a DMA engine:
    - Use VME_CPU_COPY to prevent the use of the DMA hardware.
    - Use VME_FASTEST to specify that **vme_mod_copy** should be used if the DMA hardware is busy or unavailable, rather than waiting for it.
  - whether or not to do a block transfer:
    - VME_D16 and VME_D32 transfers run as repetitive single transfers if the BLOCK options bit is not specified.
    - The BLOCK option is ignored (and **vme_mod_copy** is used) if block transfers are not supported by the underlying hardware. Otherwise, the address modifier is determined by the VME address space of the VME virtual address parameter (*from_va* or *to_va*).

**vme_copy** returns 0 on success, <0 for an error, or >0 to indicate the residual count of bytes not transferred in the case of an error (such as a VME bus error), in addition to the error values described below.

- ILLEGAL_OPTION—for any of the following combinations of options:
  - If BLOCK and VME_FASTEST are not both set when VME_IGNORE_ADM isn't, to allow fallback to a specified width when DMA hardware is unavailable.
  - Multiple data widths.
  - VME_D64 without BLOCK.
  - VME_CPU_COPY and VME_D64.
  - BLOCK and an A16 address modifier for the Model 743.
  - VME_D64 for a Model 742/747 system.
  - BLOCK and VME_CPU_COPY both for hosts that support block transfers.
- INT_ALLOC_FAILED—Memory can't be allocated to set up the DMA chain or add an entry to the wait queue.
- ILLEGAL_CONTEXT— **vme_copy** was called from interrupt context.

- ILLEGAL_CALL_VALUE—An illegal parameter or combination, e.g., neither *from_va* nor *to_va* is in kernel space, neither *from_va* nor *to_va* is in VME space, or both *from_va* and *to_va* are in I/O space.
- ILLEGAL_CALL_VALUE—Without host DMA hardware, if either *from_va* or *to_va* violates the alignment requirements imposed by the width options flag, and VME_OPTIMAL isn't set, no bytes will be transferred. With DMA, these conditions will likely cause a bus error.
- ILLEGAL_DMA_ADM—An invalid address modifier is specified.
- BUFLET_ALLOC_FAILED—Memory can't be allocated to handle a (possible) shared cache line at the beginning or end of the transfer.
- NO_D08_BLOCK—S743/744 hosts don't support D08 block transfers.

# Synchronous Local Host DMA

If you want more control than vme_copy gives you over the VME-Class's host DMA controller and various transfer characteristics, you can separately:

- Set up a chain of DMA requests
- Start DMA
- Perform the DMA transfer using vme_dmacopy()
- Clean up when the DMA completes

## Setting up a Chain for DMA Transfers

**vme_dma_setup()** called in HOST_DMA mode sets up for a VME bus-master, DMA transfer using the VME-class's DMA controller. It:

- Allocates memory for the chain
- Builds an address/count chain of VME-Class system bus physical addresses that correspond to the VME-Class's host's virtual buffer described in the *dma_parms* struct
- Requests I/O map entries

The **vme_dma_setup** routine relies on an *isc* data structure that has been filled in by your driver before this call, possibly in your *driver*_**open**, or *driver*_**attach**, by calling **vme_set_address_space()**. The syntax is:

```
vme_dma_setup (isc, dma_params)
    struct isc_table_type *isc;
    struct dma_parms *dma_params;
```

The *dma_parms* struct also contains previously-specified information about the requested DMA transaction. Its pertinent fields include:

- **dma_options**—The type of DMA for this transfer as follows:
  - VME_HOST_DMA—MUST BE SET for host DMA mode
  - DMA_READ, DMA_WRITE—Direction is from or to the card, respectively

- **flags**—Typically set to 0, but can include:
  - NO_CHECK—(Not recommended.) For host DMA, No maximum transfer length checking is to be performed by vme_dma_setup. Clear this flag forerror checking (the default).
  - NO_ALLOC_CHAIN—If set, the driver must allocate the chain pointed to by chain_ptr. If this flag is cleared, the setup routine will allocate the chain to hold the address/count pairs. If the driver allocates the chain, it must allocate the worst case size, which is:

    ((transfer count / NBPG) + 5) * sizeof(struct addr_chain_type)

    If the setup routine allocates the chain, it will allocate the size that is actually used to hold the transfer information, rather than the worst case.

***NOTE:*** Setting the NO_ALLOC_CHAIN flag without allocating sufficient space results in a trap 15 panic.

- **chain_count** and **chain_index**—Determine if there are more chains to transfer. Don't modify the **chain_count** field. For sample code, see Appendix D.

## Synchronous VME to Kernel RAM DMA with vme_dmacopy

A driver normally calls **vme_dmacopy** from its *driver*_**strategy** routine, after calling **vme_dma_setup** to set up the chain of DMA requests.

The **vme_dmacopy** routine moves data between VME and kernel RAM addresses using VME-class host DMA hardware. It can not be used to transfer data between two I/O spaces (for example, EISA to VME).

Call **vme_dmacopy** as follows:

```
vme_dmacopy(dma_parms, vme_addr, vme_adm, options)
        struct dma_parms *dma_parms;
        caddr_t vme_addr;
        int vme_adm, options;
```

The parameters have the following significance:

- *dma_parms*—The structure passed to and filled in by **vme_dma_setup**.
  - *dma_parms*->**drv_routine**—Specifies a function or subroutine to call if the DMA hardware is currently busy.
  - *dma_parms*->**dma_options**—Should set the VME_HOST_DMA flag. This field also determines the transfer direction, as in **vme_dma_setup**.
- *vme_addr*—The VME bus physical address for the transfer, not necessarily previously mapped. It must be the beginning of a contiguous region: chaining of VME buffers is not supported. **Vme_dmacopy** uses the PA-RISC address information returned by **vme_dma_setup**.
- *vme_adm*—The address modifier used in the transfer (must be supported by the DMA hardware).
- *options*—Flags that select the transfer type and data width: block (which must be specified), VME_D16, VME_D32, and VME_D64. (VME_D08 isn't supported on 743/744 hosts, and BLOCK isn't supported on 742/747 hosts.)

The **vme_dmacopy** routine returns 0 on success, < 0 for an error, or > 0 indicating the residual number of bytes not transferred in the case of an error (such as a VME bus error). It also returns the following errors:

- ILLEGAL_OPTION—Specified options bit isn't supported on the host.
- NO_DMA_HARDWARE—The host does not have DMA hardware. The S742 and S747 do not have DMA hardware.
- INT_ALLOC_FAILED—Memory cannot be allocated to set up the DMA chain or to add an entry to the wait queue.
- ILLEGAL_CONTEXT— **vme_dmacopy** was called from interrupt context.
- ILLEGAL_CALL_VALUE—One of the parameters is illegal.
- ILLEGAL_DMA_ADM—An invalid address modifier is specified.
- NO_D08_BLOCK—The host doesn't support D08 block transfers, but the VME_D08 option bit is specified with a block address modifier.

### Cleaning up after DMA

A VME driver calls the **vme_dma_cleanup** routine from its DMA completion code (**isr** or **dma_done**) to clean up after a DMA transfer.

Cleaning up is necessary because the kernel allocates special buffer pages when end portions of the transfer are not cache aligned, and because host-to-slave mapping pages that were allocated by **vme_dma_setup** (if VME_USE_IOMAP was set) need to be deallocated. The **vme_dma_cleanup** routine checks for abnormal DMA termination, transfers the non-cache-aligned bytes at the beginning and end of a transfer, and frees the system resources used for the transfer.

The **vme_dma_cleanup** routine is called as follows:

```
vme_dma_cleanup (isc, dma_parameters)
    isc_table_type *isc;
    dma_parms *dma_parameters;
```

This function always returns a 0.

# Asynchronous Local Host DMA

To use the VME-Class host DMA engine to transfer data asynchronously:

- Queue one or more buffers for transfer (do not call **vme_dma_setup**)
- Provide a callback function to be called when the DMA completes
- Initiate the transfer
- Continue processing (but without disturbing the buffers being transferred until the transfer is complete
- Clean up from the transfer within your provided callback routine (called at **spl** 2)

## Queuing Transfer Buffers

The **vme_dma_queue**() kernel routine queues a buffer for asynchronous DMA transfer using the local host's DMA engine. It automatically builds the necessary internal structures to transfer the data, much as the **vme_dma_setup()** function allocates memory and builds an address/count chain.

The **vme_dma_queue_polybuf**() kernel routine works the same way, except that the buffer is replaced by a polybuf. See **vme_polybuf** on page A-20**.**

***NOTE:*** These function do not check for cache alignment. You must make sure that each buffer you specify is cache-alignment safe at both ends.

```
void vme_dma_queue (who, buffer, size, vme_address,
        modifier, read_write, priority, callback, arg)
    caddr_t who;
    addr_t  buffer;
    int     size;
    caddr_t vme_address;
    int     modifier, read_write, priority;
    ROUTINE callback;
    caddr_t arg;
```

The parameters have the following significance:

- *who*—A pointer to the ***proc*** structure (or to any kernel driver address) that identifies this process

- *buffer*—The buffer or polybuf to be queued
- *size*—The size of the buffer or polybuf (limited to MAXPHYS). If for a polybuf, the value 0 is interpreted as "send as much as possible."
- *vme_address*—The starting VME bus address associated with the buffer
- *modifier*—Address modifier value: VME_A16, VME_A24, VME_A32, or any modifier listed in the table "Symbolic Names for Memory Address Modifiers" on page 2-10. (Use D64_ACCESS or BLK_ACCESS if your target card allows for fastest data transfers.)
- *read_write*—The direction of the transfer: DMA_READ or DMA_WRITE
- *priority*—Specifies queue priority: DMA_PRIORITY_QUEUE or DMA_NORMAL
- *callback*—Who to call back when the transfer is done
- *arg*—An argument to pass to your callback function when the DMA completes

It returns:

- NO_DMA_HARDWARE if the local host is not a 743/744
- ILLEGAL_CALL_VALUE if *size* is less than or equal to zero
- INT_ALLOC_FAILED if there was an internal **malloc** problem

## Providing a Callback Function

Your callback function should take whatever action is appropriate, such as freeing any allocated space. For example:

```
vmedrv_dma_complete (who, buffer, result, count, arg)
    int who;
    caddr_t *buffer;
    int result, count;
    void *arg;
{
    int next;
    struct vmedrv_transfer {
        caddr_t *vme_buffer;
        int     vme_am;
        int     buffer_size;
        int     total;
    } *transfer = (struct vmedrv_transfer) arg;
    if (result == 0)
```

```
        ; /* process error during DMA transfer & exit */
    /* copy count bytes from buffer to ... */
    transfer->total -= count;
    transfer->vme_buffer += count;
    if (total == 0)
        ; /* all done -- clean up and exit */
    else {
        next = total;
        if (next > transfer->buffer_size)
            next = transfer->buffer_size;
        if (vme_dma_queue (who, buffer, next,
                transfer->vme_buffer, transfer->vme_am,
                DMA_READ, DMA_NORMAL,
                &vmedrv_dma_complete, transfer) < 0)
            ; /* process error setting up DMA transfer */
    }
}
```

## Checking and Abnormally Terminating the Transfer

The following section describes functions that provide status checking and queue-related actions.

### Sleeping until the Queue Empties

The **vme_dma_wait_done**() routine finds the queue for the specified process. If the queue isn't empty, the routine sleeps until it is:

```
void vme_dma_wait_done (who)
    caddr_t who;
```

- *who*—A pointer to the ***proc*** structure (or to any kernel driver address) that identifies the process whose queue is to be checked

Calling **vme_dma_wait_done** after one or more calls to **vme_dma_queue** essentially turns the asynchronous call into a synchronous one.

### Checking DMA Status

The **vme_dma_status**() routine returns information on whether a requested DMA transfer is running, queued, or finished:

```
void vme_dma_status (who, buffer)
    caddr_t who;
    caddr_tbuffer;
```

- *who*—A pointer to identify the process whose transfer status is to be checked
- *buffer*—The buffer whose status is to be checked

**vme_dma_status** returns the following values:

- DMA_STATUS_QUEUED—The requested *who* and *buffer* are still in the queue
- DMA_STATUS_RUNNING—The transfer for the requested *buffer* is in process
- DMA_STATUS_UNKNOWN—The buffer and process could not be identified (this value is returned once DMA has completed)

If **vme_dma_queue** is called with the callback *arg* set to NULL, you can use **vme_dma_status** to poll for completion.

### Terminating DMA

The **vme_dma_nevermind**() kernel routine can be called when a process is going away. It removes any buffers from the queue if they haven't already been transferred:

```
void vme_dma_nevermind (who)
    caddr_t who;
```

- *who*—A pointer to the *proc* structure (or to any kernel driver address) that identifies the process on whose behalf the asynchronous DMA was queued

## Remote DMA Controllers

**Remote DMA controllers** run as masters on the VME bus and provide their own hardware to DMA the memory on your VME-Class host. They can be either simple DMA devices that transfer only a single, contiguous local bus address range without direct driver intervention, or smarter devices that can generate multiple ranges of local bus addresses and without driver intervention.

To use the remote DMA controller:

1. Set up a *dma_parms* struct.
2. Call **vme_dma_setup** in REMOTE_DMA mode to set up Series 700 mapping hardware on local board.
3. Call *remote_dma_setup*, which sets up the registers for a single chain and then starts the card's DMA controller.
4. When the card's DMA controller is done, it interrupts, and the *remote_isr* routine (in your driver) runs.
5. From the *remote_isr_routine* call **vme_dma_cleanup** to release Series 700 mapping resources.

The **vme_dma_setup** routine relies on an isc data structure that has been filled in by your driver before this call, possibly in your driver_open, or driver_attach, by calling vme_set_address_space(). The syntax is:

```
vme_dma_setup (isc, dma_params)
    struct isc_table_type *isc;
    struct dma_parms *dma_params;
```

The dma_parms struct also contains previously-specified information about the requested DMA transaction. Its pertinent fields include:

- **dma_options**—The type of DMA for this transfer as follows:
  - VME_A32_DMA, VME_A24_DMA—32-bit or 24-bit addressable card (mutually exclusive)
  - VME_HOST_DMA—MUST BE CLEARED for remote DMA mode
  - VME_USE_IOMAP—Use the board computer's VME slave I/O map hardware

otherwise the VME slave direct map hardware is used

- DMA_READ, DMA_WRITE—Direction is from or to the card, respectively

- **flags**—Typically set to 0, but can include:
  - NO_CHECK—(Not recommended.) No error checking is to be performed by **vme_dma_setup**. For remote dma, no maximum transfer length checking is done. In addition, the physical memory address are checked for compatability with the type of mapping selected. Clear this flag for error checking (the default).
  - NO_ALLOC_CHAIN—If set, the driver must allocate the chain pointed to by chain_ptr. If this flag is cleared, the setup routine will allocate the chain to hold the address/count pairs. If the driver allocates the chain, it must allocate the worst case size, which is:

    ((transfer count / NBPG) + 5) * sizeof(struct addr_chain_type)

    If the setup routine allocates the chain, it will allocate the size that is actually used to hold the transfer information, rather than the worst case.

NOTE: Setting the NO_ALLOC_CHAIN flag without allocating sufficient space results in a trap 15 panic.

- **chain_count** and **chain_index**—Determine if there are more chains to transfer. Don't modify the chain_count field. For sample code, see Appendix D.
- **drv_routine**—Which routine to call in the event the VME_USE_IOMAP option is selected and slave mapping resources cannot be obtained
  - If the driver calling **vme_dma_setup** is a context-switch-driven driver, it should sleep upon receiving an error return for RESOURCE_UNAVAILABLE: the function specified in **drv_routine** is issued a call to wake up, and given *drv_arg* as its argument, when the system frees DMA resources.
  - If the driver is an interrupt-driven driver, it advances its state and exits: the function specified in **drv_routine** re-invokes the driver to try again.
  - If a driver does not wish to be notified when resources become available (if they were unavailable at DMA setup time), it sets **drv_routine** to null.

If the system resources (map hardware) needed for the DMA are unavailable, **vme_dma_setup** returns resource_unavailable. See Appendix B for other error conditions and returns.

### Remote DMA Setup Routine

The following hypothetical example routine sets up the DMA registers and starts the card's DMA controller. (The *remote_card_register_set*, for instance, is a purely hypothetical structure that might be provided or required by the card's software.)

```
remote_dma_setup(isc)
struct isc_table_type *isc;
{
    struct buf *bp = isc->owner;
    int index = dma_parms->chain_index;
     if (bp->b_flags & B_READ) {
        /* Perform read-specific card set up */
    }
    else {
        /* Perform write-specific card set up */
    }
/* set up dma address and transfer count */
    card_ptr->dma_addr = (unsigned long)
                dma_parms->chain_ptr[index].phys_addr;
     card_ptr->tfr_count = (short)
                dma_parms->chain_ptr[index].count;
    dma_parms->chain_index++;
}
```

### Remote ISR Routine

The following routine provides the interrupt service for the remote DMA controller and calls your driver's *remote_dma_setup* function if there is at least one more chain to be run. The controller then does the chain and re-interrupts, and the cycle continues until there are no more chains.

```
void remote_isr (isc, arg)
struct isc_table_type *isc;
int arg;
{
    int pri, i;
    struct dma_parms *parms = isc->dma_parms;
    struct buf *bp = isc->owner;
    int idx = parms->chain_count;
```

```
    if (regs->int status & BERR){ /* clear interrupt */
        for(i=0; i< parms->chain_count - idx; i++)
            bp->b_resid +=
                    parms->chain_ptr[idx-1].count;
            bp->b_error = EIO;
            bp->b_fLags |= B_ERROR;
            pri = spl6();
        isc->owner = NULL;
        splx(pri);
        vme_dma_cleanup (isc, isc->dma_parms);
        io_free (parms, sizeof(struct dma_parms));
        iodone(bp);
        wakeup(isc);
        return;
    }
    if (idx != parms >chain_count) {
        remote_dma_setup(isc);
        return
    }
    vme_dma_cleanup(isc, isc->dma_parms);
    io_free (parms, sizeof (struct dma_parms));
    pri=spl6();
    isc->owner = NULL
    splx(pri);
    wakeup(isc);
    iodone(bp);
}
```

### **Remote DMA without** vme_dma_setup

If you choose to use your card's DMA master to do DMA without using **vme_dma_setup** and **vme_dma_cleanup**, your driver needs to set everything, maintain cache coherency, and clean up. In this case, use either **map_mem_to_bus** or one of the other, similar, mapping functions that set up the slave map entries, or use the direct mapper and call **kvtophys** to obtain the physical address of each VME-Class RAM page in the buffer so that the host hardware will slave the card's DMA master properly.

Using **map_mem_to_bus** in this manner is not appropriate for larger transfers (greater than 8 Kbytes) because you are dedicating a scarce resource (the 1 Mbyte slave mapper) to your driver. In this case, you can use **vme_map_largest_to_bus** to acquire VME shared memory in whatever size chunks are available, and transfer your data in stages.

## Cache Coherency Issues

VME bus master access to VME-Class RAM occurs without the knowledge of the VME-Class processor cache hardware. HP-UX provides two routines to validate the processor cache before and after VME bus master access: **io_flushcache**. and **io_purgecache**

### Flushing Cache

The **dma_sync** routine, as used below, writes (flushes) the current contents of cache to memory, and ensures that the next CPU access causes the cache to be loaded from memory.

**dma_sync** is defined as follows:

```
dma_sync (address_type, virtual_addr, size, hints)
    int address_type;
    caddr_t virtual_addr;
    int size;
    int hints;
```

A typical use of this call would be during the setup for a VME bus master to do a read from a VME-Class RAM. The call to **dma_sync** updates the RAM with the latest contents of cache prior to the VME bus master accessing it.

The following section of code shows the setup for the VME bus master:

```
my_strategy(buf)
struct buf *bp;
{
. . .
    /* code to set up VME bus master */
    /* address chains, counts etc.  */
    cp->dma_direction = TO_VME;
    dma_sync (KERNELSPACE, MY_BUF, MY_SIZE,
        IO_WRITE|IO_CONDITIONAL|IO_MODIFIED);
    cp->dma_trigger = START_DMA;
. . .
}
```

## Purging Cache

The **dma_sync** routine, as used below, ensures that the next CPU access will cause the cache to load from memory, discarding the previous contents in the cache.

To purge cache, call **dma_sync** as follows:

```
dma_sync (KERNELSPACE, virtual_addr, size,
        IO_READ|IO_CONDITIONAL|IO_ACCESSED);
```

A typical use of **dma_sync** would be during the clean-up for a VME bus master, after doing a write to VME-Class RAM. The call to **dma_sync** invalidates the cache lines, so that the next VME-Class processor access will load the correct RAM contents to cache.

```
my_isr(isc,arg)
struct isc_table_type *isc;
int arg;
. . .
    /* This interrupt was caused by a card */
    /* that interrupted after mastering the VME bus */

    if (arg == FROM_VME)
        dma_sync (KERNELSPACE, MY_BUF, MY_SIZE,
            IO_READ|IO_CONDITIONAL|IO_ACCESSED);
. . .
```

## Shared Cache Line Problems

If you write transfer routines for VME bus masters that do not use **vme_dma_setup** and **vme_dma_cleanup**, you must use caution if your transfers are not cache-aligned. **Cache alignment** means that your current chain begins and ends on a cache line boundary (32 bytes).

There is no indication if another process is using the remaining portion of a cache line at the end of an unaligned transfer. The other process could invalidate the cache line, causing a flush or purge during the time that your driver is accessing it. Even if the rest of the cache line is owned by your driver, writing a byte in the same cache line as, say, a DMA read, will cause the entire contents of the cache (all 32 bytes) to be written to memory, overwriting (and hence losing) the data just DMA'd into this cache line.

The **vme2** synchronous DMA calls avoid this problem by setting up a separate cache line  buffer for possible shared cache lines at the beginning or end of the transfer. The call to **vme_dma_cleanup** handles transferring the contents of the separate cache line buffers to the real cache lines.

# 5

# User-Level VME Access

This chapter describes the kernel **vme2** services available for accessing VME space from a user-level process.

All of the commands require that the **vme2** driver be opened. You will need a character device special file named **/dev/vme2** with a major number of **44** and a minor number of **0**. This should be automatically installed when VME Services are installed.

***NOTE:*** You do not need to write a driver in order to access VME address space from a user-level process.

### User-Level Access Advantages

- Good for fast prototyping to test hardware.
- No rebuild/reboot with every change (faster debugging turn-around time).
- You can use the **xdb** and **dde** debugging tools.
- You can use all routines in the user libraries.
- You do not need knowledge of operating system internals.

### User-Level Disadvantages

User-level drivers can cause a device to interrupt, but have no mechanism for handling the interrupt (this can cause system hangs or kernel panics). Conversely, the system can interrupt user-level driver execution.

## The ioctl Routine

The **vme2** driver incorporates several VME-access commands that are known as **ioctl** (I/O Control) commands. These commands are all issued as parameters in a call to the **ioctl**(2) routine.

The **ioctl** routine has the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                     /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

**Table 5-1 ioctl Commands with Structs**

| Command | Struct |
|---|---|
| VME2_ENABLE_IRQ | *vme2_int_control* |
| VME2_IO_TESTR & VME2_IO_TESTW | *vme2_io_testx* |
| VME2_LOCMON_GRAB & VME2_FIFO_GRAB<br>VME2_LOCMON_POLL & VME2_FIFO_POLL<br>VME2_FIFO_READ<br>VME2_LOCMON_RELEASE & VME2_FIFO_RELEASE | *vme2_lm_fifo_setup* |
| VME2_MAP_ADDR & VME2_UNMAP_ADDR | *vme2_map_addr* |
| VME2_REG_READ & VME2_REG_WRITE | *vme2_io_regx* |
| VME2_USER_COPY | *vme2_copy_addr* |
| VME2_CPU_NUMBER | none |

See Appendix A for definitions of these structures.

## Probe Commands

The memory probe **ioctl** commands are as follows:

- VME2_IO_TESTR
- VME2_IO_TESTW

They have the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                     /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

They probe VME space to ascertain that a VME slave is present. They are defined in **/usr/conf/wsio/vme2.h**. They use the VME adapter hardware to trap VME bus errors and return the success or failure of the probe in the **access_result** field of the *vme2_io_testx* structure (see page A-24).

The actual data written or read is internal to the **vme2** driver. In the case of a write, the value is 0xFF for a byte access, 0xFFFF for a word access, and 0xFFFFFFFF for a long access.

These commands do not require that you previously map in the requested address. Internal to the **vme2** driver, the necessary commands are made to map a single page (4 Kbytes), **vme_testr()** or **vme_testw()**, unmap the single page, and return the result to the user.

Typically, you would probe at the start address and end address of your VME interface card, and then map the actual space for use. This allows you to handle VME bus errors if the card is not present, or if you are unsure of the actual card configuration (for example, if it has optional amounts of RAM).

An example of using the VME2_IO_TESTR **ioctl** command follows.

***NOTE:*** The example code is presented in several sections, each of which is appropriate to the particular **ioctl** command being discussed—therefore you'll find **#include** statements at the beginning, and structure declarations at the beginning of **main()** that pertain to structures used in later sections.

```
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <sys/ioctl.h>
#include "/usr/conf/wsio/vme2.h"
#include <sys/io.h>
#include <stdio.h>
#include <fcntl.h>

#define MYCARDADDR 0x100000
#define EXPECTED_READ_VALUE 0x10
#define MYCARDSIZE 0x4000
#define MYCARD_LEVEL 3

main()
{
    int fd;
    struct vme2_io_testx tx;
    struct vme2_io_regx rx;
    struct vme2_map_addr ma;
```

```
    struct vme2_copy_addr ca;
    caddr_t card_memory;
    unsigned char *buffer;
    if ((fd = open("/dev/vme2", O_RDWR)) < 0) {
        perror ("ERROR: VME Bus Adapter Open FAILED");
        if (errno == ENODEV)
             printf ("ENODEV. Is driver in kernel? \
        (Is vme2 in /stand/system?)\n");
        exit(errno);
    }
/* probe hardware with vme2_io_testr */
    tx.card_type = VME_A24;
    tx.vme_addr = MYCARDADDR;
    tx.width = BYTE_WIDE;
    if (ioctl (fd, VME2_IO_TESTR, &tx)) {
        perror("VME2_IO_TESTR failed");
        exit(errno);
    }
    if (tx.error) {
        fprintf (stderr, "VME2_IO_TESTR failed:  \
        internal error %d\n", tx.error);
        exit(1);
    }
    if (tx.access_result <= 0) {
        fprintf (stderr, "VME2_IO_TESTR failed to find\
        card\n");
        exit(2);
    }
    printf("VME2_IO_TESTR succeeded\n");
. . .
}
```

## Register Access Commands

The register access **ioctl** commands are similar to the probe commands with the added feature that the user can specify the VME address modifier to be used to access VME space. The user can also specify the data written and receive the data read. The register access commands are:

- VME2_REG_READ
- VME2_REG_WRITE

They have the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                    /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

They are defined in **/usr/conf/wsio/vme2.h** and use the *vme2_io_regx* struct (see page A-22).

An example of using the VME2_REG_READ command follows (the code is a continuation of the previous code):

```
/* Alternatively, probe hardware with vme2_reg_read */
rx.vme_addr_mod = STD_NP_DATA_ACCESS;
rx.card_type = VME_A24;
rx.vme_addr = MYCARDADDR;
rx.width = BYTE_WIDE;
if (ioctl (fd, VME2_REG_READ, &rx)) {
    error("VME2_REG_READ failed");
    exit(errno);
}
if (rx.error) {
    fprintf (stderr, "VME2_REG_READ failed \
        internal error %d\n", tx.error);
    exit(1);
}
if (rx.access_result <= 0) {
    fprintf (stderr, "VME2_REG_READ failed\n");
    exit(2);
}
if (rx.value != EXPECTED_READ_VALUE) {
    fprintf (stderr, "VME2_REG_READ failed, incorrect\
        value returned.");
    fprintf (stderr, "Expected %x, read %x\n",\
        EXPECTED_READ_VALUE, rx.value);
    exit(3);
}
printf("VME2_REG_READ succeeded\n");
```

## Map and Unmap Commands

The map and unmap **ioctl** commands map VME space into a user process. They are defined in **/usr/conf/wsio/vme2.h**:

- VME2_MAP_ADDR
- VME2_UNMAP_ADDR

They have the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                     /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

You need to make probe commands prior to using these commands to avoid a possible SIGBUS error due to VME bus errors.

These commands return a user pointer to the requested VME address space. They fail if a contiguous space equal to the size requested is not available (unused) in the VME hardware mapper, or if the request is for a non-page-aligned address.

Always pair map and unmap commands so that the maximum mapper space is available for user processes. You *must* use the structure obtained in the map command when making the unmap command. You may want to trap signals so that you can clean up (unmap) on exit.

These commands use the *vme2_map_addr* struct (see page A-23)and effectively map I/O space into a user process. Be aware that this is *not* a mapping to local memory space. Do not rely on common **libc** function commands that access memory space to work properly with I/O space.

An example of using these commands follows (the code is a continuation of the previous code):

```
ma.card_type = VME_A24;
ma.size = MYCARDSIZE;
ma.vme_addr = MYCARDADDR;
if (ioctl (fd, VME2_MAP_ADDR, &ma)) {
    perror("VME2_MAP_ADDR failed");
    exit(errno);
```

```
}
if (ma.error) {
    printf("VME2_MAP_ADDR failed.\n");
    exit(ENOMEM);
}
card_memory = ma.user_addr;
```

See the following section, for the code to read the contents of VME memory into the buffer, as represented by the ellipses below:

```
. . .   /* read contents of VME RAM into buffer */
free(buffer);
finish_up:
/* unmap card memory */
if (ioctl (fd, VME2_UNMAP_ADDR, &ma))
    perror("VME2_UNMAP_ADDR failed");
close(fd);
exit(exit_value);
}
```

## The User Copy Command

The user copy command copies data between a user buffer and VME space. The methods of transfer include DMA and depend on the options field and what underlying hardware is available. This command provides user access to the **vme_copy** kernel routine:

- VME2_USER_COPY

They have the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                     /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

It is defined in **/usr/conf/wsio/vme2.h** and uses the *vme2_copy_addr* structure (see page A-21).

An example of using VME2_USER_COPY follows (the code was referenced as ellipses in the code shown above):

```
/* read contents of VME RAM into buffer */

    exit_value = 0;
    if (! (buffer=(unsigned char*)malloc(MYCARDSIZE))) {
        error("malloc failed");
        exit_value = errno;
        goto finish_up;
    }
    ca.to_va = buffer;
    ca.from_va = card_memory;
    ca.count = MYCARDSIZE;
    ca.options = VME_D32|VME_OPTIMAL|STD_NP_DATA_ACCESS;

    if (ioctl (fd, VME2_USER_COPY, &ca)) {
        perror("data copy failed");
        exit_value = errno;
    }
```

Since VME2_USER_COPY uses **vme_copy**, see “Synchronous DMA the Easy Way with vme_copy” on page 4-5 for a description of the flags such as VME_OPTIMAL.

## The Enable IRQ Command

The enable IRQ **ioctl** command enables a VME interrupt level. It is intended for systems that use a **vme_panic_isr_hook** routine to enable interrupts again, after a problem that caused an unidentified status ID is solved:

- VME2_ENABLE_IRQ

It has the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                    /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

It is defined in **/usr/conf/wsio/vme2.h** and use the *vme2_int_controil*struct (see page A-22).

It returns 0 on success, or -1 if the requested interrupt level could not be enabled.

Note that **vme_panic_isr_hook** is by default NULL. Some kernel driver must provide a function to set this pointer to allow VME Services to continue after an unidentified status ID is received in an interrupt.

## Location Monitor and FIFO Commands

The Location Monitor provides an interrupt mechanism that can alert a user process when a particular range of addresses has been read from or written into. It passes the user process a pointer to a function that can be executed to react to this information.

They have the following signature:

```
ioctl (fd, command, io_struct)
    int fd;                     /* file descriptor */
    int command;/* IO control command */
    struct structType *io_struct; /* per command */
```

The FIFO is a hardware register on 743/748 systems. Whenever a FIFO is written into, an interrupt occurs on the HP PA-RISC CPU. If the FIFO is filled, no additional data is permitted, and an overflow flag is set.

The primary differences between the FIFO and the Location Monitor are:

- The Location Monitor reports both reads and writes.
- The FIFO's address/register area contains information in the first two bytes.

Both the Location Monitor and FIFO use the *vme2_lm_fifo_setup* struct (see Appendix A).

## Location Monitor Commands

See Appendix B for more information on the functions that are related to the Location Monitor. Essentially, the user-level commands are as follows:

- VME2_LOCMON_GRAB—Locks the Location Monitor for exclusive use, thereby avoiding interference from other processes. Your *vme2_lm_fifo_setup* struct specifies the address, address modifier, the size, and a routine to be called when an interrupt is generated because the monitored area has been written into or read from.
- VME2_LOCMON_POLL—Polls the Location Monitor to find out if any interrupts have occurred, and if so, how many.
- VME2_LOCMON_RELEASE—Unlocks the exclusive grab that this process has on the Location Monitor.

## FIFO Functions

See Appendix B for more information on the functions that are related to the FIFO. Essentially, the FIFO **ioctl** commands parallel the Location Monitor functions and are as follows:

- VME2_FIFO_GRAB—Locks the FIFO for exclusive use, thereby avoiding interference from other processes. Your *vme2_lm_fifo_setup* struct specifies the address,

address modifier, the size, and a routine to be called when an interrupt is generated because the FIFO has been written into.

- VME2_FIFO_POLL—Polls the FIFO to find out if any interrupts have occurred, and if so, how many.
- VME2_FIFO_READ—Reads the contents of the FIFO register.
- VME2_FIFO_RELEASE—Unlocks the exclusive grab that this process has on the FIFO.

## The CPU Number Command

The VME2_CPU_NUMBER **ioctl** returns the CPU number (0 to 31) that has been previously set for the local host by **vme_config** (see “Processor Records” on page 8-7), or from the firmware menu while booting. If the CPU number is not set, 31 is returned.

# 6

# Writing Driver Entry Points

This chapter gives specific details about writing those routines that are system driver entry points (through one of the device switch tables). These routines include: *driver*_**attach**, *driver*_**install**, *driver*_**open**, *driver*_**read**, *driver*_**write**, *driver*_**ioctl**, *driver*_**strategy**, and *driver*_**select**.

This chapter describes how to write all routines in a device driver except for DMA and interrupt service routines. See Chapter 4 (“Direct Memory Access”) for help on writing additional functions such as *driver*_**dma_setup**, *driver*_**dma_start**, and *driver*_**isr** that work with direct memory access and interrupts**.**

Write the routines according to your type of driver—character device or block device:

- **Character device driver**—Read the sections for writing open, close, read, write, ioctl, and select routines, and the strategy routine section.
- **Block device driver**—Read the sections for writing open, close, and strategy routines.

## Attach Routines

During initialization, each VME driver installed in the kernel specifies an install function named *driver*_**install**. This install function is the only call by the kernel to your driver that is guaranteed to be made. It is responsible for calling:

- **vme_set_attach_function**—To register the driver attach function
- **wsio_install_driver**—To register the WSIO (Workstation I/O) data structures that specify the functions and configuration data associated with the driver

For example:

```
int driver_install()
{
vme_set_attach_function(driver_STRING,driver_attach);
return (wsio_install_driver (& wsio_drv_info));
}
```

(See “wsio_drv_info Structure” on page 6-4 for more information on the *wsio_drv_info* struct.)

The attach function itself is then called later during initialization. For example:

```
driver_attach (id, isc)
int id;
struct isc_table_type *isc;
{
    if (id != (int)driverSTRING)) /* is it ours? */
            return; /* nope! */
    VME_INIT_IF_INFO (isc, D08_IRQ_TYPE, VME_COMPT);
    driver_Card->isc = isc; /* save the isc */

/* link in the DMA interrupt service routine */
    if ((driver_Card->vector_number =
        vme_isrlink (isc, driver_isr, 0, 0, 0)) < 0)
    return;

/* map in the card registers */
    driver = (struct driver_regs *)
        map_mem_to_host (isc, board_addrs, board.size);
    if (driver == NULL)
        return;
    isc->card_ptr = (int) driver;

/* test if card is there */
     if(vme_testr(isc,(caddr_t)driver,BYTE_WIDE,0)<=0){
        unmap_mem_from_host(isc,driver,driver_REG_SIZE);
        isc_claim (isc, NULL); /* ours, but no card */
        return;
    }
/* do card-specific initialization */
    . . .
    isc->dma_parms = (struct dma_parms *)
        io_malloc(sizeof(struct dma_parms), IOM_NOWAIT);
    if (isc->dma_parms == NULL) {
        unmap_mem_from_host(isc,driver,driver_REG_SIZE);
        return;
    }
    isc_claim (isc, &driver_wsio_info);
    driver_Not_Attached = FALSE;
}
```

### *wsio_drv_info* Structure

You need a *wsio_drv_info* structure that references the structs that need to be defined to specify the information as follows:

- *drv_info*—Device type and class names; major device number
- *drv_ops*—Contains all driver entry points
- *drv_data*—Miscellaneous fields such as expected type of hardware

An example of defining such a struct is shown below. See Appendix A for the declaration of *wsio_drv_info*.

```
static wsio_drv_info_t wsio_drv_info = {
    &driver_info,
    &driver_ops,
    &driver_data
};
```

This structure is referenced in the **isc_claim**() function, as shown at the end of the example *driver*_**attach**() function on the previous page.

### *drv_ops* Struct

The *drv_ops* structure contains all driver entry points, as suggested in the following abstract from its definition in a sample program (see Appendix A for its declaration).

```
static drv_ops_t driver_ops = {
    driver_open,        /* your driver's open funct */
    driver_close,       /* driver's close funct name */
. . .                   /* strategy, dump, psize */
    driver_read,        /* its read function */
    driver_write,       /* its write function */
    driver_ioctl,       /* its ioctl function */
. . .                   /* misc./reserved functions */
    C_ALLCLOSES         /* its flags */
}
```

The currently-defined values for flags are listed in Appendix A.

### *drv_info* Struct

The following example *drv_info* structure shows driver-specific fields for all 10.0-compliant drivers.

```
static struct drv_info_t drv_info = {
    “driver”,       /* its device type name */
    “vme”,          /* its device class name */
    DRV_CHAR | DRV_SAVE_CONF,             /* flags */
    -1,             /* major dev num if block type */
    50,             /* minor dev num if char type */
. . .
} drv_info_t;
```

The currently defined values for flags in the preceding *drv_info* structure are listed in Appendix A.

#### ***drv_data*** **Struct**

The following example *drv_data* structure shows driver-specific fields for all 10.0-compliant drivers.

```
static wsio_drv_data_t driver_data = {
    "driver", /* for matching probes with drivers */
    T_INTERFACE,    /* expected type of hardware */
    DRV_CONVERGED,  /* convergent (or not) */
    NULL,           /* minor number formatter */
    NULL            /* major number formatter
};
```

The flag for the expected type of hardware is either:

- T_INTERFACE —the driver controls an interface card
- T_DEVICE—the driver controls a hardware device

## Open Routines

When a user uses the **open**(2) system call on a device file (e.g. in **/dev**), **open** calls the corresponding *driver*_**open** routine via one of the two device switch tables (**bdevsw** or **cdevsw**). The *driver*_**open** routine performs any actions necessary to prepare a device for I/O.

Your device driver implements the type of open required by the device:

- exclusive open
- shared open
- multiple open

**Exclusive open** drivers allow only one process at a time to access the device. To enforce this, your driver can maintain a flag to indicate whether the device is currently open. If it is inappropriate for your device to be opened by more than one process simultaneously, the driver-open routine can return an error on any **open** performed when the device is already open.

**Shared open** or **multiple open** devices allow multiple processes to access the device simultaneously. All processes in a shared open implementation share a common set of global data structures. If one process modifies a value in a data structure, the value is modified for all processes that have opened the device. In a multiple open implementation, each process that has opened your device has its own copy of the data structures associated with the device. This allows each process to modify the values independently.

Magnetic tapes and printers are exclusive open devices so that the request of one process does not get interwoven with the request of another process. Terminals are typically shared open devices so users can communicate with each other (e.g., by using the **write**(1) command).

Decide which type of open routine you need and add code to your routine as indicated in the skeleton shown on page 6-8.

## Opening Devices

The *driver*_**open** entry point is for an **open** on a character or block device file. In general, it:

- Implements the type of open needed for the device.
- Returns an error if the device is an exclusive open device and this is not the first open.
- If multiple open, allocates a set of data structures for the open.
- Finds the device.
- Sets a flag indicating the device is open.
- Initializes any necessary data structures.
- Initializes hardware (puts the device in a known state).
- Clears the open flag and returns an error if an error occurs during initialization.
- Returns zero upon successful initialization.

Although the user specifies the pathname, flags, and mode parameters in the **open** system call, the *driver*_**open** routine does not receive this exact information. Instead, the kernel invokes it with the following parameters:

```
int driver_open(dev, flag)
    dev_t dev;
    int flag;
```

The following items describe the *driver*_**open** parameters:

- *dev*—The device number of the file to be opened. The *driver*_**open** routine can extract the major and minor number via a call to **major**().
- *flag*—A value corresponding to the **oflag** field in the **open** system call. The kernel implements the functions of **oflag** according to the description given in man pages for **fcntl**(5) and **open** before your driver is called. Therefore, your driver can usually ignore these flags. The kernel translates the O_xxxx values into corresponding Fxxxx values that pass into the *driver*_**open** routine. The flags of potential interest in writing your driver include: FREAD, FWRITE, FNDELAY, and FEXCL, which are used, for example, as follows:
  - A magnetic tape *driver*_**open** routine might check the value of FWRITE. If the tape is being opened for writing, and the tape is write protected, the tape open routine could return an error to the **open** system call.
  - A terminal device file might be opened where, if FNDELAY is set, the terminal

open routine does not wait for a hardware connection before returning. See **fcntl**(5), **open**(2), and **termio**(7).

The *driver*_**open** routine should return either a zero or an **errno**(2) value to the **open** system call, respectively indicating the success or failure of the open.

If the *driver*_**open** routine is successful, the kernel returns a file descriptor to the user. If unsuccessful, the kernel returns -1 to the user, and sets **errno** to the value returned by the *driver*_**open** routine. The user can check the return value and **errno** to determine if an error occurred.

The *driver*_**open** routine might return an error if the device:

- Is off-line
- Does not exist
- Was never configured into the system
- Failed initialization

The *driver*_**open** routine might also return an error if the device is an exclusive open device and the device is already open.

```
int skel_card_opened;
int skel_card_isc;
struct buf skel_buf;
skel_open(dev, flag)
dev_t dev;
int flag;
{
    register struct isc_table_type *isc;
    register char *card;
    register int selcode;
    skel_card_isc = isc->my_isc;
    card = (char *)(isc->card_ptr);
    /* card->registerX = ....  */
    /*  If needed, enforce exclusive open of card */
    if (skel_card_opened)
        return EBUSY;
    else skel_card_opened = 1;
    return(0);
}
```

# Character Device Read and Write Routines

When a user issues a **read**(2) or **write**(2) system call to a character device, the kernel fills out *uio* and *iovec* structures and calls the corresponding *driver*_**read** or *driver*_**write** routine in the **cdevsw** table, passing the *uio* structure as one of the parameters.

The *driver*_**read** and *driver*_**write** routines can process requests using **physio**() and **uiomove**(). The following sections discuss the *driver*_**read** and *driver*_**write** routines, as well as how to use **physio** and **uiomove**.

## Performing Input from a Device

The *driver*_**read** routine performs input from a device. It is called when a **read** or **readv**(2) is performed on a character device file. The kernel fills in the *uio* structure and then passes the structure and the device number to the *driver*_**read** routine.

You can implement the *driver*_**read** routine by either:

- Calling **physio** with the appropriate parameters, allowing a character-driver strategy routine to complete the request.
- Using **uiomove** to buffer the data and then complete the request. If you use **uiomove**, the *driver*_**read** routine performs the following functions: (1) Initializes any data structures; (2) Sets a flag indicating that I/O is in progress; (3) Makes the I/O request; (4) Waits for I/O to complete; (5) Calls **uiomove** to transfer the data from kernel buffer to user buffer; and (6) Sets the return value.

See “Using physio” on page 6-11 and “Using uiomove” on page 6-14 for additional information about using these two alternatives for implementing the *driver*_**read** routine.

While the user specifies a file descriptor, buffer, and number of bytes to be read in the **read** system call, the kernel invokes the *driver*_**read** routine with the following parameters:

```
driver_read (dev, uio)
    dev_t dev;
    struct uio *uio;
```

- *dev*—The device number of the associated device file. The *driver*_**read** routine

can extract the major and minor numbers from the device number. You should verify that this parameter is valid in your *driver*_**open** routine if you intend to access kernel data structures such as an *isc* pointer.

- *uio*—A pointer to a *uio* structure that contains information about the data being read, which the driver needs to input from the device.

If you choose to code this routine without using **uiomove** or **physio**, your code needs to do the following:

- Return 0 on success.
- Return an error code (from **/usr/conf/h/error.h**) on failure.
- Set *uio*-> **uio_resid** to the number of bytes remaining to transfer.

## Performing Output to a Device

The *driver*_**write** routine performs output to a device. It is called when a **write**(2) is performed on a character device file.

You can implement the *driver*_**write** routine by using one of either:

- **physio** and a *driver*_**strategy** routine
- **uiomove**

While the user specifies a file descriptor, buffer, and number of bytes to write in the **write** system call, the kernel invokes the *driver*_**write** routine with the following parameters:

```
driver_write (dev, uio)
    dev_t dev;
    struct uio *uio;
```

As with the *driver*_**read** routine, *dev* is the device number of the associated device file, and *uio* is a pointer to a *uio* structure. If you choose to code this routine without using **uiomove** or **physio**, the code will need to do one of the following:

- Return 0 on success.
- Return an error code failure.
- Set *uio*->**uio_resid** to the number of bytes remaining to transfer.

## **Using** physio

The read and write routines for a character device driver perform many of the same functions as the **physio** kernel routine. If you transfer data directly between a device and the user's buffer, you can use **physio** to perform many related functions.

With this type of transfer, the flow of control is sometimes difficult to grasp:

1. The user program calls your driver's *driver*_**write**() function.
2. The *driver*_**write** calls **physio**().
3. **Physio** handles mapping issues, putting the caller to sleep and calling your character driver strategy-like routine.
4. The character driver strategy-like routine sets up data structures, etc., and initiates transfer on the VME card.
5. The VME card master transfers a block of data and then generates a VME interrupt.
6. Your *driver*_**isr** routine sets up and initiates the next transfer on the VME card (if there is any left to do), or it cleans up the transfer on the PA-RISC CPU's end, including cache issues, and then notifies **physio** that the transfer is done.
7. **Physio** handles mapping, and returns (via the **wakeup** function) to the caller.
8. The *driver*_**write** function returns.

The kernel passes the device number and *uio* structure as parameters to the *driver*_**read** and *driver*_**write** routines, which pass them on to **physio**, along with the parameters shown in the following signature:

```
physio (strategy, bp, dev, rw, min_count, uio)
    int (*strategy)();
    register struct buf *bp;
    dev_t dev;
    int rw;
    unsigned (*mincnt)();
    struct uio *uio;
```

- *strategy*—A character-driver strategy routine that sets up an I/O request.
- *bp*—A pointer to a *buf* structure that can be declared or allocated by your driver (if NULL, **physio** allocates a *buf* structure from the file system buffer cache).

- *dev*—The device number passed into the *driver*_**read** or *driver*_**write** routine from the kernel.
- *rw*—A read/write flag set to B_READ for a read request or B_WRITE for a write request.
- *mincnt*—The routine that sets the maximum request size allowed in a single request to a driver. For most drivers, this is the kernel routine **minphys**. If the *mincnt* routine adjusts the request size, **physio** makes multiple requests to the *driver*_**strategy** routine until all the data specified by **iov_len** in the *uio* structure have been transferred.
- *uio*—The structure that is passed into the *driver*_**read** or *driver*_**write** routine from the kernel.

The **physio** routine handles details of the I/O transfer for the *driver*_**read** and *driver*_**write** routines. It performs the following tasks:

1. Checks for user permission to access the data area (pointed to by the *uio* structure).
2. Allocates a *buf* structure for the transfer if the *bp* parameter passed into **physio** is NULL.
3. Waits for the buffer associated with the *buf* structure to become available for use if it is busy.

   Before **physio** can use the *buf* structure passed to it, it ensures that the structure is not in current use. If the B_BUSY flag is set in the *bp*->**b_bflags** field, **physio** must wait until the *buf* structure becomes available by setting the B_WANTED flag in the *bp*->**b_bflags** field and sleeping on this *buf* structure. **physio** is awakened when the process that currently has the *buf* structure releases it and calls **wakeup** to wake up all processes waiting for the buffer.
4. Marks the buffer busy (sets B_BUSY and clears B_WANTED), establishing exclusive access to the buffer.
5. Copies information from the *uio* structure into the buffer header and initializes the buffer header, i.e., the *bp* struct.
6. Calls the *mincnt* routine—usually **minphys**—to adjust the request size so that it is less than or equal to the maximum allowable transfer size.
7. Locks the user's data into memory and maps it into kernel space.
8. Calls the character driver strategy routine with the argument *bp*, where *bp* is the *buf* structure previously described.

**9** Sleeps on the buffer header (after the strategy routine returns) passed to the strategy routine. **Physio** is awakened when the B_DONE flag is set. The driver needs to wake **physio** by calling **iodone(***bp***)** when the I/O completes. This means that **physio** provides synchronous reads and writes.

**10** Unlocks the user data area (once awakened), saves the residual count(from *bp*->**b_resid**) into the *uio* structure, and interprets any errors returned in *bp*->**b_error**.

**11** Repeats steps 1-10 if another call to the strategy routine is needed to process the *iovec* structure due to limitations imposed by *mincnt***.**

**12** Releases the buffer and returns the error status.

The strategy routine called by **physio** is *not necessarily* the same routine called from the kernel when **read** or **write** is performed on a block device file. It performs similar functions, but the **d_strategy** field in the *drv_ops* structure (see “drv_ops” on page A-9) has no meaning whatsoever for character drivers. (Often, though, the block and character strategy routines for a block and character device are really similar, and they are usually combined, including a little code to differentiate between the two sources of data.)

The following example of a read routine calls **physio** to do a read:

```
struct buf driver_buf;

driver_read (dev, uio)
dev_t dev;
struct uio *uio;
{
return (physio (&char_driver_strategy, &driver_buf,
                dev,B_READ, minphys, uio));
}
```

## **Using** uiomove

The **uiomove** routine moves data from one address space to another. In a driver, it can copy data from user space to kernel space and vice versa, freeing the device driver from having to map the user data area into kernel space. We recommend that you use **uiomove** to transfer data to or from I/O space.

In general, you use **uiomove** when a driver needs to perform small data transfers.If you use **uiomove**, your *driver*_**write** routine performs the following functions, as explained below:

1. Initializes any data structures
2. Calls **uiomove** to copy the data into kernel space
3. Makes the I/O request
4. Waits for I/O to complete

**uiomove** has the following signature:

```
uiomove (cp, n, rw, uio);
    caddr_t cp;         /* ptr to kernel space data */
    int n;              /* byte count to transfer */
    enum uio_rw  rw;    /* UIO_READ or UIO_WRITE */
    struct uio *uio;    /* ptr to transfer structure */
```

When you write a routine using **uiomove**:

- If *rw* is UIO_READ, *n* bytes at address *cp* are copied into *uio*. If *rw* is UIO_WRITE, *n* bytes from *uio* are copied into *cp*.
- If a driver wants to buffer data between the device and the user's buffer, it can use **geteblk**() and **uiomove** to do the buffering. Use **geteblk** to get an empty file system buffer and an associated buffer header from the kernel buffer cache.
- The buffer returned by **geteblk** is allocated from the file system buffer cache, so the file system temporarily loses access to this buffer. The *buf* structure and associated buffer belong exclusively to the routine that called it.
- When the driver completes the request, it should release the *buf* structure and associated buffer obtained from **geteblk** by using the kernel routine **brelse**().
- When a device driver gets a buffer using **geteblk**, the device driver is borrowing a buffer that would otherwise be used by the file system to cache data. This means that a device driver should not indiscriminately allocate buffers using **geteblk**, or file system performance could be affected.

The following code segment illustrates a **uiomove** implementation of a driver's write routine:

```
struct driver_card *driver_card_ptr;
struct buf *tmp_bp;
driver_write (dev, uio)
dev_t dev;
struct uio *uio;
{
    int count;
    caddr_t addr;
    count = uio->uio_iov->iov_len;
    tmp_bp = geteblk (count);
    addr = tmp_bp->b_un.b_addr;
    uiomove (addr, count, UIO_WRITE, uio);
    while (count)
    {
        driver_card_ptr->control = ENABLE_INTERRUPT;
        driver_card_ptr->data_out_reg = *addr++;
        sleep (tmp_bp, PRIBIO);
        count--;
    }
    brelse (tmp_bp);
    return(0);
}

driver_isr (isc, arg)
struct isc_table_type *isc;
int arg;
{
    driver_card_ptr->control = DISABLE_INTERRUPT;
    wakeup(tmp_bp);
    return(0);
}
```

### Data Transfer with vme_mod_copy

You can use the **vme2** routine **vme_mod_copy()** to transfer data to and from VME space with the address modifier that your card requires. You should first check that the card responds with **vme_testr** calls (or establish a bus error handler—see "VME Bus Errors and Panics" on page 3-10). This can be done in your *driver*_**attach** or at first open in your *driver*_**open** routine. You may also check immediately prior to using the **vme_mod_copy** routine in your **read**, **write**, **strategy**, or **ioctl** routine.

The **vme_mod_copy** routine is defined as follows (see Appendix B for argument and parameter definitions):

```
vme_mod_copy (isc, direction, virtual_addr, addr_mod,
                space, buffer, width, size)
    struct isc_table_type *isc;
    int direction;
    caddr_t virtual_addr;
    int addr_mod;
    space_t space;
    caddr_t buffer;
    int width;
    int size;
```

*Direction* is the direction of transfer: HOST_TO_VME or VME_TO_HOST. The *virtual_addr* is the kernel I/O space virtual address returned from a call to **map_mem_to_host**, and *addr_mod* is the VME address modifier code required by your interface card.

You should make sure that *space* is set to 0. You can use the KERNELSPACE constant defined in **/usr/conf/machine/vmparm.h** for this parameter. If this parameter is nonzero, the call will fail with an error of -1.

The *width* can be one of the following values defined in **/usr/conf/h/io.h**: OPTIMAL, BYTE_WIDE, SHORT_WIDE, or LONG_WIDE. An error will be returned if either the initial address or the size is not an even multiple of the cycle type (*width*). If OPTIMAL is used for *width*, the largest cycle possible will be used to reduce the number of transfers. This means that byte transfers will be made until a long word boundary is reached, followed by as many long transfers as can be made, and finally using byte accesses to complete the *size* number of bytes. (*Size* is the total transfer count in bytes.)

There are some restrictions on using OPTIMAL. In particular, transfers that are not source- and destination-aligned must use byte transfers. For example, if the starting virtual address is at byte 1 of a word and the starting buffer address given is also at byte 1, OPTIMAL or BYTE_WIDE may be used. In this example, if the starting buffer address is not also at byte 1, only BYTE_WIDE may be used.

***NOTE:*** vme_mod_copy runs internally at spl6() priority, meaning that interrupts are turned off and other VME-Classi processes are prevented from running during small transfers. If the size parameter is greater than 512, the transfer will be split into 512 byte "chunks" with an spl6 and splx() call around each "chunk" until vme_mod_copy completes.

In the following example, a driver needs to write a 32-bit-wide register at VME address 0x1000 with a user-defined address modifier. The card decodes all 32 bits of address.

```
#include "/usr/conf/h/io.h"
#include "/usr/conf/machine/vmparm.h"
#include "/usr/conf/machine/vme2.h"
#define MY_AM     0x1F
#define MYREGADDR 0x1000
#define REGSIZE   0x4
    .
    .
    .
static struct isc_table_type isc;
    data = /* some value */
    isc = my_isc;
    vme_set_address_space (isc, VME_A32);
    my_reg = map_mem_to_host(isc,MYREGADDR,REGSIZE);
    vme_mod_copy(isc,HOST_TO_VME,my_reg,MY_AM,
            KERNELSPACE,&data,LONG_WIDE,REGSIZE);
```

# Character Device ioctl Routines

HP-UX provides the system call named **ioctl**(2) to allow character device drivers to perform control functions on the associated device. Since different drivers require different control functions, the system call provided by HP-UX is flexible, which means you can implement control functions according to the device.

## Using ioctl

Used exclusively on character device files, the **ioctl** (I/O control) call is driver-dependent. You use **ioctl** for the following purposes:

- Modifying driver parameters.
- Modifying the configuration of a device.
- Implementing any special processing not provided by other I/O system calls. If you do not need this capability in your driver, use **nodev** or **nulldev** in the **cdevsw** table.

The signature for **ioctl** is as follows:

```
ioctl (fildes, request, arg)
    int fildes;
    int request;
    void *arg;
```

- *fildes*—A file descriptor obtained from a previous **open** or **dup**(2).
- *request*—A 32-bit integer having various fields encoded within it to specify: the size of *arg*; whether *arg* is passed to the driver, returned by the driver, or both; and the particular command to perform. The *request* field is the **command word** (see “Defining the Command Parameter” on page 6-19), which can be split into two categories:
  - Those requests processed by more than one driver
  - Those requests processed by one particular driver
- *arg*—An argument associated with the request. The type and value of *arg* is driver dependent and can vary from one **ioctl** to another.

More than one driver often implements standard requests having the form **Fxxxx**, which indicates generality. On choosing to implement any of these requests, your driver should process the request in a standard way. Typical requests implemented by **ioctl** calls include rewind (for tape drives) and change column width (for printers). (Refer to the man page for **ioctl**(5) for a list of such requests and the processing that your driver should perform.)

## Defining the Command Parameter

**Command words** describe values used for the request argument to **ioctl**. You should use the format specified by the kernel and define them in your driver's header file, for example, *driver***_nameio.h**. Any user programs issuing **ioctl** calls to your driver must include this file.

Particular bits within the command word tell the kernel:

- The associated driver for the command
- The arguments to the driver
- How to copy the driver's data (the kernel does not copy out on failure)

Use the following format to define all commands:

```
#define CMD task('t', n, structure)
```

- *CMD*—The **ioctl** command corresponding to the *request* parameter that the user-level application uses in the call to **ioctl**
- *task***—**How the kernel should copy the data structure between the user and the driver:
  - _IO—There is no argument: the structure parameter is omitted
  - _IOR—User reads information from the driver in **arg_ptr*
  - _IOW—User writes information for the driver into **arg_ptr*
  - _IOWR—Both _IOR and _IOW
- *t***—**An arbitrary character of your choice associated with a specific driver
- *n***—**A number (0 to 127) differentiating the various commands for a particular driver
- *structure*—A pointer to a data structure to accompany the **ioctl** request. Data structures have a size limit of 16 Kbytes

The following example shows the definition of **ioctl** commands for a hypothetical device defined in **mydevice.h**:

```
#include <sys/ioctl.h>
struct mydevice_ioctl_arg{
     char reg_value;
     int location;
};
#define CLEAR  0
#define SET    1
#define CARD_RESET _IO ('X',0)
#define CARD_STATUS _IOR ('X',1,structmydevice_ioctl_arg)
#define CARD_CONTROL _IOW ('X',2,struct mydevice_ioctl_arg)
#define CARD_BUFADR _IOWR ('X',3,struct mydevice_ioctl_arg)
```

The **#defines** in the code above specify four commands:

- CARD_RESET— An **ioctl** that allows the user to reset the device to its default state. This **ioctl** has no parameters.
- CARD_STATUS—Used to return the contents of the device's status register to the user. The status register contents are returned in the **reg_value** field of the *mydevice_ioctl_arg* structure.
- CARD_CONTROL—Used to set or clear the bits in the device's control register. The **reg_value** field of the *mydevice_ioctl_arg* structure passes the *driver*_**ioctl** routine the bit(s) to be cleared or set, and the **location** field contains SET or CLEAR to tell the driver whether to set or clear the bit(s) in **reg_value**.
- CARD_BUFFER—Assigns a memory location to the buffer on the device. The **location** field of the *mydevice_ioctl_arg* structure tells the driver where the buffer should be located in memory. If **location** is 0, the driver uses a default location. The location of the buffer is returned to the user in the **location** field.

The following segment from a user program sets a control bit on the device:

```
#include <sys/errno.h>
#include "mydevice.h"
#define SET_TIMEOUT  0x04

struct mydevice_ioctl_arg *ioctl_arg;
ioctl_arg->reg_value = SET_TIMEOUT;
ioctl_arg->location = SET;
if (ioctl(fd, CARD_CONTROL, ioctl_arg) < 0)
     printf("ioctl call failed, errno = %d\n", errno);
```

### *driver*_ioctl

While the user specifies a file descriptor, a command, and an argument to **ioctl**, the kernel invokes the *driver*_**ioctl** routine with the following signature:

```
driver_ioctl(dev, cmd, arg_ptr, flag)
    dev_t dev;
    int cmd;
    caddr_t arg_ptr;
    int flag;
```

*dev* is the device number of the associated device; *cmd* the command word described in the previous section; *arg_ptr* points to any arguments accompanying the command; and *flag* is the file access flags. (Most drivers ignore this parameter, but you can use the minor number.)

The *driver*_**ioctl** routine implements the **ioctl** commands defined in the previous section. The code follows:

```
#include <sys/errno.h>
#include <sys/types.h>
#include "mydevice.h"
struct mydevice_registers *dev_rp;
driver_ioctl(dev, cmd, arg, flag)
dev_t dev;
int cmd;
struct mydevice_ioctl_arg *arg;
int flag;
{
    switch (cmd){
    case CARD_RESET:
        dev_rp->reset = 0;
        return(0);
    case CARD_STATUS:
        arg->reg_value = dev_rp->status;
        return(0);
    case CARD_CONTROL:
        switch (arg->location){
            case SET:
                dev_rp->control |= arg->reg_value;
                return(0);
            case CLEAR:
```

```
                dev_rp->control &= ~arg->reg_value;
                return(0);
            default:
                return(EINVAL);
        }
    case CARD_BUFADR:
        arg->location = set_buf_addr (arg->location);
        return(0);
    }
}
```

## Character Device Select Routines

The **select**(2) system call calls a *driver*_**select** routine to determine if I/O has completed, is ready, or an exceptional condition exists. You use **select** only for character devices.

Performing **select** on device files can have different interpretations depending on the device:

- Use the **select** routine to poll a device for read/write/event status.
- A character device driver should return true if its device is always ready for I/O. A character device driver not having a *driver*_**select** routine should always return true (non-zero). You can specify **seltrue** in the *driver*_**select** field for your driver in the /**etc/master** file. If you do so, the kernel returns true without calling your driver when **select** is performed on your device file. (See “Editing System Files for the Driver” on page 7-6.)
- The *driver*_**select** routine has no access to the *readfds*, *writefds*, and *exceptfds* values that the user passed to the **select** system call. (See the man page for **select**.) The *driver*_**select** routine is passed only the device number and ***flag***. The ***flag*** field indicates the type of readiness to check according to one of the following values:
  - FREAD—for read
  - FWRITE—for write
  - 0—zero for exception conditions
- If the *driver*_**select** routine returns true, indicating it is ready for the action requested, the kernel sets the corresponding bit in the appropriate mask field and

returns this to the user.

- If the *driver*_**select** routine does not return true (when the condition doesn't exist), the **select** system call puts the calling process to sleep, waiting for the condition to become true. The driver needs to inform the system when this condition becomes true.
- The driver must check for collisions where two or more processes have tried to select on the same file for the same condition. You accomplish this by having the driver save the pointer to the **proc** entry for the calling process and using that pointer as a parameter to the kernel call **selwakeup**() to revive the sleeping process when the condition becomes true.

The following example shows a brief, skeletal *driver*_**select** routine:

```
#include <sys/types.h>
#include <sys/param.h>              /* for user.h  */
#include <sys/user.h>               /* for u def. */
#include <sys/proc.h>               /* for proc struct
*/
#include <sys/kthread_iface.h>
#include <sys/file.h>               /* for FREAD, FWRITE
*/
#include <sys/systm.h>              /* for selwait def.
*/
struct  my_sel_struct {
    struct kthread *read_waiter;
    struct kthread *write_waiter;
    int state;
}
struct my_sel_struct *my_sel_struct
driver_select (dev, rw)
dev_t dev;
int rw;
{
    int s = spl5();
    switch (rw) {
        case FREAD:
/* Determine if the device can be read from (there is
data).  If so, return true. */
            if (there is available data){
                splx(s);
                return(1);
            }
```

The following code is executed when the device is not ready to read. See if some other process is already set to be notified upon this condition and is still blocked on the **select** call. If so, there is a collision. The **select** system call handles this if we pass a true value in the **selwakeup** call's second parameter. Save a flag to indicate the collision. The system will wake up all the processes blocked on a **select** and restart the **select** call to allow collisions to compete for the data.

```
        if ( kt_wchan(my_sel_struct->read_waiter) ==
                (caddr_t)&selwait)
            my_sel_struct->state |= READ_COLLISION;
        else
            my_sel_struct->read_waiter = u.u_kthreadp;
            break;
        case FWRITE:
/* If this if statement is true, then there is a device
that can be written to.  Return true. */
        if (device is ready for more data){
            splx(s);
            return(1);
        }
```

The following code is executed if the device is not ready to write. If some other process is already set to be notified upon this same condition, and is blocked on the **select** call, there is a collision.

```
        if ( kt_wchan(my_sel_struct->write_waiter) ==
                (caddr_t)&selwait)
            my_sel_struct->state |= WRITE_COLLISION;
        else
            my_sel_struct->write_waiter = u.u_kthreadp;
            break;
```

When we return 0, **select** sleeps on **selwait**, waiting for the driver to find the condition true and wake the process up.

```
        splx(s);
        return (0);
    }
```

When the driver knows there is more input, or it knows output can be started, it wakes up any processes that might be sleeping for this condition by calling **selwakeup**. The *driver*_**output_ready** routine is called when the driver finds that the device is ready to output more characters. The skeleton for *driver*_**output_ready** looks like this:

```
driver_output_ready (mystruct)
register struct my_sel_struct *mystruct;
{
. . .
/* if a process is sleeping on a select for this
condition, wake it up. */
    if (mystruct->write_waiter) {
        selwakeup(mystruct->write_waiter,
                    mystruct->state & WRITE_COLLISION);
        mystruct->write_waiter = 0;
        mystruct->state &= ~WRITE_COLLISION;
    }
}
```

The *driver*_**input_ready** routine is called by the driver when the device/driver has input available. The skeleton for *driver*_**input_ready** to wake up any processes sleeping on the read condition looks like:

```
driver_input_ready (mystruct)
register struct my_sel_struct *mystruct;
{
. . .
/* if a process is sleeping on a select for this
condition wake it up. */
    if (mystruct->read_waiter) {
        selwakeup(mystruct->read_waiter,
                    mystruct->state & READ_COLLISION);
        mystruct->state &= ~READ_COLLISION;
        mystruct->read_waiter = 0;
    }
. . .
}
```

The mask returned to the user applies only to the particular moment in time when the *driver*_**select** routine was invoked. If **select** returns true for a particular file descriptor, it does not necessarily guarantee that the device will still be ready when a read or write is later issued to this file descriptor.

For each file descriptor specified using **select**, the corresponding *driver*_**select** routine is invoked. If more than one file descriptor has the same major number, **select** invokes the *driver*_**select** routine multiple times.

If an error is detected by the *driver*_**select** routine, the routine should return false (zero) and an error in *u*.**u_error** if selecting for read or write, otherwise true and an error in *u*.**u_error** if selecting for an exception condition.

## Block Device Strategy Routines

This section provides an overview of block I/O and describes the strategy routine used by block device drivers and some character device drivers.

### Overview of Block I/O

When a user issues a **read**(2) or **write**(2) to a block device, the driver strategy routine in the **bdevsw** table is called. For block device files, the kernel caches the data between a user process and the block device.

Block device drivers perform many of the same functions for **read** and **write** requests. The major difference is the direction of data transfer. The *driver*_**strategy** routine does processing for both read and write requests, usually starting the I/O transaction and returning to the routine that invoked it.

For a **write**, the kernel copies the data from the user's buffer to the allocated file system buffer. The kernel then calls the *driver*_**strategy** routine, passing as a parameter a pointer to a *buf* structure (also referred to as a buffer header). The *buf* structure contains a pointer to the associated buffer in the file system buffer cache. The *driver*_**strategy** routine uses the information in the *buf* structure to process the I/O request. For a write request, the *driver*_**strategy** routine should schedule the transfer of data from the buffer to the device.

For write requests, when *driver*_**strategy** returns, if the write is asynchronous, the kernel does not wait for the I/O to complete, but immediately returns to the user process. If the write is synchronous, the kernel waits for

the I/O to complete by issuing a call to **iowait**. Therefore, the lower half of the driver must issue a call to **iodone** when the I/O request is complete. When the I/O completes, the kernel returns to the user process.

When a **read** is performed on a block device file, the kernel first checks the buffer cache for the requested data. If the kernel finds the buffer associated with the particular device and block number, the requested data can be returned to the user process without invoking the *driver*_**strategy** routine. If the kernel does not find the buffer associated with the particular device and block number, then the kernel allocates a buffer for this device and block number, and calls the *driver*_**strategy** routine to schedule the transfer of data into this buffer.

For read requests, when *driver*_**strategy** returns, if the I/O has not completed, the **read** system call waits for the I/O to complete by issuing a call to **iowait**. Therefore, the lower half of the driver must issue a call to **iodone** when the I/O request is complete. When **read** is awakened by the call to **iodone**, **read** copies the data from the file system buffer to the user's buffer.

Once a block is in the buffer cache, the kernel determines how long the block stays in the buffer cache. The kernel determines which buffers have not been accessed in a long time and either calls the *driver*_**strategy** routine to write the buffer to the device if they have been modified, or reuses the buffers if they have not been modified.

### The *driver*_strategy Routine

The *driver*_**strategy** routine can be called:

- As a result of a read or a write on an ordinary file, a directory, or a block device
- By the *driver*_**read** or *driver*_**write** routine as a result of a read or write on a character device file

Use a *driver*_**strategy** routine to perform I/O to or from the device. The functions performed by this routine include:

- Initializing any data structures
- Adding the I/O request to a queue, if necessary
- Setting a flag indicating that I/O is in progress
- Returning to the calling process

After scheduling an I/O request, the *driver*_**strategy** routine returns to the routine that invoked it. The *driver*_**strategy** routine must not issue a call to **sleep**(). The process that invokes *driver*_**strategy** is responsible for determining whether or not to wait for the I/O to complete.

On completing the I/O request, the lower half of the driver should:

1 Set B_ERROR in **b_flags,** and set **b_error** in the *buf* structure to an **errno** value, if an error occurred, for example:

```
buf->b_flags = buf->b_flags && B_ERROR;
buf->b_error = errno;
```

2 Set **b_resid** to indicate the amount of data not transferred.

3 Wake up the top half by calling **iodone,** assuming the top half called **iowait**.

The *driver*_**strategy** routine has the following signature:

```
driver_strategy (bp)
    struct buf *bp;
```

*bp* is a pointer to a *buf* structure that contains all the information about the request that the driver needs in order to perform the I/O. The *driver*_**strategy** routine uses the information in the buffer header to process the I/O request.

### *driver*_strategy **for** write()

As a result of a **write** system call to a block device file, the kernel allocates a kernel *buf* structure and a kernel buffer for the I/O request. The kernel associates the buffer with the particular device and block number the buffer represents.

The kernel fills in the buffer header with other information describing the I/O request. For example, the kernel sets the B_WRITE flag in **b_flags** to indicate to the *driver*_**strategy** routine that the request is a write request.

The kernel maps the data in the user's data area into the kernel buffer. The kernel then sets the *b_un*.**b_addr** field to point to this kernel buffer. Then the kernel calls the *driver*_**strategy** routine, passing a pointer to the *buf* structure as a parameter. The *driver*_**strategy** routine now has exclusive access to this kernel buffer.

The *driver*_**strategy** routine performs the I/O to the device. For write requests, it schedules the data in the kernel buffer to be copied to the device. It should then return to the routine that invoked it. If the write is asynchronous, the **write** system call does not wait for the I/O to complete but returns to the user, so the value returned to the user process simply indicates that the data has been successfully copied to the buffer cache and scheduled for I/O. If the write is synchronous, the **write** system call invokes **iowait** and waits for the I/O to complete.

When the I/O completes, the lower half of the driver sets **b_resid** to the amount of data not transferred. It sets B_ERROR in *bp*->**b_flags** and sets *bp*->**b_error** to an **errno** value if an error occurred, and calls **iodone**. Any process sleeping on the buffer is awakened by **iodone**. This buffer and *buf* header can now be used by another process.

### *driver*_strategy **for** read()

For **read** system calls performed on block device files, the kernel first checks the buffer cache for the requested data. If the data is in the buffer cache, the kernel copies it to the user's data area and returns without calling the *driver*_**strategy** routine. If the data is not in the buffer cache, the kernel allocates a kernel *buf* structure and a kernel buffer for the I/O request. The kernel associates the buffer with the particular device and block number the buffer represents. The kernel fills in the buffer header with other information that describes the I/O request. For example, the kernel sets the B_READ flag to indicate to the *driver*_**strategy** routine that the request is a read request.

For read requests, the *driver*_**strategy** routine schedules the data to be copied from the device to the kernel buffer. It should then return to the routine which invoked it. For read requests on block device files, the kernel always waits for the I/O to complete before returning to the user. The **read** system call invokes **iowait**, and waits for the I/O to complete.

When the I/O completes, the lower half of the driver should set *bp*->**b_resid** to the number of bytes not transferred; set B_ERROR; set *bp*->**b_error** to an **errno** if an error occurred; and call **iodone**. The **read** system call copies the data in the kernel buffer into the user's data area so that it is available to the user process. The kernel releases the *buf* structure by clearing the B_BUSY flag, and calls **wakeup** to wake up any processes sleeping on the buffer.

The code below shows the driver routine named *driver*_**strategy**:

```
#include <sys/types.h>
#include <sys/errno.h>
#include "/usr/conf/wsio/vme2.h"
#include <sys/buf.h>
struct skelregs *skel;   /* board registers     */
struct buf *skelbuf;     /* io buffer           */
char r_int_enable_reg;   /* software reg copy   */

driver_strategy(bp)
struct buf *bp;
{
    int pri;
    register caddr_t addr;
    register short   cnt;
    struct isc_table_type *isc;
    dev_t dev = bp->b_dev; /* set in physio routine */
    isc = dev->isc;
    pri= spl4();
    addr = bp->b_un.b_addr;
    cnt = bp->b_bcount;
/* set up device; if_reg_ptr is set up earlier */
    skel = (struct skelregs *)isc->if_reg_ptr;
     skel->registerX = .........
    if (bp->b_flags & B_READ){
/* This device doesn't read */
        bp->b_flags |= B_DONE;
        splx(pri);
        return(ENXIO);
    }
    else{
        /* Complete Write Transfer */
        if (~cnt) {
            bp->b_flags |= B_DONE;
            splx(pri);
            return(0);
        }
        else driver_start (bp);
    }
    splx(pri);
}
driver_start (bp)
struct buf *bp;
```

```
{
    addr = bp->b_un.b_addr;
    cnt = bp->b_bcount;
    /* special last byte setup, if needed    */
    if (cnt == 1)    /* last byte, do it now */
        skel->control = AUX_SEOI;
    skel->int_enable_reg = DOIE;
    skel->data_out_reg = *addr++;
    bp->b_bcount--;
}
driver_isr (isc, arg)
struct isc_table *isc;
int arg;
{
    skel->ch1.status_reg = D_CLEAR;
    skel->int_enable_reg = ~DOIE;
    r_int_enable_reg |= skel->int_enable_reg;
    r_int_enable_reg &= ~DO;
    cnt = skelbuf->b_bcount;
    if (cnt == 0)
        iodone(skelbuf);
    else driver_start(skelbuf);
}
```

## Writing Synchronous Host DMA Strategy Routines

If you write a *driver*_**strategy** routine to set up and initiate DMA, your *driver*_**strategy** routine should:

1. Call **vme_dma_setup** to set up the chain of DMA requests (see See “Synchronous Local Host DMA” on page 4-8). You may wish to do this from within a *dma_setup*() routine that you also write.
2. Call **vme_dma_start** to set up the chain of transfers. You may wish to do this from within a *dma_start*() routine that you also write.
3. Return.

The following code shows a *driver*_**strategy** routine:

```
driver_strategy(bp)
struct buf *bp;
{
struct isc_table_type *isc;
struct dma_parms *dma_parms;
dev_t dev = bp->b_dev; /* set in kernel physio routine */
int ret, pri;
    isc = dev->isc;
    isc->owner = bp;
    isc->dma_parms = (int)
        io_malloc(sizeof(struct dma_parms),IOM_WAITOK);
    dma_parms = (struct dma_parms *)isc->dma_parms;
    bzero(dma_parms, sizeof(struct dma_parms));
    dma_parms->addr = bp->b_un.b_addr;
    dma_parms->count = bp->bp->b_count;
    dma_parms->drv_routine = driver_wakeup;

    /* set flags according to transfer desired */
    if (DRIVER_TRANSFER TYPE == A32)
        dma_parms->dma_options |= VME_A32_DMA;
    else
        dma_parms->dma_options |= VME_A24_DMA;
    if (DRIVER_MAP_TYPE == SLAVEMAP)
        dma_parms->dma_options |= VME_USE_IOMAP;

    /* begin critical section */
    pri = spl 6();
    while (ret = vme_dma_setup(isc, dma_parms)) {
        if (ret == RESOURCE_UNAVAILABLE)
            sleep(dma_parms, PZERO+2);
        else if (ret<0) {
            bp->b_error = EINVAL;
            bp->b_flags |= B_ERROR;
            iodone(bp);
            splx(pri);
            io_free((caddr_t)dma_parms,
                    sizeof(struct dma_parms));
            return (0);
        }
    }
    splx(pri); /* end critical section */
```

```
    /* reset card */
    if (bp->b_flags & B_READ)
        /* Perform read-specific card set up */
    else
        /* Perform write-specific card set up */

    driver_dma_setup(isc);
    driver_dma_start(isc);
}
```

## Strategy Routines for a Remote Master's DMA

**Remote DMA controllers** run as masters on the VME bus and provide their own hardware to DMA the memory on your VME-Class host. Your *driver*_**strategy** and its associated DMA setup and ISR routine for a simple, remote DMA controller may be modeled on the examples shown below.

The following code shows a *driver*_**strategy** routine that:

1. Sets up a *dma_parms* struct.
2. Calls **vme_dma_setup** in REMOTE_DMA mode to set up Series 700 mapping hardware on local board..
3. Calls *remote_dma_setup*, which sets up the registers for a single chain and then starts the card's DMA controller (see "Remote DMA Setup Routine" on page 4-18).

When the card's DMA controller is done, it interrupts, and the *remote*_**isr** routine runs:

```
remote_strategy (bp)
struct buf *bp;
{
    struct dma_parms *parms;
    int pri, ret;

    parms = (struct dma_parms *)
    io_malloc(sizeof(struct dma_parms), IOM_WAITOK);
    bzero(parms, sizeof(struct dma_parms));
    if (isc->owner != NULL) /* someone else using bp? */
        sleep(isc, PRIBIO);
```

```
    isc->owner = bp;
    isc->dma_parms = parms;

    parms->dma_options |= VME USE_IOMAP;
    parms->dma_options |= VME A24 DMA;
    parms->addr = bp->b_un.b addr;
    parms->count = bp->b_bcount;
    parms->drv_routine = remote_wakeup;
    parms->drv_arg = parms;
    parms->dma_options |= DMA WRITE;

    pri = spl6();
    while (ret = vme_dma_setup (isc, parms)){
        if (ret == RESOURCE UNAVAILABLE)
            sleep(isc->dma_parms, PZERO + 2);
        else if (ret < 0) {
            io_free (parms, sizeof(struct dma_parms));
            bp->b_error = EINVAL;
            bp->b_flags |= B ERROR;
            isc->owner = NULL;
            wakeup (isc);
            splx(pri);
            iodone(bp);
            return;
        } else break;
    }
    splx(pri);
    bp->b_resid = 0;
    remote_dma_setup(isc);
    return;
}
remote_wakeup (parms)   /* Remote Card WAKEUP */
struct dma_parms *parms;
{
    wakeup(parms);
    return(0);
}
```

As seen in the above example, if a driver uses **vme_dma_setup**, its strategy routine calls **vme_dma_setup** to set up the chain of DMA requests. It then calls the *driver_dma_setup* routine to set up the chain of transfers, and then perhaps a *driver_dma_star***t** to start the first DMA transfer. The remaining

transfers on the chain are started from the *driver_isr* routine, when the card has completed and interrupted the system. This arrangement ensures that only one transfer is being initiated at a time. When your *driver*_**strategy** returns to **physio**, **physio** calls **iowait**, waiting for the I/O to complete. I/O completion is indicated by a call to **iodone**.

## Writing Asynchronous Host DMA Strategy Routines

Asynchronous DMA allows your process to continue while the DMA takes place. When the DMA finishes, a routine that you specify is called.

Asynchronous DMA strategy routines typically have three parts:

- A strategy routine that initiates DMA activities among other activities
- A launcher to initiate the DMA
- A cleanup function after the DMA terminates

The following example illustrates these latter two parts with its functions for **skel_dma_queue** and **skel_dma_complete**.

```
static int skel_dma_queue (bp)
    register struct buf *bp;
{
    caddr_t *vme_address;
    int modifier, priority, wrt, arg;
    /* printf ("skel_dma_queue: buf = 0x%x\n", bp); */

    bp->b_resid = bp->b_bcount;         /* not
transferred */
    /* validate transfer amount */
    if (bp->b_bcount){
        /* get the source/destination address */
        vme_address = . . .             /* your VME addr
*/
        /*  set up transfer mode */
        modifier=vme_get_status_id_type(bp->b_dev->isc);
        priority = DMA_NORMAL; /* set up dma priority */
        /* arg to pass to completion routine */
        arg = bp;
```

```
        /* read or write command */
        wrt = (bp->b_flags & B_READ):
                                        DMA_READ ?
DMA_WRITE;
        /* queue the dma request to be processed */
        if ((ret = vme_dma_queue(u.u_procp,
                    bp->b_un.b_addr, bp->b_bcount,
                    vme_address, modifier, wrt,
                    priority, skel_dma_complete,
                    arg)) < 0){
            goto STRATEGY_ERROR;
        }
    }
    return (0);

STRATEGY_ERROR:
    if (!(bp->b_flags & B_DONE)){
        if (bp->b_bcount){
            bp->b_error = EIO;
            bp->b_flags |= B_ERROR;
        }
    }
    return EIO;/* outer strategy routine calls iodone */
}


skel_dma_complete (who, buffer, result, count, arg)
    int who;
    caddr_t *buffer;
    int result, count, arg;
{
    struct buf *bp = (struct buf *) arg;
/*  printf ("skel_dma_complete\n");             */
    /* error during dma transfer */
    if (result < 0){
        bp->b_error = EIO;
        bp->b_flags |= B_ERROR;
    }
    /* amount not transferred */
    bp->b_resid = bp->b_bcount - count;
```

```
    /* notify your strategy routine so that it can call
            iodone() when ready */
}
```

## Transfer Routines

The *driver*_**transfer** routine is an optional interface driver routine called by the device driver (generally via the *driver*_**strategy** routine through the ***ifsw*** pointer) to perform the interface-dependent part of I/O setup and initiation. If the interface driver supports multiple types of transfers (DMA, handshake, interrupt, etc.), the *driver*_**transfer** routine determines which type of transfer to implement for each I/O request and then initiates the transfer.

The functions performed by a *driver*_**transfer** routine include:

- Determining type of transfer to perform (if appropriate)
- Setting up interface for transfer
- Initiating transfer

The parameters of a *driver*_**transfer** can be defined based on the type of information that is needed in the device driver(s) strategy routine.

```
#define INTXFER 1
#define DMAXFER 2
driver_transfer(bp,type)
struct buf *bp;
int type;
{
    if (type == INTXFER)
        /* routine to transfer using interrupt */
        skel_int_xfer(bp);
    else if (type == DMAXFER)
        /* routine to transfer using dma */
        skel_dma_xfer(bp);
    else
        ??? /* probably panic unknown xfer type */
    iodone(bp);
    return;
}
```

## Close Routines

When a user uses the **close**(2) system call on a device file, the overall process works as follows:

1. The **close** calls the corresponding driver routine via the **devsw** table.
2. The *driver*_**close** routine is called only on the last close of the device file (unless the C_ALLCLOSES flag is specified in the device switch table entry for the device).
3. Calling *driver*_**close**(*dev***)** only on the last close of the device prevents a process from closing a device if another process is accessing it. If the device has more than one unique device number, *dev*, the *driver*_**close** routine is called on the last close of each unique device number.
4. If the file being closed is a block device file that is still mounted, then the block device file cannot be closed. In this case, the **close** system call returns without calling the *driver*_**close** routine.

It is possible for a device to be referenced by two different inodes (device files). The device must remain open until all active inodes for the device are closed. If more than one active inode (e.g., there are two device files with the same major and minor number accessing the same device) exists for the same device, the *driver*_**close** routine may be called prematurely.

A driver can compensate for this by setting the C_ALLCLOSES flag and maintaining its own open count. However, setting the C_ALLCLOSES flag does not guarantee that the *driver*_**close** routine will be called on every close of the device. It will be called whenever the link count in the inode is 0 or 1. This suggests that it will not be called, for instance, when an open file descriptor is inherited by a forked child process and the child closes the device file.

### *driver*_close

The *driver*_**close** routine is the device driver entry point for a **close** performed on a block or character device file. Overall, the *driver*_**close** routine does the following:

1. Completes any in-progress I/O.
2. Releases any data structures.
3. Clears the open flag, indicating the device is closed (for exclusive open devices).

For example, actions typically performed by a **disk_close** routine for a floppy disk might include unlocking the door. Functions performed by a **tape_close** routine might include rewinding the tape.

Although the user specifies a file descriptor in the **close** system call, the *driver*_**close** routine is invoked by the kernel with:

```
driver_close (dev, flag)
    dev_t dev;
    int flag;
```

and the following parameters:

- *dev*—The device number of the file to be closed. The *driver*_**close** routine can extract the major and minor number from the device number by calling **major**().
- *flag*—A value corresponding to the **oflag** field in the **open** system call. (Refer to "Open Routines" on page 6-6 for values that can appear in the *flag* parameter, which most *driver*_**close** routines actually ignore.)

Your *driver*_**close** routine need not return a value. The kernel always returns success (zero) to the user process on a close, ignoring the return value from the driver.

```
int skel_card_opened;
int skel_card_isc;
driver_close (dev, flag)
dev_t dev;
int flag;
{
    register struct isc_table_type *isc;
    char *card;
    isc = dev->isc;
    if (isc == NULL)  /* no device at selcode */
        return;
    card = (char *)(isc->card_ptr);
/* Nobody's using the card, so disable it */
    card->registerX = ....
/* Enforce exclusive open, if implemented. */
    skel_card_opened = 0;
}
```

# 7

# Installing VME Devices

### Overview of Installing VME Devices

This chapter describes the installation of a VME device into HP-UX. It discusses the following steps:

1. Installing the VME Services product onto each HP-UX system in the cage.
2. Installing the VME card in the backplane.
3. Developing a device driver for the card so that HP-UX can access it.
4. Editing system files so that the device driver can be built into the kernel.
5. Compiling the driver.
6. Building the driver into the kernel.
7. Editing the configuration file that specifies the configuration information.
8. Using **vme_config** to determine the number designated for this board, and update the configuration file with the appropriate CPU number.
9. Running **vme_config** to read the configuration file and save its data in an EEPROM that HP-UX can read at boot time.
10. Rebooting the HP-UX system.

Note that Steps 7 through 9 are explained in Chapter 8.

## Installing the VME Product

Before configuring VME resources, you may need to install the VME Services product, which contains the **vme2** driver and the **vme_config** software.

1. Log in as superuser.
2. Check to see if the fileset is already installed, by entering the command:

   **/usr/sbin/swlist VME-Services**

   If the product is already installed, you will see a listing; otherwise a message indicates that the product was not found.
3. If the product is not installed, run the **swinstall** utility to add the **vme2** driver to the kernel and reboot:

   **/usr/sbin/swinstall VME-Services**

Accept the default parameters that **swinstall** provides, unless you want to change the depot from the default **/var/spool/sw**, to another depot—such as **/dev/rmt/0m** (DAT tape) or **/mnt/cd** (CD-ROM mount point)—supported by SD, the HP-UX Software Distributor. For more information, see **swinstall**(1M).

## Verifying the vme2 Driver

To use VME Services, you need the **vme2** driver. When you installed the VME-SERV fileset above, this driver should have been added to the HP-UX kernel. To check this:

1. Start the HP-UX system administration manager, SAM:

   **/usr/sbin/sam**

   (For more information on using SAM, see the *HP-UX System Administration Tasks* manual.)

2. Select Kernel Configuration and press the Open button.
3. Select Drivers and press Open. A list of drivers appears.
4. Scroll through the list until you see **vme2**. If **vme2** is in the kernel, its current state is marked “in,” and you can exit SAM.
5. If the current state of **vme2** is “out,” select **vme2** driver.
6. Choose Actions->Add Driver to Kernel. The entry for the driver in the column Pending State changes to “in.”
7. Reconfigure the kernel to implement the change by choosing Actions->Create a New Kernel.
8. Press Yes to confirm that you want to reconfigure the kernel now.
9. Choose Create a New Kernel Now. (This requires a reboot of your system.)
10. Follow the prompts to regenerate and reinstall the new kernel.
11. Choose the option to move the kernel into place and reboot the system now.
12. Press OK.

### Choosing a Kernel Configuration File

You can tell SAM to enable or disable the overwriting of the kernel configuration file, /**stand/system**. If you choose:

- Disable—**/stand/system** will *not* represent your current kernel (**/stand/vmunix**) when you reboot your system. Instead, **/stand/system.SAM** represents your current kernel configuration after you reboot your system.
- Enable—SAM moves **/stand/system.SAM** to **/stand/system**, overwriting any comments you have added.

If you do not want SAM to overwrite **/stand/system**, because of comments you want to retain:

1. Choose the SAM option to disable overwriting the kernel configuration file.
2. Move the kernel into place (optionally rebooting the system).
3. Copy your comments from **/stand/system** to /**stand/system.SAM**.
4. Be careful to add only your comments to the file. At this stage, you want **/stand/system.SAM** to reflect your current kernel configuration.
5. Copy or save **/stand/system.SAM** to **/stand/system**.
6. The kernel configuration file **/stand/system** now represents the current **/stand/vmunix** kernel.

***NOTE:*** HP highly recommends that you choose **Enable** so that **/stand/system** is the kernel configuration file and is up to date with the executing kernel, **/stand/vmunix**, because some system software depends on this.

## Installing a Card in the VME Backplane

The VME standard allows two card sizes:

- Half-height or 3U cards. These cards connect to the VME backplane using only one of the two backplane connectors. 3U cards often have either 3U or 6U cover plates, allowing them to be placed in either a half-height or full-height chassis.
- Full-height or 6U cards. These cards connect to the VME backplane using both

of the backplane connectors.

***NOTE:*** For information on the power available in each VME slot, see the service manual for your HP 9000 VME-Class.

Take the following steps to install a VME card:

1. Ensure that your VME card configuration matches the configuration specified in the **vme.CFG** file (see “vme.CFG, the VME Configuration File” on page 8-2).

   The file **/var/adm/vme/system.log** contains a description of your current configuration. The documentation for your card should explain how to configure the card. You may need to set card switches and jumpers, or configure the card from an application program.

2. If the card provides a system controller capability, disable it with a switch or jumper. The card’s documentation should explain how to do this.
3. Also see your VME card’s installation manual, then set any configurations that may be required for your application.
4. Before physically installing the VME cards, shut down the system. See the appropriate hardware owner’s guide for installation instructions.
5. See your HP-UX and VMEbus manuals for information on the VMEbus accessory card’s application, loading the software, and running it.

## Making Device Files for Your Drivers

The device file supplies information about the major device type (the device driver number) and device location. For cards and processors, the card’s supplier should provide specific information to create the device files.

For more information, see **mknod**(1M).

To create a device special file for your device driver in the **/dev** directory:

1. Use the **lsdev** command to identify the major numbers assigned to device drivers in your current kernel.
2. Use the **su** command to become the superuser.
3. Use the **mknod()** command to create the necessary device file(s) for the card’s driver and any peripherals that will be attached to the card:

**mknod /dev/***my_driver device_type major_number* [*minor_number*]

In the syntax above, *my_driver* is the name of your driver; *device_type* is either **b** for block or **c** for character devices; *major_number* is your choice of a number that has not already been assigned to a device driver; and *minor_number* is a number that can be represented in 24 bits, either specified as a decimal or hex.

For example, the command

**mknod /dev/my_driver c 59 0**

creates a node for a character driver with a major number of 59 and a minor number of 0. The major number provides an index into the character (or block) device switch table. The major and minor number together are represented in 32 bits and are used as the first argument (*dev*) to all driver entry points.

## Editing System Files for the Driver

Multiple master files in **/usr/conf/master.d** replace the pre-10.0 **/etc/master** file in providing entry points for the device switch tables **bdevsw** and **cdevsw**, as well as configuration information about drivers. From these files, the **config** command generates a comprehensive master file.

After you create a *master* file to describe your driver, you also need to modify a build file named **system**.

### Creating a Master File for Your Driver

Copy another master file from the **/usr/conf/master.d** directory to use as a template. Name your master file uniquely so that it identifies your driver, so that customers, other programmers, or other administrators will recognize the connection. Make the file name no longer than 14 characters, not containing the words *core* or *rmtbranch*, nor the characters for a period (.), tilde (~), or pound sign (#).

***CAUTION:*** Don't add your driver's information to one of the existing master files supplied by Hewlett-Packard in **/usr/conf/master.d**. These files may likely be overwritten by a future HP-UX release.

Four sections of a master file need to be edited: the install, dependencies, and two library sections.

#### The Install Section

The install section of the table is identified by the $DRIVER_INSTALL variable and needs to contain a line that specifies the name of your device driver, so that the name can later be supplied on the output line for the driver when you issue the **ioscan** command. (It's also used elsewhere.)

#### The Driver Dependency Section

If your driver has dependencies on other drivers, such as **vme2**, you need to list them in this part of the master file.

#### The Driver Library Section

Use this section to specify libraries that contain functions used by your driver. You only need to specify them if they aren't specified in other master files such as **core-hpux**.

## Editing the Build File

The kernel looks for the name of your driver in **/stand/system**, which replaces the pre-10.0 **/etc/conf/dfile** as a means of describing a particular system's components. See **config**(1M) for more information about its format.

To make an entry for your driver in the file **/stand/system**:

1. Change directory to the build environment (**/stand/build**).

   **cd /stand/build**

2. Execute the system-preparation script to extract the system file from the current kernel, as follows:

   **/usr/lbin/sysadm/system_prep -s system**

   The **system_prep** script writes a system file in your current directory—in this case, it creates **/stand/build/system**.

3. Edit the first section of **/stand/build/system** to add your driver's name, which should match the name you gave in the **/usr/conf/master.d/***master* file.

# Compiling the Driver

Before you can build your driver into the HP-UX kernel, you must compile your driver to generate a new object file.

## Declaring Variables and Functions

You may provide tunable parameters for on-site system administration. In this section we give you an example of how VME services variables and functions are declared.

The driver developer defines tunable values:

- By declaring variables in a *mydriver*.**h** file in the **/usr/conf/space.h.d** directory.

```
unsigned int vmebpn_sockets          = VMEBPN_SOCKETS;
unsigned int vmebpn_tcp_ip           = VMEBPN_TCP_IP;
unsigned int vmebpn_total_jobs       = VMEBPN_TOTAL_JOBS;
```

- By supplying default values in your master file **/usr/conf/master.d/***mydriver*. You can put tunable parameters and their default values into the section labeled $TUNABLE, as is done for VME BPN's tunable parameters:

```
vmebpn_sockets       VMEBPN_SOCKETS          1
vmebpn_tcp_ip        VMEBPN_TCP_IP           1
vmebpn_total_jobs    VMEBPN_TOTAL_JOBS      16
vmebpn_public_pages  VMEBPN_PUBLIC_PAGES     1
. . .
```

The default values can be changed by the user:

- By editing values or by adding new lines to the **/stand/system** file. For example, the following part of a **/stand/system** file sets **vmebpn_sockets** to 0 (see "Tuning VME BPN Pipes" on page 9-6 for what this might mean):

```
* Generic system file for new computers
* Drivers/Subsystems
graph3
. . .
vme2
vmebpn_sockets  0
```

- By using SAM to produce editable values in **/stand/build/conf.SAM.c**.

### Compiling the driver

To compile a driver, e.g., **mydriver.c**:

1. Change to the directory that contains your source files:

   **cd** *source_directory*

2. Use the compiler options shown below:

   **/usr/ccs/bin/cc -c -g -DKERNEL -D_KERNEL** \
   **-DKERNEL_BUILD -D_UNSUPPORTED -D__hp9000s700** \
   **-Dhp9000s700 -D__hp9000s800 -Dhp9000s800 -D_WSIO** \
   **-I/usr/conf/io +XixduV +Hx0 +R25 +DA1.0 +DS1.0** ***mydriver*.c**

   (You can also use the compiler options that you find in **/stand/build/config.mk**.)

   Note that you must use HP's ANSI C compiler for the **-g** debug flag to work.

3. A successful compilation produces a *mydriver*.**o** file in your current directory.

4. Optionally, after you have debugged your driver, you can delete the **-g** option in the compile command.

## Building the Driver into the Kernel

After you have created or edited the master file **/usr/conf/master.d/** *mydriver*, edited the file **/stand/system**, and compiled the driver, if necessary, see the *HP-UX System Administration Tasks Manual*, and then build a kernel that includes it by taking the following steps:

1. Choose one of the following two approaches to arrange for your driver's object file to be found by the **config.mk** file:

   - Assign the object file's name to the OFILES macro definition in **/stand/build/config.mk**, for example:

     ```
     OFILES = mydriver.o
     ```

     See **config(**1M**)** for additional help. You can generate the file **/stand/build/config.mk** by using the command:

     **/usr/sbin/config -s system**

- Add your driver to **/usr/conf/libusrdrv.a** by issuing the command:

  **ar -r /usr/conf/libusrdrv.a** *mydriver***.o**

  This approach uses **config.mk**'s LIBUSRDRV macro to link the kernel and your driver.

**2** Build the kernel by moving your *master* and *space* files into standard areas and executing the command to make the **config.mk** file:

**mv** *mydriver* **/usr/conf/master.d/***mydriver*
**mv** *myspace* **/usr/conf/space.h.d/***myspace*
**make -f config.mk**

You may have to wait a moment while **config.mk** links your driver in with the rest of HP-UX and builds a file named **vmunix_test** in your current directory.

If you have tunable parameters in a *myspace* file, and the compiler fails to link **vmunix** because they are undefined, supply a **master.d** file ( see "Compiling the Driver" on page 7-8) and repeat Step 2.

**3** Move the new system file and kernel into place, after first moving the old system file and kernel to a safe place (thus, if anything goes wrong, you still have a bootable kernel):

**mv /stand/system /stand/***system.previous.rev_num*
**mv /stand/vmunix /stand/***vmunix.previous.rev_num*
**mv system /stand/system**
**mv vmunix_test /stand/vmunix**

**4** If you haven't already configured the device in the appropriate **vme.CFG** configuration file, see Chapter 8.

**5** If you haven't updated the workstation's EEPROM from the configuration file, see "Updating the Series 700's EEPROM" on page 8-13.

**6** Reboot your system with the new kernel:

**exec reboot**

# 8

# Configuring VME Devices

To configure a VME device into HP-UX, take the following steps:

1 Edit the **vme.CFG** configuration file that specifies the configuration information.

2 Use **vme_config** to set the appropriate CPU number for this board.

3 Run **vme_config** to read save the configuration information in an EEPROM.

## vme.CFG**, the VME Configuration File**

**vme_config**(1M) reads a file that describes the VME devices in a backplane card cage, in order to store the information into an EEPROM that HP-UX reads when the system boots. The **vme.CFG** file specifies the address range for each card that provides memory space, a location monitor, or FIFO; the interrupt levels for each processor card; and so on.

Edit the configuration file according to the general steps listed here and described more fully in the next few subsections:

1 Make sure there is a link from **/etc/vme/vme.CFG** to **/sbin/lib/vme/vme.CFG** so that when you create your own file **/sbin/lib/vme/vme.CFG** it will be found by **vme_config** (since **/etc/vme/vme.CFG** is the configuration file that **vme_config** reads by default).

2 Change to the directory containing the sample configuration files:

**cd /sbin/lib/vme**

**(**The file **/var/adm/vme/system.log** contains the current system configuration.)

3 Choose one of the example files as a template and copy it to **./vme.CFG**, which is the default configuration file for **vme_config**:

**cp /etc/vme/***selected_example* **vme.CFG**

The example files describe the following single card-cage configurations:

- **example.CFG**—An HP-UX processor and a single card
- **example2.CFG** —Two HP processors and shared memory
- **example3.CFG** —Multiple processors and cards, and shared memory
- **example4.CFG**—VME Backplane Networking

4 Edit the **vme.CFG** file

# General Format for Configuration Records

The configuration **records** have the following types of formatting:

- Two slashes (//) at the beginning of a line introduce a comment, which may be a reminder of the fields to be specified for a given type of record, for example:

```
// Processor Records
//  Name    CPU/Card        ID      Options
```

- Curly braces { and } enclose multi-line, complex record definitions.
- A back slash (\) at the end of a line is used to continue records.
- Numbers are decimal, unless preceded by 0x for hex or 0 for octals.
- **M** and **K** suffixes indicate megabytes and kilobytes, respectively.

## Common Fields for Configuration Records

Many of the record types specify fields that have a common definition or meaning, as listed here (not all of which apply to any given record):

- Keyword—an alphanumeric string at the beginning of a record, specifying its type (**proc**, **card**, **memory**, **dma_params**, **interrupt**, or **slot_1_functions**.
- Name—a symbolic reference to a processor, card, memory entity, or DMA-capable HP CPU, expressed as a string of one or more alphabetic characters or an underscore (_) followed by the same possibilities or the digits 0-9.
- CPU/Card Number—a CPU number, unique from 0 to 31, or a card number in the same range, so that there can be at most 32 VME cards and/or CPUs, with a maximum of 32 boards described in the file. (This number does not need to correspond in any way to the actual location of the card in the VME cage.)

***NOTE:*** The provided configuration files contain required records for HP processors. Do not delete these records. Instead, modify them to suit your needs and add any records required for additional cards or processors.

## Card Records

A card record is required for each card (for example, for data acquisition or instrument control) other than an HP Series 700 processor in the VME bus. Cards can be VME bus masters or slaves, and they may also supply local memory.

For third-party cards, if the supplier provides complete card records, you can append them to the end of **vme.CFG**, or include them with a **#include** *path-name* C pre-processor declaration, for example:

`#include` *`Card_Record_Filename`*

The fields in this record, for example,

```
//Keyword   Name    CPU/Card    Options
card        my742i  5
```

define a card named **my742i**, and assign it a CPU number of 5.

In the example above,the following fields apply:

card - Required - keyword indicating the start of a card record

Name - Required -

CPU/card number

opt_

## Memory Records

Memory records are required in the VME configuration file for all entities such as processors and cards that need VME address space. Memory records reserve a range of addresses for each processor or card, and specify the card or processor supplying the physical RAM or registers that the memory entity uses.

You may still need to set jumpers and switches on the card to correspond to the reserved memory address, but some cards are hard-coded for particular addresses, and others are programmable.

***NOTE:*** Before you specify the memory record, first specify a card record, as explained in "Card Records" on page 8-4.

The fields in the following record specify a memory address space, starting address, address range, name, and the VME board containing the memory:

```
memory A24 {
//Address        Name    Card/Proc   Options
//                       Name
0x800000:0x10    mem1    io_board
0x800010:0x10    mem2    io_board
}
```

The **address modifier** (shown above as *A24*) indicates which VME address space the object is to occupy. A16, A24, and A32 indicate all subspaces in their respective address space. They provide a convenient approximation for cards that occupy several but not necessarily all subspaces of a space. This field can be one or more of various names—see Table 8-1.

**Table 8-1 Names for Memory Address Modifier Declarations**

| Name | Name | Name | Explanation |
|---|---|---|---|
| A16[a] | A24 | A32 | Standard address modifiers, including all subspaces |
| A16L[a] | A24L[b] | A32L[b] | Lock cycles, for multi-card synchronization |
| | A24NB64 | A32NB64 | Non-privileged, 64-bit block transfer |
| A16N[a] | A24ND | A32ND | Non-privileged data space |
| | A24NB | A32NB | Non-privileged, 24- and 32-bit block transfer |
| | A24SB64 | A32SB64 | Supervisory, 64-bit block transfer |
| A16S[a] | A24SD | A32SD | Supervisory data space |
| | A24SB | A32SB | Supervisory, 24- and 32-bit block transfer |

a. Not valid for HP Models 742/747 and 743/744.
b. Not valid for HP Models 742 and 747.

The **address** (alignment) field (shown above as 0x800000:0x10) can be specified in the following formats:

- *starting-ending* addresses for cards that only operate at a certain address range. For example, 0x001000000-0x0001FFF means that the card occupies the address space between these two values. (Note that a hyphen separates the addresses.)
- *starting_address*:*size* for cards that can operate at a certain address range. For example, 0x00100000:8K means that the card occupies the 8 Kbytes of address space from 0x00100000 through 0x00101FFF. (Note that a hyphen separates the addresses.)
- **align** *num*:*size* (alignment and size) for processors that can be configured to their specific address boundaries. For example, **align 8K:8K** means that the card occupies 8 Kbytes of address space at *any* 8-Kbyte address boundary.

  **Align** lets you specify an address boundary requirement and range size instead of exact addresses. Any memory entities to be accessed through the slave mapper (see below) should have alignments that are integral multiples of 4 Kbytes.

  For HP-UX 74x*i* processors, if **align** is used for the direct or slave mapper, **vme_config** will assign a location and will place this location into the board's EEPROM so that VME Services will configure itself correctly. It is important that memory statements about all of the cards are put into your **.CFG** file so that **vme_config** does not accidentally assign memory that another card occupies.

The **name** field (shown above as *mem1*) specifies a name for this segment of memory.

The **card/proc name** (shown above as *io_board*) is the name of the previously declared card or processor that provides the physical memory, whether shared RAM, card registers, hardware FIFOs, etc.

An optional **access=***type* for HP-processor memory entities can specify **slave** and **direct_map_to** *address*, where

- **slave** (default) says that the processor's slave map is used to access the memory
- **direct_map_to** *address* indicates that the processor's A24 and/or A32 direct map window(s) are to be used to access this memory's entity, mapping it to the processor's RAM, starting at physical address *address*.
- **fifo**—To reserve space for the FIFO.
- **loc_mon**—To reserve space for the location monitor.

## Configuring Processor-Related Records

The processor-related records include:

- Processor records
- Interrupt records
- Slot 1 function records (optional)

***NOTE:*** Before you specify the processor-related records, first specify a card record, as explained in "Card Records" on page 8-4.

### Processor Records

A processor record is required in the VME configuration file for each CPU in the VME bus. This record contains the required fields for the **proc** keyword, the name of the processor, the CPU number (set to an asterisk "*" for third-party processors), and the revision code ID, as seen in the example below:

```
//Keyword    Name     CPU/          ID           Options
                      Card
proc         hp747i   0             ID_hp747
req_lv=BR3\

req_mode=RWD
```

The revision code ID uniquely identifies HP hardware for internal consistency checking. (For third-party processors, use ***** to specify that internal checking doesn't apply.) You can use the **cpp** C pre-processor to **#define** IDs at the top of the file, for example:

```
#define ID_hp743 0xd1
#define ID_hp748 0xd1
#define ID_hp742 0x51
#define ID_hp747 0x51
```

Several options are pertinent to processor cards:

- **keep_bus**—An optional decision to keep the bus or not.
- **pb_reset=***value*—Where a value of **true** (default) enables the push-button reset switch. (On HP Models 742 and 747, this option may not be set to **false**.)
- **req_lv**—An optional bus request level for VME transfers: **BR0**, **BR1**, **BR2**, or **BR3** (the default). This option defines the level at which a VME master or an interrupt handler requests the bus. Arbitration between these levels is specified by setting an arbitration mode—see "Slot 1 Function Records" on page 8-9. This value may be different in the **proc** record (which is for non-DMA transfers) than in the **dma_params** record.
- **req_mode**—An optional request: **RWD**, **ROR** (the default), or **FAIR**, specifying when a VME master releases the bus: when it's done (**RWD**), when it's done *and* another master requests the bus (**ROR**), or at the end of the current cycle (**FAIR**). (See "Releasing the VME Bus" on page 8-10 and "DMA Parameter Records" on page 8-11 for a slight difference in meaning for these values when configuring a DMA-capable board.)
- **vme_sysreset=***value*—Causes a VME SYSRESET signal to reset the processor (*value* defaults to **true**)

### Interrupt Records

Interrupt records assign interrupt request lines to processors. There can be only one CPU per interrupt line. For example, if processor 1 handles interrupt line 3, no other processor can handle line 3.

The interrupt record specifies a number or interrupt range or list, and the name of a processor or card. The fields in this record, for example,

```
interrupt {
    1-3,5   hp747i  // interrupt range and list
    4,7     hp742rt // interrupt list
    6       hp743rt // interrupt handler
}
```

specify that the first processor handles interrupt lines 1, 2, 3, and 5; the second handles lines 4 and 7, and the third processor handles interrupt line 6.

### Slot 1 Function Records

If Slot 1 records aren't preset, defaults are used. These records do not apply to third-party slot 1 controllers. They set up configuration parameters for the entire system, since Slot 1 is the arbiter board. The fields in this record, for example,

```
slot_1_functions timeout = 160, arb_mode =PRI
```

specify the:

- **time-out**—A value in microseconds (one of: 10, 20, 40, 80, 160, 320, 640, or 1000), defaulting to 160, for how long the bus will wait for a data transfer acknowledge before generating a bus error. Set this value as small as possible, but allow sufficient time for the slowest card in the system.
- **arb_mode**—The arbitration mode: **PRI (**the default) for priority arbitration,

**RRS** for round-robin. For **PRI**, request level **BR3** has the highest priority and is serviced first, followed by **BR2**, **BR1**, and **BR0,** which has the lowest priority. (See “req_lv” on page 8-8.) If **RRS**, the request rotates through all the levels, as described below in “Requesting the VME Bus,” below.

- **powerup_reset=***value*—Option (not shown above), defaulting to **true** to specify that the Slot 1 controller will assert the VME SYSRESET when it initializes its VME subsystem. This causes a reset on all the other VME cards. However, if some of them are active when the controller powers up, this will disable them.

## Requesting the VME Bus

You can request the level at which a VME master or an interrupt handler will request the bus, by setting the optional **req_lv=***value* parameter of a processor record to a *value* of **BR0**, **BR1**, **BR2**, or **BR3**. However, this is only a request, and how it is honored depends on the **arbitration mode**.

The VME bus requires an arbiter to mediate bus requests, since more than one device or interrupt handler might request the bus at the same level. Each card in the bus may have an arbiter, but only the arbiter on the system controller card in Slot 1 is used.

The arbitration mode defines how the system determines which request level to handle first. This can be set in the Slot 1 function record of the VME configuration file by using **arb_mode=***value*, where *value* is one of the following choices:

- **PRI**—Priority arbitrary (the default) gives the **BR3** request the highest priority, and other levels are serviced in the order **BR2**, **BR1**, and **BR0** when and if they are present.
- **RRS**—Round robin select gives each of the four request levels an equal priority, but it rotates through them in ascending numeric order, from **BR0** to **BR1**. If more than one card uses the same request level, the card in the slot closes to Slot 1 has priority because of hardware daisy chaining.

## Releasing the VME Bus

You can specify an optional **req_mode=***value* in the processor record on the Series 700, where *value* can be one of the following:

- **RWD**—Release When Done, the default, specifies that a VME bus master gets

the bus and finishes its task before releasing it so that another master can use it.

- **ROR**—Release On Request specifies that a VME bus master can finish its task and keep the bus until it's requested by another master. This mode may be useful if a particular master in the system is doing most of the work and frequently requests the bus.
- **FAIR**—Fair Request is used when there are multiple bus masters on a single request level, to specify that a VME bus master should release the bus at the end of the current cycle and wait to re-request it until no other masters (at the FAIR requester's level) need the bus. Thus, all masters on a single level have an equal chance to obtain the bus. If any requester on a level implements FAIR, all requesters on that level should implement FAIR.

### Combining Levels and Modes

Set the bus arbitration mode to **PRI** so that the most critical work gets done first. For instance, if you have two VME bus masters, and Master 1 is doing the more important task:

1. Assign Master 1 to the **BR3** request level
2. Set the bus arbitration mode to Priority
3. Set Master 1's request mode to Release on Request.

This ensures that Master 1 gets the highest priority request level (which the bus arbiter services first), and that it does not have to release the bus until another master requests it.

If you must put more than one bus master on the same request level, ensure that neither master is doing critical work, and use the FAIR request mode.

## DMA Parameter Records

The optional DMA parameter records specify non-default values for HP processors with VME DMA capability.

DMA parameter records are only needed for unusual situations, such as a system with more than five DMA-capable processors, each producing frequent, large DMA bursts. In such cases, bus-own time could be decreased, or wait time could be increased, to prevent starving other processes.

***NOTE:*** Before you specify the DMA record, first specify a card record and processor record, as explained in "Card Records" on page 8-4 and "Configuring Processor-Related Records" on page 8-7. There can be only one DMA record per processor.

The fields in the following record specify **dma_parms** in the required **proc** keyword field, the name of the processor (hp743i), how long to own the bus (40 microseconds), how long to wait before requesting it again (2 microseconds), and request level and mode options as described above.

```
// Proc Bus Own Wait Options
//          Name Time Time
//               (usec)  (usec)
dma_params       hp743i  40       2        req_lv=BR2 \
                                           req_mode=ROR \
                                           keep_bus=FALSE
```

Because long DMA transfers might hold the bus longer than other requesters can wait, DMA records include **Bus Own Time** and **Wait Time** values, which break the transfers into short bursts and give other cards an opportunity to request the bus. These and the optional fields can be specified as follows:

- **Bus Own Time**—From 0 to 51 microseconds, defaulting to 51: the maximum time that the DMA controller will hold the VME bus per DMA burst.
- **Wait Time**—From 0 to 51 microseconds, defaulting to 0: how long the DMA controller waits between bursts after having released the bus and before requesting it again.
- **keep_bus**—An optional decision to keep the bus or not. False by default, it can be set to **true** to keep the bus, slightly improving VME block transfer perfor-

mance (at the expense of all other subsystems**).**

- **req_lv**—An optional bus request level for VME transfers: **BR0**, **BR1**, **BR2**, or **BR3** (the default). This value may be different in the **proc** record (which is for non-DMA transfers) than in the **dma_params** record. This option defines the level at which a VME master or an interrupt handler requests the bus. If arbitration mode is set or defaulted to **PRI** (see "Slot 1 Function Records" on page 8-9), **BR3** has the highest priority level and is serviced first. If round robin mode is set in the Slot 1 function record (**RRS**), the request rotates though all the levels, from the highest to the lowest.
- **req_mode**—An optional request: **RWD**, **ROR**, or **FAIR**, specifying:
  - **RWD (Release When Done)**—That the DMA controller will request the bus, then transfer data until the specified **Bus Own Time** has elapsed, then release the bus, wait for **Wait Time**, and re-request the bus.
  - **ROR** (**Release on Request**)—That the DMA controller will again request the bus and transfer data until the bus ownership time has elapsed, but at that point, if there are no other requests for the VME bus, will continue to transfer data for another **Bus Own Time**. Otherwise, it will release the bus, wait for **Wait Time**, and re-request the bus. (**ROR** is set by default.)
  - **FAIR**—That the DMA controller will request the bus, transfer data until **Bus Own Time** has elapsed, and release the bus. After **Wait Time** has elapsed, it will continue to wait until there are no other requesters on the same bus request level, and then it will re-request the bus.

## Updating the Series 700's EEPROM

After you have edited the **vme.CFG** file, configure VME resources.

1. Log in as root.
2. Check the **vme.CFG** file for syntax errors without affecting the system configuration, by using the command:

   **/sbin/vme_config -c**

3. Use the following key sequence:

   ***/error_number***

to search for specific error messages by number in the **vme_config** man page. (Use **man vme_config** to bring up the man page, then search.)

**4** After fixing any errors in the **vme.CFG** file, type the following command:

**/sbin/vme_config**

By default, this reads configuration information from **/etc/vme/vme.CFG (**which is normally a link to **/sbin/lib/vme/vme.CFG**) and writes to the non-volatile memory of the processor on which it is executed. If you have multiple HP processors, you must run **vme_config** on each of them.

***NOTE:*** The file **/var/adm/vme/system.log** contains a description of your current configuration.The new configuration will not take effect until you have rebooted the system.

## vme_config Options

Options to the **vme_config** command let you check the configuration file syntax for correctness, specify an alternate file, and view memory assignments—again, see **vme_config(**1M**)**. These options are as follows:

- **-f** *filename*—For specifying an alternate configuration file. For example:

  ```
  /sbin/vme_config -f example.CFG
  ```

- **-c**—For checking the **vme.CFG** file for correct syntax without affecting the current configuration. For example, to check **file.CFG**:

  ```
  /sbin/vme_config -cf file.CFG
  ```

- **-m** *address_modifier*—For viewing the location of all memory entities in the specified memory space. *address_modifier* can be any entry from Table 8-1 on page 8-6. For example:

  ```
  /sbin/vme_config -m A16
  ```

- **-h** *address_modifier*—For displaying the empty locations in the specified memory space. For example:

  ```
  /sbin/vme_config -h A16
  ```

- **-e** *entity_name*—For viewing the attributes of the specified entity. *Entity_name* is the name of a processor, card, or memory entity declared in the configuration file. For example:

  ```
  /sbin/vme_config -e i7fifo
  ```

- **-N**—For displaying the CPU/card number of an HP processor.

- **-N** *number*—For changing the CPU/card number of an HP processor, e.g,

  ```
  /sbin/vme_config -N 2
  ```

- **-D** *define*—For defining a label to be used in a **#ifdef** statement in the configuration file for **cpp** preprocessor commands.

- **-U** *define*—For un-defining previously defined labels.

# 9

# Backplane Networking at HP-UX 10.20 and later releases

At HP-UX 10.20, BPN (backplane networking) provides two communications interfaces for the Models 743 and 744 (it is not available for Models 742/747):

- TCP/IP, providing typical ARPA services such as **ftp** and **telnet**, as well as interprocess communication via Berkeley socket programming and the AF_INET address family. TCP/IP provides services between one or more hosts on the backplane and hosts on Ethernet networks external to the backplane.
- VME BPN pipes between BPN hosts on a single VME backplane, via Berkeley socket programming using AF_VME_LINK, BPN's new socket domain and address family that is several times faster than AF_INET.

**To Install BPN:**

1. Follow the instructions in the next section to install the fileset for each processor (see "Installing the HP-UX BPN Fileset" on page 9-3).
2. Add **bpn** to the **system** file and rebuild the kernel (see"Editing System Files for the Driver" on page 7-6 and "Building the Driver into the Kernel" on page 7-9).
3. Edit the VME configuration file on each processor (see "Configuring the Shared Memory Area" on page 9-4) to configure:
   - A single shared memory area if using TCP/IP
   - Slave-mapped I/O memory records on each processor using VME BPN pipes
4. Run **vme_config** on each processor using the identical configuration file.
5. Run **bp_config** on each processor that will use TCP/IP (see "Running the bp_config Configuration Utility" on page 9-10).
6. Reboot each processor—for TCP/IP use, reboot after specifying each host's TCP/IP address in its **/etc/hosts** file (see "Specifying TCP/IP Addresses" on page 9-8).

**To Use TCP/IP:**

1. Use **ifconfig** to bring up the backplane network (see "Bringing up the Backplane Network for TCP/IP" on page 9-11).
2. Use standard ARPA utilities—or socket programming with AF_INET as the address family (see "Using Berkeley Socket Communications" on page 9-12).

**To Use VME BPN Pipes:**

1. Use HP-UX socket programming and BPN's new AF_VME_LINK socket domain to send and receive data between hosts on the backplane (see "Using Berkeley Socket Communications" on page 9-12).

## Installing the HP-UX BPN Fileset

Before configuring the BPN resources, you need to install the fileset that provides the communication layers between remote processes.

1. Log in as root on your Model 743/744.
2. Check that VME Services software is installed, by following the install and verification steps described in “Installing the VME Product” on page 7-2.
3. Check to see if the fileset for backplane networking is already installed, by entering the command:

   **/usr/sbin/swlist VME-Services.VME-BPN**

   If the product is already installed, you will see an appropriate listing as confirmation; otherwise a message indicates that the product was not found.
4. If the product is not installed, run the **swinstall** utility to add the fileset and reboot:

   **/usr/sbin/swinstall VME-Services.VME-BPN**

   Accept the default parameters that **swinstall** provides, unless you want to change the depot from the default **/var/spool/sw**, to another depot—such as **/dev/rmt/0m** (DAT tape) or **/mnt/cd** (CD-ROM mount point)—supported by SD, the HP-UX Software Distributor. For more information, see **swinstall**(1M).

## Configuring VME Memory for BPN Communication

Whether you choose to use communications via VME BPN pipes or TCP/IP, you need to:

- Configure a shared memory area for each VME BPN pipes host
- Configure a single memory area shared by all VME hosts for TCP/IP
- Update each host’s EEPROM by running **vme_config** (and **bp_config** for TCP/IP**)**
- Reboot for the configuration to take effect

## Configuring the Shared Memory Area

You should have the same configuration file for each processor in the backplane. To edit this file for a proper configuration, follow these steps:

1. Edit a *vme.CFG* file as explained in "vme.CFG, the VME Configuration File" on page 8-2, and the sections following that, so that all the processors and cards in the backplane are described appropriately for your HP-UX VME configuration.
2. Set **powerup_reset=**FALSE in the Slot 1 Function record, so that powering up the system controller does not reset other cards in the VME backplane and thereby disable them—see "Slot 1 Function Records" on page 8-9.
3. Set **vme_sysreset=**FALSE in the Processor Declaration record for Model 743 and 744 systems, so that the VME SYSRESET signal doesn't reset them.
4. If your configuration file does not include a section titled "Backplane Networking Shared Memory Declarations," you may want to copy and edit that portion from **/etc/vme/example4.CFG**.
5. Edit the "Backplane Networking Shared Memory Declarations" record to specify the memory's starting address, size, associated address modifier(s), and so on, as displayed in the following extract and described in the subsections below.

### Specifying Address Modifiers and Memory Locations

The **address modifier** (shown below as the default *A24SD . . .*) indicates which VME address space the object is to occupy—see the table "Names for Memory Address Modifier Declarations" on page 8-6.

***NOTE*** For VME BPN pipes using the AF_VME_LINK socket domain, all three address modifier types(AxxSD, AxxSB, and AxxSB64, where xx is 24 or 32) must be specified for best performance.

The **address entry** is expressed as *starting_address:size* to specify the location and size of the shared memory areas.

### Configuring Shared Memory for TCP/IP Communications

For TCP/IP communications, the shared memory must reside on a memory card or on CPU 0, and be no larger than 1 MB. The default address is 0x200000 in A24_SUP_DATA_ACCESS. (For A24 space, the starting address is truncated by removing the upper byte.)

```
#define SHM_ADDR    0x200000    /* edit as appropriate */
#define SHM_SIZE    0x50000     /* edit as appropriate */

#define BP_SIZE     0x21000     /* do not change BP defines */
. . .
#define BP_Meg1     0x1DF000    /* do not change */
#define BP_Meg2     0x2DF000    /* do not change */
. . .
#define BP_MegF     0xFDF000    /* do not change */


memory A24SD A24SB A24SB64 {
//  address         Name          Card or Proc   Options
//                                Name
SHM_ADDR:SHM_SIZE   bpSharedMem   hp748i3        opt_bpn_rmw=

BP_Meg6:BP_SIZE     bphp743i1     hp743i1
BP_Meg7:BP_SIZE     bphp743i2     hp743i2
BP_Meg2:BP_SIZE     bphp748i3     hp748i3
}
```

To calculate the shared memory size for **bpSharedMem**, use the formula:

$$Size = 2144 + (168 \times num_CPUs) + (2208 \times Slots_in_Free_Ring)$$

where $2 \le NumCPUs \le 32$ and $16 \le SlotsInFreeRing \le 256$ (in powers of 2).

- **Name** must be **bpSharedMem** (the compiler is case-sensitive).
- **Card or Proc Name** must be the same as specified for **Name** in the Card Record, and it must be for CPU 0, unless a memory card provides the memory. (See “Card Records” on page 8-4.)
- ***RMW* Options** must be specified by setting **opt_bpn_rmw** to:
  - TRUE_RMW if the memory is on a Model 743 or 744 computer, or on a VME memory card that can accept VME RMW cycles
  - PSEUDO_RMW if the memory is on a memory card that can not accept VME RMW cycles

### Configuring Shared Memory for VME BPN Pipes

Each processor that supports VME BPN pipes using the AF_VME_LINK socket domain must declare slave memory at the top of its own 1-MB space chosen from the first 16 MB of address space.

- The same 1-MB range must not be chosen by two processors.
- The **bpSharedMem** area may occupy one of the 1-MB ranges—provided that it doesn't overlap with the AF_VME_LINK memory at the top of the range.
- Address modifiers A24SD, A24SB, and A24SB64 must all three be specified for best performance (or their A32 variants A32SD, A32SB, and A32SB64 if A32 space is preferred).
- The memory declarations in the example file assume that the tunable value for **vmebpn_public_pages** is set to its maximum value of 32 (see below). It is best to use the declarations as they appear in the **example4.CFG** file, even if fewer pages are actually declared.

## Tuning VME BPN Pipes

You can specify several "tunable" parameters for your backplane network. See the file **/usr/comfy/master.d/vmebpn** for a listing of the defaults for these four parameters (which are described below). To change the defaults, you can choose one of the following methods:

- Edit a build version of the **system** file, as in **/stand/build/system** (see "Editing the Build File" on page 7-7) and then rebuild the kernel (see "Building the Driver into the Kernel" on page 7-9).
- Use SAM to specify the parameters and create a new kernel (see"Choosing a Kernel Configuration File" on page 7-4).

The tunable parameters are as follows:

- **vmebpn_sockets**—Specifies by its default value of 1 that you will use VME BPN pipes for communications between HP processors on the same backplane. Set it to 0 if you do not wish to use VME BPN pipes.
- **vmebpn_total_jobs**—Specifies how many pipes you may open concurrently. This defaults to 16, but may be set up to a maximum of 8092.
- **vmebpn_public_pages**—Specifies how many 4-KB pages will be reserved by the slave mapper on each HP processor for VME BPN pipes. This defaults to 1 and may be assigned a maximum of 32. Tune larger if you have many pipes send-

ing small chunks of data.

- **vmebpn_tcp_ip**—Specifies by its default value of 1 that you will use TCP/IP communications between HP processors on the VME backplane and other host machines on a connected Ethernet LAN. Set it to 0 if you do not wish TCP/IP.

The values for the total number of pipes (**vmebpn_total_jobs**) and the number of public pages (**vmebpn_public_pages**) interact to affect the overall performance of communications. In general, the more concurrently open pipes that you expect, the higher you should set your number of public pages, unless all pipes are calling **send**() or **recv**() with a buffer length of one page or larger in size.

## Establishing TCP/IP Communications

TCP/IP communications are based on *de facto* standard Ethernet protocols incorporated into 4.2BSD Unix. Although TCP and IP specify two protocols at specific layers, TCP/IP is often used to refer to the entire protocol suite based upon these, including **telnet**, **ftp**, UDP, and RDP.

TCP/IP on the HP-UX BPN enables multi-vendor processors to communicate with one another using the usual TCP/IP networking method of communication. The backplane networking driver (**bp0**) provides and acts like the ethernet driver (**lan0**) running on top of the physical LAN.

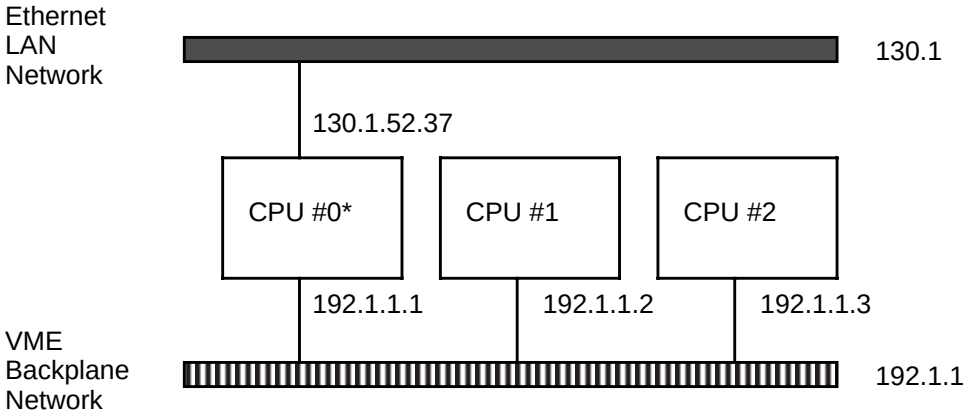
For TCP/IP communications across the BPN and for communications between HP-UX hosts in the BPN and TCP/IP hosts elsewhere on a connected LAN, you need to:

- Configure a system-wide, shared-memory area named **bpSharedMem** (as explained above—see “Configuring the Shared Memory Area” on page 9-4)
- Configure IP addresses and host names into the **/etc/hosts** file (see below)
- Run the **bp_config** configuration utility (as explained in “Running the bp_config Configuration Utility” on page 9-10)
- Bring up the backplane network for TCP/IP (as explained in “Bringing up the Backplane Network for TCP/IP” on page 9-11)

## Specifying TCP/IP Addresses

The VME backplane forms a unique subnet, with the backplane itself assigned a unique Internet network number, and each processor a unique Internet Protocol (IP) address, much as if the backplane were a LAN with multiple processors.

In the example shown in Figure 9-1, the Ethernet LAN has a Class B network number of 130.1; and the VME backplane has a Class C network number of 192.1.1. CPU Number 0 was arbitrarily assigned an IP address of 192.1.1.1. Then, CPU Numbers 1 and 2 were assigned incremental IP addresses of 192.1.1.2 and 192.1.1.3.




* CPU #0 is the master processor and gateway to the LAN.

**Figure 9-1** **Example VME Backplane Network**

The Ethernet LAN has a Class B network number of 130.1; and the VME backplane has a Class C network number of 192.1.1. CPU Number 0 was arbitrarily assigned an IP address of 192.1.1.1. Then, CPU Numbers 1 and 2 were assigned incremental IP addresses of 192.1.1.2 and 192.1.1.3.

Frequently, the backplane master processor (CPU Number 0) has two IP addresses—one on the Ethernet LAN (e.g., 130.1.52.37) and one on the backplane network (e.g., 192.1.1.1). The master processor thus can act as a

gateway between the VME backplane network and the LAN. The other CPUs have only VME backplane networking IP addresses, but they can use CPU 0 as a gateway to communicate with any system on the LAN.

To specify TCP/IP addresses, see the man page for **hosts**(4), and follow these steps:

1 As root, edit the **/etc/hosts** file on each processor in the BPN (Backplane Network), and each in the LAN, so that each copy contains information regarding all the known hosts on the network, as suggested by the following example.

```
#
# Internet Address  Hostname    Alias    # Comments
#
127.0.0.1           localhost   loopback # Required

130.1.52.37         LAN-Gateway # CPU 0's IP addr on the LAN
192.1.1.1           BPN-Gateway # CPU 0's IP addr on the BPN
192.1.1.2           Client1     # CPU 1's IP addr on the BPN
192.1.1.3           Client2     # CPU 2's IP addr on the BPN
memory A24SD {
```

2 For each host on the BPN, specify a single line with the following information:

- **Internet address**—A network IP address that uniquely identifies the node and is not in use or accessible on the LAN. Don't end the address in 0 or 255 (e.g., 130.1.1.255).
- **Official host name**—The unique name of the node.
- **Aliases**—Optionally, one or more unique common names for the node.

3 For the single BPN master processor (the slot controller, a.k.a. CPU Number 0), specify:

- An IP address to identify the processor to the LAN.
- A hostname unique to both the LAN and the BPN.

4 The information in **/etc/hosts** doesn't take effect until you reboot each host.

If your HP-UX system uses the Internet Domain naming environment, the default is to use the hostname in the Domain name server. If the system does not find the hostname there, it searches for the hostname in the **/etc/hosts** file. For more information, see the **hosts**(4) and **named**(1M) man pages.

### Running the bp_config Configuration Utility

You need to run the **vme_config** utility regardless of the type of backplane communications. However, to support TCP/IP communications, you also need to run **bp_config** to add to **vme_config**'s EEPROM information for each processor in the backplane network. If you don't run **bp_config**, shared memory variables default to a starting address of 0x200000, a size of 36 pages (0x24000), and an address modifier of STD_SUP_DATA_ACCESS.

On each processor, re-update the EEPROM by running **bp_config**(1M), which reads the information that you have previously specified (see "Configuring the Shared Memory Area" on page 9-4),

**/sbin/bp_config**

By default, bus interrupts are assumed (when available), unless you have previously run **bp_config -poll**. To restore bus interrupts in case you have previously specified the **-poll** option, use the command

**/sbin/bp_config -bus**

Both **bp_config** and **vme_config** output the same messages, except that **bp_config** adds a line of information about the shared memory's location, size, and type, and a confirmation of the current CPU number, for example:

```
Generating configuration for processor: hostname
CPU/Card number: 0
Anchor = 0x200000, Size = 0x50000, AM Code = 0x3d, CPUs = 3
Overwrite system eeprom data (Y/N):Y

Configuration successfully generated.
Configuration saved in /etc/vme/system.log
```

For help resolving configuration error messages, see **bp_config(**1M) and use the man page viewer's search facility— a slash "/" followed by the error number.

## Bringing up the Backplane Network for TCP/IP

Follow the steps below to bring up the backplane network and set up system files so that it will be brought up automatically when the system is later rebooted.

**1** Activate backplane networking by entering:

**/usr/sbin/ifconfig bp0 *IP_address* up**

where:

- **bp0**—Names the network interface, the backplane networking driver.
- *IP_address*—The Internet address, expressed in Internet standard “dot notation.”
- **up**—Marks the backplane networking driver as active.

For example:

**/usr/sbin/ifconfig bp0 192.1.1.1 up**

**2** To see the result of Step 1, execute the following command:

**/usr/sbin/ifconfig bp0**

The output will look similar to:

bp0: flags=63<UP,BROADCAST,NOTRAILERS,RUNNING>
inet 192.1.1.1 netmask ffffff00 broadcast 192.1.1.255

Look for the words `UP` and `RUNNING` in the output. Check that the IP address is correct. In this example, the IP address is 192.1.1.1.

To bring up the network automatically when the system is booted, edit the **/etc/rc.config.d/netconf** file as described below.

**1** Locate the Internet configuration parameters for the LAN. They look like this example entry:

```
INTERFACE_NAME[0]=lan0
IP_ADDRESS[0]=“130.1.52.37"
SUBNET_MASK[0]=“255.255.255.0"
BROADCAST_ADDRESS[0]=“”
LANCONFIG_ARGS[0]=“ether”
```

**2** Copy the *first four lines* of the LAN entry and insert them below the LAN entry. Do *not* copy the last line of the LAN entry.

**3** Make the changes shown in this example:

```
INTERFACE_NAME[1]=bp0
IP_ADDRESS[1]=“192.1.1.1"
SUBNET_MASK[1]=“255.255.255.0"
BROADCAST_ADDRESS[1]=“”
```

where **bp0** is the backplane networking interface name, and the [1] indicates that **bp0** is the second networking interface.

(Change the example IP address to the actual IP address you are using for backplane networking.)

The **ifconfig** utility or the **netconf** startup file needs to be run for each HP-UX system that needs to bring up BPN. Be sure that you use a unique backplane networking IP address for each HP-UX system.

## Using Berkeley Socket Communications

VME BPN supports both the AF_INIT TCP/IP socket domain and a new, high-performance, Unix-like socket domain that includes a new address family and communications protocol. This new domain, AF_VME_LINK, appears to socket programmers as a cross between the:

- UNIX domain, which supports pipes between applications running on a single processor, and the
- INET domain, which supports connections (TCP, UDP, etc.) between applications typically running on different processors.

The AF_VME_LINK domain establishes pipe-like connections for applications typically on different 743/744 systems connected by a VME backplane. It takes maximum advantage of the 743/744 DMA engines, performing more like single-processor UNIX domain communications than TCP/IP.

This new protocol provides process-to-process communication via simple, standard socket programming using calls to **socket**(2), **bind**(2), **accept**(2), **connect**(2), **listen**(2), **recv**(2), and **send**(2). It appears to programmers very much like the AF_UNIX protocol, except that it can connect listeners on up to 16 different processors on the VME backplane.

## Client/Server Connection-Oriented Protocol

The AF_INIT and AF_VME_LINK domains support a connection-oriented transfer in which a server and client(s) communicate using the Berkeley socket calls (see Figure 9-2).

The **server**:

- Starts up and issues a sequence of socket calls (**socket**, **bind**, **listen**)
- Forks a child process that blocks, listening for a client's **connect** request before issuing an **accept**, and then **read** and **write** requests before closing

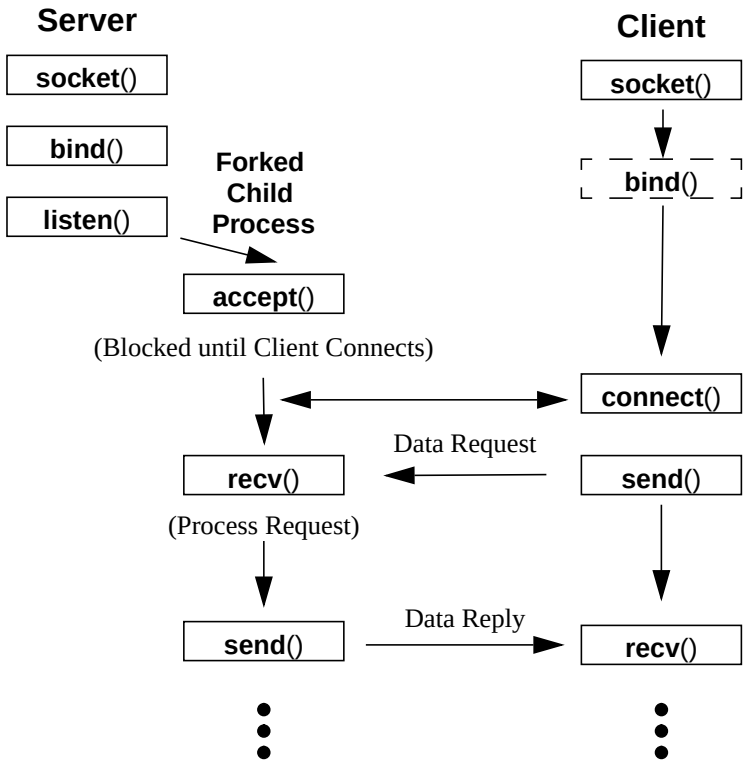



**Figure 4-2** **Client/Server, Connection-Oriented Protocol**

The **client** in the transfer:

- Starts up and issues a **socket** call
- Can optionally issue a **bind**(), specifying Port 0
- Issues a **connect** call to the socket address that identifies the server's service
- Issues **read**s and **write**s to communicate to transfer the data
- Uses **close** to end the connection

## Socket-Related Structures

AF_VME_LINK identifies the address family and protocol for socket communication across a VME backplane. This form of communication, called **VME BPN pipes**, requires the installation of the VME Services product, **vme2**, and the specification during the installation of both VME-KRN and VME-BPN filesets for the VME and BPN drivers.

- This address family uses the SOCK_STREAM socket type.
- AF_VME_LINK is also the socket protocol for this address family.

Many of the supported socket calls require a pointer to a socket address structure, which is defined in the file <**sys/socket.h**>:

```
struct sockaddr {
u_short sa_family; /* address family: AF_VME_LINK */
char sa_data[14]; /* protocol-specific address */
}
```

For the AF_INET and AF_VME_LINK address families, the contents of the **sa_data** area are specified by two structures defined in <**netinet/in.h**>:

```
struct in_addr {
u_long s_addr;
}
```

and

```
struct sockaddr_in {
short sin_family;       /* AF_INET or AF_VME_LINK */
u_short sin_port;               /* Port number */
struct in_addr  sin_addr; /* IP address or CPU number */
char sin_data[8];               /* Unused */
}
```

### The socket() Call

The **socket**(2) system call creates an endpoint for communication and returns a descriptor that is used in all subsequent socket-related system calls. The **socket** call has the following form:

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

int socket (af, type, protocol)
    int af;  /* addr family: AF_VME_LINK or AF_INIT */
    int type;   /* only SOCK_STREAM for AF_VME_LINK */
    int protocol;   /* set to 0 */
```

where:

- *af*— Specifies an address family to be used to interpret addresses in later operations that specify the socket. The only currently supported address families for VME BPN are:
  - AF_INET—DARPA Internet addresses for TCP/IP communication
  - AF_VME_LINK—backplane communications on VME bus
- *type*—Specifies the semantics of communication for the socket; either SOCK_STREAM, which provides sequenced, reliable, two-way-connection-based byte streams, or SOCK_DGRAM sockets (for AF_INET only).
- *protocol*—Specifies a particular protocol to be used with the socket. For VME BPN, set this to zero.

If the **socket** call fails, it returns -1 and sets **errno** to one of the values that you can find in **man socket**. As you can see from the figure “Client/Server, Connection-Oriented Protocol” on page 9-13, both the client and server make a call to **socket**().

### The bind() Call

The **bind**() call binds an address to an unbound socket. (It must be called by servers, but clients of AF_VME_LINK can omit this call or bind with a port number of 0, as described below. When a socket is created with **socket**(), it exists in an address space (address family) but has no address assigned.

```
#include <sys/socket.h>
#include <netinet/in.h> /* AF_INET & AF_VME_LINK only */
```

```
int bind (sock_desc, *addr, addrlen)
    int sock_desc;      /* socket descriptor */
    const void *addr;   /* socketaddr_in struct */
    int addrlen;        /* size of the struct */
```

**bind**() causes the socket whose descriptor is *sock_desc* to become bound to the address specified in the socket address structure pointed to by *addr.*

*addrlen* must specify the size of the address structure, which, because it varies with each family, should be **sizeof(struct *sockaddr_in*)** for AF_INET and AF_VME_LINK.

When binding an AF_INET or AF_VME_LINK socket, the **sin_port** field in the *sockaddr_in* struct can be a port number or it can be zero. If **sin_port** is zero, the system assigns an unused port number automatically.

For AF_VME_LINK servers, port numbers in the range 0-4095 are "well-known" ports that may be associated with services, as requested by the **bind**() call.

The **sin_addr** field in the *sockaddr_in* struct must be an IP address for AF_INET or a CPU number for AF_VME_LINK. In the latter case, the CPU number may be 0-31, with -1 also available for **connect**() calls as explained below.

**bind**() returns 0 on success, and -1 on failure, with **errrno** set to the values shown in **man bind**.

### The listen() Call

The **listen**() call listens for connections on a socket. It is only used by servers, and it has the form:

```
#include <sys/socket.h>
int listen (sock_desc, backlog)
    int sock_desc;      /* socket descriptor */
    int backlog;        /* no. pending connects */
```

*backlog* defines the desirable queue length for pending connections. If a connection request arrives when the queue is full, the client will receive an ETIMEDOUT error. See **man listen** for more information on the values that can be assigned to *backlog*.

Sockets must be bound to an address by using **bind**() before connection establishment can continue, otherwise an EADDREQUIRED error is returned.

### The accept() Call

After the server issues the **listen**() socket call, it forks a process that issues an **accept**() call and blocks until it receives a **connect**() request from a client (see “Client/Server, Connection-Oriented Protocol” on page 9-13).

This call takes the following form:

```
#include <sys/socket.h>
int accept (sock_desc, addr, addrlen)
    int sock_desc;
    void *addr;
    int *addrlen;
```

The argument, *sock_desc*, is a socket descriptor created with **socket**(), bound to a local address by **bind**(), and listening for connections after a **listen**(). **accept**() extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *sock_desc*, and returns a new file descriptor, *new_sock_desc*, for the socket.

The argument *addr* should point to a socket address structure. The **accept**() call fills in this structure with the address of the connecting entity, as known to the underlying protocol.

*addrlen* is a pointer to an **int** that should initially contain the size of the structure pointed to by *addr*. On return, it contains the actual length (in bytes) of the address returned.

### The connect() Call

When a **connect**() is issued by a client to connect to a server’s services, the port number and CPU number are both specified by the **connect**(). For AF_VME_LINK, a wildcard CPU number of -1 results in a connection to the first responding CPU that supports the requested port number.

This call has the form:

```
#include <sys/socket.h>
#include <netinet/in.h>
```

```
int connect (sock_desc, addr, addrlen)
    int sock_desc;
    const void *addr;
    int addrlen;
```

- *sock_desc*—A socket descriptor
- *addr*—A pointer to a socket address structure containing the address of a remote socket to which a connection is to be established
- *addrlen*—The size of this address structure, typically **sizeof(struct *sockaddr_in*)**

If the socket does not already have a local address bound to it, **connect** binds the socket to a local address chosen by the system. (For AF_VME_LINK, this local address is in the range 4096-8191.) If **bind**() was called previously, then the *sockaddr_in*->**sin_port** value must have been 0.

## The recv() and recvmsg() Calls

These calls receive messages from the socket and have the following forms:

```
#include <sys/socket.h>
int recv (sock_desc, buf, len, flags)
    int sock_desc;
    void *buf;
    int len;
    int flags;
int recvmsg (sock_desc, msg, flags)
    int sock_desc;
    struct msghdr msg[];
    int flags;
```

- *sock_desc*—The socket descriptor from which messages are received.
- *buf*—A pointer to the buffer into which the messages are placed.
- *len*—The maximum number of bytes that can fit in the buffer.
- *flags*—If no data is available to be received, **recv**() and **recvmsg**() wait for a message to arrive unless nonblocking mode is enabled by the flags O_NONBLOCK or O_NDELEY (see **man recv**). The flags parameter can also be set to MSG_PEEK or zero. If it is set to MSG_PEEK, any data returned to the user still is treated as if it had not been read. The next **recv**() rereads the same data.
- *msg*—**recvmsg**() performs the same action as **recv**(), but scatters the read data into the buffers specified in the *msghdr* structure (see **man recv**).

**recv**() and **recvmsg**() can only be used after the connection has been established.

### The send() and sendmsg() Calls

These calls send a message to a socket. They have the form:

```
#include <sys/socket.h>
int send (sock_desc, msg, len, flags);
    int sock_desc;
    const void *msg;
    int len;
    int flags;
int sendmsg (sock_desc, msg[], flags);
    int sock_desc;
    const struct msghdr msg[];
    int flags;
```

The **send**() and **sendmsg**() calls transmit a message to another socket. These calls can only be used after the connection has been established. **sendmsg**() allows the send data to be gathered from several buffers specified in the *msghdr* structure. See **man recv** for the parameter descriptions, and **man send** for the *flags* parameter.

For **send**(), the length of the message is given by *len* in bytes, and the function returns the number actually sent. For AF_INET, if the message is too long to pass atomically through the underlying protocol, the message is not transmitted, and -1 is returned, with **errno** is set to EMSGSIZE. For AF_VME_LINK, there is no size limitation.

When **send**() returns a positive value, it only indicates this number of bytes have been sent to the local transport provider. It does not mean they have been delivered to the peer socket application. Eventual end-to-end delivery is guaranteed, but if the underlying transport provider later detects an irrecoverable error, it will return a value of -1 at another socket function call.

### The shutdown() Call

This call shuts down a socket. It has the form:

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

```
int shutdown (sock_desc, how);
    int sock_desc;
    int how;
```

*how* can be one of the following:

- SHUT_RD (disallow further receives)
- SHUT_WR (no further sends)
- SHUT_RDWR (no further sends or receives).

Once the socket has been shut down for receives, all further **recv**() calls return an end-of-file condition. A socket that has been shut down for sending causes further **send**() calls to return an EPIPE error and send the SIGPIPE signal. After a socket has been fully shut down, operations other than **recv**() and **send**() return appropriate errors, and the only other thing that can be done to the socket is a **close**().

Multiple shutdowns on a connected socket and shutdowns on a socket that is not connected may not return errors.

For a *how* of SHUT_WR or SHUT_RDWR, the connection begins to be closed gracefully in addition to the normal actions, but **shutdown**() call does not wait for the completion of the graceful disconnection. The disconnection is complete when both sides of the connection have done a **shutdown**() with a *how* of SHUT_WR or SHUT_RDWR. Once the connection has been completely terminated, the socket becomes fully shut down.

The SO_LINGER option—see **socket**(2)—does not have any meaning for the **shutdown**() call, but does for the **close**() call. For more information on how the **close**() call interacts with sockets, see **socket**(2).

If the socket has a **listen**() pending on it, it becomes fully shut down when *how* is SHUT_WR.

# A

# Structures

This appendix presents a quick-reference overview of the structs used in writing a VME device driver. The structs provided here are from the latest HP-UX release. Check the structs in your operating system as there are differences depending on the release.

Note that most of the structs whose name begins with "**vme_**" are declared in **/usr/conf/wsio/vme2.h**, which needs to be included in your prolog:

```
#include /usr/conf/wsio/vme2.h
```

## *bdevsw*

Each block device driver needs a *bdevsw* entry in the **bdevsw** table, which is declared in **/usr/conf/h/conf.h** and looks like this:

```
struct bdevsw

{

    d_open_t        d_open;
    d_close_t       d_close;
    d_strategy_t    d_strategy;
    d_dump_t        d_dump;
    d_psize_t       d_psize;
    int             d_flags;
    int             (*reserved) __(());
    void            *d_drv_info;
    pfilter_t       *d_pfilter_s;
    aio_ops_t       *d_aio_ops;
};
```

The first five fields are pointers to routines within a device driver. The **d_flags** field is defined in the same way as in the **cdevsw** table (see below).

Implement all functions in the **bdevsw** structure appropriate for the device being controlled. Functions not implemented by your driver must contain **nulldev** or **nodev** in the appropriate device switch table entry; and **d_flags** must contain C_ALLCLOSES and/or C_NODELAY, or 0.

## *buf*

The *buf* data structure describes a system buffer and the operations performed on it.

A pointer to a *buf* structure is passed as a parameter to the driver_strategy routine, which should schedule the transfer of data into or out of the buffer pointed to by the *buf* structure.

Letting your driver modify some fields in a file system buffer header could corrupt the buffer cache, and possibly the file system. In the descriptions that follow, note which ones are free to be used by drivers and which ones the system expects to be modified by drivers.

The *buf* structure is defined in **/usr/conf/h/buf.h**:

```
struct buf
{
    long b_flags;
    struct buf *b_forw, *b_back;/* 2-way hash chain */
    struct buf *av_forw, *av_back;/* free list pos. */
    . . .
    long b_bcount;      /* transfer count */
    long b_bufsize;     /* size of allocated buffer */
    . . .
    short b_error;      /* returned after I/O */
    . . .
    dev_t b_dev;        /* major+minor device name */
    union {
        caddr_t b_addr; /* low order core address */
            /* (remainder of address is in b_spaddr) */
    . . .
    } b_un;
    struct isc_table_type *b_sc;/* select code ptr */
    int (*b_action)();  /* only WSIO: next activity */
    . . .
    struct iobuf *b_queue; /* which IO queue it's on */
    long b_s2;             /* scratch area */
    char b_s0;             /* scratch area */
    char b_s1;             /* scratch area */
    char b_s3;             /* scratch area */
```

```
    char b_ba;              /* bus address */
    unsigned int b_resid;   /* bytes not transferred */
    . . .
    daddr_t b_blkno;        /* block # on device */
    . . .
    int (*b_iodone)();      /* called by iodone */
}
```

There are many fields in this struct, but these are the most important ones:

- **av_forw**, **av_back**—Two available or freelist pointers. The freelist is a list of available buffers that can be reused.
- **b_action**—Used by the Finite State Machine mechanism. It contains the address of the routine containing the FSM. The select code management and the DIO DMA routines use this field. This field is free for other use if none of these features is used.
- **b_ba**—Can be the bus address of the device associated with this *buf* structure. This value can be used by the driver as it sees fit.
- **b_bcount**—Indicates the number of bytes to be transferred. The driver strategy routine should get its buffer count from this field (see the code in the section "Performing Input from a Device" on page 6-9 and "Performing Output to a Device" on page 6-10).
- **b_blkno**—Specifies the location of the buffer in terms of the number of DEV_BSIZE units, starting from the beginning of the device. (The type **daddr_t** is declared as a long disk address in **types.h**.)
- **b_bufsize**—Specifies the size of the buffer. The buffer size can range from zero to four pages.
- **b_dev**— Indicates the device associated with this particular buffer. This field is set by the kernel **physio** routine. When the buffer has no particular associated device, the field is set to NODEV. (The type **dev_t** is declared in **types.h** as a long.)
- **b_error**— Contains a zero if there are no errors. If the **b_flags** B_ERROR bit is set, then **b_error** contains the error number. This field can be set by a driver to indicate when an error has occurred. **Physio** returns this value to the caller if it was set by the driver's strategy routine.
- **b_flags**—Contains information about the state of the buffer. The flags contained in **b_flags** include:
  - B_ASYNC—Do not wait for I/O completion.
  - B_BUSY—Indicates the buffer is being used. If this flag is set, no process ex-

cept the process owning the buffer should access the buffer.

- B_CALL—Indicates the function contained in **b_iodone** should be called by **iodone**().
- B_DELWRI—Indicates a delayed write.
- B_DONE—Indicates the I/O transaction on this buffer has completed. The **iodone**(2) routine sets this flag.
- B_ERROR— Indicates an error occurred when the driver attempted to satisfy this request. When this flag is set, data in this buffer is not valid. Set this flag in the driver's strategy routine before returning to **physio** if an error occurred during I/O.
- B_PHYS—Indicates physical I/O is in progress. This bit is always set if a driver uses **physio** to get the buffer.
- B_READ—Indicates the buffer is being used for a read request.
- B_WANTED—Issues a wakeup on clearing the B_BUSY flag.
- B_WRITE—A non-read, pseudo flag indicating current non-use of the buffer for a read request. Since the value of B_WRITE is 0, do *not* use a test such as if (bp->b_flags & B_WRITE); instead, use if (!(bp->b_flags & B_READ)).

- **b_forw**, **b_back**—Two buffer list pointers. The buffer list is a hashed list used by the kernel to quickly locate a block in the buffer cache. All buffers are linked onto a hashed buffer list.
- **b_iodone**—Contains a pointer to a function to be called by **iodone** if the B_CALL flag is set in **b_flags**. If this feature is not used by a driver, this field is free for driver use.
- **b_queue**—A pointer to the *iobuf* structure associated with this *buf* structure. It is set by **enqueue**().
- **b_resid**—Is the number of non-transferred bytes of the request. If appropriate, the *driver*_**strategy** routine should set this field to the amount of non-transferred data before returning to **physio**.
- **b_s**[**0-8**]—**b_s0**, **b_s1**, **b_s2**, up to **b_s8** are scratch fields that can be used by the driver for any purpose.
- **b_sc**—A pointer to the select code structure of the device associated with this buf structure. This value can be set by the driver. The select code management routines expect this value to be set.
- *b_un* —A structure of which *b_un*.**b_addr** is the low order core address (**b_spaddr** contains the remainder of the core address) of the kernel buffer asso-

ciated with this buffer header. All other fields of the *b_un* structure are used by the file system and should not be used by drivers.

The kernel uses the buffer list and the available list pointers to connect a buffer header into two doubly linked lists. The kernel uses these pointers to manage the placement and usage of buffers in the buffer cache. If your driver queues buffers, you can use the **av_forw** and **av_back** pointers to maintain a queue. The select code management routines use these fields as the forward and backward pointers in the select code queue. You must not modify the **b_forw** and **b_back** pointers if you are using a file system buffer. The kernel uses them to locate buffers in the buffer cache.

Drivers have historically used the remaining *buf* structure fields to store information. These fields are not guaranteed to be available for your driver in later releases of HP-UX.

## *cdevsw*

Each character device driver needs a **cdevsw** entry in the **cdevsw** table, which is an array of *cdevsw* structures constructed from information in files in the **/usr/conf/master.d/** directory. The *cdevsw* structure is declared in **/usr/conf/h/conf.h** as follows:

```
struct cdevsw {
    d_open_td_open;
    d_close_td_close;
    d_read_td_read;
    d_write_td_write;
    d_ioctl_td_ioctl;
    d_select_td_select;
    d_option1_td_option1;
    int     d_flags; /* C_MGR_IS_MP is valid */
    void    *d_drv_info;
    pfilter_t*d_pfilter_s;
    aio_ops_t*d_aio_ops;
};
```

The various fields have the following characteristics:

- All fields except **d_flags** point to device driver routines. For instance, performing

the **open**() system call on a device file calls the driver routine pointed to by ***d_open** in the appropriate *cdevsw* entry.

- The **d_flags** field is not a pointer, nor is it passed to the driver. It contains flags indicating special operation features of the device. Currently defined flags are:
  - C_ALLCLOSES forces a call to the driver's close routine on every closing of the device. (By default, **close**() only calls the driver's close routine on the last close of the device.)
  - C_NODELAY cues the kernel not to wait for a write request to complete on this device. The default action is to wait for a write request to complete before returning to the calling process.
  - C_EVERYCLOSE is the same as C_ALLCLOSES.

Drivers need not provide a corresponding routine for every I/O system call; instead, **cdevsw** table entries can point to one of two special routines as follows:

- The **nulldev**() routine returns a zero. Use **nulldev** when it is okay for the routine in this field to be called, but the field for the device has no functionality.
- The **nodev**() routine returns an ENOVDEV error (no such device). Use **nodev** when it is not okay for the routine in this field to be called. For example, for a read-only device, you could use **nodev** in the entry for **d_write** so an error is returned when a user process performs a **write**() on the corresponding device file.

Your driver should implement all functions in the *cdevsw* structure appropriate for the controlled device. Functions not implemented in a driver must contain **nulldev** or **nodev** in the appropriate device switch table entry.

## *dma_parms*

The DMA parameters struct contains information about the requested DMA transaction. It is defined in **/usr/include/sys/io.h**.

```
struct dma_parms {
     int channel;
     int dma_options;
     int flags;
     int key;
     int num_entries;
     buflet_info_type *buflet_key;
```

```
    struct iovec *chain_ptr; /*pointer to first DMA element*/
    int chain_count; /*number of DMA elements in a chain*/
    int chain_index;
    int (*drv_routine)() __((caddr_t drv_arg));
    int drv_arg;
    int transfer_size;
    caddr_t addr;     /* host offset address */
    space_t spaddr; /* host space address */
    int count;         /* total number of bytes */
}
```

- **chain_count** and **chain_index**—Determine if there are more chains to transfer. The **chain_count** field should not be modified by the driver.
- **dma_options**—One or more of the following: VME_A32_DMA, VME_A24_DMA, DMA_READ, DMA_WRITE, VME_HOST_DMA, VME_USE_IOMAP, or VME_SKIP_IOMAP.

***NOTE:***

VME_A32_DMA and VME_A24_DMA are mutually exclusive bits used to specify the address capability of the card requesting a DMA if the address modifier provided by the *isc* is set to VME_COMPT. VME_USE_IOMAP specifies the use of the VME slave map hardware. If VME_USE_IOMAP is not set, the VME direct map hardware is used.

- **drv_routine—**Used as follows:
  - If the system resources (map hardware) needed for the DMA are unavailable, **vme_dma_setup()** returns RESOURCE_UNAVAILABLE.
  - If the driver calling **vme_dma_setup** is a context-switch-driven driver, it sleeps upon receiving this error return; **drv_routine** issues wakeup to wake up the driver. This routine is called with *drv_arg* as its argument when the system frees DMA resources.
  - If the driver is an interrupt-driven driver, it advances its state and exits; **drv_routine** re-invokes the driver to try again.
  - If a driver does not wish to be notified when resources become available (if they were unavailable at DMA setup time), it sets **drv_routine** to NULL.

## *drv_info*

The following example shows driver-specific fields for a Version 10.0-compliant driver:

```
typedef struct drv_info {
    char *name;         /* device type name */
    char *class;        /* device class name */
    ubit32 flags;       /* driver type info */
    int b_major;        /* block driver major dev num */
    int c_major;        /* char driver major dev num */
    . . .
} drv_info_t;
```

- **name**—The name of the driver, e.g., “skel”
- **class**—The name of the device class, e.g., “VME Driver”
- **flags**—The type of driver, as follows (may be OR’d):
  - DRV_CHAR—character device driver
  - DRV_BLOCK—Block device driver
  - DRV_PSEUDO—Pseudo device driver
  - DRV_SCAN—Supports scanning
  - DRV_SAVE_CONF—Save configuration information to **ioconfig**
- **b_major, c_major**—The major device number (use -1 for **b_major** or **c_major** to “de-specify” the opposite type of block or character device, respectively)

## *drv_ops*

The *drv_ops* struct contains all the driver entry points. It is defined in **/usr/conf/h/conf.h**:

```
typedef struct drv_ops {
    d_open_t        d_open;
    d_close_t       d_close;
    d_strategy_t    d_strategy;
```

```
    d_dump_t        d_dump;
    d_psize_t       d_psize;
    int             (*reserved0) __(());
    d_read_t        d_read;
    d_write_t       d_write;
    d_ioctl_t       d_ioctl;
    d_select_t      d_select;
    d_option1_t     d_option1;
    pfilter_t       *pfilter;
    int             (*reserved1) __(());
    int             (*reserved2) __(());
    aio_ops_t       *d_aio_ops;/* reserved3 replaced */
    int             d_flags;
} drv_ops_t;
```

The defined flag values are:

- C_ALLCLOSES—Call device close on all closes
- C_NODELAY—If no write delay on block devices
- C_EVERYCLOSE—Call device close on all closes
- C_CLONESMAJOR—Driver clones major and minor num
- C_DYN_MAJOR—Reserved for dynamic allocation
- C_ASSIGN—Needs to remap a buffer to kernel space
- C_MAP_BUFFER_TO_KERNEL—Needs to remap a buffer to kernel space in **physio**

## *iobuf*

Each device has an associated *iobuf* structure containing private-state information for each device driver. This structure is optional and can contain any information the driver needs to store. The structure is defined in /**usr/include/sys/iobuf.h**.

```
struct iobuf
{
    uint32_t     b_flags;/* see buf.h */
    struct buf *b_forw;/* first buffer */
    struct buf   *b_back;/* last buffer */
    struct buf *b_actf;/* head of I/O queue */
```

```
    struct buf   *b_actl;/* tail of I/O queue */
    int    b_queuelen;/* KI current queue length to device */
/*** struct iostat*io_stp;/* unit I/O statistics */
    struct  timeout timeo;/* for timeouts */
    struct  sw_intloc intloc;/* for soft trigger on timeouts */
    int (**markstack)();/* for timeout escapes */
    char   timeflag; /* timeout has occurred */
    char   b_active; /* busy flag */
    char   b_errcnt; /* error count (for recovery) */
    char   b_state;  /* state for FSMs */
    char   io_s0;     /* space for drivers to leave things */
    char   in_fsm;    /* indicates flow in FSM for timeout */
    caddr_tb_xaddr;  /* transfer address */
    long   b_xcount; /* transfer count */
    long   b_headpos;/* head position for scheduling */
    struct eblock*io_erec;/* error record */
    int    io_nreg;  /* number of registers to log on errors*/
    physadrio_addr;  /* csr address */
    long   io_s1;/* space for drivers to leave things */
    long   io_s2;/* space for drivers to leave things */
    long   io_s3;/* space for drivers to leave things */
};
```

- **b_actf**, **b_actl**—Used by the select code management routines as the head and tail pointers of the select code queue. If your driver is not using these routines, it can use these fields as scratch space.
- **b_active**—Indicates the busy status of the device; set to non-zero during I/O.
- **b_dev**—Typically contains the major and minor number for the device.
- **b_errcnt**—Typically contains the current error count; typically used for recovery or retry purposes.
- **b_flag**— Can contain information corresponding to the **b_flags** field in the *buf* structure.
- **b_state**—Used by the Finite State Machine mechanism to indicate the current state of the FSM (value initialized by **enqueue**()). If your driver is not using this mechanism, it can use this field as scratch space.
- **b_xaddr**—Typically contains the address of data to be transferred to/from the device.
- **b_xcount**—Typically contains the number of bytes to be transferred to/from **b_xaddr**.
- **in_fsm**—Used by the Finite State Machine mechanism as a flag to indicate being

currently in a FSM. If your driver is not using this mechanism, it can use this field as scratch space.

- **intloc**—Can be used for setting up software triggers.
- **timeflag**—Used by the Finite State Machine mechanism. If your driver is not using this mechanism, it can use this field as scratch space.
- **timeo**—Used by the Finite State Machine mechanism START_POLL macro. If your driver is not using this mechanism, it can use this field as scratch space.

Use of the remaining fields of the *iobuf* structure is driver-defined.

## *io_parms*

The *io_parms* structure defines some variables used by **map_mem_to_bus**() and **vme_map_mem_to_bus2**() to specify an address and the number of bytes to map. It is defined in **<sys/io.h>**, as follows:

```
struct io_parms {
    int     flags;
    int     key;
    int     num_entries;
    int     (*drv_routine) __((caddr_t drv_arg));
    caddr_t drv_arg;
    caddr_t host_addr;
    space_t spaddr;
    u_int   size;
};
```

- **host_addr**—The host virtual address or polybuf to be mapped to the I/O bus space
- **size**—The size of the space to be mapped in bytes.

## *iovec*

The *iovec* structure contains a pointer to the user's data area and the length of the data to be transferred. It is defined as follows in **/usr/conf/wsio/vme2.h**:

```
struct iovec {
    caddr_t iov_base;      /* user address */
    int iov_len;           /* num bytes to transfer */
}
```

## *isc*

*isc* is the common way of referring to a struct that is actually an *isc_table_type* struct. The Interface Select Code table contains entries for each card in the system, and an *isc* is a particular entry. This struct is defined in **/usr/include/sys/io.h** as follows:

```
struct isc_table_type {
    struct buf *b_actf;
    struct buf *b_actl;
    . . .
    char card_type;
    unsigned char my_isc;
    char my_address;
    char active;
    char int_lvl;
    . . .
    struct buf *owner;
    . . .
    int resid;
    . . .
    int count;
    struct sw_intloc intloc; /* for software triggers */
    struct sw_intloc intloc1;/* for software triggers */
    struct sw_intloc intloc2;/* for software triggers */
    . . .
    char bus_type;
```

```
    struct bus_info_type *bus_info;
    int if_id;
     . . .
    struct gfsw gfsw;
    caddr_t ifsw;
    . . .
    caddr_t if_drv_data;
    struct dma_parms *dma_parms;
    . . .
}
```

Information in an ISC table is maintained for interface drivers corresponding to each card (Core I/O select code, EISA slot, or VME card) in use in the system. Each driver can reference its own entry by using the select code/slot field of its **dev** number as an index. Since each select code has an entry, an interface driver can get information on each of its cards by using the select code from the minor number. If appropriate, other drivers can use the struct for their own private information.

This struct has the following fields of interest:

- **active**—Used by the select code management routines to indicate current use of the select code.
- **b_actf**, **b_actl**—Used by the select code management routines (free if mechanism not used).
- **buffer**—Typically contains the address of the data buffer to be transferred to/from this card.
- **bus_info**—A pointer to a structure describing features of the I/O bus this card resides on. Used by the system I/O services, this must not be modified by drivers.
- **bus_type**—Contains the type of I/O bus the interface card resides on. Current bus types are defined in **/usr/conf/h/io.h**.
- **card_ptr**—A pointer to the interface card's registers.
- **card_type**—Typically contains the part number identified with this interface card.
- **count**—Contains the amount of data to be transferred to/from the buffer.
- **dma_parms**—Pointer used by drivers for VME DMA.
- **gfsw**—Contains a pointer to the driver’s initialization routine.
- **if_drv_data**—This is a card-specific pointer reserved for a driver's use.

- **if_id**—Contains the hardware ID associated with this interface card.
- **if_reg_ptr**—A pointer to the interface card's registers.
- **ifsw**—Contains a pointer to the general I/O switch structure corresponding to this entry, if any.
- **intloc***—You can optionally use **intloc**, **intloc1**, and **intloc2** for setting up software triggers, or you can declare your own.
- **int_lvl**—Typically contains the interrupt level of the interface card.
- **iosw**—Maintains a pointer to the I/O switch structure corresponding to this entry, if any.
- **my_address**—If needed, usually contains the interface bus address (free for use by the driver).
- **my_isc**—Offset of this *isc* structure in the ISC table.
- **owner**—Used by the select code management routines to point to the buffer header belonging to the current owner of the select code.
- **resid**—Set to hold the residual count from a data transfer.

The driver *should not* use other fields of the *isc* structure for any purpose.

***NOTE***

The **if_info** field used by HP-UX 9 drivers is no longer available.

## *proc*

A *proc* structure is allocated for each process on the system. It is always in memory, since it contains information that must be available to the kernel at all times, for example: process priority, status, ID. Drivers generally never access any of these fields, but they do need to know that the *proc* structure exists, because several kernel services require a pointer to one.

```
#include <sys/proc.h>
struct proc *p;
. . .
```

## *sw_intloc*

The kernel implements software triggers by checking a linked list of structures during each processor bus interrupt so see which process to call.

```
struct sw_intloc {
     struct sw_intloc *link;    /* must be first */
     int (*proc)();             /* to call on trigger */
     caddr_t arg;               /* argument */
     char priority;             /* level of interrupt */
     char sub_priority;         /* interrupt sub-level*/
};
```

The fields of this struct are initialized by a call to **sw_trigger**().

## *uio*

For all **read**(2) and **write**(2) system calls on character device files, the kernel allocates and fills out the *uio* structure, defined in **usr/include/sys/io.h**. For **read** and **write** system calls performed on a character device file, the kernel passes this *uio* structure to the *driver***_read** and *driver***_write** routines of the driver. The *uio* structure contains a pointer to the user's data area. The *driver***_read** and *driver***_write** routines can either:

- Transfer the data between the user's buffer and the device using **physio()**.
- Buffer the data using **uiomove**().

Character device drivers seldom access individual members of the *uio* structure because **uiomove** or **physio** take care of many details for you.

```
struct uio {
    struct iovec *uio_iov; /* pointer to iovec */
    int uio_iovcnt;        /* number of iovecs */
    int uio_offset;        /* pointer offset in file */
    int uio_segflg;        /* space or quadrant ID */
    int uio_resid;         /* bytes left to transfer */
};
```

- **uio_iov**—A pointer to an *iovec* (I/O vector) structure, often known as the argument name *status_id*, which contains a pointer to the user's data area and the length of data to be transferred.
- **uio_iovcnt**—Specifies the number of *iovec* structures pointing to valid data areas to be transferred. **Read** and **write** requests use only one *iovec* structure and can specify a maximum of MAXIOV (currently 16) *iovec* structures.
- **uio_offset**—Points to the current location in the file where I/O is taking place. The meaning of **uio_offset** depends on the particular device, but it usually has meaning only to devices capable of seeking. The value for **uio_offset** is obtained from the file pointer (offset) entry in the system file table.

  The file pointer in the system file table entry is set to zero on **open** by the kernel, and its value is incremented with each **physio** or **uiomove** call. If your device does not seek, reset the **uio_offset** field in the *driver*_**read** and *driver*_**write** routines so the kernel does not increment it past its maximum value.
- **uio_segflg**—A flag telling the virtual memory system how to transfer data. Device drivers should ignore this field.
- **uio_resid**—Contains the number of bytes of data remaining to be transferred. At the start, this equals the sum of all the **iov_len** lengths.

## *user*

The *user* structure provides kernel information about a process that needs to be available only when it is running and can otherwise be swapped out: pointer to the *proc* struct; arguments to system calls; user credentials, etc.

```
#include <sys/proc.h>
#include <sys/user.h>

extern struct user u;
struct proc *p;
    . . .
    p=u.u_procp;
```

Drivers generally never access any of these fields except the pointer to the *proc* structure. When a driver entry point is called, the global variable *u* contains the *user* structure for the current process.

## *vme_hardware_map_type*

The **vme_hardware_map_info**(2) routine fills in the *vme_hardware_map_type* structure with the parameters for the direct and slave I/O maps, according to the following structure (defined in **/usr/conf/wsio/vme2.h**):

```
struct vme_hardware_map_type {
    unsigned int a32_dirwin_addr; /*dflt:= 0x10000000 */
    unsigned int a24_dirwin_addr; /*dflt:= 0x00c00000 */
    unsigned int a32_dirwin_host_base; /* dflt:= 0x0 */
    unsigned int a24_dirwin_host_base; /* dflt:= 0x0 */
    int  a32_dirwin_size; /* dflt: 32 Meg */
    int  a24_dirwin_size; /* dflt: 0 */
/* next four fields may be 0b0xxxxxxx - dflt: 0x3f* /
    unsigned char a32_dirwin_addr_mods;
    unsigned char a24_dirwin_addr_mods;
    unsigned char a32_iomap_addr_mods;
    unsigned char a24_iomap_addr_mods;
    unsigned int a32_iomap_addr;  /* dflt:= 0x100000 */
    unsigned int a24_iomap_addr;  /* dflt:= 0x100000 */
    int  iomap_size;                /* default: 1 Meg */
};
```

- **a32_dirwin_addr**—The VME A32 base address of the A32 direct mapper. Its default value is 0x10000000.
- **a24_dirwin_addr**—The VME A24 base address of the A24 direct mapper. Its default value is 0x00c00000.
- **a32_dirwin_host_base**—Local physical memory base address of the A32 direct mapper. On the 742/747, this is 0. On the 743/744, this is settable to any multiple of the size, with a default of 0x10000000 (256MB).
- **a24_dirwin_host_base**—Local physical memory base address of the A24 direct mapper. On the 742/747, this is 0. On the 743/744, this is settable to any multiple of the size, with a default of 0x100c0000 (12MB).
- **a32_dirwin_size**—The size of the A32 direct mapbper. On the 742/747, this is 256MB. On the 743/744, this is settable to 0, 64 KB,128 KB, . . ., 512 MB, with a default size of 32 MB..
- **a24_dirwin_size**—The size of the A24 direct mapper. On the 742/747, this is

1 MB. On the 743/744, this is settable to 0, 64 KB, 128 KB, . . ., 16 MB, with a default size of 0.

- **a32_dirwin_addr_mods**, **a24_dirwin_addr_mods**, **a32_iomap_addr_mods**, **a24_iomap_addr_mods**—These four bit-fields specify which area modifiers are supported by the direct or slave mapper in A24 or A32 space respectively. Valid values (or'd together) are: NP_D64_ACCESS_AM, NP_DATA_ACCESS_AM, NP_BLK_ACCESS_AM, SUP_D64_ACCESS_AM, SUP_DATA_ACCESS_AM, SUP_BLK_ACCESS_AM, and for 743/744, LOCK_CYCLE_AM.
- **a32_iomap_addr**—The VME bus A32 base address of the slave mapper. This is ton a 1-MB address boundary: 0xXXXXX000.h.
- **a24_iomap_addr**—The VME bus A24 base address of the slave mapper. This is the A32 base address of the slave mapper with the high byte masked off: 0x00XXX000.
- **iomap_size**—The size of the slave mapper, currently 1 MB made up of 256 4-KB entries. This mapper is accessible from A24 or A32 VME bus masters and its location is set with the **vme_config** program.

## *vme_hardware_type*

The **vme_hardware_info(**2**)** routine fills in the *vme_hardware_type* structure with the parameters for the system on which the driver is currently running, according to the following structure (defined in **/usr/conf/wsio/vme2.h)**:

```
struct vme_hardware_type {
    int iomap_size;     /* size of the IO map (1 MB) */
    int vme_expander;   /* indicates VME board system */
    int cpu_number;     /* CPU/card number */
    unsigned int a32_dirwin_addr;  /* VME starting addr
                          of direct-mapped window */
    int avail_interrupts;/* which ones are available */
    int features;       /* is CPU a slot 1 controller */
}
```

- **iomap_size**—The size of the slave mapper, currently 1 Mbytes made up of 256 4-Kbyte entries. This mapper is accessible from A24 or A32 VME bus masters

and its location is set with the **vme_config** program.

- **vme_expander**—Returns the type of **expander** (just another name for the 700's VME bus adapter/controller): ISPAVME1 (the Series 742/747), or IS_PAVME2 (Series 743/744).
- **cpu_number**—For multiprocessor configurations. The **cpu_number** is set with either the VME-Class*i* PDC (boot ROM) Boot Administration mode command BPN_CONFIG, or with the **vme_config** routine.
- **a32_dirwin_addr**—The VME base address of the direct mapped window capability on a 742/747, or 743/744. Its default value is 0x10000000 (256 MB) on a 742/747, and 32 MB on a 734/744. Its location and size are set with the **vme_config** program.
- **avail_interrupts**—Indicates which interrupts are being handled by the current CPU. The field is essentially a bit mask with a 1 in a bit position indicating it is available. Bit 0 is ignored and bits 1 through 7 represent possible interrupt levels.
- **features**—Indicates whether the current CPU is a VME slot 1 controller and whether it contains a location monitor: SLOT_1 or LOC_MON (or both) may be set.

## *vme_polybuf*

This struct refers to arrays of pointers and sizes of (usually) multiple buffers for use in **vme_map_polybuf_to_bus** and **vme_dma_queue_polybuf**.

```
struct vme_polybuf {
    int num_entries;    /* count of arrays used */
    int num_allocated;  /* arrays allocated */
    char**buffers;      /* ptr to start of addr array */
    int *sizes;         /* ptr to array of sizes */
};
```

- **num_allocated**—Indicates the size of the *buffers* and *sizes* arrays..
- **num_entriess**—Indicates how many of the entries of the *buffers* and *sizes* arrays contain valid data.

## *vme2_copy_addr*

The data structure *vme2_copy_addr* contains information used by the VME2_USER_COPY **ioctl** command. It is defined in **/usr/conf/wsio/vme2.h**:

```
struct vme2_copy_addr {
    caddr_t from_va;    /* transfer source */
    caddr_t to_va;      /* transfer destination */
    int count;          /* transfer size in bytes */
    int options;        /* method of transfer */
    int residual;       /* num bytes not transferred */
}
```

Either **from_va** or **to_va** should be an address returned from a previous call to the VME2_MAP_ADDR **ioctl**, and the other should be an address of a buffer in user space.

The **options** field indicates the method of transfer:

- Bit 0-5 must specify a VME address modifier.
- One, and only one, transfer bit (VME_D08, VME_D16, VME_D32, or VME_D64) must be set. (Transfer bit values that can be "OR'd" together.)
- The VME_OPTIMAL bit may be set.

For more information, see “Synchronous DMA the Easy Way with vme_copy” on page 4-5.

A **residual** value of 0 indicates the copy completed; a value > 0 indicates the number of bytes not transferred due to a VME bus error or error on the call.

## *vme2_int_control*

The VME2_ENABLE_INT and VME2_ENABLE_IRQ **ioctl** commands use the following struct to enable interrupts:

```
struct vme2_int_control {
    int int_level;              /* level to enable */
    int available_interrupts;   /* no. of levels */
    int error;                  /* num from vme2 */
}
```

- **Available_interrupts** returns the number of available interrupt levels.

## *vme2_io_regx*

The register access **ioctl** calls VME2_REG_READ and VME2_REG_WRITE let you specify a VME address modifier, as well as the data that was read or is to be written. This struct is defined in **/usr/conf/wsio/vme2.h** as follows:

```
struct vme2_io_regx {
    int card_type;          /* card's address space */
    caddr_t vme_addr;       /* VME address to probe */
    unsigned int value;     /* value read or written
    int width;              /* width of the address */
    int access_result;      /* returns 1 for success */
    int vme_addr_mod;       /* addr modifier for reg */
    int error;              /* error no. from vme2 */
}
```

**Card_type**, **vme_addr**, **width**, **access_result**, and **error** are the same as in the *vme2_io_testx* struct.

## *vme2_map_addr*

The third data structure *vme2_map_addr* is used for VME “map” and “unmap” calls and is as follows:

```
struct vme2_map_addr {
    int             card_type;
    caddr_t         vme_addr;
    unsigned int    size;
    caddr_t         user_addr;
    int             error;
}
```

The **card_type**, **vme_addr**, **width**, and **error** fields are the same as in the *vme2_io_testx* struct.

The **size** is the size ib bytes to map into the user process.

The **user_addr** field contains the kernel virtual address that can be used in the user process.

## *vme2_io_testx*

The probe **ioctl** commands VME_TESTR and VME_TESTW use the VME adapter hardware to trap VME bus errors and return the success or failure of the probe in the **access_result** field of the *vme2_io_testx* structure.

The actual data written or read is internal to the **vme2** driver.

```
struct vme2_io_testx {
    int card_type;          /* card's address space */
    caddr_t vme_addr;       /* VME bus addr to probe */
    unsigned int width;     /* width of the address */
    int access_result;      /* returns 1 for success */
    int error;              /* error num from vme2 */
}
```

**Card_type** is one of the following: VME_A16, VME_A24, or VME_A32, or any of the specific area modifiers in the table "Symbolic Names for Memory Address Modifiers" on page 2-10.

**Width** is BYTE_WIDE, SHORT_WIDE, or LONG_WIDE (defined in **/usr/conf/h/io.h**).

A value of 0 in **access_result** indicates a VME bus error, and a -1 indicates incorrect usage.

## *vme2_lm_fifo_setup*

*vme2_lm_fifo_setup* is used by the Location Monitor and FIFO **ioctl** commands (see “Location Monitor and FIFO Commands” on page 5-11).

```
struct vme2_lm_fifo_setup {
    caddr_t address;            /* start address */
    unsigned short modifiers;   /* address modifiers */
    short size;                 /* range from start */
    int signal;                 /* signal to return */
};
```

- **address**—The starting address to be monitored by the Location Monitor or to contain the FIFO information.
- **modifiers**—Address modifier bit values that may be OR'd together: A32_LOCK_BIT, A24_LOCK_BIT, A32_SUP_DATA_BIT, A32_NP_DATA_BIT, A24_SUP_DATA_BIT, A24_NP_DATA_BIT, A16_SUP_DATA_BIT, A16_NP_DATA_BIT, A32_BIT, A24_BIT, and A16_BIT.

## *wsio_drv_data*

The *wsio_drv_data* structure specifies driver-specific fields for all 10.0-compliant drivers. It is defined in **/usr/conf/wsio/wsio.h**:

```
typedef struct wsio_drv_data {
    char *drv_path;         /* to match probe & driver */
    sbit8 drv_type;         /* type of hardware */
    ubit32 drv_flags;       /* convergent driver? */
    int (*drv_minor_build)();   /* minor num formatter*/
    int (*drv_minor_decode)();  /* minor num decoder */
} wsio_drv_data_t;
```

- **drv_path**—A character string that you can define to confirm that this driver is the one responding to probes.
- **drv_type**—The type of hardware:
  - T_INTERFACE—The driver controls an interface card

  - T_DEVICE—The driver controls a hardware device
- **drv_flags**—Whether this is a convergent driver: DRV_CONVERGED or NOT_CONVERGED.
- **drv_minor_build**(), **drv_minor_decode**()—Leave NULL.

## *wsio_drv_info*

The *wsio_drv_info* struct simply references the structures *drv_info*, *drv_ops*, and *wsio_drv_data*. It is defined in **/usr/conf/wsio/wsio.h**:

```
typedef struct wsio_drv_info{
    drv_info_t *drv_info;
    drv_ops_t *drv_ops;
    wsio_drv_data_t *drv_data;
} wsio_drv_info_t;
```

The *wsio_drv_info* structure is referenced in the **wsio_install_driver** function and in the **isc_claim** function, as shown in the following example:

```
wsio_install_driver (&driver_wsio_info);
isc_claim (isc, &driver_wsio_info);
```

# B

# Kernel and Driver Routine Summaries

The purpose of this appendix is to present quick-reference information on VME Services routines as well as other HP-UX kernel routines most useful to a writer of a VME device driver.

See the Preface of this manual for pointers to other appropriate documentation for these and additional calls.

As a general rule:

- The functions have a type **int**.
- The functional declarations are shown in K&R format.
- All calls to **vme2** services need to include the **vme2.h** header file:

```
#include "/usr/conf/wsio/vme2.h
```

- Many of the VME functions are discussed in more detail in the text of previous chapters.

## acquire_buf(), release_buf()

Marks (or releases) the buffer pointed to by *bp* as busy for the exclusive use of the calling process.

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

int acquire_buf (bp)
    struct buf *bp;

int release_bus (bp)
```

If the buffer is already busy, the calling process blocks until it's available. When it's acquired, B_BUSY is set in *bp*->**bflags**, and *bp*->**b_error** is cleared. When it's released, B_BUSY and B_WANTED are cleared in *bp*->**bflags**.

## bcopy()

Copies ***count*** bytes from one buffer to another in the same virtual space.

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

bcopy (from, to, count)
    caddr_t from;       /* addr of buff copied from */
    caddr_t to;         /* addr of buff copied to   */
    unsigned count;     /* num bytes to copy */
```

To copy data between user and kernel space, use **copyin**, **copyout**, or **uiomove**.

## brelse()

Releases a buffer cache buffer.

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

brelse (bp)
    struct buf *bp; /* ptr to buff to release */
```

## bzero()

Zeroes *size* bytes starting at the location specified by *to*.

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

bzero (to, size)
    caddr_t to;         /* address of buffer to zero */
    unsigned size;      /* size of buffer to zero */
```

## copyin()

Copies *nbyte*s from the address *user_from* to address *to*.

```
copyin (user_from, to, nbytes)
    char *to, *user_from;
    int nbytes;
```

Returns EFAULT by setting an error in ***u*.u_error** if a bus error occurs during the copy operation.

## copyout()

Copies *nbytes* to the address *user_to* in the user data area from the kernel address *from*.

```
copyout (from, user_to, nbytes)
    char *user_to, *from;
    int nbytes;
```

Returns EFAULT by setting an error in ***u*.u_error** if a bus error occurs during the copy operation.

## dma_sync()

**dma_sync** is defined as follows:

```
dma_sync (address_type, virtual addr, size, hints);
    int address_type
    caddr_t virtual addr;
    int size;
    int hints;
```

To flush cache, call **dma_sync** as follows:

```
dma_sync (KERNELSPACE, virtual_addr, size,
    IO_WRITE|IO_CONDISTIONAL|IO_MODIFIED);
```

Ensure cache coherency by writing (flushing) the current contents of cache to memory, or by purging the cache to cause its reload.

Typically, **dma_sync** is called this way during the setup for a VME bus master to do a read from VME-Class RAM. This call updates the RAM with the latest contents prior to the VME bus's access.

To flush cache, call **dma_sync** as follows:

```
dma_sync (KERNELSPACE, virtual_addr, size,
    IO_READ|IO_CONDISTIONAL|IO_ACCESSED);
```

This call to **dma_sync**, on the other hand, ensures that the next CPU access will cause the cache to be loaded from memory. (The previous cache contents are discarded.) Typically, **dma_sync** is called during the clean-up for a VME bus master after doing a write to the VME-Class RAM. The call invalidates the cache lines so that the next PA-RISC processor access loads the correct RAM contents to cache.

These routines are not needed for DMA if **vme_dma_setup** and **vme_dma_cleanup** are used.

## free_isc()

Frees an *isc* structure and returns it to the system.

```
int free_isc (isc)
    struct isc_table_type *isc;
```

Useful if errors are encountered during driver initialization that would cause the driver not to be usable.

## geteblk()

Allocates a buffer cache buffer.

```
#include <sys/types.h>
#include "/usr/conf/h/buf.h"
```

```
(struct buf *) geteblk (size)
    int size;      /* size of the requested block */
```

## geterror()

Checks for and returns I/O errors.

```
geterror (bp)
    register struct buf *bp;
```

## ioctl()

Allows access to VME services from a user-level process.

```
#include <sys/ioctl.h>
#include "/usr/conf/machine/vme2.h"
```

```
ioctl (fd, ioctl_command, struct_ptr)
    int fd;
    int ioctl_command;
    struct struct_type *struct_ptr;
```

**ioctl**() allows access to VME services from a user-level process, given a file descriptor, *fd*, for a previously-opened file; an *ioctl_command* from the set VME2_*xx* where *xx* can be ENABLE_IRQ, FIFO_GRAB, FIFO_POLL, FIFO_READ, FIFO_RELEASE, IO_TESTR, IO_TESTW, LOCMON_GRAB, LOCMON_POLL, LOCMON_RELEASE, MAP_ADDR, REG_READ, REG_WRITE, UNMAP_ADDR, and USER_COPY; and a pointer to a structure specifically used by the **ioctl** command—see “ioctl Commands with Structs” on page 5-3.

## iodone()

Marks a buffer as done and wakes any processes waiting for the buffer.

```
#include <sys/types.h>
#include "/usr/conf/h/buf.h"

iodone (bp)
    struct buf *bp; /* pntr to buf marked done */
```

## io_malloc(), io_free()

Allocates kernel I/O memory for *size* bytes, suitably aligned.

```
#include "/usr/conf/h/buf.h"

caddr_t io_malloc (size, flags)
    int size;
    int flags;

int io_free (ptr, size)
    caddr_t ptr;
    int size;
```

- *flags*—Bit values as follow:
  - IOM_WAITOK—**io_malloc** sleeps until memory is available.
  - IOM_NOWAIT—**io_malloc** fails if no memory is available; for this flag, it can be called on the interrupt stack.

**io_malloc** returns zero on failure and >0 on success; **io_free** always returns success.

## iowait()

Waits until I/O is done and returns any errors.

```
#include <sys/types.h>
#include "/usr/conf/h/buf.h"

iowait (bp)
    struct buf  *bp; /* pntr to buf to sleep on */
```

## isc_claim()

Accepts or rejects the *isc* received by your *driver*_**attach** function.

```
isc_claim (isc, driver_wsio_info);
    struct isc_table_type *isc;
    struct wsio_drv_info *driver_wsio_info;
```

- *isc*—The card pointer received as an argument by the *driver*_**attach** function that calls **isc_claim**.
- *driver_wsio_info*—Either NULL to reject the *isc*, or a pointer to a struct containing three individual structs of driver-specific fields for all 10.0-compliant drivers.

## isrlink()

Installs an interrupt service routine (**isr**).

```
int isrlink (isr,level,regaddr,mask,value,parm1,parm2)
    int (*isr)();
    int level;
    char *regaddr;
    char mask, value, parm1;
    int parm2;
```

- *isr*—The routine to call if the interrupt is for our driver.
- *level*—The interrupt level of the card.
- *regaddr*—The card's register to examine when polling to determine if the card interrupted.
- *mask* —Applied to the register's contents which, after masking, will equal *value* if it is the interrupting device.
- *parm1*, *parm2*—The parameters handed to the interrupt service routine. These can be anything that you want passed to it.

## issig()

Returns true if the current process has a signal pending. The signal to process is put in the **p_cursig** field of the *proc* structure.

```
issig ()
```

See also **signal**(2).

## kvtophys()

Takes a VME-Class kernel virtual address and returns the corresponding physical address in the same 700 Series machine's RAM.

```
caddr_t kvtophys (virtual_addr)
    caddr_t virtual_addr;
```

## major(), minor()

Macros that return the major number from the device number.

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

major (dev)
    dev_t dev;          /* device number */

minor (dev)
    dev_t dev;          /* device number */
```

## map_mem_to_bus()

Maps a host physical address to I/O bus space.

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

caddr_t map_mem_to_bus (isc, io_parms)
    struct isc_table_type *isc;
    struct io_parms_type *io_parms;
```

- *isc*—The pointer for the interface card wishing to access this memory.
- *io_parms*—Pointer to the struct that specifies the host virtual address to be mapped to the I/O bus space, and the size of the space to be mapped, in bytes.

Returns an I/O bus physical address on success; returns a negative number indicating the error condition on failure. If the mapping can't be obtained, the *io_parms*->**drv_routine** field is called when the mapping becomes available. See **unmap_mem_from_bus**(), **vme_map_largest_to_bus**(), **vme_map_pages_to_bus**(), and **vme_map_polybuf_to_bus**().

## map_mem_to_host()

Maps physical system bus space to host virtual address.

```
#include "/usr/conf/h/io.h"

caddr_t map_mem_to_host (isc, phys_addr, size)
    struct isc_table_type *isc;
    caddr_t phys_addr;
    int size;
```

- *phys_addr*—The I/O bus physical address for an area of memory on an I/O bus
- *size*—The size of the memory space in bytes
- *isc*—The pointer that corresponds to an interface card associated with memory

Returns the host virtual address for accessing the space specified. See **unmap_mem_from_bus**(), **vme_set_mem_error_handler**(), and **vme_remap_mem_to_host**().

## minphys()

Checks and adjusts the transfer count.

```
#include <sys/types.h>
#include "/usr/conf/h/buf.h"

minphys (bp)
    struct buf *bp;     /* Pointer to buf structure */
```

This routine limits a single transfer to MAXPHYS bytes by assigning MAXPHYS to *bp*->**b_count** if **b_count** is larger. **Minphys** is typically passed as a parameter to **physio**, which calls it and then tracks any partial transfers.

## msg_printf()

Puts diagnostic error information into the kernel message buffer rather than printing it to the console.

```
msg_printf (/* see printf(3) for list of parameters */)
```

See **printf** in this appendix, below.

## panic()

Soft crash routine. Prints *str* to the console and halts the system, giving control to the kernel debugger if one is present.

```
int panic (str)
    char *str;
```

**panic** halts the system and may cause file system damage; it should be used only to flag catastrophic and irrecoverable failures. It dumps to the system console the processor status register, program counter register, and trap type on processor exceptions, and part of the kernel stack in all other cases.

## physio()

Sets up raw I/O and calls the driver's strategy routine.

```
#include <sys/types.h>
#include "/usr/conf/h/buf.h"
#include <sys/uio.h>

physio (strat, bp, dev, flag, mincnt, uio)
    int (*strat)();         /* Strategy I/O routine */
    struct buf *bp;         /* Ptr to buf structure */
    dev_t dev;              /* Dev major/minor number */
    int rw_flag;            /* B_READ or B_WRITE */
    unsigned (*mincnt)();   /* Size limit routine */
    struct uio *uio;        /* Ptr to uio structure */
```

The **physio**() routine accepts raw I/O requests in the form of a *uio* structure, performs some of the more "intricate" aspects of I/O, and builds the request into a *buf* structure that it passes to the driver strategy routine specified by *strat*. The "intricate" aspects of I/O are discussed below and include waiting for the *buf* structure to become available (if necessary), checking memory access permissions, locking and unlocking memory, mapping the user buffer into kernel space, and blocking the process until the I/O complete.

If *bp* does not have an allocated buffer, **physio** allocates one by setting the *buf*->**b_flags** B_WANTED bit in the selected *buf* and sleeping on the buffer. It then checks that the user has access permission for the buffer specified by *iovec*->**iov_base** and *iovec*->**iov_len**. If B_READ is set in *rw_flag*, the memory page(s) where the buffer resides are checked for write [*sic*] access, otherwise for read access. If they have the wrong access permissions, EFAULT is returned to the caller.

If B_BUSY is set in *buf*->**b_flags**, **physio** waits for the buffer by setting the B_WANTED bit and putting the calling process to sleep, waiting to be awakened by the process that currently has the *buf*. The process that called **physio** is typically awakened when a driver calls **iodone** on that *buf*.

For each *iovec* structure that is associated with the passed *uio* structure, **physio** performs the following steps:

1. It places the request information in the *buf* structure: *buf*->**b_dev** is set to the device major/minor number *dev; buf*->**b_error** is set to zero; the B_BUSY, B_PHYS and B_READ or B_WRITE (selected by the *rw_flags* argument) are set in *buf*->**b_flags**; *b_un*.**b_addr** is set to *iovec*->**iov_base**; *buf*->**b_bcount** is set to *iovec*->**iov_len**; and *buf*->**b_blkno** is set to the DEV_BSIZE block number in which **uio_offset** resides. The B_PHYS bit in *buf*->**b_flags** is set to indicate that the I/O request is raw as opposed to block.
2. It calls the request size-adjusting routine specified in the *mincnt* parameter, to adjust (if necessary) the request size. The *mincnt* routine in most cases should be **minphys,** which sets the maximum request size allowed in a single request to a driver in an attempt to limit the amount of memory that a process can have locked for I/O purposes. If the *mincnt* routine adjusts the request size, **physio** will make multiple requests to the driver until all the data specified by *iovec*->**iov_len** has been transferred (or an error occurs).
3. It locks the page(s) of user space that the buffer resides in and maps the user buffer into kernel space.
4. It calls the strategy routine *strat*, passing the *buf* pointer *bp* as a parameter.
5. After *strat* returns, if the I/O request is not completed, **physio** puts the process to sleep, waiting for the completion of the I/O request. The driver must call **iodone** or **biodone** when the I/O completes to signal I/O completion and to awaken the process.
6. After the I/O request has completed, the page(s) of memory that were locked for the request are unlocked.
7. If another call to the *strat* routine is needed to process the *iovec* structure due to the request size limitations imposed by *mincnt*, Steps 1 through 6 are repeated.

After all the *iovec* structures for the passed *uio* structure are processed, the *buf*->**b_flags** are cleared, and those processes that are asleep waiting for the buffer to become available are awakened, and **physio** returns any errors to its caller.

## printf()

Prints diagnostic information to the console.

```
printf (/* see printf(3) for list of parameters  */)
```

**printf** and **msg_print** are scaled-down versions of the C library **printf** routine, the former printing information to the console, the latter to a kernel message buffer. Both of these functions accept only the following formats:

```
%l, %x, %d, %u, %o, %c, %b, %s, %%
```

The VME-Class kernel **printf** routine is buffered, so console error messages may not accurately reflect the current kernel state when debugging.

## release_buf()

Marks the buffer pointed to by *bp* as no longer busy and wakes any processes waiting for it.

```
#include "/usr/conf/h/buf.h"

int release_buf (bp)
    struct buf *bp;
```

When it's released, B_BUSY and B_WANTED are cleared in *bp*->**bflags**.

## selwakeup()

Wakes a process that is sleeping while waiting for a *select* condition.

```
selwakeup (p, coll)
    register struct proc *p;
    int coll;
```

This routine should be called to wake up a process that was put to sleep waiting on a condition, in order to wake it up and any other processes waiting on the condition. *Coll* indicates whether there was a collision on the **select**(2) call, i.e., if more than one process is waiting. (*Coll* is expected to have been set previously by the driver's **select** routine.)

## sleep(), wakeup()

Sleeps on an address until **wakeup**() on that address occurs.

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

sleep (address, priority)
    caddr_t address;/* address to sleep on */
    int priority;   /* priority of sleeping process */

wakeup (address)
    caddr_t address;
```

- *address*—Typically an address associated with the reason for sleeping; must not be zero.
- *priority*—If less than or equal to PZERO, a signal cannot disturb the sleep; only a corresponding call to **wakeup** can. If greater than PZERO, signals are processed—if PCATCH is OR'd into *priority*, **sleep** returns 1 when a signal occurs, otherwise the system call in progress is aborted. If PCATCH is not set and control returns to the user program because a signal occurs, your driver has no opportunity to clean up any data structures or complete the request.

The **param.h** file lists valid priorities for *priority.* Most drivers use PRIBIO and do not set PCATCH.

Returns zero on success. A wakeup occurs on *address* when another process calls **wakeup** with the same address. The sleeping process then enters the scheduling queue at the *priority* specified. If your driver calls **sleep**, it should also call **wakeup**. Do not call **sleep** from an interrupt context or from a driver strategy routine.

## snooze()

Delays for a specified number of microseconds, useful to busy wait with no interrupts. (The routine has an accuracy of 4 microseconds.)

```
int snooze (usecs)
    int usecs;
```

## spl*()

Sets and restores the processor interrupt level.

```
spl0 () . . . spl7 ()

splx (x)
    int x;      /* new interrupt level */
```

**spl0** through **spl7** set the processor interrupt level to 0 through 7, respectively, and return the current interrupt level before the function was called. If you save the return value, you can later restore interrupts to it by using a call to **splx**. When the interrupt level is set to level *n*, all interrupts at that level or lower are turned off.

The **spl*** routines are designed to protect critical sections of code to guarantee that an interrupt does not occur unexpectedly. Before doing any manipulation of a data structures in a critical section of code, set the proper processor priority level via a call to one of the **spl*** routines. Immediately after the critical section, use **splx** to restore the previous level.

## suser()

Checks whether the current user is the superuser (i.e., root).

```
    int suser ()
```

Returns non-zero if user is superuser.

## sw_trigger()

This routine arranges the calling of an interrupt service routine in the interrupt context at a given priority level. The list of waiting triggers is checked whenever an interrupt is processed, and if their priority is greater than the current interrupt level, they are processed.

```
sw_trigger (intloc, proc, arg, level, sublevel)
    struct sw_intloc *intloc; /* struct to add */
    int (*proc)();          /* call proc on trigger */
    caddr_t arg;            /* argument to routine */
    int level;              /* priority of interrupt */
    int sublevel;           /* interrupt sub-priority */
```

where:

- *intloc*— A pointer to the *sw_intloc* structure to be added to the software trigger queue. Its fields are initialized by **sw_trigger**. If you want to have multiple triggers pending, you must use separate *sw_intloc* structs.
- *proc* —The address of the routine to call when the software trigger is processed.
- *arg* —The argument to be passed to proc.
- *level*—The priority level of the software trigger.
- *sublevel* —The sub-priority of the software trigger. This field is not used by the VME-Classi HP-UX kernel.

## timeout()

Schedules a routine to be called at a later time if a request has been made to a device that hasn't answered within the specific number of clock ticks.

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

timeout (func, arg, t, timeout)
    int (*func)();              /* routine to call */
    caddr_t arg;                /* arg to pass it */
```

```
    int t;                      /* cycles to wait */
    struct timeout *timeout;    /* NULL */
```

Typically called to set a timeout on a request for I/O to a device so that *func* gets called after the time specified

The cycle count or number of clock ticks to wait, *t*, should be expressed in terms of fractions or multiples of HZ for compatibility on different machines.

## uiomove()

Moves data from kernel space to user space or from user space to kernel space.

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

uiomove (cp, n, rw, uio)
    caddr_t cp;         /* ptr to buffer in kernel */
    int n;              /* num bytes to transfer */
    enum uio_rw rw;     /* direction of transfer */
    struct uio *uio;    /* ptr to the uio structure */
```

**uiomove** moves data from user to kernel space or vice versa, transparently collapsing the *uio* struct into a single vector, and maintaining its own pointers for keeping track of the amount of data transferred. **uiomove** maintains and updates fields in the *uio* struct as needed, so you don't have to worry about manipulating them.

- *rw*—specifies the direction of the copy. If UIO_READ, the contents of *cp* are copied into *uio*; if UIO_WRITE, the contents of *uio* are copied into *cp*.
- *cp*—Pointer to buffer in kernel to be copied.

If the transfer is from user space to kernel space or vice versa, **copyin** or **copyout** is called to complete the transfer. If the transfer is contained in kernel space, **bcopy_prot** is used. To transfer a large amount of data, a driver can make many calls to **uiomove**.

## unmap_mem_from_bus()

Unmaps the portion of address space that was mapped using **map_mem_to_bus**.

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

unmap_mem_from_bus (isc, io_parms)
    struct isc_table_type *isc;
    struct io_parms_type *io_parms;
```

***isc*** is the pointer for the interface card wishing to access this memory.

Refer to Appendix A for a description of the *io_parms* struct.

Returns zero on success. See **map_mem_to_bus**().

## unmap_mem_from_host()

Unmaps previously mapped virtual space.

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

unmap_mem_from_host (isc, virt_addr, size)
    struct isc_table_type *isc;
    int virt_addr, size;
```

- *isc*—Pointer that corresponds to an interface card associated with memory.
- *virt_addr*—The virtual address previously obtained with a **map_mem_to_host** call.
- *size*—The size of the memory space in bytes.

This routine panics the kernel if an invalid *virt_addr* is passed in. See **map_mem_to_host**().

## untimeout()

Cancels a **timeout** request.

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

untimeout (func, arg)
    int (*func)();  /* previous timeout routine addr */
    caddr_t arg;    /* previous timeout call arg */
```

## vme_change_adm()

Allows a driver to change the current address modifier in use for a mapping of a VME address onto a host virtual address.

```
vme_change_adm (isc, virt_addr, adm, size)
    struct isc_table_type *isc;
    caddr_t virt_addr;
    int adm, size;
```

*Virt_addr* is the address returned from a previous **map_mem_to_host** call. *Adm* is the new address modifier to use in the mapping. *Size* is the size of the memory space in bytes.

Returns zero on success. This routine is for use by drivers that need to interact with a slave mapper on a card that uses a non-standard address modifier.

## vme_clr()

Clears the most significant bit set by **vme_test_and_set** or **vme_rmw**, to ensure that local caching issues are handled for semaphores in local memory.

```
void vme_clr (isc, address)
    struct isc_table_type *isc;
    caddr_t address;
```

The *isc* is a pointer that corresponds to an interface card. *Address* is the byte address whose most significant bit is to be cleared; it can be either a VME address mapped in via **map_mem_to_host** or a VME-Class memory address. *Address* must not be a local automatic variable to **vme_create_isc**.

## vme_copy()

The **vme_copy** routine moves data between VME and kernel addresses, and between kernel addresses. Host DMA hardware will be used for the transfer if available and appropriate, otherwise **vme_mod_copy** will be used. **vme_copy** will fail if either or both addresses are in user space.

```
vme_copy (to_va, from_va, count, options)
    caddr_t to_va;
    caddr_t from_va;
    unsigned int count;
    int options;
```

*From_va* and *to_va* are virtual addresses representing the transfer source and destination, respectively. At least one must be a kernel RAM address and the other must be a kernel address or a VME address. *Options* contains flags used to select the data width of the transfer, whether to try to use a DMA engine, and whether or not to do a block transfer: VME_D08, VME_D16, VME_D32 (at least one of which must be specified), VME_IGNORE_ADM, BLOCK, VME_OPTIMAL, VME_CPU_COPY, and VME_FASTEST. VME_OPTIMAL specifies that there are no alignment restrictions on the source and destination addresses.

Returns zero on success, <0 for an error, or >0 indicating the residual count of bytes not transferred in the case of an error (such as a VME bus error).

## vme_create_isc()

VME drivers may call **vme_create_isc** to create and use additional ISC entries, but **vme_set_attach_function** returns an *isc* that is sufficient for most purposes.

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

vme_create_isc ()
```

Returns a new *isc* pointer on success, or a null pointer on failure.

## vme_dma_cleanup()

Performs the system cleanup required by a previous, synchronous DMA transfer.

```
#include <sys/io.h>
vme_dma_cleanup (isc, dma_parms)
    struct isc_table_type *isc;
    struct dma_parms *dma_parms;
```

Returns zero. See **vme_dma_setup**().

## vme_dmacopy()

Copies data synchronously between kernel space and VME space by using DMA hardware.

```
vme_dmacopy (dma_parms, vme_addr, vme_adm, options)
    struct dma_parms *dma_parms;
    caddr_t vme_addr;
    int vme_adm, options;
```

- *dma_parms* —The PA address information returned by **vme_dma_setup**, as well as a pointer to the routine to call if the DMA hardware is currently busy. The

*dma_parms*->**dma_options** field is checked for the flags VME_HOST_DMA, DMA_READ, and DMA_WRITE.

- *vme_addr*—The VME bus physical address beginning a contiguous block of memory to be used in the transfer.
- *vme_adm* —The address modifier to use in the transfer. It must be supported by the DMA hardware. These include the A*xx*_LOCK_BIT, A*xx*_SUP_DATA_BIT, A*xx*_NP_DATA_BIT, and A*xx*_BIT modifiers (where *xx* can be 16, 24 or 32).
- o*ptions*—Flags used to select the data width of the transfer: BLOCK, VME_D08, VME_D16, VME_D32, and VME_D64, only one of which must be set.

Returns zero on success, <0 for an error, or >0 indicating the residual number of bytes not transferred in the case of an error (such as a VME bus error). (See See “Synchronous VME to Kernel RAM DMA with vme_dmacopy” on page 4-9..)

## vme_dma_nevermind()

Removes any buffers from the asynchronous DMA queue if they haven’t already been transferred:

```
void vme_dma_nevermind (who)
    caddr_t who;
```

- *who*—A pointer to the *proc* structure (or to any kernel driver address) that identifies the process on whose behalf the asynchronous DMA was queued

Can be called when a process is going away. See **vme_dma_queue**() and **vme_dma_queue_polybuf**().

## vme_dma_queue(), vme_dma_queue_polybuf()

Queues a buffer (or, for **vme_dma_queue_polybuf**, a polybuf) for asynchronous DMA transfer using the local host's DMA engine.

```
void vme_dma_queue (who, buffer, size, vme_address,
        modifier, read_write, priority, callback, arg)
```

```
void vme_dma_queue_polybuf (who, polybuf, size,
        vme_address, modifier, read_write, priority,
        callback, arg)


    caddr_t who;
    addr_t buffer;/* polybuf in vme_dma_queue_polybuf */
    int     size;
    caddr_t vme_address;
    int     modifier, read_write, priority;
    ROUTINE callback;
    caddr_t arg;
```

This function builds needed internal structures to transfer the data. It doesn't check for cache alignment, which you must ensure at both ends of the buffer that you specify.

The parameters have the following significance:

- *who*—A pointer to the ***proc*** structure (or to any kernel driver address) that identifies this process
- *buffer*—The buffer or polybuf to be queued
- *size*—The size of the buffer (limited to MAXPHYS). For **vme_dma_queue_polybuf**(), a *size* of 0 has the effect of asking to DMA as much of the polybuf as possible.
- *vme_address*—The starting VME bus address associated with the buffer
- *modifier*—Address modifier value (VME_A16, VME_A24, VME_A32)
- *read_write*—The direction of the transfer: DMA_READ or DMA_WRITE
- *priority*—Specifies queue priority: DMA_PRIORITY_QUEUE or DMA_NORMAL
- *callback*—Who to call back when the transfer is done
- *arg*—An argument to pass to your callback function when the DMA completes

It returns:

- NO_DMA_HARDWARE if the local host is not a 743/744*i*
- ILLEGAL_CALL_VALUE if *size* is zero, or greater than MAXPHYS.
- INT_ALLOC_FAILED if there was an internal **malloc** problem

## vme_dma_setup()

Takes a description of a DMA transfer to be performed and generates a chain of address/count pairs for use by a VME master.

```
#include "/usr/conf/h/io.h"
```

```
vme_dma_setup (isc, dma_parms)
    struct isc_table_type *isc;
    struct dma_parms *dma_parms;
```

If the system resources (map hardware) needed for the DMA are unavailable, **vme_dma_setup** returns RESOURCE_UNAVAILABLE. Other error indications include:

- UNSUPPORTED_FLAG—An invalid flag is specified in the flags field of the *dma_parms* structure
- RESOURCE_UNAVAILABLE—The request for I/O map entries fails.
- MEMORY_ALLOC_FAILED—Could not allocate the memory required to set up the DMA chain structures.
- TRANSFER_SIZE—Requested transfer size larger than 4 Mbytes is not acceptable because a larger area can't be referenced by the I/O map table in hardware.
- INVALID_OPTIONS_FLAG—An illegal combination of the *dma_options* bits was passed in (for example, a direct-map DMA requested in A16 space).
- BUFLET_ALLOC_FAILED—Unable to allocate space to handle a (possible) shared cache line at the beginning or end of the transfer.

## vme_dma_status()

Returns information on whether a requested, asynchronous DMA transfer is running, queued, or finished:

```
void vme_dma_status (who, buffer)
    caddr_t who;
    caddr_tbuffer;
```

- *who*—A pointer to identify the process whose transfer status is to be checked

- *buffer*—The buffer or polybuf whose status is to be checked

**vme_dma_status** returns the following values:

- DMA_STATUS_QUEUED—The requested *who* and *buffer* are still in the queue.
- DMA_STATUS_RUNNING—The transfer for the requested *buffer* is in process.
- DMA_STATUS_UNKNOWN—The buffer and process could not be identified, usually indicating that the DMA has completed.

If **vme_dma_queue** is called with the callback *arg* set to NULL, you can use **vme_dma_status** to poll for completion. See **vme_dma_queue**() and **vme_dma_queue_polybuf**().

## vme_dma_wait_done()

Finds the asynchronous DMA queue for the specified process. If the queue isn't empty, the routine sleeps until it is.

```
void vme_dma_wait_done (who)
    caddr_t who;
```

- *who*—A pointer to the ***proc*** structure (or to any kernel driver address) that identifies the process whose queue is to be checked

Calling **vme_dma_wait_done** after one or more calls to **vme_dma_queue** essentially turns the asynchronous call into a synchronous one. See **vme_dma_queue**() and **vme_dma_queue_polybuf**().

## vme_fifo_copy()

Transfers data to or from a FIFO.

```
vme_fifo_copy (isc, direction, virtual_addr,
                addr_mod, space, buffer, width, size)
    struct isc_table_type *isc;
    int direction;      /* HOST_TO_VME or VME_TO_HOST */
    caddr_t virtual_addr;
    int addr_mod;
```

```
    space_t space;      /* should always be zero */
    caddr_t buffer;     /* buffer read or written */
    int width;
    int size;           /* num bytes transferred */
```

This is a clone of **vme_mod_copy**, except that the VME side of the transfer is expected to be a FIFO. In other words, only the PA-RISC CPU's side of the transfer's addresses will be auto-incremented.

## vme_fifo_grab()

Locks the FIFO for exclusive use, thereby avoiding interference from other processes.

```
vme_fifo_grab (who,address,modifiers,size,routine,arg)
    struct proc *who;
    caddr_t address;
    int modifiers, size;
    void (*routine)();
    int arg;
```

- *who*—A pointer to the *proc* struct that identifies this process. It need merely be a value unique to the process or drive, as it only serves to help identify which interrupt is being announced.
- *address*—The VME address at which to place the FIFO.
- *modifiers*—Address modifier bit values.
- *size* —The size in bytes of the FIFO address space.
- *routine*—A pointer to a function to be called when the interrupt occurs.
- *arg*—An argument to be passed to *routine*.

Returns non-zero on success, or zero when the FIFO is locked by someone else.

## vme_fifo_poll()

Polls the FIFO to find out if any interrupts have occurred, and if so, how many.

```
vme_fifo_poll (who, answer)
    struct proc *who;
    int *answer;
```

*Who* is a pointer to the same proc struct or the same string used to grab the FIFO and identify this process. *Answer* is a pointer to the returned FIFO status, which can be FIFO_EMPTY, FIFO_FULL, or FIFO_OVERFLOW. If *answer* is on input non-zero and the FIFO is empty when the call is made, the calling process will sleep until the FIFO contains something.

Returns non-zero on success, or zero if the FIFO is locked by someone else.

## vme_fifo_read()

Reads the contents of the FIFO register.

```
vme_fifo_read (who, data)
    struct proc *who;
    int *data;
```

*Who* is a pointer to the same proc struct or the same string used to grab the FIFO and identify this process. *Data* is a pointer to the contents of the FIFO buffer.

Returns non-zero on success, or zero if the FIFO is locked by someone else.

## vme_fifo_release()

Unlocks the exclusive grab that this process has on the FIFO.

```
vme_fifo_release(who)
          struct proc *who;
```

*Who* is a pointer to the same proc struct or the same string used to grab the FIFO and identify this process.

Returns non-zero on success, or zero if the FIFO is locked by someone else.

## vme_generate_interrupt()

Generates an interrupt on the VME bus. This interrupt is to be handled by a VME interrupt handler other than the VME-Class processor on which the call is made.

```
int vme_generate_interrupt(isc, level, width, status_id)
    struct isc_table_type *isc;
    int level;
    int width;
    unsigned int  status_id;
```

This routine generates a VME interrupt on interrupt line *level* of the VME bus. The VME interrupt acknowledge (IACK) cycle will be a *width*-wide cycle with the specified *status_id*.

*Width* can be one of the following: D08_IRQ_TYPE, D016_IRQ_TYPE, or D032_IRQ_TYPE.

Returns zero on success, or non-zero on failure.

## vme_get_address_space()

Gets available VME space.

```
int vme_get_address_space (isc)
    struct isc_table_type *isc;
```

*isc* is a pointer for the interface card and will later be used as a parameter to **map_mem_to_host**.

Returns an address space type (in the *isc)* on success; otherwise returns zero.

## vme_get_cpu_number()

Returns the CPU number (0 to 31) that has been previously set for the local host by **vme_config** (see “Processor Records” on page 8-7), or from the firmware menu while booting.

```
int vme_get_cpu_number()
```

## vme_get_dirwin_host_address()

Returns the host address mapped by the direct mapper to the specified *vme_address* according to whether the address *modifier* is in A24 or A32 space and is enabled.

```
int vme_get_dirwin_host_address (vme_address, modifier)
    caddr_t vme_address;
    unsigned int modifier;
```

Returns 0 if the address is not mapped.

## vme_get_iomap_host_address()

Returns the host address mapped by the slave mapper to the specified *vme_address*, according to whether the address *modifier* is in A24 or A32 space.

```
int vme_get_iomap_host_address (vme_address, modifier)
    caddr_t vme_address;
    unsigned int modifier;
```

Returns 0 if the area is not mapped, or if the system is not a 743/744.

## vme_get_status_id_type()

Gets VME interrupt handling types.

```
int vme_get_status_id_type (isc)
           struct isc_table_type *isc;
```

*isc* is the pointer to the interface card.

Returns an address space type (in the *isc*) on success; otherwise returns zero.

## vme_hardware_info()

Fills in the *vme_hardware_type* structure with the parameters for the system on which the driver is currently running.

```
int vme_hardware_info (hw_type)]
    struct vme_hardware_type *hw_type;
```

*hw_type* is a pointer to the data structure to hold the returned parameters.

Returns non-zero.

## vme_hardware_map_info()

**vme_hardware_map_info**() obtains information, such as address range, base, and address modifiers, that is pertinent to the direct and slave I/O maps on the current system.

```
int vme_hardware_map_info (hw_map_type)
    struct vme_hardware_map_type *hw_map_type;
```

See “vme_hardware_map_type” on page A-18 for a description of the structure.

## vme_isrlink()

Links the interrupt service routine and VME-generated status ID.

```
int vme_isrlink (isc, isr_addr, status_id,
                 arg, sw_trig_lvl)
    struct isc_table_type *isc;
    caddr_t isr_addr;       /* addr of driver ISR */
    int status_id;          /* VME status ID */
    int arg;                /* passed to the ISR /*
    int sw_trig_lvl;        /* currently ignored */
```

Returns zero if the requested status ID could not be obtained. If *status_id* is zero, the next available status ID for this type interrupter will be returned, if available.

If *arg* is zero, the status ID from the interrupt is passed to the driver ISR, otherwise the value of *arg* is passed.

## vme_isrunlink()

Disassociates the interrupt service routine and the VME-generated status ID.

```
int vme_isrunlink (isc, status_id)
    struct isc_table_type *isc;
    int status_id;
```

Before this call is made, interrupts should be disabled on the card generating this status ID, or a system panic will occur.

## vme_locmon_grab()

Locks the Location Monitor for exclusive use for this process.

```
vme_locmon_grab (who, address, modifiers,
                 size, routine, arg)
    struct proc *who;
    caddr_t address;
    int modifiers, size;
    int (*routine)();
    int arg;
```

- *who*—A pointer to the *proc* struct that identifies this process. It need merely be a value unique to the process or drive, as it only serves to help identify which interrupt is being announced.
- *address*—The VME address of the start of the area to be monitored.
- *modifiers*—Address modifier bit values that may be OR'd together: A*xx*_LOCK_BIT, A*xx*_SUP_DATA_BIT, A*xx*_NP_DATA_BIT, and A*xx*_BIT (where *xx* can be 16, 24 or 32).
- *size* —The size in bytes of the VME address space to monitor.
- *routine*—A pointer to a function to be called when the interrupt occurs to announce that the range has been written into or read from.
- *arg*—An argument to be passed to *routine.*

Returns non-zero on success; returns zero on failure.

## vme_locmon_poll()

Polls the Location Monitor to find out if any interrupts have occurred, and if so, how many.

```
vme_locmon_poll (who, answer)
    struct proc *who;
    int *answer;
```

*Who* is a pointer to the same *proc* struct or other ID used to grab the Location Monitor and identify this process. *Answer* is the count representing the number of times that the Location Monitor has noticed a read or write to the address or address range.

Returns non-zero on success; returns zero on failure.

## vme_locmon_release()

Unlocks the exclusive grab that this process has on the Location Monitor.

```
vme_locmon_release(who)
    struct proc *who;
```

*Who* is a pointer to the same *proc* struct or the same string used to grab the Location Monitor and identify this process.

Returns non-zero on success; returns zero on failure.

## vme_map_largest_to_bus ()

Maps the largest possible number of pages beginning at the host address that you specify.

```
caddr_t vme_map_largest_to_bus(isc, io_parms)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
```

This function interrogates the slave mapper for the number of pages required for the size buffer that you specify, then calls **vme_map_pages_to_bus**.

Before calling **vme_map_largest_to_bus**, first set *io_parms*->**host_addr** and *io_parms*->**size** to specify the host address and the size of the buffer to be mapped. If the size can't be mapped, *io_parms*->**size** is adjusted down to the largest buffer that can be mapped.

## vme_map_mem_to_bus2()

Maps a host virtual address to specific a VME bus space address.

```
caddr_t vme_map_mem_to_bus2 (isc, io_parms, vme_addr)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
    caddr_t vme_addr;
```

- *isc*—The pointer for the interface card needing to access this memory.
- *io_parms*—A struct defining certain variables used by **map_mem_to_bus2**, especially *io_parms*->***host_addr*** (the host virtual address to be mapped to I/O space) and *io_parms*->**size** (the size of the space to be mapped in bytes).
- *vme_addr*—The VME bus physical starting address desired.

Returns null on failure or a pointer in VME space corresponding to the PA-RISC CPU's host memory location.

## vme_map_pages_to_bus()

Maps a specified number of pages and returns a pointer to the base of the slave area that is mapped.

```
caddr_t vme_map_pages_to_bus (isc, io_parms, pages,
                              num_pages_p, all_or_nothing)
    struct isc_table_type *isc;
    struct io_parms *io_parms;
    char **pages;
```

```
    unsigned int *num_pages_p; /*how many to map */
    BOOLEAN all_or_nothing; /* map all pages or none */
```

This is similar to **map_mem_to_bus,** except that it maps not necessarily contiguous pages instead of a contiguous buffer, and *io_parms*->**host_addr** does not need to be set.

## vme_map_polybuf_to_bus()

Maps an array of pointers to multiple buffers, to the beginning of the host address that you specify.

```
caddr_t vme_map_polybuf_to_bus (isc, io_parms,
                                max_pages)
struct isc_table_type *isc;
struct io_parms *io_parms;
uint max_pages;/* 0 implies a max of 256, 4-KB pages */
```

Before calling this routine, set *io_parms*->**host_addr** to the *polybuf* struct containing pointers to the regions that you want to map;
set *io_parms*->**drv_routine** to a callback function when the mapping is complete, and *io_parms*->**arg** to a value to be sent to the callback.

## vme_mod_copy()

Transfers data to or from VME with VME address modifier.

```
vme_mod_copy (isc, direction, virtual_addr,
                addr_mod, space, buffer, width, size)
    struct isc_table_type *isc;
    int direction;      /* HOST_TO_VME or VME_TO_HOST */
    caddr_t virtual_addr;
    int addr_mod;
    space_t space;          /* should always be zero */
    caddr_t buffer;         /* buffer read or written */
    int width;
    int size;               /* num bytes transferred */
```

The *isc* is a pointer to the structure returned from a **vme_init_isc**() call. *Direction* is the direction of transfer: HOST_TO_VME or VME_TO_HOST. The *virtual_addr* is the kernel virtual address returned from a call to **map_mem_to_host**, and *addr_mod* is the VME address modifier code required by your interface card. *Width* can be OPTIMAL, BYTE_WIDE, SHORT_WIDE, or LONG_WIDE.

Returns zero on success. Returns an error indication of -1 if the initial address and the size are not an even multiple of the cycle type (*width*). This routine runs internally at **spl6**() priority.

## vme_reg_read()

Does a protected, single-transfer read or write.

```
vme_reg_read (isc, to_va, from_va, width, addr_mod)
    struct isc_table_type *isc;
    caddr_t to_va, from_va;
    int width, addr_mod;
```

*Isc* is the pointer that corresponds to an interface card associated with the VME address. *To_va* is the destination address (user or kernel space). *From_va* is the source address (previously mapped VME address). *Width* specifies the width of the transfer. *Addr_mod* is the address modifier to use for the transfer.

Returns the number of bytes transferred.

## vme_reg_write()

Does a protected single transfer write.

```
vme_reg_write (isc, to_va, from_va, width, addr_mod)
    struct isc_table_type *isc;
    caddr_t to_va, from_va;
    int width, addr_mod;
```

*To_va* is the destination address (previously mapped VME address). *From_va* is the source address (user or kernel space address).

Refer to **vme_reg_read** for the remaining field definitions.

Returns the number of bytes transferred.

## vme_remap_mem_to_host()

Updates the master mapper's list of entries and their address modifiers.

```
int vme_remap_mem_to_host (isc, pa_address, size,
                           vme_address)
    struct isc_table_type *isc;
    caddr_t pa_address;
    unsigned int size, vme_address;
```

The *pa_address* is the local address for which a mapper entry has already been assigned. The *size* must be the same size used in an earlier call to **vme_map_mem_to_host**. The *vme_address* is the new beginning of the VME region.

Be aware that **map_mem_to_host** assigned the I/O addresses in 4-MB chunks, which can only be moved or remapped in 4-MB increments. If your address range was smaller than 4 MB, only part of the 4-MB entity is mapped by the I/O map. Moving the VME address range by a multiple of 4-MB does not change the portion of the I/O map range that has been mapped.

## vme_rmw()

Allows implementation of VME semaphores using RMW (read-modified-write) cycles.

```
int vme_rmw (isc, address)
    struct isc_table_type *isc;
    caddr_t address;
```

Returns zero if the most significant bit of the byte pointed to by *address* is clear; returns >0 if it is set.

## vme_set_address_space()

Sets available VME space.

```
int vme_set_address_space (isc, value_addr)
    struct isc_table_type *isc;
    int value_addr;
```

*Value_addr* is an address space of VME_COMPT, VME_A16, VME_A24, VME_A32, or VME_UD. Or an interrupter type of UNKNOWN_IRQ_TYPE, D08_IRQ_TYPE, D16_IRQ_TYPE, or D32_IRQ_TYPE.

Returns non-zero on success, or zero on failure.

## vme_set_attach_function()

Registers a function to call at system initialization.

```
void vme_set_attach_function (your_attach_string,
                              your_attach_function)
    char *your_attach_string;
    int (*your_attach_function)();

int your_attach_function (id, isc)
    int id;
    struct isc_table_type *isc;
```

*Your_attach_string* points to a string that the kernel associates with the particular attach function being registered. It is returned when the attach function is called. *Your_attach_function* points to the attach function that is called later by VME Services during initialization. *Id* is a pointer to a string that VME Services associates with the attach function being registered. This is the same *your_attach_string* that was specified in the call to **vme_set_attach_function**(). *Isc* is the pointer for the interface card.

## vme_set_mem_error_handler()

Associates an error handler for a master-mapped address range.

```
void vme_set_mem_error_handler (isc, pa_address, size,
                                handler, arg)
    struct isc_table_type *isc;
    unsigned int pa_address;
    int size;
    ROUTINE handler;
    caddr_t arg;
```

This routine assigns an error handler to each 4-MB entry of the master mapper for the specified A24 or A32 address space.

The *pa_address* and *size* of the buffer to associate with the error handler were obtained by a prior call to **map_mem_to_host**. The *handler* is a pointer to the error handler for the master mapper entry (or entries), and *arg* is an argument to be passed to the error handler when a VME bus error occurs.

This routine is an alternative to **vme_testr** and **vme_testw**.

## vme_set_status_id_type()

Sets VME interrupt handling types.

```
int vme_set_status_id_type (isc, value_type)
    struct isc_table_type *isc;
    int value_type;
```

- *isc*—A pointer for the interface card needing to access this memory, previously initialized by a call to **map_mem_to_host**().
- *value_type*—An interrupter type: UNKNOWN_IRQ_TYPE, D08_IRQ_TYPE, D16_IRQ_TYPE, or D32_IRQ_TYPE.

Returns non-zero on success, or zero on failure.

## vme_test_and_set()

Tests for zero in MSB and sets MSB if it is not set.

```
int vme_test_and_set (isc, address)
    struct isc_table_type *isc;
    caddr_t address;
```

Returns >0 if the most significant bit of the byte pointed to by *address* is set. Returns zero and sets the most significant bit of the byte pointed to by *address*, if the bit is clear.

## vme_testr()

Tests for the presence of a device acknowledging a *width*-wide read access to *virtual_addr* .

```
int vme_testr (isc, virtual_addr, width, options)
    struct isc_table_type *isc;
    caddr_t virtual_addr;       /* address to check */
    int width;  /* BYTE_WIDE, SHORT_WIDE, LONG_WIDE */
    int options;
```

*Options* are defined in **vme2.h** and provide for various mixes of privileged and non-privileged data and program access in short, standard, and extended modes: SHORT_NP_ACCESS, SHORT_SUP_ACCESS, STD_NP_DATA_ACCESS, STD_NP_PRGRM_ACCESS, STD_SUP_DATA_ACCESS, STD_SUP_PRGRM_ACCESS, EXT_NP_DATA_ACCESS, EXT_NP_PRGRM_ACCESS, EXT_SUP_DATA_ACCESS, and EXT_SUP_PRGRM_ACCESS.

Returns >0 if the access was acknowledged by the VME device, or -1 for usage errors. Returns zero if a VME bus error is generated.

## vme_testw()

Tests for the presence of a device acknowledging a *width*-wide write access to *virtual_addr*.

```
int vme_testw (isc, virtual_addr, width, options)
    struct isc_table_type *isc;
    caddr_t virtual_addr;       /* address to check */
    int width;  /* BYTE_WIDE, SHORT_WIDE, LONG_WIDE */
    int options;
```

*Options* are the same as for **vme_testr**. Returns >0 if the access is acknowledged by the VME device; returns -1 for usage errors; and returns zero if a VME bus error is generated.

## wakeup()

Wakes all processes sleeping on *address*.

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

wakeup (address)
    caddr_t address; /* Wakes process on this address */
```

The **wakeup** routine awakens all processes sleeping on *address* and puts them on the scheduling queue with the *priority* they specified in their calls to **sleep**. If two or more have the same priority, which one executes first is indeterminate.

# C

# Porting Device Drivers

This appendix provides basic pointers to help you port your VME driver to the release of HP-UX and VME Services described in this manual.

# Porting HP-UX 9.x Drivers

This section contains notes on updating pre-10.0 VME drivers.

As of 10.0, there are six major differences: a new **vme_set_attach_function**(); new access functions to select addresses and interrupter types; a new **vme_create_isc**() function to replace **vme_init_isc**(); new **vme_** calls to replace several **io_** calls; a new **vme_hardware_info**() function; and new steps in building and installing your driver.

(For the new functions at 10.1, see “New 10.1 Functions” on page C-3.)

## New Attach Function and Process

**vme_set_attach_function**(2) specifies an attach function to be registered with VME Services when the HP-UX kernel is initialized during system start-up. This attach function and the initialization process involving it are the primary features that distinguish a 10.0-compliant driver and its initialization by VME Services.

During initialization, each VME driver installed in the kernel specifies an install function named *driver*_**install**(). This install function is responsible for calling **vme_set_attach_function** to register the driver attach function and then **wsio_install_driver** to register the *wsio* data structures that specify the functions and configuration data associated with the driver.

When VME Services calls your *driver*_**attach**() function, you can perform whatever initialization is appropriate for your driver, and then use **isc_claim**() to accept or reject the *isc* that your function receives.

## Other New Functions

### New Access Functions

There are new access functions to select an address and interrupter type:

- **vme_get_address_space**(*isc*) and **vme_set_address_space** (*isc*, *value*)
- **vme_get_status_id_type**(*isc*) and **vme_set_status_id_type**(*isc*, *value*)

Prior to 10.0, you could use the *isc*->**if_info** field to get this information, but now it points to a *wsio* structure and the old VME *if_info* struct has been obsoleted.

### New Create ISC Function

Prior to 10.0, you would call **vme_init_isc**(0) to create an *isc*, but this has been replaced by **vme_create_ isc**(), which takes no arguments. This new function assigns you an *isc*.

### New 10.0 vme_ Calls that Replace io_ Calls

Six **io_** calls (**io_dma_setup**, **io_dma_cleanup**, **io_testr**, **io_testw**, **io_isrlink**, and **io_isrunlink**) have been replaced by calls with **vme_** in their names (**vme_dma_setup**, **vme_dma_cleanup**, **vme_testr**, **vme_testw**, **vme_isrlink**, and **vme_isrunlink**).

### New vme_hardware_info() Function

Replacing **get_vme_hardware**(), **vme_hardware_info**() is used to fill in the *vme_hardware_type* structure with the parameters for the system that the driver is currently running on.

### New 10.1 Functions

For 10.1, VME Services has been updated to provide additional function calls for

- Asynchronous DMA purposes relating to queues, status, and interrupts: **vme_dma_cleanup**(), **vme_dma_nevermind**(), **vme_dma_queue**(), **vme_dma_status**(), and **vme_dma_wait_done**()
- Mapping pages and buffers to the VME bus: **vme_map_largest_to_bus**() and

**vme_map_pages_to_bus**()

- Remapping memory: **vme_remap_mem_to_host**()
- Obtaining information on the current direct and slave I/O maps: **vme_hardware_map_info**()
- Obtaining the CPU number of the current process: **vme_get_cpu_number**()

#### New Include File Pathnames

See “Modifying Header File Inclusion” on page C-5.

### Building and Installing Your Driver

There are new procedures to build and install your driver. See “Editing System Files for the Driver” on page 7-6, and the two sections that follow it.

## Porting Third-Party Drivers

Depending on the origin of the driver that you port, you might need to change it so that it conforms to HP-UX semantics. You will also need to add the *isc* (Interface Select Code) table entries required to use the services provided by the kernel **vme2** driver, for instance in **vme_dma_setup**().

### Modifying Entry Point Routine Names

If your driver contains device driver routines, make the routine names conform to the HP-UX convention:

- Driver routine names for open, close, read, write, strategy, ioctl, and select have names conforming to the convention *driver_routine* where *driver* is a name given to the driver, and *routine* is the routine name (for example, give your open and close routines the names *driver***_open** and *driver***_close**).
- Choose your driver name, *driver*, so that it does not collide with an existing driver name in the second field in the lines in **/usr/conf/master.d/***master*. It is good practice to prepend all global variables in your driver with your driver name because this reduces the risk of colliding with a kernel global variable.

## Modifying Header File Inclusion

You probably need to add or rearrange some `#include` statements to include all the header files needed for an HP-UX driver.

Virtually all drivers need to include the files shown below:

```
#include <sys/types.h>
#include <sys/errno.h>
#include "/usr/conf/wsio/vme2.h"
#include <sys/uio.h>
```

In general, see the manual pages and sections of the manual for each kernel call/structure your driver uses to find out which header files they require.

The minimal list (in order) will usually include:

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/signal.h>
#include <sys/timeout.h>
#include <sys/io.h>
#include <sys/uio.h>
#include "/usr/conf/h/conf.h"
#include "/usr/conf/wsio/vme2.h
#include "mydriver.h"
```

## Checking Driver Entry-Point Routines

### Checking Return Values

Make sure that your driver entry-point routines return a value of 0 if no error occurs. HP-UX generates spurious and meaningless error messages if the return value of these routines is not set to 0 (for no error) or the error number indicating the error that occurred.

### Checking Entry Point Routine Parameters

The parameters to some of the driver entry point routines of your device driver might need to be modified to reflect the routine declarations for *driver*_**open**, *driver*_**close**, *driver*_**read**, *driver*_**write**, *driver*_**ioctl**, and *driver*_**select**.

The declaration of the **open**, **close**, and **select** routines should look like:

```
driver_{open,close,select}(dev,flag)
    dev_t dev;
    int flag;
```

The declaration of the **read** and **write** routines should look like:

```
driver_{read, write}(dev,uio)
    dev_t dev;
    struct uio *uio;
```

The declaration of the **ioctl** routine should look like:

```
driver_ioctl (dev, cmd, arg_ptr, flag)
    dev_t dev;
    int cmd;
    caddr_t arg_ptr;
    int flag;
```

#### Checking Kernel Routine Parameters

For all kernel support routines called by your driver, check the parameters passed to the routine. For example, **physio**() should have the following parameters:

```
physio (strat, bp, dev, rw, mincnt, uio)
    int (*strat)();
    register struct buf *bp;
    dev_t dev;
    int rw;
    unsigned (*mincnt)();
    struct uio *uio;
```

#### Checking the ioctl Command Format

If your driver has an **ioctl** routine, modify the declarations of your **ioctl**(2) commands (usually defined in your driver's header file), if necessary, so they conform to the HP-UX format. See “Defining the Command Parameter” on page 6-19.

#### Checking Timeout Parameters

HP-UX has a fourth parameter in the call, as seen in the signature below:

```
timeout (func, arg, t, timeout)
    int (*func)();  /* routine to call on timeout */
    caddr_t arg;    /* argument passed to it */
    int t;          /* cycle count to wait */
    struct timeout *timeout; /* NULL */
```

This parameter is a pointer to a *timeout* struct. Usually this is called with a NULL parameter and the kernel then allocates the struct to be used by the call.

#### Checking Interrupt Service Routine Parameters

The HP-UX VME services always call VME interrupt service routines with two parameters. These parameters are an *isc* pointer and a second parameter which, depending upon the call made to **vme_isrlink**(), may be a predetermined argument or the status vector provided in the IACK cycle by your VME interface card.

## Miscellaneous Suggestions

#### Kernel Mapping Routines

The only supported mapping calls are those contained in Appendix B. Specifically, **map_mem_to_host**(), **map_mem_to_bus**(), **vme_map_largest_to_bus**(), **vme_map_mem_to_bus2**(), and **vme_map_pages_to_bus**(). You will have to change other systems mapping calls accordingly.

#### Kernel Address Conversion and Buffer Alignment

If you are not using the **vme_dma_setup** and **vme_dma_cleanup** routines when doing DMA, it is up to your driver to ensure, for example, processor cache consistency by using the **io_flushcache** and **io_purgecache** routines described in Appendix B, in “Flushing Cache” on page 4-21, and in “Purging Cache” on page 4-22. Hewlett-Packard recommends that you use the provided **vme_dma_setup** and **vme_dma_cleanup** routines for all DMA transactions.

If a driver needs to do a kernel logical-to-physical address conversion, the **kvtophys**() kernel utility routine is provided. You pass it a logical memory address and it returns the corresponding physical address.

If your driver is performing word or long-word DMA (16 or 32 bit), your driver may need to check the alignment and size of the data buffer to ensure that your hardware can handle the transfer. If a buffer is not aligned properly for DMA for your device, it is usually possible to line it up by handshaking one or more bytes at the beginning or the end of the buffer.

**Unique Routines**

Finally, the various mapping and copy routines are in general unique to HP-UX VME.

# D

# Skel Device Drivers

This appendix shows the skeletons for two VME device drivers. They're called **skeleton** because they're just basic instructional sketches or models rather than actual drivers that can be compiled and executed.

The first is an example of a non-DMA driver.

## Non-DMA Skeleton Driver

The handle chosen is **skel**.

```
#include <sys/errno.h>
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include "/usr/conf/wsio/timeout.h"
#include <sys/io.h>
#include "/usr/conf/wsio/vme2.h"
#include <sys/ioctl.h>
#include <sys/user.h>
#include "skel.h"

#define TRUE 1
#define FALSE 0

/* global data structure declarations */

static struct skelregs *Skel_Regs; /* board address */
static struct buf Skel_Buf; /* io buffer     */
static int Skel_Opened = FALSE; /* driver has been opened */

#define SKELIRQ          0x2 /* interrupt level */
#define SKEL_CARD_ADDR   0x10000 /* VME bus address */
#define SKEL_REG_SIZE    0x200

#define SKEL_STRING      "Acme Skeleton Device"

#define SKEL_IOCTL1   _IO('s', 1)
#define SKEL_IOCTL2   _IO('s', 2)
#define SKEL_IOCTL3   _IO('s', 3)
```

### skel_isr()

```
int skel_isr(isc, arg)
struct isc_table_type *isc;
int arg;

{
     /* determine if board caused interrupt */
     /* reset card here */
     /* clear interrupt on card */
     if (/* card indicates transfer is complete */)
         iodone(&Skel_Buf);
     else
         wakeup(&Skel_Buf);
}
```

### skel_open() and skel_close()

Perform initialization on first open. Allow only one open at a time.

```
skel_open(dev, flag)
dev_t dev;
int flag;

{
    /* The following implements exclusive open */
    if (Skel_Opened)
        return(EACCESS);

    Skel_Opened = TRUE;
    return(0);
}


skel_close(dev)
dev_t dev;
{
     Skel_Opened = FALSE;
    return(0);
}
```

### skel_strategy()

The strategy routine is responsible for setting the hardware up for a transfer and starting the transfer.

```
skel_strategy(bp)
struct buf *bp;

{
    int pri;
    register caddr_t dmaddr;
    short count;

    pri = spl4();
    dmaddr = bp->b_un.b_addr;
    count = bp->b_bcount;
     if (bp->b_flags & B_READ) {
         /* prepare card for read transfer */
    }
    else {
      /* prepare card for write transfer */
    }

    /* start transfer of data */
      splx(pri);
}
```

### skel_read()

Read up to *count* bytes of data from the device into *buf*.

```
skel_read(dev, uio)
dev_t dev;
struct uio *uio;

{
    /* The read is implemented through the strategy routine */
    return(physio(skel_strategy, &&Skel_Buf, dev, B_READ,
minphys, uio));
}
```

### skel_write

Write *count* bytes of data from *buf* to the device.

```
skel_write(dev, uio)
dev_t dev;
struct uio *uio;

{
    /* The write is implemented through the strategy routine */
    return(physio(skel_strategy, &&Skel_Buf, dev, B_WRITE,
                                minphys, uio));
}
```

### skel_ioctl()

The **ioctl** routine is used to implement the remaining functions. Since they require at most one argument, the third address is taken to be an argument passed by value.

```
skel_ioctl(dev, command, arg)
dev_t dev;
int command;
caddr_t arg;

{
    switch(command) {
         case SKEL_IOCTL1:
            /* Call routine or insert code to do first ioctl */
            break;
         case SKEL_IOCTL2:
            /* Call routine or insert code to do second ioctl */
            break;
         case SKEL_IOCTL3:
            /* Call routine or insert code to do third ioctl */
            break;
         default:
            return(EINVAL);
    }
return(0);
}
```

## Structs

```
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static drv_ops_t skel_ops =
{
    skel_open,
    skel_close,
    skel_strategy,
    NULL,
    NULL,
    NULL,
    skel_read,
    skel_write,
    skel_ioctl,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    C_ALLCLOSES
};
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static drv_info_t skel_info =
{
    "skel",
    "skel class",
    DRV_CHAR | DRV_SAVE_CONF,
    -1,
    SKEL_CHAR_MAJOR,
    NULL,
    NULL,
    NULL
};
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static wsio_drv_data_t skel_data =
{
    "skel",
    T_INTERFACE,
    DRV_CONVERGED,
    NULL,
    NULL
};
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static wsio_drv_info_t skel_wsio_info =
{
    &skel_info,
```

```
    &skel_ops,
    &skel_data
};
/*-------------------------------------------------------------*/
```

## skel_attach()

```
skel_attach(id, isc)
int id;
struct isc_table_type *isc;

{
     VME_INIT_IF_INFO(isc, D08_IRQ_TYPE, VME_COMPT);
    Skel_Card->isc = isc;   /* squirrel away the isc */
     /* link in the DMA interrupt service routine */
    if ((Skel_Card->vector_number =
              vme_isrlink(isc, skel_isr, 0, 0, 0)) < 0)
         return;
/* map in the card registers */
    skel = (struct skel_regs *)
          map_mem_to_host(isc, board_addrs, board.size);
    if (skel == NULL)
         return;
     isc->card_ptr = (int) skel;

    /* test if card is there */
    if (vme_testr(isc, (caddr_t) skel, BYTE_WIDE, 0) <= 0) {
         unmap_mem_from_host(isc, skel, SKEL_REG_SIZE);
         isc_claim(isc, NULL);   /* ours but no hardware there */
    }

    /* do card specific initialization
       .
       .
       .
    */

     isc_claim(isc, &skel_wsio_info);
    Skel_Not_Attached = FALSE;
}
```

### skel_install()

```
skel_install()
{
    vme_set_attach_function(SKEL_STRING, skel_attach);
    return(wsio_install_driver(&skel_wsio_info));
}
```

## VME DMA Driver Skeleton

This section shows the skeleton for a VME driver with DMA. The parts of the skeleton appear under subsections that name the routines.

```
#include <sys/errno.h>
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/io.h>
#include <sys/ioctl.h>
#include <sys/param.h>
#include <sys/user.h>
#include “/usr/conf/wsio/vme2.h”

#define TRUE 1
#define FALSE 0

/* global data structure declarations */

static struct skelregs *Skel_Regs;      /* board address */
static struct buf Skel_Buf;             /* io buffer     */
static struct skel_card_info *Skel_Card;
static int Skel_Not_Attached = TRUE;      /* code to make sure
attached */

#define SKEL_IOCTL1   _IO('s', 1)
```

### skel_isr()

```
skel_isr(isc, arg)
struct isc_table_type *isc;
int arg;

{
    struct buf *bp = isc->owner;
    struct dma_parms *dp = (struct dma_parms *) isc->dma_parms;
    int index;

    /* determine if board caused interrupt
       reset card here
       clear interrupt on card
    */
/* was there a bus error */

    if (card_ptr->status & BERR) {
        card_ptr->control &= ~BERR;
        bp->b_error |= EIO;
    } else if (card_ptr->tfr_count) {   /* did DMA complete? */
        /* not complete calculate resid */
        bp->b_resid += card_ptr->tfr_count;
        for (index = dp->chain_index + 1;
                      index < dp->chain_count; index++)
            bp->b_resid += dp->chain_ptr[index].count;
    } else if (dp->chain_index != dp->chain_count) {
     /* more elements to do */
        skel_dma_setup(isc);
        skel_dma_start(isc);
        return;
    }

    iodone(bp);
    io_dma_cleanup(isc,dp);
    isc->owner = NULL;
}
```

### skel_open() and skel_close()

```
skel_open(dev, flag)
dev_t dev;
int flag;

{
    if (Skel_Not_Attached)
        return(ENXIO);

    /* enforce exclusive open, if exclusive open device  */
    /* do any open time driver/hardware initialization */

    return(0);
}
skel_close(dev)
dev_t dev;

{
    /* clear exclusive open flag, if exclusive open device */
    /* do any close time driver/hardware cleanup           */

    return(0);
}
```

### skel_strategy()

```
skel_strategy(bp)
struct buf *bp;

{
    struct isc_table_type *isc = bp->sc;
    struct dma_parms *dma_parms;
    int ret, pri;

    isc->owner = bp;
    dma_parms = isc->dma_parms;
    bzero(dma_parms, sizeof(struct dma_parms));
    dma_parms->addr = bp->b_un.b_addr;
    dma_parms->count = bp->b_un.b_count;
    dma_parms->flags = 0;
    dma_parms->drv_routine = skel_wakeup;

    /* set flags according to transfer desired */
    if (SKEL_TRANSFER TYPE == A32)
        dma_parms->dma_options |= VME_A32_DMA;
```

```
    else
        dma_parms->dma_options |= VME_A24_DMA;
    if (/* want to use the slave mapper */)
        dma_parms->dma_options |= VME_USE_IOMAP;

    pri = spl6();
    while (ret = io_dma_setup(isc, dma_parms))
        if (ret = RESOURCE_UNAVAILABLE)
            sleep(dma_parms, PZERO+2);
        else if (ret < 0) {
            bp->b_error = EINVAL;
            iodone(bp);
            splx(pri);
            return(0);
        }
    splx(pri);

    /* reset card */
    if (bp->b_flags & B_READ)
        /* Perform general read specific card set up */
    else
        /* Perform general write specific card set up */

    skel_dma_setup(isc);
    skel_dma_start(isc);
}
```

## skel_read() and skel_write()

```
skel_read(dev, uio)
dev_t dev;
struct uio *uio;

{
    return(physio(skel_strategy, &&Skel_Buf, dev, B_READ,
minphys, uio));
}
skel_write(dev, uio)
dev_t dev;
struct uio *uio;

{
    return(physio(skel_strategy, &&Skel_Buf, dev,
             B_WRITE, minphys, uio));
}
```

## skel_dma_setup() and skel_dma_start()

```
skel_dma_setup(isc)
struct isc_table_type *isc;

{
    /* local variable declarations */
    struct dma_parms *dmaparms =
                  (struct dmaparms *)isc->dma_parms;
    struct buf * bp = (struct buf *)isc->owner;
    int index = dmaparms->chain_index;

    /* Reset card */

    if (bp->b_flags & B_READ) {
        /* Perform read specific card set up */
    } else {
        /* Perform write specific card set up */
    }

    /* set up dma address and transfer count */

    card_ptr->dma_addr = (unsigned long)
              dma_parms->chain_ptr[index].phys_addr;
    card_ptr->dma_count = (short)
              dma_parms->chain_ptr[index].count;

/* Advance to next transfer on dma chain (for next transfer) */
    dma_parms->chain_index++;
}
skel_dma_start(isc)
struct isc_table_type *isc;

{
    struct buf *bp = (struct buf *) isc->owner;

    if (bp->b_flags & B_READ)
        /* Do any read specific start-up here  */
    else
        /* Do any write specific start-up here */

    /* Enable interrupts, if not already enabled   */
    /* Start card doing dma transfer               */
}
```

### skel_ioctl()

```
skel_ioctl(dev, command, arg)
dev_t dev;
int command;
caddr_t arg;

{
    switch(command) {
        case SKEL_IOCTL1:
            /* Call routine or insert code to do first ioctl */
            break;

        default:
            return(EINVAL);
    }

    return(0);
}
```

### Structures

```
static drv_ops_t skel_ops =
{
    skel_open,
    skel_close,
    skel_strategy,
    NULL,
    NULL,
    NULL,
    skel_read,
    skel_write,
    skel_ioctl,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    C_ALLCLOSES
};
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static drv_info_t skel_info =
{
    "skel",
    "skel class",
```

```
    DRV_CHAR | DRV_SAVE_CONF,
    -1,
    SKEL_CHAR_MAJOR,
    NULL,
    NULL,
    NULL
};
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static wsio_drv_data_t skel_data =
{
    "skel",
    T_INTERFACE,
    DRV_CONVERGED,
    NULL,
    NULL
};
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
static wsio_drv_info_t skel_wsio_info =
{
    &skel_info,
    &skel_ops,
    &skel_data
};
/*-------------------------------------------------------------*/
```

## skel_attach()

```
skel_attach(id, isc)
int id;
struct isc_table_type *isc;

{
    if (strcmp(id, SKEL_STRING))    /* is this for us? */
        return;                     /* nope! */

    VME_INIT_IF_INFO(isc, D08_IRQ_TYPE, VME_COMPT);
    Skel_Card->isc = isc;   /* squirrel away the isc */

    /* link in the DMA interrupt service routine */
    if ((Skel_Card->vector_number =
              vme_isrlink(isc, skel_isr, 0, 0, 0)) < 0)
        return;

    /* map in the card registers */
    skel = (struct skel_regs *)
              map_mem_to_host(isc, board_addrs, board.size);
```

```
    if (skel == NULL)
        return;

    isc->card_ptr = (int) skel;

    /* test if card is there */
    if (vme_testr(isc, (caddr_t) skel, BYTE_WIDE, 0) <= 0) {
        unmap_mem_from_host(isc, skel, SKEL_REG_SIZE);
        isc_claim(isc, NULL);   /* ours but no hardware there */
    }

    /* do card specific initialization
       .
       .
       .
    */

    isc->dma_parms =
        (struct dma_parms *)
             io_malloc(sizeof(struct dma_parms), IOM_NOWAIT);
    if (isc->dma_parms == NULL) {
        unmap_mem_from_host(isc, skel, SKEL_REG_SIZE);
        return;
    }

    isc_claim(isc, &skel_wsio_info);
    Skel_Not_Attached = FALSE;
}
```

## skel_install()

```
skel_install()
{
    vme_set_attach_function(SKEL_STRING, skel_attach);
    return(wsio_install_driver(&skel_wsio_info));
}
```

# Glossary

**A16** The mnemonic symbol for 16-bit addressing mode (VME address space). Bus masters with A16 capability can generate bus cycles with 16-bit addresses. Bus slaves or location monitors with A16 capability can accept bus cycles with 16-bit addresses. The total available address space is 64KB.

**A24** The mnemonic symbol for 24-bit addressing mode (VME address space). Bus masters with A24 capability can generate bus cycles with 24-bit addresses. Bus slaves or location monitors with A24 capability can accept bus cycles with 24-bit addresses. The 24-bit addresses are sometimes called **standard addresses**. The total available address space is 16MB.

**A32** The mnemonic symbol for 32-bit addressing mode (VME address space). Bus masters with A32 capability can generate bus cycles with 32-bit addresses. Bus slaves or location monitors with A32 capability can accept bus cycles with 32-bit addresses. The 32-bit addresses are sometimes called **extended addresses**. The total available address space is 4GB.

**anchor** A common point of reference for all processors connected to a common VME backplane. See **backplane anchor**.

**arbiter** A VME board that accepts bus requests and grants control of the data-transfer bus to one requester at a time. The slot 1 controller functions as the arbiter.

**arbitration** The process of assigning control of the data-transfer bus to a requester. In the HP VME configuration file, the method or mode (**arb_mode**) for defining how the system will determine which request level (BR0, BR1, BR2, or BR3) to handle first. The two arbitration modes are round robin select (RRS) and priority (PR). When the arbitration mode is RRS, the request rotates through the four request levels. When the arbitration mode is PR, BR3 is serviced first.

**ARPA** Advanced Research Projects Agency, a U.S. government research agency instrumental in developing and using the original ARPA services on the ARPANET.

**ARPA/Berkeley services** The networking services originating from the Advanced Research Projects Agency (ARPA) and from the University of California at Berkeley (UCB). ARPA services are used to communicate in HP-UX, UNIX, and non-UNIX environments. UCB services are used for HP-UX or UNIX operations only. ARPA services include File Transfer Protocol (**ftp**), Telnet, and Simple Mail Transfer Protocol (SMTP). Berkeley services include Remote copy (**rcp**), Remote login (**rlogin**), Remote who (**rwho**), Remote execution (**rexec**), Remote shell (**remsh**), and Remote uptime (**ruptime**).

**ASIC** An application-specific integrated circuit such as the VME bus adapter.

**backplane anchor** A common point of reference for all processors connected to a common VME backplane. All processors using VME backplane networking must know the VME address of the back-

plane anchor. By convention, the anchor occupies the beginning of the backplane networking swsm. All processors on the backplane access this swsm by first accessing the backplane anchor, which points to the backplane header.

**backplane header** A part of the backplane networking SWSM. It contains the hardware address of the backplane network and other pointers.

**backplane master processor** The processor that initializes the backplane networking swsm and increments the heartbeat. This processor is assigned CPU number zero.

**block device interface** An interface that accommodates only block-oriented I/O and is generally used for mass storage devices such as disk drives. This type of interface uses a buffer cache, which minimizes the number of I/O requests that actually require an I/O operation to the device.

**block device driver** A device driver that accesses a structured device such as a disk or magnetic tape.

**board computer** A board that contains the CPU, memory subsystem, and I/O subsystem. Board computers, such as the HP 9000 VME-Class*i*, plug into a VME backplane.

**bus** The hardware communication path between different system elements within a given system.

**bus arbiter** A functional module on VMEbus systems that monitors bus requests and grants control of the data transfer bus to one bus requester (master) at a time. It is often called the **system controller**. Each processor on the VME backplane can have an arbiter, but only the arbiter in slot 1 is used for HP 9000 VME-Class workstations. Arbiters are classified by the type of VME arbitration scheduling mode they use. (See the glossary entry “arbitration mode.”)

**bus device** A card or built-in I/O function connected to the local bus.

**character device interface** An interface that accommodates character-oriented I/O, that is, one character at a time, such as a serial communications port. I/O operations are not channeled through a buffer cache. Instead, they take place between the device and your application’s virtual address space. For a block-oriented device, a character device interface becomes an unstructured (raw device) interface.

**character device driver** A device driver that accesses a device a character at a time. Character devices do not use the HP-UX buffer cache.

**configuration** The arrangement of VME boards according to their nature, number, and chief characteristics.

**device number** A 32-bit number that includes both the **major** and **minor numbers** that uniquely identify a particular device driver and instance of a hardware device. Also called a **node device number**.

**DMA** Abbr. for **Direct Memory Ac-**

**cess**, an I/O method that permits data to be transferred directly from memory to an interface card (or vice versa) without CPU intervention.

**driver** The software interface between the HP-UX kernel and a hardware device. A device driver, which resides in the kernel, is either a **block device driver** or a **character device driver**. Generally, block device drivers are for mass storage devices such as disks; and character device drivers are for any other type of I/O peripheral, such as serial communications ports. All mass storage devices must have both a block and a character (raw) device driver.

**EEPROM** System memory that retains its contents even after workstation power is turned off.

**Ethernet address** Ethernet is a local area network (LAN) system developed by Xerox Corporation. It is implemented at the data link and physical layers of networking software. It is used to uniquely identify nodes on the LAN. In a VME environment, each backplane network is assigned a unique 48-bit Ethernet address. Each processor on the VMEbus modifies the last byte of this address to contain its own CPU number.

**expander** Another name for the VME bus adapter.

**FIFO** A first-in, first-out queue written to by other VME bus masters and read by the HP's VME bus adapter chip. You must reserve address space for the FIFO buffer so that it does not conflict with other objects.

**host bus** The physical bus via which main memory is accessed.

**IACK** An interrupt acknowledgment. An interrupt acknowledge cycle is initiated by an interrupt handler in response to an interrupt request. The VME IACK handshake identifies which I/O device on a particular interrupt level has signaled an interrupt. If the interrupt level is set to **normal mode**, the hardware performs the IACK handshake before the CPU receives the interrupt. If the interrupt level is set to **fast mode**, the CPU is interrupted in parallel with the IACK handshake. Fast mode is used when only one I/O device is associated with a particular interrupt level.

**inode** An entry in a table maintained by the kernel for each existing file, identifying such aspects as owner, group, mode, last access time, etc.

**Initial System Loader** Implements the operating system independent portion of the bootstrap process. It is loaded and executed after self-test and initialization have completed successfully. See the **isl (1M)** man page for more information.

**Internet** All ARPA networks that are registered with the Network Information Center.

**Internet address** The network address of a computer node. It has two parts: a network number and a node address. The VME backplane is assigned a unique Internet network number. Each processor on the backplane is assigned a unique Internet Protocol (IP) address.

**interrupt handler** A routine that determines which slave has interrupted and handles the interrupt in a device-specific manner. Interrupt handlers can detect requests on any of the seven interrupt lines of the priority interrupt bus.

**interrupt level** One of seven prioritized hardware lines on which a interrupter or bus master can generate an interrupt. (For more information, refer to the glossary entry “VMEbus interrupt.”)

**interrupter** A VME entity that generates an interrupt request on the priority interrupt bus and then provides status and ID information when the interrupt handler requests it. Interrupters can use any of the seven interrupt lines.

**IP** Abbreviation for Internet Protocol.

**ISL** Abbreviation for Initial System Loader.

**ISR** The interrupt service routine (ISR) that handles interrupts from a device. The kernel calls the ISR when it receives an interrupt from a device.

**kernel threads** A category of threads that execute only in kernel code, including I/O drivers, through the use of system calls.

**kernel virtual address** The address actually generated by a CPU in kernel mode, which is the context in which a driver runs. It is the address which drivers use when referencing memory objects, as well as the address returned from translation services such as when mapping a bus interface device’s registers for access by the driver.

**LAN** Abbreviation for Local Area Network.

**link-level address** See **Ethernet address**.

**local bus** The physical bus to which a bus interface device is attached, either as a system board core I/O device or as an expansion device via a card in an expansion slot.

**location monitor** Monitors a 1KB or 2-byte window of VME memory space for activity, and generates a system interrupt whenever it is written into. Used as a general monitor of activity on the bus, or as a **mailbox interrupt** that generates an interrupt to a board without disrupting VME bus traffic.

**master** A VME entity that can initiate data transfer bus cycles between itself and a slave. Examples include CPU modules and peripherals with DMA controllers.

**network address** The network portion of an IP address. For Class A, B, and C networks, respectively, the network address is the first, first two, and first three bytes of the IP address. In each case, the remainder is the host address. In the Internet, assigned network addresses are globally unique.

**Network File System** A networking service that allows many systems to share the same files. It is not limited to specific hardware or software. It supports Remote File Access.

**NFS** Abbreviation for Network File System.

**PA-RISC** Abbr. for Precision Architecture—Reduced Instruction Set Computer (or Computing).

**panic** What the kernel does when a critical internal consistency check fails in such a way that Unix cannot continue. With VME, this typically happens when an attempt is made to access a location for which VME memory has not been mapped.

**polling** A method for handling input and output (I/O). When using this method, a device does not interrupt. I/O is accomplished by periodically checking the device to see if it is ready to send or receive data. If a processor on the VME backplane uses polling, it periodically polls the swsm data structure for input packets.

**processor** A VME bus master that handles interrupts and may supply local memory.

**pseudo Read-Modify-Write (pseudo_rmw)** A simulated VME RMW cycle in which the processor locks the VMEbus, executes a separate read and write, and then unlocks the VMEbus. Only Model 742*i*, 742*rt*, and 747*i* computers implement this method.

**remote host** A computer that is accessible through the network or via a gateway.

**requester** A functional module that resides on the same board as a master or interrupt handler and requests use of the data transfer bus whenever its master or interrupt handler needs it.

**request level** The request line or level (**req_lev**) on which an interrupt handler will make a request for the VMEbus. The four request levels are BR0, BR1, BR2, and BR3.

**request mode** The manner or mode (**req_mode**) in which a VME master will release the VMEbus. The three modes are release when done (RWD), release on request (ROR), and fair request (FAIR).

**RMW** Abbr. for Read-Modify-Write.

**shared memory** A communication buffer accessible to multiple processors, and used to pass data among them. This memory may be located on one of the processors or on a separate VME memory card.

**slave** A VME board that detects bus cycles generated by masters. When those cycles specify its participation, it transfers data between itself and the master. Examples include memory and I/O modules.

**slot one controller** See **system controller**.

**smart DMA device** DMA for which the bus interface device, generally using asynchronous DMA, is capable of accessing specific control structures in memory to direct its operation to include the generation of multiple ranges of local bus addresses.

**socket** A special type of file: the Berkeley Unix mechanism for creating a virtual connection between processes. Sockets form the interface between Unix standard I/O and network communication facilities. They can be of two types, stream (bi-directional) or datagram (fixed length destination-addressed messages). The socket library function **socket(2)** creates a communications endpoint or socket and returns a file descriptor with which to access that socket. The socket has associated with it a socket address, consisting of a port number and the local host's network address.

**STATUS_ID** A word that contains the return value for each interrupter on a particular interrupt level. When there are different service routines on the same interrupt level, the STATUS_ID is used by the interrupt handler during the IACK routine to determine which driver initiated the interrupt.

**SWSM** Abbreviation for **system-wide shared memory**.

**synchronous DMA** DMA occurring in lock-step with the driver state. The driver sets up the DMA engine and then directs it to execute the I/O, the completion of which is signaled asynchronously via an interrupt or by the driver's polling of the bus interface device status. The driver state, or at least the state of the I/O, remains synchronized with the hardware state.

**system call** A kernel process invoked by a user program.

**system controller** The module on the VME backplane that serves as the bus arbiter. Its function is to mediate bus requests by preventing more than one master from using the data transfer bus at any one time. This module must reside in slot 1 of the VME backplane. It is sometimes referred to as the **slot one controller**.

**system wide shared memory (SWSM)** A contiguous block of memory that resides on a processor or memory board connected to a VME backplane. A SWSM may exist anywhere in the VME address space, and all supported processors on a common backplane have access to this memory. This memory is assumed to have a fixed allocation in some physical address space for the life of the system. At least one SWSM is reserved for backplane networking. HP-UX sets up access to this area when the system is booted.

**TAS** Abbreviation for test-and-set.

**TCP/IP** Abbreviation for Transmission Control Protocol/Internet Protocol.

**test-and-set** The mechanism used to protect the shared memory resources on a VME backplane. The test-and-set methods used for backplane networking are **true_rmw** and **pseudo_rmw**.

**threads** Collections of multiple streams of execution within normal processes. A thread includes a complete machine state; that is, a set of registers, a program counter (next instruction counter), and a private stack. All of these are used by the CPU for instruction execution. A thread, by itself, does not "own" code or data,

but it does share code and data areas with other threads running within the same process. See also **kernel threads**.

**Transmission Control Protocol** A protocol that provides the underlying communication support for stream sockets. TCP is used to implement reliable, sequenced, flow-controlled, two-way communication based on byte streams similar to pipes.

**true Read-Modify-Write** A VME Read-Modify-Write cycle used for protecting the shared memory resources on a VME backplane. A processor that can accept this type of hardware RMW cycle performs a read and write of the semaphore protecting the shared memory all in one cycle. Models 743*i*, 743*rt*, and 744*i* VMEbus computers implement this type of routine.

**vector number** Every VME bus interrupt requestor must supply an interrupt vector number, which in the VME Services software is identified with the *status_ID* argument in the various calls that service interrupts.

**VME** Abbreviation for Versa Module Eurocard.

**VME backplane** A backplane board for a VME enclosure (usually called a **cardcage**, **crate**, or **chassis**). It uses a mechanical standard known as the **Eurocard format**. The Eurocard format describes the rack and connector standards. The modules that plug into the VME backplane come in two sizes: single height (3U) and double height (6U).

**VME master** A functional module that can initiate data transfer cycles to and from a VME slave: for example, CPU modules and peripherals with Direct Memory Access (DMA) controllers. Before a VME master can transfer data, it must acquire the bus from the **bus arbiter**. Some processors can be configured as both a VME master and a VME slave.

**VME slave** A functional module that can detect bus cycles generated by a bus master. If selected, it can participate in the cycles by transferring data between itself and the master. Some processors can be configured as both a VME slave and a VME master.

**VMEbus** The Versa Module Eurocard industry standard device interfacing bus. The VMEbus is an asynchronous bus that allows communication between processor boards and memory cards from a variety of vendors when they are plugged into a common VME backplane and observe VMEbus protocol specifications. This is accomplished without disturbing the internal activities of other devices in the system. The VMEbus does not distinguish between I/O and memory space, and it supports multiple address spaces. The VMEbus uses a master-slave architecture. Functional modules called **VME masters** transfer data to and from functional modules called **VME slaves**.

**VMEbus interrupt** A method for handling interrupts in a VME backplane system. The VMEbus specification allows prioritized interrupts on seven hardware levels (level 1 through 7). Level one has the lowest priority; level seven has the highest priority. Modules that generate

interrupts have an **interrupter**; modules that service interrupts have an **interrupt handler**. An interrupter can interrupt on any of the seven interrupt levels. An interrupt handler can monitor one or more interrupt levels; for example, a particular interrupt handler could be assigned to handle levels 1, 2, 3, 4, and 7. Each interrupt level can only be monitored by one interrupt handler.

**wsio** Workstation I/O level.

# Index


**Symbols**
#include pre-processor declaration, 8-4
$DRIVER_INSTALL, 7-7
,/etc/rc.config.d/netconf, 9-11
/etc/vme/vme.CFG, 8-14
/sbin/lib/vme/vme.CFG, 8-14
/sbin/vme_config, 8-14
/usr/sbin/ifconfig, 9-11
/var/adm/vme/system.log, 8-14
_attach_string, B-39
_IO, 6-19
_ioctl_arg, 6-20
_IOR, 6-19
_IOW, 6-19
_IOWR, 6-19

**A**
A24 address space
  defined, Gl-1
a24_dirwin_addr, A-18
a24_dirwin_addr_mods, A-19
a24_dirwin_host_base, A-18
a24_dirwin_size, A-18
a24_iomap_addr, A-19
a24_iomap_addr_mods, A-19
A32 address space, Gl-1
  defined, Gl-1
a32_dirwin_addr, A-18, A-20
a32_dirwin_addr_mods, A-19
a32_dirwin_host_base, A-18
a32_dirwin_size, A-18
a32_iomap_addr_mods, A-19
access-result, A-22
acquire_buf, B-2
active, A-14
addr_mod, B-37
  definition, 6-16
address, A-25, B-27, B-33, B-41
  A16 space, 1-7, 2-2
  A24 space, 1-7, 2-2
  A32 space, 1-7, 2-2
  boundary, 8-6
  lines, 2-2
  modifiers, 1-7
  physical page, 1-4
  range, 8-2
  range size, 8-6
  space, 1-7, 8-5, 9-4
  space allocation, 1-7
  space, types of, 1-7
  virtual, 1-4
  virtual page, 1-4
address field formats
  align, 8-6
  starting_address
    size, 8-6, 9-4
  starting-ending, 8-6
address modifier, 2-3
address modifiers, 8-5, 9-4
address space
  user level access, 1-11
addressing
  compatibility mode, 2-9
  enhanced mode, 2-9
adm, B-20
align, 8-6
answer, B-34
Application-Specific Integrated Circuit (ASIC), 1-3
arb_mode, 8-10
arb_mode value
  PRI, 8-10, 8-11
  RRS, 8-10
arbiter, Gl-1
arbiter,, see also bus arbiter
arbitration, Gl-1
arbitration mode, 8-10
arbitration modes, 1-8, 8-10
arbitration modes and request levels
  combining, 8-11
arg, B-17, B-27, B-33
ARPA/Berkeley services
  defined, Gl-1
attach routine
  writing a, 6-2
av_back, A-4, A-6
av_forw, A-4, A-6
avail_interrupts, A-20
available_interrupts, A-22

**B**
b_act, A-11
b_actf, A-11, A-14
b_action, A-4
b_active, A-11
b_actl, A-14
B_ASYNC, A-4
b_ba, A-4
b_back, A-5, A-6
b_bcount, A-4
b_blkno, A-4
b_bufsize, A-4
B_BUSY, 6-12, A-4, B-13
B_CALL, A-5
B_DELWRI, A-5
b_dev, A-4, A-11
B_DONE, A-5
b_errcnt, A-11
B_ERROR, 6-28, A-4, A-5
b_error, 6-13, 6-29, A-4
b_flag, A-11
b_flags, 6-12, A-4
b_forw, A-5, A-6
b_iodone, A-5
b_major, A-9
B_PHYS, A-5, B-13
b_queue, A-5
B_READ, 6-12, A-5, B-12
b_resid, 6-13, 6-29, A-5
b_s, A-5
b_sc, A-5
b_spaddr, A-5
b_state, A-11
b_un, A-5
b_un.b_addr, A-5
B_WANTED, 6-12, A-5, B-12
B_WRITE, 6-12, A-5
b_xaddr, A-11
b_xcount, A-11
backplane anchor
  defined, Gl-1
backplane header
  defined, Gl-2
backplane master processor
  defined, Gl-2
bcopy, B-3
bcopy_prot, B-18
bdevsw table, 6-26, 7-6, A-2
biodone, B-13
BLOCK, 4-6
block device driver, Gl-1, Gl-3
block drivers, 1-12
board computer
  defined, Gl-2

# Index


BPN_CONFIG, A-20
brelse, 6-14, B-3
buf, 6-11, 6-12, 6-28, A-3, B-12
buffer, A-14
BUFLET_ALLOC_FAILED, 4-7, B-25
build file, 7-6
  editing, 7-7
bus
  defined, Gl-2
bus arbiter
  defined, Gl-2
bus arbitration
  arbitration mode, 8-13
  round robin mode, 8-13
bus errors and panics, 3-10
bus request, 1-8, 8-10
bus request levels
  BR0, 8-8, 8-10
  BR1, 8-8, 8-10
  BR2, 8-8, 8-10
  BR3, 8-8, 8-10
bus_info, A-14
bus_type, A-14
BYTE_WIDE, 6-17, A-24
bzero, B-3

**C**
C_ALLCLOSES, 6-38, A-7
C_EVERYCLOSE, A-7
c_major, A-9
C_NODELAY, A-7
cache
  flushing, 4-21
  purging, 4-22
  shared line problems, 4-21
cache alignment, 4-22
cache coherency, 4-21
cache memory
  and DMA, 1-6
card cage, 1-2
card records, 8-4
card/proc name, 8-7
card_ptr, A-14
CARD_RESET, 6-20
CARD_STATUS, 6-20
card_type, A-14, A-22, A-23, A-24
cards, 8-4
  hard-coded, 8-5
  programmable, 8-5
  third-party, 8-4
cdevsw, 1-16, A-6, A-7
cdevsw table, 6-9, 6-18, 7-6, A-6
chain_count, A-8
chain_index, A-8
character device
  writing read/write routines, 6-9
character drivers, 1-12
class, A-9
close, 1-13, 6-38, A-7
CMD, 6-19
cmd, 6-21
code
  protecting critical sections, 3-5, B-16
coll, B-15
commands
  BPN_CONFIG, A-20
  cd, 7-9
  CMD, 6-19
  config, 7-6
  ioctl, 2-9
  ioscan, 7-7
  lsdev, 7-5
  mknod, 1-15, 7-5
  sync, 1-17
  user-level ioctl, 2-2
  vme_config, 8-15
  write, 6-6
config, 7-6
config file
  editing, 8-2
config.mk file, 7-9
configuration, Gl-2
configuration records
  fields, 8-3
  format, 8-3
configuring
  VME and EEPROM, 1-5
copyin, B-4, B-18
copyout, B-4, B-18
count, A-14
cp, B-18
cpu_number, A-20

**D**
d_flags, A-2, A-6
d_open, A-7
d_write, A-7
D08_IRQ_TYPE, 3-9, B-39
D16_IRQ_TYPE, 3-9, B-39
D32_IRQ_TYPE, 3-9, B-39
ddb, 1-17
debugging tools
  xdb, 5-2
dev, 6-7, 6-9, 6-12, 6-21, 6-38, 6-39, A-14
DEV_BSIZE, B-13
device
  relative priority of, 3-17
device and driver integration, 1-15
device driver
  definition, 1-11
  writing a, 1-11
device files, 7-5
device_type, 7-6
direct mapper, 2-6
direct mapping, 2-6
Direct Memory Access (DMA), 4-2
direct_map_to, 8-7
direction, B-37
disk_close, 6-39
DMA
  parameter records, 8-11
  strategy routine, 6-31
  transfers, 6-34
DMA capabilities
  series 743i/748i, 4-3
dma_options, A-8
dma_params record, 8-8, 8-13
dma_parms, 6-33, 8-12, A-7, A-14, B-22
dma_sync, B-4
documentation conventions, xx
driver
  compiling, 7-8
driver entry points
  driver_attach, 6-2
  driver_install, 6-2
  driver_ioctl, 6-2
  driver_open, 6-2
  driver_read, 6-2
  driver_select, 6-2
  driver_write, 6-2
driver functions
  user written, 1-13
driver read parameters
  uio, 6-10

# Index


driver structures, 6-4
driver_attach, 6-2, 6-16, C-2
driver_close, 6-38, C-4
driver_close routines parameters
  dev, 6-39
driver_dma_setup, 6-34
driver_dma_start, 6-34
driver_entry points
  driver_strategy, 6-2
driver_input_ready, 6-25
driver_install, 6-2, C-2
driver_ioctl, 6-2, 6-21
driver_ioctl parameters
  arg_ptr, 6-21
  cmd, 6-21
  dev, 6-21
  flag, 6-21
driver_isr, 6-35
driver_open, 6-2, 6-6, 6-7, 6-8, 6-16, C-4
driver_open parameters
  dev, 6-7
  flag, 6-7
driver_output_ready, 6-25
driver_read, 6-2, 6-11, 6-12
  parameters
    dev, 6-9
driver_select, 6-2, 6-22, 6-25
driver_strategy, 6-2, 6-10, 6-12, 6-26, 6-27, 6-28, 6-31, 6-33, 6-35, 6-37
  calling, 6-27
driver_strategy, example, 6-31
driver_transfer, 6-37
  functions of, 6-37
driver_write, 6-2, 6-9, 6-10, 6-11, 6-12
drivers
  block, 1-12
  character, 1-12
DRV_BLOCK, A-9
DRV_CHAR, A-9
drv_data, 6-4
drv_flags, A-26
drv_info, 6-4, A-9, A-26
drv_minor_build, A-26
drv_minor_decode, A-26
drv_ops, 6-4, A-9, A-26
drv_path, A-25
DRV_PSEUDO, A-9
drv_routine, A-8
DRV_SAVE_CONF, A-9
DRV_SCAN, A-9
drv_type, A-25
dup, 6-18

**E**
EEPROM, 1-5, 1-15, 7-2, 7-10, 8-2
ENABLE_IRQ, B-6
enqueue, A-11
entry point routine names
  modifying, C-4
entry points, 1-12
errno, 6-8, 6-28, 6-29
error, A-22, A-23
error values
  B_ERROR, 6-28, 6-29
  BUFLET_ALLOC_FAILED, 4-7
  ILLEGAL_CALL_VALUE, 4-7
  ILLEGAL_CONTEXT, 4-6
  ILLEGAL_DMA_ADM, 4-7
  INT_ALLOC_FAILED, 4-6
  INVALID_OPTIONS_FLAG, B-25
  MEMORY_ALLOC_FAILED, B-25
  NO_D08_BLOCK, 4-7
  RESOURCE_UNAVAILABLE, A-8, B-25
  UNSUPPORTED_FLAG, B-25
Ethernet address
  defined, Gl-3
example files
  example.CFG, 8-2
  example2.CFG, 8-2
  example3.CFG, 8-2
example4.CFG, 8-2
exceptfds, 6-22
exclusive open, 6-6
External Interrupt Register(EIR), 3-3

**F**
FAIR, 8-13
fcntl, 6-7
features, A-20
FEXC, 6-7
fields
  a24_dirwin_addr, A-18
  a24_dirwin_host_base, A-18
  a24_dirwin_size, A-18
  a24_iomap_addr, A-19
a24_iomap_addr_mods, A-19
a32_dirwin_addr, A-18, A-20
a32_dirwin_addr_mods, A-19
a32_dirwin_host_base, A-18
a32_dirwin_size, A-18
a32_iomap_addr_mods, A-19
access_result, 5-3
access-result, A-22, A-23
active, A-14
address, A-25
address alignment, 8-6
av_back, A-4
av_forw, A-4
avail_interrupts, 3-6, A-20
available_interrupts, A-22
b_actf, A-11, A-14
b_action, A-4
b_active, A-11
b_actl, A-11, A-14
b_ba, A-4
b_back, A-5
b_bcount, A-4
b_blkno, A-4
b_bufsize, A-4
b_dev, A-11
b_errcnt, A-11
b_error, 6-13, A-4
b_flag, A-11
b_flags, 6-12, 6-28, A-4
b_forw, A-5
b_iodone, A-5
b_major, A-9
b_queue, A-5
b_resid, 6-13, 6-29, A-5
b_s, A-5
b_sc, A-5
b_state, A-11
b_un.b_addr, 6-28
b_xaddr, A-11
b_xcount, A-11
buf_info, A-14
buffer, A-14
bus_type, A-14
c_major, A-9
card_ptr, A-14
card_type, A-14, A-22, A-23, A-24
chain_count, A-8
chain_index, A-8

# Index


class, A-9
count, A-14
CPU/card number, 8-3
cpu_number, A-20
d_flags, A-2, A-6
dev, 6-39, A-14
dma_options, A-8
dma_parms, A-14
driver_data, 6-4
driver_ops, 6-4
drv_flags, A-26
drv_minor_build, A-26
drv_minor_decode, A-26
drv_path, A-25
drv_routine, A-8
drv_type, A-25
error, A-22, A-23
features, A-20
flag, 6-22, 6-39
flags, A-9
gfsw, A-14
if_drv_data, A-14
if_id, A-15
if_info, C-3
if_reg_ptr, A-15
ifsw, A-15
in_fsm, A-11
int_lvl, A-15
intloc, A-12, A-15
iomap_size, A-19
iosw, A-15
keep_bus, 8-8, 8-12
keyword, 8-3
location, 6-20
modifiers, A-25
my_address, A-15
my_isc, A-15
name, 8-3, 8-7, A-9
options, A-21
owner, A-15
parm1, B-8
proc, 8-12
reg_value, 6-20
req_lv, 8-8, 8-13
resid, A-15
timeflag, A-12
uio, 6-10
uio_iov, A-17
uio_iovcnt, A-17
uio_offset, A-17
uio_resid, A-17
uio_segflg, A-17
vme_addr, A-22, A-23
vme_expander, A-20
width, A-22, A-23, A-24
FIFO, 1-9, 3-19, Gl-3
fifo, 8-7
FIFO functions, 5-12
FIFO_GRAB, B-6
FIFO_POLL, B-6
FIFO_READ, B-6
FIFO_RELEASE, B-6
fimeflag, A-12
flag, 6-7, 6-21, 6-39
flags, A-9
B_ASYNC, A-4
B_BUSY, 6-12, A-4, B-13
B_CALL, A-5
B_DELWRI, A-5
b_dev, A-4
B_DONE, A-5
B_ERROR, A-5
b_error, 6-29
B_PHYS, A-5, B-13
B_READ, 6-12, A-5, B-12
B_WANTED, 6-12, A-5, B-12
B_WRITE, 6-12, A-5
BLOCK, 4-6
BYTE_WIDE, A-24
C_ALLCLOSES, 6-38, A-7
C_EVERYCLOSE, A-7
C_NODELAY, A-7
DEV_BSIZE, B-13
DRV_BLOCK, A-9
DRV_CHAR, A-9
DRV_PSEUDO, A-9
DRV_SAVE_CONF, A-9
DRV_SCAN, A-9
FEXC, 6-7
FNDELAY, 6-7
FREAD, 6-7
FWRITE, 6-7
IOM_NOWAIT, B-7
IOM_WAITOK, B-7
LONG_WIDE, A-24
SHORT_WIDE, A-24
T_DEVICE, A-26
T_INTERFACE, A-25
timeo, A-12
VME_CPU_COPY, 4-6
VME_FASTEST, 4-6
VME_IGNORE_ADM, 4-5, 4-6
VME_OPTIMAL, 4-5, 4-7, A-21
FNDELAY, 6-7
FREAD, 6-7, 6-22
from_va, 4-5, A-21, B-21, B-37, B-38
functions, 7-8
FWRITE, 6-7, 6-22

**G**
geteblk, 6-14, B-5
gfsw, A-14

**H**
hardware
mappers, 5-8
series 700i VME DMA, 4-3
host_addr, B-35
HOST_TO_VME, B-37
HPUX command, 9-11
HPUX file, 9-11
HP-UX kernel routines, B-2
hw_type, B-31
, 6-9

**I**
I/O control, 1-2
I/O space, 1-5
I/O system calls
close, 1-13
ioctl(2), 1-13
open, 1-13
read, 1-13
select, 1-13
write, 1-13
id, B-39
if_drv_data, A-14
if_id, A-15
if_info, C-3
if_reg_ptr, A-15
ifconfig(1M), 9-11
ifsw, 6-37, A-15
ILLEGAL_CALL_VALUE, 4-7
ILLEGAL_CONTEXT, 4-6

# Index


ILLEGAL_DMA_ADM, 4-7
ILLEGAL_OPTION, 4-6
in_fsm, A-11
Initial System Loader (ISL)
  defined, Gl-3
Initial System Loader (ISL),, see also ISL
INT_ALLOC_FAILED, 4-6
int_lvl, A-15
integrating device and driver, 1-15
Interface Acknowledge (IACK) register, 3-3
interface card, 3-4
interface driver, 3-4
Internet
  defined, Gl-3
Internet address
  defined, Gl-3
Internet address,, see also IP address
Internet Domain server, 9-10
interrupt
  handling an, 3-2
  level, 3-7
  range, 8-9
  records, 8-9
  request line, 8-9
  steps in processing, 3-2
interrupt handler, 1-8, Gl-4
interrupt handling, 1-8, 3-2
  interrupt-driven I/O, 3-3
interrupt handling, VME, 1-6
interrupt level
  defined, Gl-4
  fast mode, Gl-3
  normal mode, Gl-3
  software, 3-5
interrupt levels, 8-2
  range, 3-5
interrupt service routine (isr)
  writing an, 3-9
interrupt service routine(isr), 3-3
interrupt vector
  setting up the VME card, 3-7
interrupter, Gl-4
interrupts
  finding available, 3-6
  PA-RISC, 3-13
intloc, A-12, A-15, B-17
INVALID_OPTIONS_FLAG, B-25
io_dma_cleanup, C-3
io_dma_setup, C-3
io_flushcache, 4-21
io_free, B-7
io_isrlink, C-3
io_isrunlink, C-3
io_parms, 2-12, B-35
io_purgecache, 4-21, 4-22
IO_TESTR, B-6
io_testr, C-3
IO_TESTW, B-6
io_testw, C-3
IO_WAITOK, B-7
iobuf, A-10
ioctl, 1-2, 6-16, B-6
  calls, 1-11
  commands, 1-11
  parameters
    arg, 6-18
    fildes, 6-18
    request, 6-18
ioctl calls, 5-2
  VME_TESTR, A-24
  VME_TESTW, A-24
  VME2_ENABLE_INT, A-22
  VME2_ENABLE_IRQ, A-22
  VME2_REG_READ, A-22
  VME2_REG_WRITE, A-22
ioctl calls()
  IRQ, 5-11
ioctl() calls
  map, 5-6, 5-8, 5-11
  probe, 5-3
  register access, 5-6
  unmap, 5-8
  user copy, 5-10
  VME2_FIFO_GRAB, 5-12
  VME2_FIFO_POLL, 5-13
  VME2_FIFO_READ, 5-13
  VME2_FIFO_RELEASE, 5-13
  VME2_LOCMON_GRAB, 5-12
  VME2_LOCMON_POLL, 5-12
  VME2_LOCMON_RELEASE, 5-12
  VME2_MAP_ADDR, 5-8
  VME2_UNMAP_ADDR, 5-3, 5-8
  VME2_USER_COPY, 5-10
ioctl() commands
  ENABLE_IRQ, B-6
  FIFO_GRAB, B-6
  FIFO_POLL, B-6
  FIFO_READ, B-6
  FIFO_RELEASE, B-6
  IO_TESTR, B-6
  IO_TESTW, B-6
  LOCMON_GRAB, B-6
  LOCMON_POLL, B-6
  MAP_ADDR, B-6
  REG_READ, B-6
  REG_WRITE, B-6
  UNMAP_ADDR, B-6
  USER_COPY, B-6
  VME2_ENABLE_IRQ, 5-3
  VME2_IO_TESTR, 5-3
  VME2_IO_TESTW, 5-3
  VME2_MAP_ADDR, 1-11, 2-9, 5-3
  VME2_REG_READ, 1-11, 5-3
  VME2_REG_WRITE, 1-11, 5-3
  VME2_UNMAP_ADDR, 1-11, 2-9
  VME2_USER_COPY, 1-11, 5-3
ioctl(2), 1-13
ioctl(2) commands
  CARD_CONTROL, 6-20
  CARD_RESET, 6-20
ioctls
  VME2_MAP_ADDR, A-21
  VME2_USER_COPY, 4-2, 4-4
iodone, 3-10, 6-27, 6-28, 6-29, 6-35, A-5, B-6, B-12
IOM_NOWAIT, B-7
iomap_size, A-19
ioscan, 7-7
iosw, A-15
iov_len, 6-12, A-17
iovec, 6-13, A-13
iowait, 6-27, 6-28, 6-29, 6-35, B-7
IP,, see also Internet Protocol
IRQ ioctl() call
  VME2_ENABLE_IRQ, 5-11
is_pavme2, A-20
isc, 2-9, 2-12, A-13, B-10, B-19, B-21, B-30, B-31, B-35, B-37, B-39, B-40, C-2
isc_claim, 6-4, A-26, C-2
isc_table_type, A-13
ISL,, see also Initial System Loader

# Index


ispavme1, A-20
isr, 3-3, 3-4, 3-6, 3-7, 3-8, 3-9, B-8
isrlink, B-8
isrlink arguments
  isr, B-8
  level, B-8
  mask, B-8
  parm1, B-8
  parm2, B-8
  regaddr, B-8
issig, B-9

**K**
kernel
  building, 7-9
kernel code, 1-15
kernel configuration file
  choosing a, 7-4
kernel vme2 services, 5-2
kvtophys, B-9

**L**
level, B-8, B-17
libc, 5-8
LIBUSRDRV macro, 7-10
linklevel address, Gl-4
loc_mon, 8-7
local bus
  cache, 1-3
  memory, 1-3
  system, 1-3
local bus memory, 1-3
location, 6-20
location monitor, 1-9, 3-19, 5-11, Gl-4
LOCMON_GRAB, B-6
LOCMON_POLL, B-6
LOCMON_RELEASE, B-6
LONG_WIDE, A-24
lsdev, 7-5

**M**
macros
  major, B-9
  minor, B-9
  START_POLL, A-12
major, B-9
major number, 5-2, 7-6
major_number, 7-6
map, 5-8
MAP_ADDR, B-6
map_mem_to_bus, 2-9, 2-11, 2-13, B-10, B-36
map_mem_to_bus arguments
  isc, B-10
map_mem_to_bus routine
  using, 2-13
map_mem_to_host, 2-9, 2-10, 2-11, 6-16, B-10, B-20, B-21, B-30, B-37, B-40
  using, 2-10
map_mem_to_host arguments
  phys_addr, B-10
  size, B-10
mapper space
  allocation by kernel, 2-3
mapping
  local address space, 1-4
mask, B-8
master, Gl-4
master file, 7-6
  creating, 7-6
  driver dependency section, 7-7
  install section, 7-7
  library section, 7-7
  naming, 7-6
master mapper
  setting up, 2-10
master mappers, 2-3
master mapping, 1-5, 2-3
  unmapping, 2-11
me_map_mem_to_bus2, 2-12
mechanisms
  timeout, 3-17
memory
  cache, 1-6
  virtual, 1-6
memory management
  basic concepts, 2-2
memory mapping
  non-local, 2-3
  routines, 2-9
  to a specific address, 2-13
  unmapping slave memory, 2-15
memory records, 8-5
  fields, 8-5, 8-12
MEMORY_ALLOC_FAILED, B-25
mincnt, 6-12, B-13
minor, B-9
minor number, 5-2, 7-6
minor_number, 7-6
minphys, 6-12, B-11, B-13
mknod, 7-5
modifiers, A-25, B-27, B-33
multiple open, 6-6
my_address, A-15
my_isc, A-15

**N**
name, A-9
network controller, Gl-5
Network File System services (NFS),, see also NFS
new
  access functions, C-3
  attach function and process, C-2
  create ISC function, C-3
  include file pathnames, C-4
  vme_ calls, C-3
  vme_hardware_info, C-3
NFS
  defined, Gl-4
NO_D08_BLOCK, 4-7
node, Gl-5
nodev, 6-18, A-2, A-7
non-DMA transfers, 8-8, 8-13
nonvolatile memory, Gl-5
nulldev, 6-18, A-2, A-7

**O**
object file, 7-8
oflag, 6-7
open, 1-13, 1-15, 6-6, 6-7, 6-18, 6-39, A-7
OPTIMAL, 6-16
  restrictions using, 6-17
options, A-21, B-21, B-23
owner, A-15

**P**
panic, B-11
parameters
  tunable, 7-8
PA-RISC, 1-3, 2-2, 3-3
parm1, B-8
parm2, B-8
pb_reset, 8-8

phys_addr, B-10
physio, 3-4, 6-10, 6-12, 6-13, 6-35, A-4, A-5, A-10, A-16, A-17, B-12
  using, 6-11
polling
  defined, Gl-5
porting
  HP-UX 9.x driver, C-2
  third-party drivers, C-4
printf, B-14, B-15, B-16, B-17, B-18, B-19, B-20, B-21, B-22, B-25, B-26, B-27, B-28, B-29, B-30, B-31, B-32, B-33, B-34, B-35, B-36, B-37, B-38, B-39, B-40, B-41, B-42
probe
  end address, 5-4
  start address, 5-4
probe ioctl() calls
  VME2_IO_TESTR, 5-3
  VME2_IO_TESTW, 5-3
problems, questions, and suggestions, xx
proc, 6-23, B-17
proc record, 8-8, 8-13
process
  forked child, 6-38
processor, Gl-5
processor card options, 8-8
processor records, 8-7
  CPU number, 8-7
  proc keyword, 8-7
  processor name, 8-7
  revision code ID, 8-7
pseudo ReadModifyWrite, Gl-5
PSEUDO_RMW, Gl-5

**R**

read, 1-12, 1-13, 1-15, 3-4, 6-9, 6-16, 6-26, 6-27, 6-29, A-16
readfds, 6-22
record type
  card, 8-3
  dma_params, 8-3
  interrupt, 8-3
  memory, 8-3
  proc, 8-3
  slot_1_functions, 8-3
records
  card, 8-4
  DMA parameter, 8-11
  memory, 8-5
  slot 1 function, 8-9
REG_READ, B-6
REG_WRITE, B-6
regaddr, B-8
register access ioctl() calls
  VME2_REG_WRITE, 5-6
release documents, xvii
release_buf, B-2, B-14
remote DMA controllers, 4-16, 6-33
remote host
  defined, Gl-5
remote_dma_setup, 6-33
remote_isr, 6-33
req_lv, 8-10
req_mode, 8-8, 8-13
req_mode values
  FAIR, 8-13
  ROR, 8-13
  RWD, 8-10, 8-13
request level, 1-8
  defined, Gl-5
request levels and arbitration modes
  combining, 8-11
request mode
  defined, Gl-5
  FAIR, 8-8, 8-11
  fair (FAIR), 8-11
  Release on Request, 8-11
  release on request (ROR), 8-11
  release when done (RWD), 8-10
  ROR, 8-8
  RWD, 8-8
request mode values
  FAIR, 8-11
  ROR, 8-11
request modes, 1-8
requester, Gl-5
resid, A-15
residual value, A-21
RESOURCE_UNAVAILABLE, A-8, B-25
revision history, xix
ROR, 8-13
round robin select, 8-10
routine, B-27, B-33
routines
  acquire_buf, B-2
  bcopy, B-3
  bcopy_protect, B-18
  biodone, B-13
  brelse, 6-14, B-3
  bzero, B-3
  close, 6-38
  copyin, B-4, B-18
  copyout, B-18
  disk_close, 6-39
  driver_attach, 6-2, 6-16, C-2
  driver_close, 6-38
  driver_dma_setup, 6-2, 6-34
  driver_dma_start, 6-2, 6-34
  driver_input_ready, 6-25
  driver_install, C-2
  driver_ioctl, 6-2, 6-20, 6-21
  driver_isr, 6-2, 6-35
  driver_link, 6-2
  driver_open, 6-2, 6-6, 6-7, 6-8, 6-16
  driver_output_ready, 6-25
  driver_read, 6-2, 6-9, 6-11, 6-12
  driver_select, 6-2, 6-22, 6-25, 6-26
  driver_strategy, 6-2, 6-10, 6-12, 6-26, 6-27, 6-28, 6-31, 6-33, 6-35, 6-37
    calling, 6-27
    read, 6-29
  driver_strategy, example, 6-31
  driver_transfer, 6-37
  driver_write, 6-2, 6-9, 6-10, 6-11, 6-12
  drv_routine, A-8
  dup, 6-18
  enqueue, A-11
  get_vme_hardware, C-3
  geteblk, 6-14, B-5
  geterror, B-6
  io_dma_cleanup, C-3
  io_dma_setup, C-3
  io_flushcache, 4-21
  io_free, B-7
  io_isrlink, C-3
  io_isrunlink, C-3
  io_malloc, B-7
  io_purgecache, 4-21, 4-22
  io_testr, C-3
  io_testw, C-3
  ioctl, 6-16, B-6

# Index


writing, 6-18
ioctl(2), 6-19
character device, 6-18
iodone, 3-10, 6-13, 6-27, 6-28, 6-29, 6-35, B-6, B-12
iowait, 6-27, 6-28, 6-29, 6-35, B-7
isc_claim, 6-4, A-26, C-2
isr, 3-3, 3-4, 3-6, 3-8, 3-9
isr_link, 3-3
isrlink, B-8
issig, B-9
kvtophys, B-9
libc, 5-8
map_mem_to_bus, 2-9, 2-11, 2-13, B-10, B-36
map_mem_to_host, 2-9, 2-11, 6-16, B-10, B-20, B-21, B-30, B-37, B-40
mincnt, 6-12, B-13
minphys, 6-12, B-11, B-13
msg_printf, B-11
nodev, A-7
nulldev, A-7
open, 1-15, 6-6, 6-18
panic, B-11
physio, 3-4, 6-9, 6-10, 6-12, 6-13, 6-35, A-4, A-5, A-10, A-16, A-17, B-12
using, 6-11
printf, B-14
read, 1-12, 1-15, 3-4, 6-9, 6-13, 6-16, 6-26, 6-27, 6-29
readv, 6-9
release_buf, B-14
release_bus, B-2
remote_dma_setup, 6-33
select, 6-22, 6-25, 6-26
writing a, 6-22
selwakeup, 6-23, 6-25, B-14
skel_isr, 3-9
sleep, 4-3, 6-28, B-15
snooze, B-16
spl, 3-5, 3-6
spl*, B-16
strategy, 6-16
suser, B-16
sw_trigger, 3-13, 3-15, B-17
system driver entry points, 6-2
tape_close, 6-39
timeout, 3-17, B-17
transfer
writing a, 6-37
uiomove, 6-9, 6-10, 6-14, A-16, A-17, B-18
unmap_mem_from_bus, 2-9, 2-15, B-19
unmap_mem_from_host, 2-9, 2-11, B-19
untimeout, 3-17, B-20
user written
driver_attach, 1-13
driver_close, 1-14
driver_install, 1-13
driver_ioctl, 1-14
driver_open, 1-13
driver_read, 1-14
driver_select, 1-14
driver_size, 1-14
driver_strategy, 1-14
driver_write, 1-14
vme_change_adm, B-20
vme_clr, B-21
vme_config, 8-2, A-20
vme_copy, 4-2, 4-4, 4-5, 4-6, 5-10, B-21
vme_create_isc, B-21, B-22, C-2, C-3
vme_dma_cleanup, 4-2, 4-22, 4-23, B-5, B-22, C-3
vme_dma_setup, 4-2, 4-22, 6-33, 6-34, A-8, B-5, B-25, C-3
vme_dma_start, 6-31
vme_dmacopy, 4-2, 4-4, B-22
vme_fifo_copy, 3-20, B-26
vme_fifo_grab, 3-20, B-27
vme_fifo_poll, 3-20, B-28
vme_fifo_read, 3-20, B-28
vme_fifo_release, 3-20, B-29
vme_generate_interrupt, B-29
vme_get_address_space, B-30, C-3
vme_get_status_id_type, 3-9, B-31, C-3
vme_hardware_info, 3-6, A-19, B-31, C-2
vme_hardware_map_info, 2-12, B-32
vme_init_isc, B-37, C-2, C-3
vme_isrlink, 3-3, 3-7, 3-8, B-32, C-3
vme_isrunlink, B-33, C-3
vme_locmon_grab, 3-19, B-33
vme_locmon_poll, 3-19, B-34
vme_locmon_release, 3-19, B-34
vme_map_largest_to_bus, 2-14, B-34
vme_map_mem_to_bus2, 2-9, 2-11, 2-13, B-35
vme_map_pages_to_bus, 2-13, B-35
vme_map_polybuf_to_bus, 2-14
vme_mod_copy, 4-5, 4-6, 6-16, 6-17, B-21, B-36
vme_panic_isr_hook, 5-11
vme_reg_read, B-37
vme_reg_write, B-37
vme_rmw, B-21, B-38
vme_set_address_space, 2-9, B-39, C-3
vme_set_attach_function, 6-2, B-22, B-39, C-2
vme_set_status_id_type, 3-9, B-40, C-3
vme_test_and_set, B-21, B-41
vme_testr, 5-4, 6-16, B-41, C-3
vme_testw, 5-4, B-42, C-3
wakeup, 3-10, 6-12, 6-29, B-42
write, 1-12, 6-9, 6-13, 6-16, 6-26, 6-29
wsio_install_driver, 6-2, A-26, C-2
rw, B-18
RWD, 8-13

**S**

safe page, 2-5
SAM, 7-3, 7-4
SCSI, Gl-6
select, 1-13, 6-22, 6-24
seltrue, 6-22
selwait, 6-24
selwakeup, 6-23, 6-25, B-14
selwakeup arguments
coll, B-15
semaphores, B-38
shared memory, Gl-5
shared open, 6-6
SHORT_WIDE, A-24
size, B-10, B-19, B-20, B-27, B-33, B-35
slave, 8-7, Gl-5
slave mapper, 2-4, 8-6
slave mapping, 1-5, 2-4
sleep, 4-3, 6-28, B-15
slot 1 function records, 8-9
arb_mode, 8-9
powerup_reset, 8-10

time-out, 8-9
slot one controller,, see also system controller
snooze, B-16
software trigger
selecting the level, 3-16
spl, 3-5, 3-6, 3-17
spl*, B-16
START_POLL, A-12
status ID, 3-7
STATUS_ID
defined, Gl-6
status_id, 1-8, 3-3, 3-6, 3-7, 3-8, A-17
strategy, 6-16
structures
_ioctl_arg, 6-20
b_un, A-5
bdevsw, A-2
buf, 6-11, 6-12, 6-14, 6-26, 6-28, A-3, B-12
b_error, 6-28
b_flags, 6-28
b_resid, 6-28
cdevsw, A-6
dma_parms, 6-33, 8-12, A-7
drv_info, 6-4, A-9, A-26
drv_ops, 6-4, A-9, A-26
if_info, C-3
io_parms, 2-12
iobuf, A-10
iovec, 6-9, 6-13, A-13
isc, 2-9, 2-12, A-13
isc_table_type, A-13
uio, 6-9, 6-10, 6-12, 6-13, A-16, B-12, B-18
vme_hardware_type, 3-6, A-19, B-31
vme2_copy_addr, 5-3, 5-10, A-21
vme2_int_control, 5-3, A-22
vme2_io_regx, 5-3, 5-6, 5-11, A-22
vme2_io_testx, 5-3, A-22, A-23
vme2_lm_fifo_setup, A-25
vme2_map_addr, 5-3, 5-8
wsio_drv_data, 6-5, A-25, A-26
wsio_drv_info, A-26
wsio_drv_info_t, 6-2, 6-4
sublevel, B-17
sub-space modifiers, 2-2
subspaces, 8-5
suser, B-16
sw_intloc, A-16
sw_trigger, 3-13, 3-15, B-17
sw_trigger arguments
arg, 3-14, B-17
intloc, 3-13, 3-14, B-17
level, 3-14, B-17
proc, 3-14, B-17
sublevel, 3-14, B-17
switch table, 1-16
swsm,, see also system wide shared memory
SYSRESET, 8-8, 8-10
system administration manager (SAM), 7-3
system call, Gl-6
system calls
close, 6-38, 6-39
I/O, 1-12
ioctl(2), 6-18
open, 6-6, 6-39, A-7
read, 6-9, 6-13, 6-29, A-16
select, 6-22
write, 6-9, 6-13, 6-28, A-16
system controller
defined, Gl-6
system file, 7-7
system wide shared memory (swsm)
defined, Gl-6
system.log, 8-2
system_script, 7-7
system-preparation script, 7-7

**T**

T_DEVICE, A-26
T_INTERFACE, A-25
tape_close, 6-39
TAS,, see also test-and-set
task, 6-19
TCP/IP
defined, Gl-7
termio, 6-8
testandset (TAS), Gl-6
timeo, A-12
timeout, 3-17, B-17
timeout mechanism, 3-17
to_va, 4-5, A-21, B-21, B-37, B-38
TRANSFER_SIZE, B-25
transfer_size bits
VME_D64, 4-6
true ReadModifyWrite
defined, Gl-7
defined,, see also TRUE_RMW

**U**

u.u_error, 6-26
uio, 6-10, 6-12, 6-13, A-16, B-12, B-18
uio_iov, A-17
uio_iovcnt, A-17
uio_offset, A-17
UIO_READ, B-18
uio_resid, 6-10, A-17
uio_segflg, A-17
UIO_WRITE, B-18
uiomove, 6-9, 6-10, 6-14, A-16, A-17, B-18
buffering using, 6-14
uiomove arguments
cp, B-18
rw, B-18
UNKNOWN_IRQ_TYPE, 3-9
UNKOWN_IRQ_TYPE, B-39
unmap, 5-8
UNMAP_ADDR, B-6
unmap_mem_from_bus, 2-9, 2-15, B-19
unmap_mem_from_bus arguments
isc, B-19
unmap_mem_from_host, 2-9, 2-11
using, 2-11
unmap_mem_from_host arguments
isc, B-19
size, B-19
virt_addr, B-19
UNSUPPORTED_FLAG, B-25
untimeout, 3-17, B-20
USER_COPY, B-6
user-level driver
disadvantages, 5-2
utility
vme_config, 7-2, 8-2

**V**

value_addr, B-39
variables, 7-8
Versa Module Eurocard (VME), 1-2

# Index


virt_addr, B-19, B-20
virtual memory
  PA-RISC, 1-5
virtual_addr, B-37
  definition, 6-16
VME
  backplane, 1-2
  bus, 1-2
  bus slaves, 2-4
VME address space
  A24, Gl-1
  A32, Gl-1
VME backplane
  defined, Gl-7
  installing cards, 7-4
VME bus, 1-6
  releasing after interrupt, 1-8
VME card
  installing, 7-5
VME chip, 2-2, 3-3
VME devices
  installation overview, 7-2
VME DMA engine, 1-6
VME DMA strategy routine
  writing a, 6-31
VME driver concepts
  introduction, 1-1
VME fileset
  installing, 7-2
VME interrupts
  VMEbus, Gl-7
VME master
  defined, Gl-7
VME services routines, B-2
VME slave
  defined, Gl-7
VME system administration, 1-5
VME user programming, 1-11
vme.CFG, 7-5, 7-10, 8-4, 8-13
VME_A16, B-39
VME_A24, B-39
VME_A32, B-39
vme_addr, A-22, A-23, B-23, B-35
vme_adm, B-23
vme_change_adm, B-20
vme_change_adm arguments
  adm, B-20
  size, B-20
  virt_addr, B-20
vme_clr, B-21
vme_clr arguments
  address, B-21
  isc, B-21
vme_config, 1-5, 1-7, 1-15, 2-7, 3-6, 7-2, 8-2, 8-6, 8-15, A-20
VME_COPT, B-39
vme_copy, 4-2, 4-4, 4-5, 4-6, 5-10, B-21
  using, B-21
vme_copy arguments
  from_va, B-21
  options, B-21
  to_va, B-21
VME_CPU_COPY, 4-6
vme_create_isc, B-21, B-22, C-2, C-3
vme_dma_cleanup, 4-2, 4-22, 4-23, B-5, B-22, C-3
vme_dma_setup, 4-2, 4-22, 6-33, 6-34, A-8, B-5, B-25, C-3
vme_dma_start, 6-31
vme_dmacopy, 4-2, 4-4, B-22
vme_dmacopy arguments
  dma_parms, B-22
  options, B-23
  vme_addr, B-23
  vme_adm, B-23
vme_expander, A-20
VME_FASTEST, 4-6
vme_fifo_copy, 3-20, B-26
vme_fifo_grab, 3-20, B-27
vme_fifo_grab arguments
  address, B-27
  arg, B-27
  modifiers, B-27
  routine, B-27
  size, B-27
  who, B-27
vme_fifo_poll, 3-20, B-28
vme_fifo_poll arguments
  who, B-28
vme_fifo_read, 3-20
vme_fifo_read arguments
  who, B-28
vme_fifo_release, 3-20
vme_fifo_release arguments
  who, B-29
vme_generate_interrupt, B-29
vme_generate_interrupt arguments
  width, B-29
vme_get_address_space, B-30, C-3
vme_get_status_id_type, 3-9, B-31, C-3
vme_get_status_id_type arguments
  isc, B-31
vme_hardware_info, 3-6, A-19, C-2, C-3
vme_hardware_map_info, 2-12, B-32
vme_hardware_type, 3-6, A-19, B-31
vme_hardware_type arguments
  hw_type, B-31
VME_IGNORE_ADM, 4-5, 4-6
vme_init_isc, B-37, C-2, C-3
vme_isrlink, 3-3, 3-7, 3-8, 3-9, B-32, C-3
vme_isrunlink, B-33, C-3
vme_locmon_grab, 3-19, B-33
vme_locmon_grab arguments
  address, B-33
  arg, B-33
  modifiers, B-33
  routine, B-33
  size, B-33
  who, B-33
vme_locmon_poll, 3-19, B-34
vme_locmon_poll arguments
  answer, B-34
  who, B-34
vme_locmon_release, 3-19, B-34
vme_locmon_release arguments
  who, B-34
vme_map_largest_to_bus, 2-14, B-34
vme_map_mem_to_bus2, 2-11, 2-13, B-35
vme_map_mem_to_bus2 arguments
  host_addr, B-35
  io_parms, B-35
  isc, B-35
  size, B-35
  vme_addr, B-35
vme_map_pages_to_bus, 2-11, 2-12, 2-13, B-36
vme_map_polybuf_to_bus, 2-11, 2-14
vme_mod_copy, 4-5, 4-6, 6-16, 6-17, B-21, B-36
  using, 6-16
vme_mod_copy arguments
  addr_mod, B-37
  direction, B-37

isc, B-37
virtual_addr, B-37
VME_OPTIMAL, 4-5, 4-7, A-21
vme_panic_isr_hook, 5-11
vme_polybuf, A-20
vme_reg_read, B-37
vme_reg_read arguments
addr_mod, B-37
from_va, B-37
isc, B-37
to_va, B-37
width, B-37
vme_reg_write, B-37
vme_reg_write arguments
from_va, B-38
to_va, B-38
vme_remap_mem_to_host, B-38
vme_rmw, B-21, B-38
vme_set_address_space, 2-9, B-39, C-3
vme_set_address_space arguments
value_addr, B-39
vme_set_attach_function, 6-2, B-22, B-39, C-2
vme_set_attach_function arguments
your_attach_function, B-39
your_attach_string, B-39
vme_set_mem_error_handler, 3-10, B-40
vme_set_status_id_type, 3-9, B-40, C-3
vme_set_status_id_type arguments
isc, B-40
vme_sysreset, 8-8
vme_test_and_set, B-21, B-41
vme_test_and_set arguments
address, B-41
VME_TESTR, A-24
vme_testr, 5-4, B-41, C-3
VME_TESTW, A-24
vme_testw, 5-4, B-42, C-3
VME_TO_HOST, B-37
VME_UD, B-39
vme2 driver
verifying, 7-3
vme2 macros
VME_HIGHEST_PRI_AVAIL_IRQ, 3-7
VME_LOWEST_PRI_AVAIL_IRQ, 3-7
vme2 services, B-2
vme2.h, B-2
vme2_copy_addr, 5-3, 5-10, A-21
VME2_ENABLE_IRQ, 5-3
VME2_FIFO_GRAB, 5-12
VME2_FIFO_POLL, 5-13
VME2_FIFO_READ, 5-13
VME2_FIFO_RELEASE, 5-13
vme2_int_control, 5-3, A-22
vme2_io_regx, 5-3, 5-6, 5-11, A-22, A-23
VME2_IO_TESTR, 5-3
VME2_IO_TESTW, 5-3
vme2_io_testx, 5-3, A-22, A-23
vme2_lm_fifo_setup, A-25
VME2_LOCMON_GRAB, 5-12
VME2_LOCMON_POLL, 5-12
VME2_LOCMON_RELEASE, 5-12
VME2_MAP_ADDR, 1-11, 2-9, 5-3, 5-8, A-21
vme2_map_addr, 5-3, 5-8
VME2_REG_READ, 1-11, 5-3, A-22
VME2_REG_WRITE, 1-11, 5-3, A-22
VME2_UNMAP_ADDR, 1-11, 2-9, 5-3, 5-8
VME2_USER_COPY, 1-11, 4-2, 4-4, 5-3
VMEbus
defined, Gl-7
VMEbus interrupt
defined, Gl-7
VME-SERV fileset, 7-2, 7-3, 9-3
vmunix file, 7-10

**W**

wakeup, 3-10, 6-12, B-15, B-42
who, B-27, B-28, B-29, B-33, B-34
width, A-22, A-23, A-24, B-29, B-37
write, 1-12, 1-13, 6-6, 6-10, 6-16, 6-26, 6-28, 6-29, A-7, A-16
writefds, 6-22
wsio_drv_data, 6-5, A-25, A-26
wsio_drv_info, 6-2, 6-4, A-26
wsio_install_driver, 6-2, A-26, C-2

**Y**

your_attach_function, B-39

# Index