sun
microsystems

Writing Device Drivers *for the* Sun Workstation®

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part Number: 800-1304-05
Revision: A of 19 September 1986

Sun equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of Sun equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

Contents

Chapter 1 Introduction **3**

1.1. Device Independence 3

1.2. Types of Devices 3

1.3. System V Compatibility 5

1.4. Major Development Stages 6

1.5. Warning To Microcomputer Programmers 6

1.6. Address-Space Terminology 6

1.7. Manual Overview 7

Chapter 2 Hardware Context **11**

2.1. Multibus Machines 11

Multibus Memory Address Space and I/O Address Space 11

Allocation of Multibus Memory 14

Allocation of Multibus I/O Space 15

2.2. VMEbus Machines 16

Sun-2 VMEbus Address Spaces 16

Sun-3 Address Spaces 18

Allocation of VMEbus Memory 19

The Sun VMEbus to Multibus Adapter 21

Interrupt Vector Assignments 21

2.3. Hardware Peculiarities to Watch Out For 22

Multibus Device Peculiarities 22

Multibus Byte-Ordering Issues 22

Other Multibus-related Peculiarities 24

Other Device Peculiarities 25

2.4. DMA Devices 27

Sun Main-Bus DVMA 27

Chapter 3 Overall Kernel Context **33**

3.1. The System Kernel 33

3.2. Devices as ''Special'' Files 34

3.3. Run-Time Data Structures 40

The Bus-Resource Interface 41

Autoconfiguration-Related Declarations 47

Other Kernel/Driver Interfaces 48

Chapter 4 Kernel Topics and Device Drivers **53**

4.1. Overall Layout of a Character Device Driver 53

4.2. User Space versus Kernel Space 55

4.3. User Context and Interrupt Context 55

4.4. Device Interrupts 56

4.5. Interrupt Levels 57

4.6. Vectored Interrupts and Polling Interrupts 58

4.7. Some Common Service Routines 60

Timeout Mechanisms 61

Sleep and Wakeup Mechanism 61

Raising and Lowering Processor Priorities 62

Main Bus Resource Management Routines 62

Data-Transfer Functions 62

Kernel printf Function 63

Macros to Manipulate Device Numbers 63

Chapter 5 Driver Development Topics **67**

5.1. Installing and Checking the Device 67

Setting the Memory Management Unit 67

Selecting a Virtual Address 68

Finding a Physical Address 70

Selecting a Virtual to Physical Mapping 70

Sun-2 Address Mapping 72

Sun-3 Address Mapping 75

A Few Example PTE Calculations 77

Getting the Device Working and in a Known State 78

A Warning about Monitor Usage 79

5.2. Installation Options for Memory-Mapped Devices 80

Memory-Mapped Device Drivers 80

Mapping Devices Without Device Drivers 81

Direct Opening of Memory Devices 85

5.3. Debugging Techniques 86

Debugging with `printf` 87

Event-Triggered Printing 89

Asynchronous Tracing 90

`kadb` — A Kernel Debugger 91

5.4. Device Driver Error Handling 92

Error-Handling Mechanisms 92

Error Recovery 92

Error Returns 93

Error Signals 93

Error Logging 93

Kernel Panics 93

5.5. System Upgrades 94

Chapter 6 The ''Skeleton'' Character Device Driver **97**

6.1. General Declarations in Driver 100

6.2. Autoconfiguration Procedures 101

Probe Routine 101

Attach Routine 103

6.3. Open and Close Routines 103

6.4. Read and Write Routines 105

Some Notes About the UIO Structure 106

6.5. Skeleton Strategy Routine 107

6.6. Skeleton Start Routine — Initiate Data Transfers 108

6.7. Interrupt Routines 110

6.8. Ioctl Routine 112

6.9. DMA Variations 112

Multibus or VMEbus DVMA 112

A DMA Skeleton Driver 112

Chapter 7 Configuring the Kernel **119**

7.1. Background Information 119

7.2. An Example 121

7.3. Devices that use Two Address Spaces 125

7.4. Booting Kernels on Diskless Workstations 126

Appendix A Using the Sun CPU PROM Monitor **129**

A.1. PROM Monitor Command Syntax 129

A.2. PROM Monitor Syntax for Memory and Register Access 129

A.3. PROM Monitor Command Descriptions 131

Appendix B Summary of Device Driver Routines **137**

B.1. Standard Error Numbers 137

B.2. Device Driver Routines 137

xx`attach` — Attach a Slave Device 137

xx`close` — Close a Device 138

xx`intr` — Handle Vectored Interrupts 138

xx`ioctl` — Miscellaneous I/O Control 139

xx`mmap` — Mmap a Page of Memory 140

xx`minphys` — Determine Maximum Block Size 141

xx`open` — Open a Device for Data Transfers 141

xx`poll` — Handle Polling Interrupts 142

xx`probe` — Determine if Hardware is There 142

xx`read` — Read Data from Device 143

xx`strategy` Routine 143

xx`write` — Write Data to Device 144

Appendix C Kernel Support Routines **147**

copyin — Move Data From User to Kernel Space 147

copyout — Move Data From Kernel to User Space 147

CDELAY — Conditional Busy Wait 147

DELAY — Busy Wait for a Given Period 148

iodone — Indicate I/O Complete 148

iowait — Wait for I/O to Complete 148

getkpgmap — get PTE for Virtual Address 148

gsignal — Send Signal to Process Group 148

kmem_alloc — Allocate Space from Kernel Heap 149

kmem_free — Return Space to Kernel Heap 149

MBI_ADDR — Get Address in DVMA Space 149

mbrelse — Free Main Bus Resources 149

mbsetup — Set Up to Use Main Bus Resources 149

panic — Reboot at Fatal Error 150

peek, peekc — Check and Read an Address 150

physio — Block I/O Service Routine 150

poke, pokec — Check and Write an Address 152

printf — Kernel Printf Function 152

pritospl — Convert Priority Level 153

psignal — Send Signal to Process 153

rmalloc — General-Purpose Resource Allocator 153

rmfree — Recycle Map Resource 154

sleep — Sleep on an Event 154

spl*n* — Set CPU Priority Level 155

splx — Reset Priority Level 155

swab — Swap Bytes 155

timeout — Wait for an Interval 155

uiomove — Move Data To or From an **uio** Structure 156

untimeout — Cancel **timeout** Request 156

uprintf — Nonsleeping Kernel Printf Function 156

ureadc and **uwritec** — uio Structure Read and Write 157

vac_disable_kpage — Stop Caching of a Kernel Page 157

wakeup — Wake Up a Process Sleeping on an Event 157

Appendix D User Support Routines **161**

free — Free Allocated Memory 161

getpagesize — Return Pagesize 161

mmap — Map Memory from One Space to Another 161

munmap — Unmap Pages of Memory 162

valloc — Allocate Virtual Memory 162

Appendix E Sample Driver Listings **165**

E.1. Skeleton Board Driver 166

E.2. Sun-2 Color Graphics Driver 174

E.3. Sky Floating-Point Driver 186

E.4. Versatec Interface Driver 194

Index 207

Tables

Table 1-1 VMEbus Address-space Names 7

Table 2-1 Sun-2 Multibus memory types 12

Table 2-2 Sun-2 Multibus Memory Map 15

Table 2-3 Sun-2 Multibus I/O Map 15

Table 2-4 Sun-2 VMEbus Memory Types 16

Table 2-5 Generic VMEbus (Full Set) 18

Table 2-6 Sun-3 VMEbus Address Types 18

Table 2-7 16-bit VMEbus Address Space Allocation 20

Table 2-8 24-bit VMEbus Address Space Allocation 20

Table 2-9 32-bit VMEbus Address Space Allocation (Sun-3 Only) 20

Table 2-10 VMEbus Address Assignments for Some Devices 21

Table 2-11 Vectored Interrupt Assignments 22

Table 3-1 A Sample Listing of the /dev Directory 35

Table 3-2 Current Major Device Number Assignments 39

Table 5-1 Sun-2 PTE Masks 73

Table 5-2 Sun-3 PTE Masks 76

Table 5-3 Virtual Memory Devices 82

1

Introduction

Introduction 3

1.1. Device Independence 3

1.2. Types of Devices 3

1.3. System V Compatibility 5

1.4. Major Development Stages 6

1.5. Warning To Microcomputer Programmers 6

1.6. Address-Space Terminology 6

1.7. Manual Overview 7

1

Introduction

This manual is a guide to adding software drivers for new devices to the Sun UNIX† kernel.

1.1. Device Independence

One of the UNIX Operating System's major services to application programs is to provide a device-independent view of the I/O hardware. In this view, user processes (application programs), see devices as "special" types of files that can be opened, closed and manipulated just like regular files. The user process manipulates devices as it would files, by making *system calls.*

Once a system call carries process execution into the UNIX kernel, however, it becomes clear just how "special" devices really are. The kernel distinguishes between real files and device special files, and translates operations on the later into calls to their corresponding device drivers. These drivers control *all* device operations; devices do nothing until their drivers tell them to.

Thus, system calls provide the interface between user processes and the UNIX kernel, while device drivers provide an interface between the kernel itself and its peripheral devices. Device drivers are thus crucial elements in the UNIX operating system's overall device-independent scheme of things. Device-drivers are the *only* parts of the system that know, or care, if a device is DMA (Direct Memory Access), PIO (Programmed I/O), or memory-mapped.

The kernel supplied with the Sun system is a *configurable* kernel, meaning that it is possible to add new device driver modules to your system by rebuilding your kernel, even if you don't have access to the system source code. For more information on how to reconfigure your kernel to include new device drivers, see the *Configuring the Kernel* section of this manual, the *Adding Hardware to Your System* section of the *System Administration Manual* and the config(8) man page.

1.2. Types of Devices

This document is aimed at Sun users who wish to connect new VMEbus or Multibus devices to their system. It does *not*, however, explain how to write drivers for all possible Sun devices.

We can classify devices into eight major categories:

† UNIX is a trademark of AT&T Bell Laboratories.

1. Co-processors.
2. Disks and tapes.
3. Network interface drivers such as Ethernet or X.25.
4. Serial communications multiplexors.
5. General DMA devices such as driver boards for raster-oriented printers or plotters. DMA devices contain their own processors and, once dispatched, perform I/O independently of the system CPU by stealing memory cycles.
6. Programmed I/O devices, that is, devices which send and receive data on the main system bus under direct control of the system CPU.
7. Frame buffers and other memory-mapped devices. Such devices are typically mapped into user-process memory and then accessed directly.
8. So called *pseudo devices*, which are actually drivers without associated hardware devices.

This manual *only* covers driver development for devices in categories 5, 6, 7 and 8. In these categories, however, will be found the great majority of the devices which users will want to add to their systems. These include:

□ input devices like mice, digital tablets and analog-to-digital converters,

□ output and display devices like frame buffers, printers, and plotters,

□ and utility peripherals like array and graphics processors.

This manual doesn't support the development of co-processor drivers for the simple reason that co-processors, while certainly devices, are so intimately linked to the CPU that they are integrated below the driver level of the kernel.

It also excludes tape and disk drivers, or indeed drivers for any *structured* or *block I/O* devices, for such drivers are quite difficult to write well. Most customers will find the structured-device drivers provided with the standard system software to fill their needs as well, if not better, than drivers that they could develop themselves. The extensive use of standards within the Sun product line will allow them to use hardware interfaces already provided by Sun to drive whatever tape and disk units they wish to use. If this turns out not to be the case, an experienced driver developer will have to be consulted. (You'll also want to start with an existing driver, and will thus need a source-code license).

Finally, this manual doesn't discuss the issues relevant to serial communications and local network interface drivers. Again, such drivers are rather involved, and users will almost certainly find the Sun product line to contain devices adequate to their task. (And again, you'll need a source license to go it alone).

This manual is concerned with *unstructured* or *character* (as opposed to *structured* or *block*) devices. This distinction is often made, but seldom clearly, and it may be helpful then to consider *structured* devices as only those upon which UNIX filesystems can be mounted. Such devices (almost always disks, but tape drives are possible) support random-access I/O by way of the system buffer-caching mechanism. They almost always support a second, character-oriented style of I/O, often called *raw I/O*, but this doesn't make them character devices. Their drivers tend to implement raw I/O with the same mechanisms constructed for the main task of supporting block I/O.

Character devices, on the other hand, do not support random-access I/O, and filesystems cannot be mounted upon them. Their drivers typically support *read* and/or *write* operations, but these operations are fundamentally different than in block devices. *Sometimes character drivers use mechanisms, routines and structures that are primarily intended for block drivers, but this shouldn't be allowed to confuse matters; they use them only because it's convenient to do so.*[1]

The techniques described in this manual can also be used to build *pseudo-device drivers*. Such drivers can be useful in a variety of ways. They can be used to implement virtual devices (for example, windows that behave as virtual terminals) or for extending the capabilities of the kernel in highly localized and portable fashions (for example, by building a pseudo device to implement a specific kind of semaphore facility). What they all have in common is the absence of hardware; the driver actually implements and controls virtual software devices.

1.3. System V Compatibility

Sun has embarked on a effort aimed at making its operating system compatible with AT&T's System V UNIX system. In general, this effort will have a negligible impact on the structure of Sun device drivers.

System V compatibility doesn't involve massive driver rewrites because, with the exception of drivers for pseudo devices, drivers are far more sensitive to the architectural details of the machines upon which they run than to the details of the kernels to which they interface. (The System V interface will be built by rewriting system calls and utilities, and by providing compatibility libraries).

Sun device drivers differ from typical System V drivers because the Sun operating system is evolved from 4.2BSD and, in 4.2BSD, the kernel driver interface was significantly restructured. This doesn't mean that programmers with experience developing System V drivers will find Sun drivers to be altogether foreign. In fact, the overall structure of Sun drivers is largely identical to the structure of System V drivers. Nevertheless, there are differences, and from some perspectives they are quite significant. See the *Overall Kernel Context* chapter of this manual for the details of the Sun driver/kernel interface.

The greatest differences between Sun drivers and drivers for other systems are due not to operating system differences but rather to differences between the Sun Memory-Mangement Unit (MMU) and the MMUs of other systems. Consequently, drivers which map addresses require a lot of Sun-specific code.

[1] To jump ahead for a moment, the kernel routines which, though written for block drivers are also used for character drivers are `physio`, `mbsetup` and `mbrelse`. The driver *xx*`strategy` routine is also intended primarily for block devices, though it can be used in character drivers which buffer their I/O (typically those which don't support a tty-style interface). In such cases it's not, as it is in block drivers, an entry point, and it doesn't implement any strategy to speak of. But `physio` requires its existence, as it does the use of the `buf` structure, and so they are used. The main point to keep in mind is that character drivers use block-driver mechanisms because it's convenient for them to do so, but this doesn't make them block drivers. In particular, character drivers never have anything to do with the kernel buffer cache.

1.4. Major Development Stages

To add a new device and its driver to the system you must:

1. Get the device hardware into a state where you know it works as advertised. It is *extremely* difficult to debug the driver software if the device hardware isn't first working properly.
2. Write the device driver itself.
3. Add the driver to a kernel's configuration file to specify a system containing the new driver, and compile this system.
4. Debug the driver.
5. Repeat steps 2 to 4 as necessary. Drivers are often written (and debugged) by stages, with development proceeding long after early versions are configured into the kernel.

1.5. Warning To Microcomputer Programmers

Sun computers are virtual-address machines, and, as such, their addressing schemes are far more complex than anything that microcomputer programmers typically confront. In virtual-address machines, physical addresses have a complex and rapidly changing relationship to the virtual addresses which user programs manipulate. The kernel continually maps, remaps and unmaps pages of virtual memory to accommodate the limits of system physical memory. This means that the kernel (including its device drivers) cannot assume that any physical address in user memory will not be snatched away by the paging daemon unless it explicitly locks the physical page containing that address into memory. The details of how this locking is done will be given later, in discussions of the kernel support routine `physio`; for the moment simply note that physical addresses have a complex and transient relationship to virtual addresses. Specifically:

□ Each user process (and, on Sun-2 machines, the kernel as well) has its own virtual address space. A user process (or the kernel) can make arrangement to share memory with another process — that is, to have part of its address space mapped to the same physical memory as a part of the address space of another process — but this must be done explicitly.

□ In similar regard, a user process can elect to have a bus address mapped into its address space, but this doesn't happen automatically.

1.6. Address-Space Terminology

In this manual, we'll adopt a VMEbus address-space naming convention that makes both address size and data size explicit. The first number in the name indicates the number of bits in the address and the second number indicates the number of bits in the data length. For example, the space with a 24-bit address and a 16-bit data length will be known as `vme24d16`. This naming convention is used elsewhere, but others are as well, as indicated in the following table.

Table 1-1 *VMEbus Address-space Names*

Address-Space Name	Other Name(s)
vme16d16	VME D16A16 and vme16
vme24d16	VME D16A24 and vme24
vme32d16	VME D16A32
vme16d32	VME D32A16
vme24d32	VME D32A24
vme32d32	VME D32A32 and vme32

The short names in the second column (vme16, vme24 and vme32) are commonly used, but they can seem ambiguous to the novice, and will consequently be avoided in this manual.

Note that there are two situations where the system expects the name of a VMEbus address space as input. In these situations, either the vme16d16 or the vme16 forms are acceptable. These situations are:

- within the kernel config file, and
- when naming actual memory devices ("special" files in the /dev directory). See the *Mapping Devices Without Device Drivers* section of this manual for more information.

1.7. Manual Overview

Chapter 2 is an overview of the hardware environment provided by Sun Workstations to their drivers. The emphasis is on bus and address-space related issues.

Chapter 3 is an overview of the kernel environment within which drivers operate.

Chapter 4 covers a number of topics relevant to drivers: address spaces, interrupts and so on, in greater detail. It also surveys the most important classes of services provided by the kernel to its drivers.

Chapter 5 covers development topics, including the initial installation and checkout of devices, driver debugging and error handling.

Chapter 6 explains how to configure a kernel that contains new drivers.

Chapter 7 provides a detailed discussion of driver for a very simple hypothetical character device.

Finally, a few appendices are provided. These include a reference on the Sun PROM monitor, a summary of kernel support functions useful in developing device drivers, and descriptions of user-level routines useful in driver development. It also contains a number of annotated driver listings.

Remember, spend as much time as you need in the Sun PROM monitor poking, prodding and cajoling your device until you're thoroughly familiar with its behavior. This will save you a lot of grief later. The details on how to proceed with a monitor checkout of your device are found in the *Installing and Checking the Device* section.

And finally, note that if you have no previous experience writing UNIX device drivers, you should expect to seek some advice from the Sun technical support organization or from an outside consultant experienced with driver development.

2

Hardware Context

Hardware Context **11**

2.1. Multibus Machines 11

Multibus Memory Address Space and I/O Address Space 11

Allocation of Multibus Memory 14

Allocation of Multibus I/O Space 15

2.2. VMEbus Machines 16

Sun-2 VMEbus Address Spaces 16

Sun-3 Address Spaces 18

Allocation of VMEbus Memory 19

The Sun VMEbus to Multibus Adapter 21

Interrupt Vector Assignments 21

2.3. Hardware Peculiarities to Watch Out For 22

Multibus Device Peculiarities 22

Multibus Byte-Ordering Issues 22

Other Multibus-related Peculiarities 24

Other Device Peculiarities 25

2.4. DMA Devices 27

Sun Main-Bus DVMA 27

2

Hardware Context

Computer I/O architectures are far more dependent upon bus structure than they are upon CPU type, and device drivers, oriented as they are towards I/O, must have intimate knowledge of the bus characteristics of the machines on which they are running. For example, many Multibus machines do not support vectored interrupts [2] and thus drivers for interrupt driven devices which are intended to run on Multibus machines must provide polling facilities. Fortunately, the Sun kernel provides facilities (described in *Other Kernel/Driver Interfaces)* by which a driver can determine the type of the machine upon which it's running.

2.1. Multibus Machines

Multibus Memory Address Space and I/O Address Space

The MC680X0 family of processors do all of their I/O via a process known as "memory mapping." What this means is that the processor sees no difference between memory and peripheral devices — all input-output operations are performed by storing data and fetching data from the same memory space. The Multibus, on the other hand, was originally designed for processors, like those of the Intel 8080 family, which have two separate address spaces. Such processors have one kind of instruction for storing data in memory or fetching data from memory (instructions such as MOV), and another, different kind of instruction (such as IN and OUT) for transferring data to or from peripheral devices. Reflecting the architecture of such processors, the Multibus has two address spaces.

Multibus memory space

is used for memory or devices that look like memory. Many devices — commonly known as "memory mapped" devices — are designed to be accessed as memory, and drivers for such devices can "map" them into user virtual memory space and then perform device I/O by simply reading and writing the device's memory as part of normal address space. Such memory-mapped drivers tend to be quite simple, and so it's notable that devices not explicitly designed to be memory mapped can, under a restricted set of circumstances, be driven by memory mapping. The restrictions are, however, fairly severe. Such drivers cannot, for example, have *xx*`ioctl`

[2] The Multibus itself, as it turns out, actually does support vectored interrupts, but not in a way that can reasonably be used with the MC680X0 family of processors)

routines. See *Mapping Devices Without Device Drivers* for more details. The Sun-2 Color Board is a good example of a device that *is* designed to be memory mapped, and a listing of its driver can be found in the *Sample Driver Listings* appendix.

Multibus I/O address space

is another "space" entirely separate from normal memory. Typically used as an area to which device registers can be mapped, I/O space was originally introduced to keep such registers out of limited primary address space by providing a means of making peripherals, rather than system memory, respond to the bus whenever given I/O control lines were asserted by the CPU. (Such a setup also reduces hardware costs by keeping the number of address lines small.) Devices which have their control and status registers mapped to Multibus I/O address space are said to be "I/O mapped" devices.

The MC680X0 family, of course, no longer suffers the addressing limitations that made the dual-space architecture of the Multibus so attractive. The VMEbus, in similar regard, is no longer structured around separate "memory" and "I/O" spaces. (The term "I/O space" does continue to be used, from time to time, with reference to VMEbus-based systems and devices. Such use, however, is largely by way of analogy with Multibus systems, and it shouldn't be taken too literally).

Be aware that while generic Multibus memory space can be either 20-bit or a 24-bit. (Sun normally uses 20-bit Multibus memory addresses, though when a Multibus card is installed in a VMEbus system with a VMEbus/Multibus adapter, 24-bit addresses are used). In similar regard, a generic Multibus can provide either a 8 or 16 bit I/O space, and Sun uses only the 16-bit Multibus I/O space. Note, however, that some older Multibus boards accept only 8-bit Multibus I/O addresses.

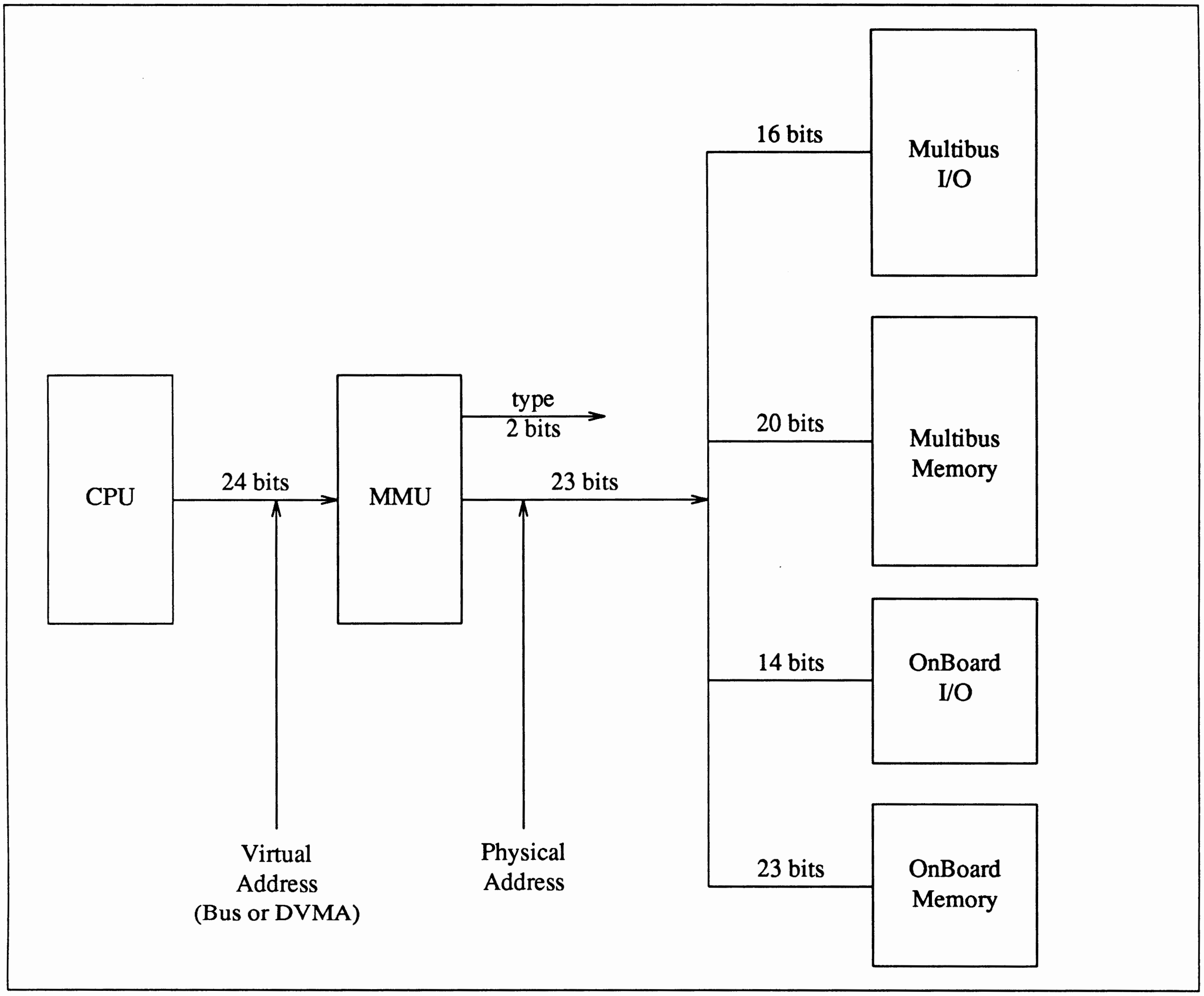
Sun Multibus systems actually have four "address spaces," corresponding to the four types of memory (each type has an identifying number associated with it, a number which is used by the MMU in computing PTE's (Page Table Entries). See *Sun-2 Address Mapping* for details. Though you will seldom deal with the on-board address spaces, you're best off understanding what they are. The following table thus contains not only the two Multibus spaces, but the "on board" memory and I/O spaces as well. It's within these spaces, resident on the CPU board itself, that UNIX is run.

Table 2-1 *Sun-2 Multibus memory types*

Type	*Description*	*Address Size*	*Address Range*
0	On-Board Memory	23 bits	`0x0 – 0x7FFFFF`
1	On-Board I/O Space	14 bits	`0x0 – 0x3FFF`
2	Multibus Memory	20 bits	`0x0 – 0xFFFFF`
3	Multibus I/O Space	16 bits	`0x0 – 0xFFFF`

The following schematic view of the Sun-2 Multibus may help the driver developer to visualize the larger hardware context within which drivers operate (when running on a Sun-2 Multibus machine.)

Figure 2-1 *Sun-2 Multibus Address Spaces*



Note some significant aspects of addressing layout as indicated in this table.

- The Memory Management Unit is at the *center* of the picture, a position that reflects its importance in the addressing scheme of all Sun machines, VMEbus based as well as Multibus based. (The centrality of the MMU will become quite clear when you later set out to allocate a physical address to your device, and then examine/set it with the PROM monitor.)
- Secondly, the input address of the MMU is a 24-bit *virtual address*. It may originate with the CPU, or come from a DMA bus master; it makes no difference.

- The output is a 23-bit *physical address* and a 2-bit *address type*. The address type specifies one of the four address spaces indicated at the right of the diagram.
- The four address spaces are to the right. The space corresponding to the incoming virtual address is a function of both the address and the memory type. Note that only the top two memory spaces (Multibus I/O and Multibus Memory) are accessible by way of the Multibus; the two On-Board memory spaces are accessed directly and are seldom of concern to non-Sun driver developers.

Programs can only reference driver address spaces in terms of virtual addresses which are then translated by the MMU into physical addresses within the appropriate physical address space.

Allocation of Multibus Memory

Here are some notes about the allocation of Multibus Memory resources in the Sun system.

No devices may be assigned addresses below 256K in Multibus memory space since the CPU uses these addresses for DVMA.

The table on the next page shows a map of how Multibus Memory space is laid out in the Sun system. Note that this memory map, as well as all of those that follow, is only a general guide. To be sure that you are not installing a device at a location that will put it in conflict with other, already installed devices, it's necessary to check the configuration of the specific systems into which it will be installed. The best way to do so is to check the local config file for the physical addresses of the devices installed within the bus of interest. This will probably give you enough information, but if you still think that there may be a conflict, and if you have a Sun source license, you can check the driver header files to determine the amount of space consumed on the bus by already installed devices. With the exception of the Sky board, these device can be rearranged. Also note the possibility that your machine will have devices attached to it, and taking up bus space, even though those devices do not appear in the config file. This possibility exists because the *xx*mmap system call can sometimes be used to drive a device without installing it in the formal sense — see the *Mapping Devices Without Device Drivers* section of this manual for more details.

Table 2-2 *Sun-2 Multibus Memory Map*

Address	*Device*
0x00000 – 0x3FFFF	DVMA Space (256 Kilobytes)
0x40000 – 0x7FFFF	Sun Ethernet Memory (#1) (256 Kilobytes)
0x80000 – 0x83800	SCSI (#1) (16 Kilobytes)
0x84000 – 0x87800	SCSI (#2) (16 Kilobytes)
0x88000 – 0x8B800	Sun Ethernet Control Info (#1) (16 Kilobytes)
0x8C000 – 0x8F800	Sun Ethernet Control Info (#2) (16 Kilobytes)
0x90000 – 0x9F800	*** FREE *** (64 Kilobytes)
0xA0000 – 0xAF800	Sun Ethernet Memory (#2) (64 Kilobytes)
0xB0000 – 0xBF800	*** FREE *** (64 Kilobytes)
0xC0000 – 0xDF800	Sun Model 100
0xE0000 – 0xE1800	3COM Ethernet (#1)
0xE2000 – 0xE3800	3COM Ethernet (#2)
0xE4000 – 0xE7C00	*** FREE *** (16 Kilobytes)
0xE8000 – 0xF7800	Reserved for Color Devices (64 Kilobytes)
0xF8000 – 0xFF800	*** FREE *** (16 Kilobytes)

Allocation of Multibus I/O Space

Multibus I/O address space is specified in the config file as `mbio`. From the PROM monitor, Multibus I/O space begins at `0xEB0000`, and extends to `0xEC0000`.

Prior to Sun Release 3.0, the system made the assumption that any address lower than 64K that it found in its config file was a Multibus I/O address. With the advent of Release 3.0, this is no longer true; now the bus type of every address must be explicitly given.

The following table of generic Multibus I/O usage, like the table above, is intended only as a guide.

Table 2-3 *Sun-2 Multibus I/O Map*

Address	*Device Type*
0x0040 – 0x0047	Interphase Disk Controllers
0x00A0 – 0x00A3	CPC TapeMaster Controllers
0x0200 – 0x020F	Archive Tape Drives
0x0400 – 0x047F	Ikon 10071-5 Multibus/Versatec Interface
0x0480 – 0x057F	Systech VPC-2200 Versatec/Centronics Interfaces
0x0620 – 0x069F	Systech MTI-800/1600 terminal Interface
0x2000 – 0x200F	Sky Board
0xEE40 – 0xEE4F	Xylogics 450/451 Disk Controller
0xEE60 – 0xEE6F	Xylogics 472 Multibus Tape Controller

2.2. VMEbus Machines

VMEbus machine architecture is generally more complex than Multibus machine architecture — it makes no distinction between I/O space and Memory space, but on the other hand it supports multiple address spaces. It does so for reasons of both cost and flexibility. The VMEbus was designed to be cost-effective for a range of applications. It is expensive (in terms of money, power, and board space) to provide the hardware for a full 32-bit address space. If installed devices only respond to 16-bit addresses, it makes sense to be able to put them all into a 16-bit address space and save the cost of 16-bits' worth of address decoders and the like. The 24 and 32-bit address spaces are similar compromises between cost and flexibility.

The driver writer has to understand which address space his board uses (generally, this is completely out of his/her control), and make an appropriate entry in the config file. For DMA devices, the driver writer has to know the address space that the board uses for its DMA transfers (this is usually a 32 or 24-bit space).

Sun-2 VMEbus Address Spaces

The Sun-2 VMEbus machines are based upon the 24-bit subset of the generic VMEbus — they support only a 16-bit and a 24-bit address space. These address spaces are known as `vme16d16` (16 data bits and 16 address bits) and `vme24d16` (16 data bits and 24 address bits). Sun-2 VMEbus machines also contain on-board memory and I/O space, of course, but these aren't accessed by way of the VMEbus and are only barely relevant to the driver developer.

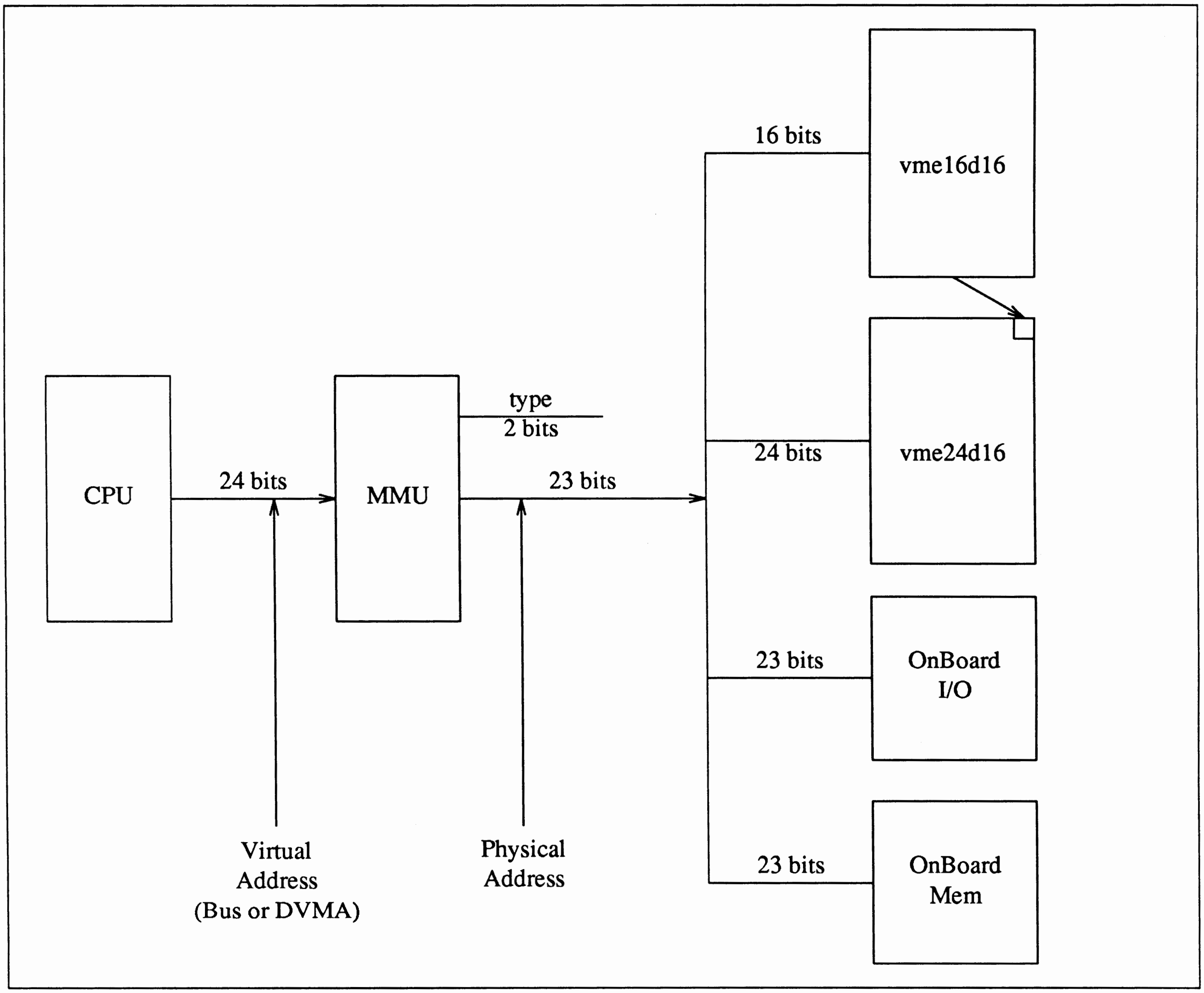
There are four types of memory on Sun-2 VMEbus machines:

Table 2-4 *Sun-2 VMEbus Memory Types*

Description	*Address Size*	*Address Range*
On-Board Memory	23 bits	0x0 - 0x7FFFFF
On-Board I/O Space	23 bits	0x0 - 0x7FFFFF
`vme24d16`	23+1 bits	0x0 - 0xFEFFFF
`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		

The four address spaces are laid out as follows:

Figure 2-2 *Sun-2 VMEbus Address Spaces*



Note a few details:

- In all Sun-2 and Sun-3 machines, the address input into the MMU is a virtual address, and may originate with the CPU or a DVMA bus master.
- Unlike Sun-2 Multibus systems, in which each memory type maps cleanly to one address space, `vme24d16` maps to two different types. Addresses from `0x0` to `0x7FFFFF` are "type 2" memory, while those from `0x8000000` and up are "type 3". This is because Sun-2 VMEbus machines have only 23 output address bits, and this trick is necessary to generate the full range of a 24-bit address space. (See *Sun-2 Address Mapping* for more details).

- Multibus boards, connected to VMEbus to Multibus adapters, can be plugged into physical memory anywhere within `vme24d16` (which means that they can also be in `vme16d16`).
- The 24 bits in the `vme24d16` address space are referred to in the above table as 23+1 bits. This is because, as should be clear in the diagram below, the Sun-2 MMU outputs only the lower 23 bits of the address, and the 24th bit is actually one of the MMU's type bits.
- Note especially that `vme16d16` *is stolen from* `vme24d16`. It's selected by addresses in the form `0xFFXXXX`, that is, addresses which have the 8 high bits set.

Sun-3 Address Spaces

The Sun-3 machines are all based on the full 32-bit VMEbus, so let's begin their discussion with a listing of the address types supported by the generic VMEbus.

Table 2-5 *Generic VMEbus (Full Set)*

VMEbus-Space Name	*Address Size*	*Data Transfer Size*	*Physical Address Range*
`vme32d16`	32 bits	16 bits	`0x0 – 0xFFFFFFFF`
`vme24d16`	24 bits	16 bits	`0x0 – 0xFFFFFF`
`vme16d16`	16 bits	16 bits	`0x0 – 0xFFFF`
`vme32d32`	32 bits	32 bits	`0x0 – 0xFFFFFFFF`
`vme24d32`	24 bits	32 bits	`0x0 – 0xFFFFFF`
`vme16d32`	16 bits	32 bits	`0x0 – 0xFFFF`

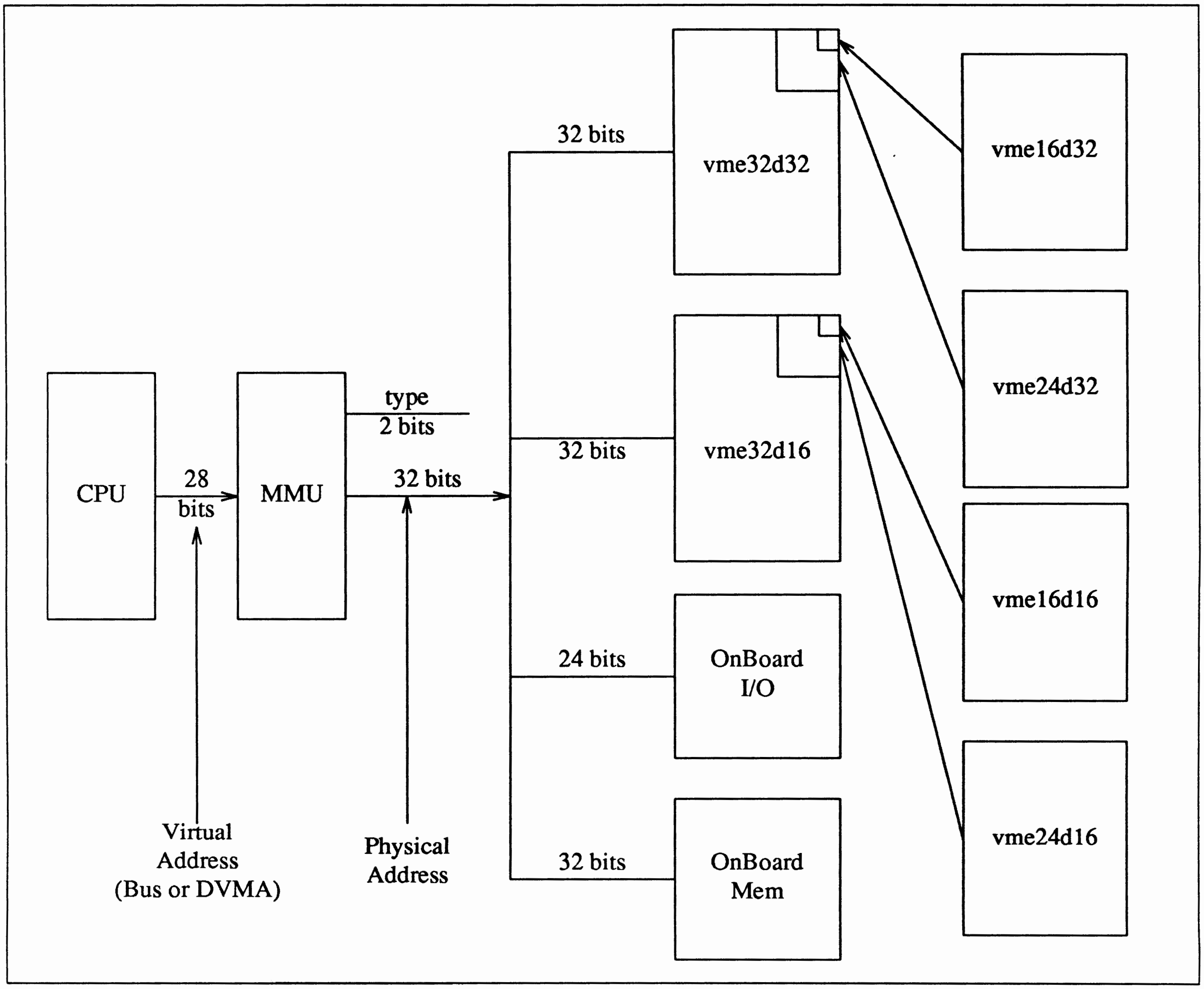
Not all of these spaces are commonly used, but they are all nevertheless supported by the Sun-3 line. The following table indicates their sizes and physical address mappings within Sun-3 computers.

Table 2-6 *Sun-3 VMEbus Address Types*

Type	*Address-Space Name*	*Address Size*	*Address Range*
0	On-board Memory	32 bits	`0x0 – 0xFFFFFFFF`
1	On-board I/O	24 bits	`0x0 – 0xFFFFFF`
2	`vme32d16`	32 bits	`0x0 – 0xFEFFFFFF`
3	`vme32d32`	32 bits	`0x0 – 0xFEFFFFFF`
2	`vme24d16` — Stolen from top 16M of `vme32d16` (0x0 - 0xFFFFFF)		
2	`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		
3	`vme24d32` — Stolen from top 16M of `vme32d32` (0x0 - 0xFFFFFF)		
3	`vme16d32` — Stolen from top 64K of `vme24d32` (0x0 - 0xFFFF)		

The Sun-3 space overlays are much more complex than those of the Sun-2, as is evident from both the table above and the diagram below. The principle, however, is the same — when a space overlays a larger space, its memory is stolen from that larger space and is considered by the MMU to be in the the overlaid space. One simply cannot address above 0xFF000000 in 32-bit VMEbus space or above 0xFF0000 in 24-bit VMEbus space.

Figure 2-3 *Sun-3 VMEbus Address Spaces*



Allocation of VMEbus Memory

This section summarizes the typical use of the 16, 24 and 32-bit VMEbus address spaces by Sun devices. Note well that the usages summarized here are only for the generic configuration, and there's no guarantee that they match the exact usage on your machine. They will, however, help you to decide where to attach your device. The "Allocated From" field shows whether bus space is allocated from the high end of the given range or from the low end. The idea is to keep the maximum size "hole" in the middle in case the boundary later needs to be shifted later.

Table 2-7 *16-bit VMEbus Address Space Allocation*

Address Range	*Allocated From*	*Description of Use*
`0x0000-0x7FFF`	Low	Reserved for OEM/user devices
`0x8000-0xFFFF`	High	Reserved for Sun devices

16-bit VMEbus space is mapped into the topmost 64K of 24-bit VMEbus space at `0x00FF0000` to `0x00FFFFFF` (on the Sun-2) or `0xFFFF0000` to `0xFFFFFFFF` (on the Sun-3). Note: The Multibus/VMEbus Adapter will map the Multibus I/O addresses of Multibus cards that use Multibus I/O into the same addresses in the 16-bit VMEbus space. This may place the standard Multibus addresses for some cards into the OEM/user area in the above table. These addresses can be changed, if necessary, by physically readdressing the device and then changing its entry in the config file.

Table 2-8 *24-bit VMEbus Address Space Allocation*

Address Range	*Allocated From*	*Description of Use*
`0x000000-0x0FFFFF`		CPU board DVMA space
`0x100000-0x1FFFFF`		Reserved by Sun
`0x200000-0x2FFFFF`	Low	Reserved for small Sun devices
`0x300000-0x3FFFFF`	High	Reserved for large Sun devices
`0x400000-0x7FFFFF`	(Taken)	Reserved for huge Sun devices
`0x800000-0xBFFFFF`	High	Reserved for huge OEM/user devices
`0xC00000-0xCFFFFF`	Low	Reserved for large OEM/user devices
`0xD00000-0xDFFFFF`	High	Reserved for small OEM/user devices
`0xE00000-0xEFFFFF`		Multibus-to-VMEbus memory space
`0xF00000-0xFEFFFF`		Reserved for the Future
`0xFF0000-0xFFFFFF`		Stolen by 16-bit VMEbus space

Table 2-9 *32-bit VMEbus Address Space Allocation (Sun-3 Only)*

Address Range	*Description of Use*
`0x00000000 - 0x000FFFFF`	DVMA Space
`0x00100000 - 0x7FFFFFFF`	Reserved by Sun
`0x80000000 - 0xFEFFFFFF`	Reserved for OEM/user devices
`0xFF000000 - 0xFFFFFFFF`	Stolen by `vme24d16`

These same assignments apply to both 16-bit-data and 32-bit-data VMEbus accesses.

Table 2-10 *VMEbus Address Assignments for Some Devices*

Device	*Addressing*	*Addresses Used*
VMEbus SKY Board	`vme16d16`	0x8000 - 0x8FFF (Sun-2 only)
VMEbus SCSI Board	`vme24d16`	0x200000 - 0x2007FF
VMEbus TOD Chip	`vme24d16`	0x200800 - 0x2008FF (Sun-2 only)
Graphics Processor	`vme24d16`	0x210000 - 0x210FFF
Sun-2 Color Board	`vme24d16`	0x400000 - 0x4FF7FF

The VMEbus Sky board occupies addresses 8000-9000 in 16 bit address space, and it requires that the high nibble of the address be '8'. Unlike other pre-installed devices, it cannot be moved.

This table is, of course, not complete. There are a variety of devices on the bus in both the Sun-2 and Sun-3, as can be easily determined by examining the config file. This table, however, does include the standard devices that use a significant amount of space on the VMEbus. Note that in the Sun-3 several of them have disappeared, being replaced by on-board devices.

The Sun VMEbus to Multibus Adapter

Multibus devices that are to be attached to VMEbus machines must be attached to a VMEbus to Multibus adapter. (The Adapter works for most, but not all, Multibus boards). An adapter can be used to take over *one and only one* chunk of `vme24d16`. However, that chunk can overlap all or part of `vme16d16` (because `vme16d16` is a proper subset of `vme24d16`). In any case, the adapter must be told how much space the board attached to it actually expects, for by default it'll take over a full Megabyte. Note that the Multibus Adapter supports fully vectored interrupts, and that drivers for Multibus devices attached by way of adapters need not poll, since the adapters contain switches by which Multibus devices can be assigned vectors.

Interrupt Vector Assignments

The table below shows the assignments of interrupts vectors for those devices that can supply interrupts through the VMEbus vectored interrupt interface. To pick one for your device, examine the kernel config file for an unused number in the range reserved for customer use, `0xC8` to `0xFF`.

Table 2-11 *Vectored Interrupt Assignments*

Vector Numbers	*Description*
0x40 *thru* 0x43	sc0, sc? si0, si? — SCSI Host Adapters
0x48 *thru* 0x4B	xyc0, xyc1, xyc? — Xylogics Disk Controllers
0x4C *thru* 0x5F	future disk controllers
0x60 *thru* 0x63	tm0, tm1, tm? — TapeMaster Tape Controllers
0x64 *thru* 0x67	xtc0, xtc1, xtc? — Xylogics Tape Controllers
0x68 *thru* 0c6F	future tape controllers
0x70 *thru* 0x73	ec? — 3COM Ethernet Controller
0x74 *thru* 0x77	ie0, ie1, ie? — Sun Ethernet Controller
0x78 *thru* 0x7F	future ethernet devices
0x80 *thru* 0x83	vpc? — Systech VPC-2200
0x84 *thru* 0x87	vp? — Ikon Versatec Parallel Interface
0x88 *thru* 0x8B	mti0, mti? — Systech Serial Multiplexors
0x8C *thru* 0x8F	dcp1, dcp? — SunLink Comm. Processor
0x90 *thru* 0x9F	zs0, zs1 — Sun-3 Terminal/Modem Controller
0xA0 *thru* 0xA3	future serial devices
0xA4 *thru* 0xA7	pc0, pc1, pc2, pc3 — SunIPC
0xA8 *thru* 0xAB	future frame buffer devices
0xAC *thru* 0xAF	future graphics processors
0xB0 *thru* 0xB3	sky0, ? — SKY Floating Point Board
0xB4 *thru* 0xB7	Sun-3 SunLink Channel Attach
0xB8 *thru* 0xC7	Reserved for Sun Use
0xC8 *thru* 0xFF	Reserved for Customer Use

2.3. Hardware Peculiarities to Watch Out For

There are a variety of device peculiarities that the driver developer must be aware of. The most common of them are related to the Multibus and Multibus-based devices, but there are others as well.

Multibus Device Peculiarities

The IEEE Multibus is a source of problems for two separate reasons. The first of these, discussed immediately below, is the fact that the Multibus has a different notion of byte order than does the Motorola MC680X0 family of processors (which are used in Sun machines). The second is simply that the Multibus has been around for a long time, and thus brings with it a variety of older devices, many of which have addressing limitations and other characteristics which make for a less than perfect fit with the Sun architecture.

Multibus Byte-Ordering Issues

Sun processors are all members of the Motorola MC680X0 family, and these processors address bytes within words by what we shall call *IBM conventions* — the most significant byte of an word is stored at the lowest addresses byte of the word. The Multibus, on the other hand, uses *DEC conventions* — the least significant byte of an word is stored at the lowest address, and significance increases with address.

This class of byte-addressing conventions leads to two separate problems, with two separate solutions:

- □ The first problem occurs when you're moving a single *byte* across the interface between the MC680X0 and the IEEE Multibus. Because the two devices don't agree about the end of the word that the byte actually appears in, you have to change the byte address before the move — what you want to do, in effect, is move every byte to the other side of the word which it occupies — the most CPU-efficient way of doing so is to toggle the least significant bit of every byte address.
- □ The second problem, also related to the Multibus, is a higher level version of the first. It occurs when 16-bit *words* with significant internal structure (or structures that contain 16-bit words) are moved across the bus interface. (If you write only words, and the device uses only words, there's no problem). The Multibus/MC680X0 byte-ordering incompatibility will cause structures to be scrambled when they're moved across the bus interface, unless the bytes within them are physically swapped first.

Here are a few pictures describing the problems in detail:

Motorola (IBM) Byte Ordering

bit 15	bit 0
Byte 0	Byte 1

Multibus (DEC) Byte Ordering

bit 15	bit 0
Byte 1	Byte 0

That is, the MC680X0 places byte 0 in bits 8 through 15 of the word, whereas the Multibus places byte 1 in those bits. If you did everything with a MC680X0, or everything on the Multibus, there wouldn't be any conflict, since things would be consistent. However, as soon as you cross the boundary between them, the byte order is reversed. Thus, you have to toggle the least significant bit of the address of any *byte* destined for the Multibus — this will have the effect of swapping adjacent addresses and thus reordering the bytes.

To clarify this, consider an interface for a hypothetical Multibus board containing only two 8-bit I/O registers, namely a control and status register (csr) and a data register (we actually use this design later on in our example of a simple device driver). In this board, we place the command and status register at Multibus byte location 600, and the data register at Multibus byte location 601. The Multibus picture of that device looks like this:

Hypothetical Board Registers

bit 15	bit 0
Location 601 DATA	Location 600 CSR

But the MC680X0 processor family views that device as looking like this:

Hypothetical Board Registers

bit 15	bit 0
Location 600 CSR	Location 601 DATA

so that if you were to read location 600 from the point of view of the MC680X0 processor, you'd really end up reading the DATA register off the Multibus instead. So, when we define the *skdevice* data structure for that board, we define it by starting with the register definition in the device manual, and then swapping bytes to take account of the expected byte swapping:

```
struct skdevice {
        char     sk_data;        /* 01: Data Register */
        char     sk_csr;         /* 00: command(w) and status(r) */
};
```

This rule (flipping the least significant bit of the address) holds good for all *byte* transfers which cross the line between the MC680X0 and the Multibus.

Other Multibus-related Peculiarities

□ Many Multibus device controllers are geared up for the 8-bit 8080 and Z80 style chips and don't understand 16-bit data transfers. Because of this, such controllers are quite happy to place what's really a word quantity (such as a 16-bit address which must be two-byte aligned in the MC680X0) starting on an odd byte boundary. Some devices use 16-bit or 20-bit addresses (many don't know about 24-bit addresses), and it often happens that you have to chop an address into bytes by shifting and masking, and assign the halves or thirds of the address one at a time, because the device controller wants to place word-aligned quantities on odd-byte boundaries. Note also that many Multibus boards are geared up for the 8086 family with its segmented address scheme. An 8086 (20-bit) address really consists of a 4-bit segment number and a 16-bit address; you usually have to deal with the 4-bit part and the 16-bit part separately. For a good example of what we're talking about here, see the code for `vp.c` in the *Sample Driver Listings* appendix to this manual.

- Although there are a myriad of vendors offering Multibus products, remember that the Multibus is a "standard" that evolved from a bus for 8-bit systems to a bus for 16-bit systems. Read vendors' product literature *carefully* (especially the fine print) when selecting a Multibus board. The memory address space of the Multibus is *supposed to be* 20 or 24 bits wide and the I/O address space of the Multibus is *supposed to be* 16 bits wide. In practice, some older boards are limited to 16 bits of address space and 8 bits of I/O space. In particular, watch for the following addressing peculiarities:

  - For a memory-mapped board, ensure that the board can actually handle a full twenty bits of addressing. Older Multibus boards often can only handle sixteen address lines. The Sun system assumes there is a 20-bit Multibus memory space out there. If the Multibus board you're talking to can only handle 16-bit addresses, it will ignore the upper four address lines, and this means that such a board "wraps around" every 64K, which means that on a Sun the addresses that such a board responds to would be replicated sixteen times through the one-Megabyte address space on the Multibus. This may conflict with some other device.

  - Some Sun-2 Multibus systems, notably Sun-2/170's, have a backplane structure that complicates the installation of 24-bit memory-mapped Multibus. The internal "bus" on these systems (often called the P2 bus) is divided into multiple segments, each mapped to a portion of the backplane slots. In such systems, 24-bit memory-mapped devices must be installed in a different segment than that used by standard Sun-2 devices. See the *Sun-2/170 Configuration Guide* for more information.

  - For an I/O-mapped board (one that uses I/O registers), make sure that the board can handle 16-bit I/O addressing. Some older boards can't cope and only use eight-bit I/O addressing. In our system, the address spaces of such boards would find themselves replicated every 256 bytes in the I/O address space. Trying to fit such a board into the Sun system would severely curtail the number of I/O addresses available in the system.

- Finally, watch out for boards containing PROM code that expects to find a CPU bus master with an Intel 8080, 8085, or 8086 on it. Such boards are of course useless in the Sun system.

Other Device Peculiarities

There are other device peculiarities of interest to the driver developer that are not specific to the Multibus and Multibus-based devices. These peculiarities are particularly unfortunate in that they tend to require special handling of various kinds — byte swapping, bit shuffling, timing delays, etc. — whenever the driver contacts the device. Such special handling precludes the most obvious and desirable means of interfacing the driver to the device, by mapping the device registers into a C-structure declaration and then accessing them by way of references to structure fields.

- One of the most infuriating of these peculiarities is internal sequencing logic. Devices with this strange characteristic (a vestige of microcomputer systems with extremely limited address space) map multiple internal registers to the same externally addressable address. There are various kinds of internal sequencing logic:
  - The Intel 8251A and the Signetics 2651 alternate the same external register between *two* internal mode registers. Thus, if you want to put something in the first mode register of a 8251, you do so by writing to the external register. This write will however, have the invisible side effect of setting up the sequencing logic in the chip so that the next read/write operation refers to the alternate, or second, internal register.
  - The NEC PD7201 PCC has multiple internal data registers. To write a byte into one of them, it's necessary to first load the first (register 0) with the number of the register into which the following byte of data will go — you then send that byte of data and it goes into the specified data register. The sequencing logic then automatically sets up the chip so that the next byte sent will go into data-register 0.
  - Another chip of a similar ilk is the AMD 9513 timer. This chip has a data pointer register for pointing at the data-register into which a data byte will go. When you send a byte to the data register, the pointer gets incremented. The design of the chip is such that you *can't read the pointer register to find out what's in it*!
- In fact, it's often true that device registers, when read, don't contain the same bits that were last written into them. This means that bitwise operations (like `register &= ~XX_ENABLE`) that have the side effect of generating register reads must be done in a software copy of the device register, and then written to the real device register.
- Another problem is timing. Many chips specify that they can only be accessed every so often. The Zilog Z8530 SCC, which has a "write recovery time" of 1.6 microseconds, is an example. This means that a delay has to be enforced (with DELAY) when writing out characters with a 8530. Things can get worse, however, for there are instances when it's unclear what delays are needed, and in such cases it's left to the driver developer to determine them empirically.
- And peripheral devices can contain chips that use a byte-ordering convention different from that used by the Sun system into which they're installed. The Intel 82586, for example, supports DEC byte-ordering conventions; this makes it perfectly compatible with Multibus-based, but not VMEbus-based, Sun machines. Drivers for such peripheral devices will have to swap bytes, as indicated above, and to take care that, in doing so, they don't inadvertently reorder the bits in any control fields greater than 16 bits in length.
- Finally, there are some common interrupt-related peculiarities worth noting:
  - When a controller interrupts, it does *not* necessarily mean that both it *and* one of its slave devices are ready. Some controllers are designed in this way, but others interrupt to indicate that the controller or one of its

devices *but not necessarily both* is ready.

- Not all devices power up with interrupts disabled and then start interrupting only when told to do so.
- While there should be a way to determine that a board has actually generated an interrupt — an attention bit or something equivalent — some devices have no such thing.
- Finally, an interrupting board should shut off its interrupts when told to do so (and also after a bus reset). Not all do.

2.4. DMA Devices

Many device controller boards are capable of what is known as Direct Memory Access or DMA. This means that the CPU can tell the device controller for such devices the address in memory where a data transfer is to take place and the length of the data transfer, and then instruct the device controller to start the transfer. The data transfer then takes place without further intervention on the part of the processor. When it's complete, the device controller interrupts to say that the transfer is done.

Sun Main-Bus DVMA

Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun Memory Management Unit to allow devices on the Main Bus to perform DMA directly to Sun processor memory. It also allows Main Bus master devices to do DMA directly to Main Bus slaves without the extra step of going through processor memory. DVMA works by ensuring that the addresses used by devices are processed by the MMU, just as if they were virtual addresses generated by the CPU. This allows the system to provide the same memory protection and mapping facilities to DMA devices as it does to the system CPU (and thus to programs).

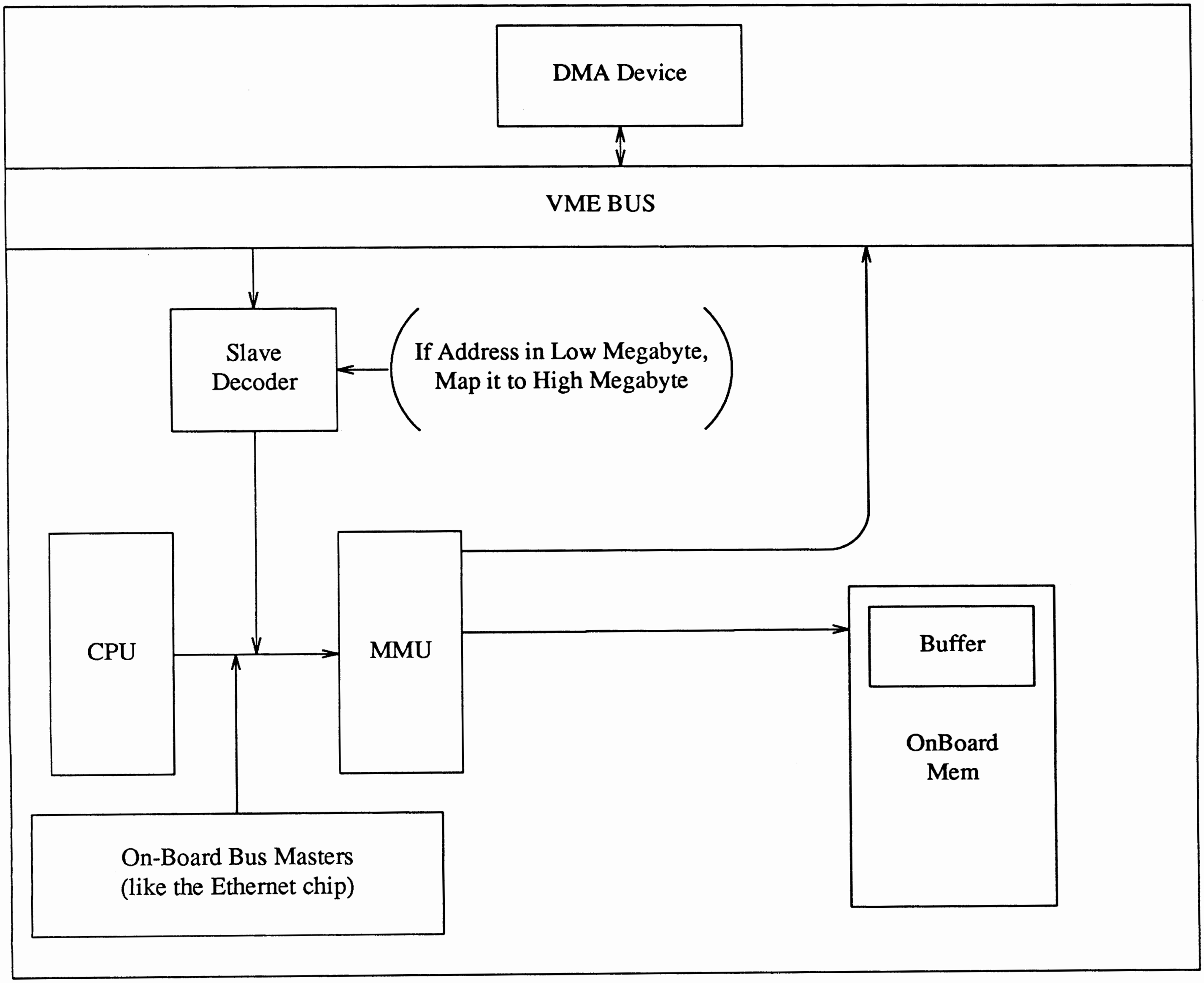
When setting up a driver to support DMA, it's necessary to know the device's DMA address size. This address size is the primary factor used in determining which of the system address spaces will host the device. Multibus devices generally have a DMA address size of 20 bits, while VMEbus devices generally have a 24 or 32-bit DMA address size.

- Since, on Sun-2 Multibus machines, DMA addresses are generally 20-bits long, the system DVMA hardware responds to the first 256K of Multibus address space (`0x0` to `0x3FFFF`). When an address in this range appears on the bus, the DVMA hardware adds `0xF00000` to it (the system places the Multibus memory address space at `0xF00000` in the system's virtual address space) and then uses the MMU to map to the location in physical memory that will be used for the data transfer.
- On Sun-2 VMEbus systems, the DVMA hardware responds to the entire lower Megabyte of VMEbus address space (`0x0` to `0xFFFFF`). The system maps addresses in this range into the most significant Megabyte of system virtual address space (`0xF00000` to `0xFFFFFF`).
- On Sun-3 systems, the DVMA hardware responds to the lowest Megabyte of VMEbus address space *in both the 24-bit and 32-bit VMEbus spaces*. It

maps addresses in this megabyte into the most significant Megabyte of system virtual address space (`0xFF00000` to `0xFFFFFFF`). The Sun-3 DVMA hardware uses supervisor access for checking protection.

The driver writer must account for these mappings, as should be evident from the diagram below.

Figure 2-4 *System DVMA*


DMA Device
VME BUS
Slave Decoder
If Address in Low Megabyte, Map it to High Megabyte
CPU
MMU
Buffer
OnBoard Mem
On-Board Bus Masters (like the Ethernet chip)

Device can only make DVMA transfers in memory buffers which are from (or redundantly mapped into — see below) the low-memory areas reserved as DVMA space. The memory-management hardware will then recognize references to these areas and map them into the high Megabyte of system virtual address space, an area known as DVMA space. Likewise, if a driver needs to allocate space for a DMA transfer, it must do so by way of a mechanism that guarantees its allocation from DVMA space. There are several ways of making this guarantee:

- rmalloc can be used with the iopbmap argument. This will get a small block of memory from the beginning of the DVMA space. Such small blocks of memory are usually used for control information, and not for large blocks of data.
- For a large buffer, the driver can statically declare a buf structure (which is a buffer header that contains a pointer to the data) and then use mbsetup to allocate a buffer for it from DVMA space. This mechanism is primarily intended for block devices but is perfectly adaptable for use by character devices that need large DMA buffers.

When dealing with buffers which, like those allocated by mbsetup, are in DVMA space, the driver itself need do nothing unusual. However, before passing the buffer address to the device, the driver must strip the high bits from its address. It does so by subtracting the external variable DVMA, which is declared as a character array and initialized by the system to either 0xF00000 (for Sun-2's) or 0xFF00000 (for Sun-3's). If it fails to do so the device will attempt to use the null address — in the high Megabyte — and the CPU board will not respond to it. (Note that addresses received by way of mbsetup and MBI_ADDR do *not* have to be adjusted in this fashion, as mbsetup will have already adjusted them to be relative to the start of DVMA space). When the device, in turn, uses the address, the address reference comes down the bus and through a slave decoder, which adds the machine-specific offset to it to map it back into the high Megabyte of system virtual memory.

Sun DMA is called DVMA because the addresses which the device uses to communicate with the kernel are virtual addresses like any others. The driver, as part of the kernel, is privy to implementation dependent information, and knows that it must chop off the high-bits of any address intended for the device. This allows the MMU to recognize the addresses destined for the Main Bus and to act accordingly. The device, however, knows nothing of this except that its buffers are mapped to the high Megabyte of system virtual memory.

User processes, it should be noted, cannot do DVMA directly into their own address spaces. The kernel, however, provides a way of getting around this limitation by supporting the redundant mapping of physical memory pages into multiple virtual addresses. In this way, a page of user memory (or, for that matter, a page of kernel memory) can be mapped into DVMA space in such a way that transferred data immediately appears in (or immediately comes from) the address space of the process requesting the I/O operation. All that a driver need do to support such direct user-space DVMA is to set up the kernel page maps with the routine mbsetup — the details of the mapping will then be automatically handled by the kernel.

If you wish to do DMA over the Main Bus, you must make the appropriate entries in the kernel memory map. There are two functions, mbsetup and mbrelse, to help with this chore.

3

Overall Kernel Context

Overall Kernel Context **33**

3.1. The System Kernel 33

3.2. Devices as ‘‘Special’’ Files 34

3.3. Run-Time Data Structures 40

The Bus-Resource Interface 41

Autoconfiguration-Related Declarations 47

Other Kernel/Driver Interfaces 48

3

Overall Kernel Context

3.1. The System Kernel

Device drivers are parts of the UNIX system kernel, a fact that must be appreciated to understand the ways in which drivers differ from user-level programs. The kernel is the crucial system program responsible for the control and allocation of system resources, including the processor, primary memory and the I/O devices. In most ways it's just like any user program, being a more or less cleverly constructed structure shaped to its particular goals. In other ways, however, it's significantly different from a user program:

- For one thing, the kernel is thick with the details of hardware implementation and function. This tends *not* to be true of user programs, precisely because the kernel shields them from the need to consider device-specific details.
- For another, the kernel (and thus its drivers) runs in supervisor mode. This means that drivers can often perform privileged device operations that can't be performed by user processes, even if those processes have access to the necessary device registers.
- The kernel memory context is *not paged.* The entire kernel, and all of its drivers, are memory resident at all times. This may change in the future, but it'll remain possible for drivers to assume that they are resident and stationary within physical memory.
- Programmers of ordinary user processes rarely need to concern themselves with physical addresses and virtual-to-physical address mappings. Device-driver developers, however, deal simultaneously with user virtual addresses, kernel virtual addresses and physical bus addresses. Special functions (see the *Kernel Support Routines* appendix) are provided to help drivers with the various address mappings they're called upon to perform.
- Finally, the kernel provides a far different external interface than do user processes. It's possible for user processes to communicate with and dispatch tasks to other user processes by way of system inter-process communications mechanisms (like signals and pipes) but to do so they must first make special arrangements with those other processes. The kernel, on the other hand, exists to provide services to user processes and it provides a special mechanism — the system call — by which user processes can call upon it to do so. This is not to say that user processes and the kernel (that is, the drivers) can't also use system inter-process communications mechanisms like sockets and signals. It's certainly possible, for example, to write a driver so that it will

send a signal to a user process as part of its handling of a specified event. However, in the norm, user processes and the kernel communicate by way of system calls.

Note: system calls are defined in `/usr/sys/sys/init_sysent.c`, *which is shipped with all systems so that users can add system calls if they wish.*

System calls can, for all intents and purposes, be understood as calls by user processes to kernel subroutines; they involve, however, far more profound system state changes that do regular subroutine calls. When system calls are processed, the processor is placed in supervisor state (and, in Sun-2 systems, the kernel virtual address space replaces the user virtual address space). The user process is suspended and the kernel begins to run, but since it runs on behalf of that user process which issued the system call, it can be viewed as that user process continuing execution in kernel mode. Such "kernel-mode" processes continue to run (with pauses whenever they sleep or yield to higher-priority process) until the system call processing is completed. At this time the scheduler is called to choose the next user process to be dispatched.

Some system calls can be completely processed without calling any device driver routines. The system call `lseek` is in this class, it requires only that a software file position indicator be reset. Like many system calls — those related to process control, inter-process communication, timing services, and status information — it can be handled entirely in software. Requests for I/O, however, usually involve some action on the part of a peripheral device. In this case the kernel calls (through a branch table mechanism described below) a routine within the I/O device's driver. The driver will then initiate the I/O operation and, if necessary, `sleep` until the data is available; in the meantime the kernel will dispatch another user process.

3.2. Devices as "Special" Files

When a user process issues a system call, execution shifts to the kernel. Then, for I/O-related system calls, the kernel distinguishes requests related to regular named files (that is, files on a block device like a disk) from requests related to other kinds of I/O devices (like terminals or printers). In the interests of uniformity, UNIX views these other devices as "special" files which (by convention) are collected in the */dev* directory. These special files are not created in the usual way. The information in their *i-nodes* (the system structures that define the state of files) is quite different from the information maintained for regular files, and, as a consequence, special files can only be created with the `mknod` (make a node) administration command. Instead of the addresses that will locate the contents of a regular file on a disk, the i-nodes of special files (devices) contain the information necessary to determine the corresponding device driver (the major device number), the device class (block or character), and the minor device number.

When a file of any type is accessed, the kernel needs to determine which device driver is responsible for it. To make this determination, it must get the name of the device associated with the file. From that name it can derive (using a device-independent kernel subsystem) an i-node and thus a major device number (as well as a minor device number and a device class).

The connection between the device name and its major number is made by way of the device entry in the */dev* directory (more specifically, by way of the i-node information associated with the device entry). The i-node for a device special file contains a major device number, which is used to index one of the two *device switches*. These switches, `bdevsw` (the block device switch) and `cdevsw` (the character device switch) are actually arrays of structures, and the major device number selects a driver by indexing one of these structures. (The minor device number is then passed to the driver for local interpretation).

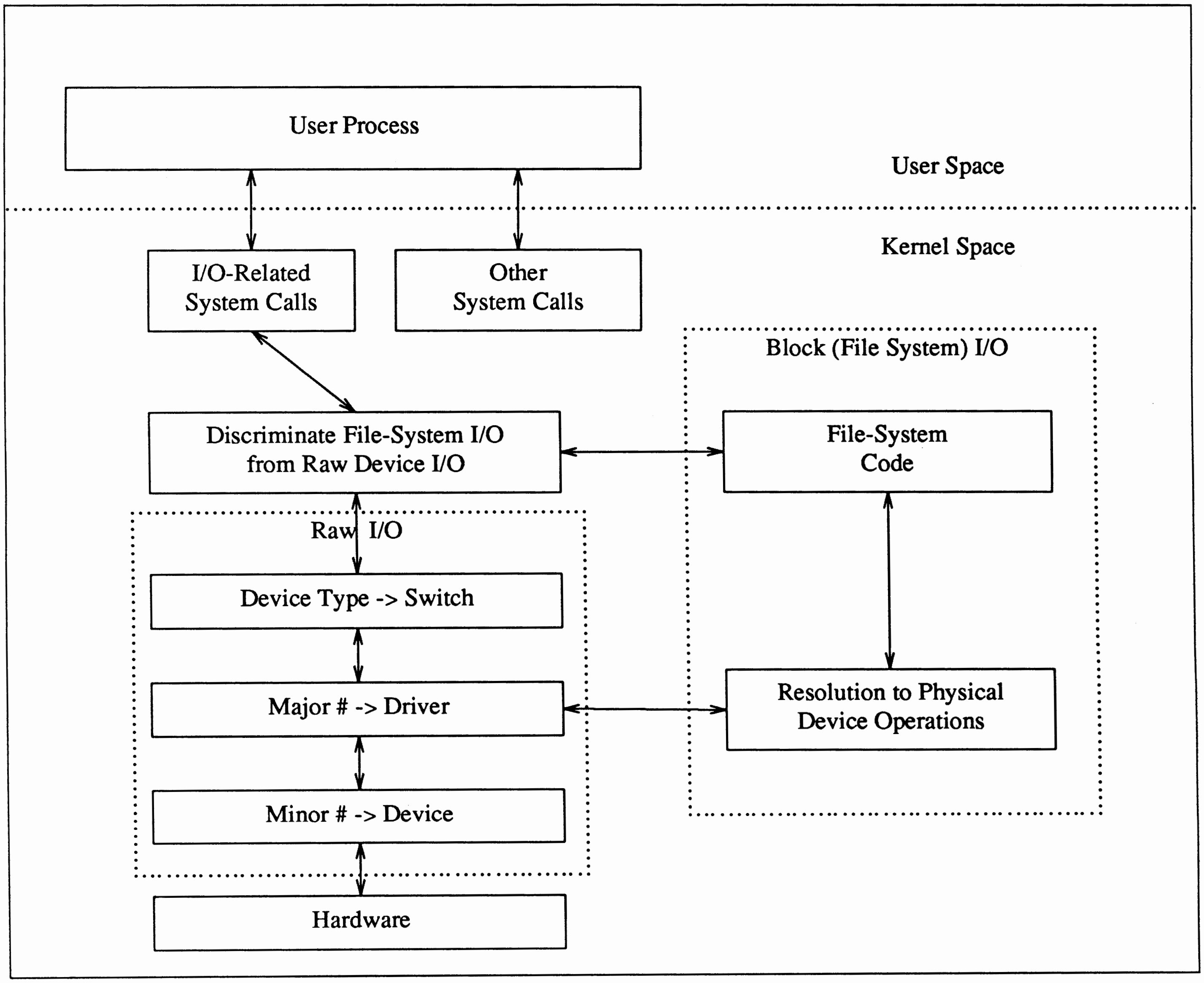
Using the `ls -l` command on the `/dev/` directory shows you the i-node information associated with special files:

Table 3-1 *A Sample Listing of the /dev Directory*

T y p e	*per- mis- sions*	*s i z e*	*own- er*	*maj- or #*	*min- or #*	*date*	*name*
c	rw--w--w-	1	henry	0,	0	Feb 21 09:45	console
c	rw-r--r--	1	root	3,	1	Dec 28 16:18	kmem
c	rw-------	1	root	3,	4	Jan 13 23:07	mbio
c	rw-------	1	root	3,	3	Jan 13 23:07	mbmem
c	rw-r--r--	1	root	3,	0	Dec 28 16:18	mem
c	rw-rw-rw-	1	root	13,	0	Dec 28 16:18	mouse
c	rw-rw-rw-	1	root	3,	2	Feb 22 16:40	null
c	rw-------	1	root	9,	0	Dec 28 16:19	rxy0a
c	rw-------	1	root	9,	1	Dec 28 16:19	rxy0b
							. . .
c	rw-------	1	root	9,	6	Feb 25 1984	rxy0g
c	rw-------	1	root	9,	7	Dec 28 16:19	rxy0h
b	rw-------	1	root	3,	0	Feb 25 1984	xy0a
b	rw-------	1	root	3,	1	Jan 17 20:12	xy0b
							. . .
b	rw-------	1	root	3,	6	Dec 28 16:19	xy0g
b	rw-------	1	root	3,	7	Dec 28 16:19	xy0h

When a user process wishes access to a system service, it makes a system call. The subsequent flow of control looks somewhat like this:

Figure 3-1 *Unix I/O Paths*



When you add a new device driver you must add entries to one or both of the device switches. Since we are discussing only character-oriented devices in this manual, we'll ignore the `bdevsw` structure and concentrate on the `cdevsw` structure. But note that it's common for drivers to appear in both tables; this happens because block-devices almost always support raw character I/O.

Application programs make calls upon the operating system to perform services such as opening a file, closing a file, reading data from a file, writing data to a file, and other operations that are done in terms of the file interface. The operating system code turns these requests into specific requests on the device driver involved with that particular file. The glue between the specific file operation involved and the device driver entry-point is through the `bdevsw` and cdevsw tables.

Each entry in bdevsw or cdevsw contains pointers to a driver's entry-point functions. The position of an entry in the structure corresponds to the major device number assigned to the device. The minor device number is passed to the device driver as an argument. Usually, the driver uses it to access one of several identical physical devices, but it is also possible for it to be encoded so that multiple minor numbers indicate the same device, but different operating modes. For example, one minor number might indicate a specific tape device, as well as the fact that the device is to be rewound when being closed, while another indicates the same device without the rewind. A minor number may also indicate a controller/device pair. Such breadth of interpretation is possible because the minor number has no significance other than that attributed to it by the driver itself.

The cdevsw table specifies the interface routines present for character devices. Each character device may provide seven functions: *xx*open, *xx*close, *xx*read, *xx*write, *xx*ioctl, *xx*select, and *xx*mmap. (While character drivers sometimes have "strategy" routines, this name is simply a carryover from the world of block drivers, and cdevsw thus has no *xx*strategy entry point). If you wish calls on a routine to be ignored — for example *xx*open calls on non-exclusive devices that require no setup — the cdevsw entry for that driver can be given as nulldev; if a call should be considered an error — for example *xx*write on read-only devices — nodev, which returns immediately with an error code, can be used. For terminals, the cdevsw structure also contains a pointer to an array of tty structures associated with the driver.

Note: the device switch tables do not include pointers to the driver initialization and interrupt handler functions. Pointers to these functions appear in separate mbvar structures (discussed below).

Here's what the declaration of an entry in the character device switch looks like. Each entry (row) is the only link between the main UNIX code and the driver. The declaration and initialization of the device switches is in /usr/sys/sun/conf.c:

```
struct cdevsw {
    int (*d_open)();          /*  routine to call to open the device   */
    int (*d_close)();         /*  routine to call to close the device   */
    int (*d_read)();          /*  routine to call to read from the device   */
    int (*d_write)();         /*  routine to call to write to the device   */
    int (*d_ioctl)();         /*  special interface routine   */
    int (*d_stop)();          /*  flow control in tty's  */
    int (*d_reset)();         /*  reset device and recycle its bus resources  */
    struct tty *d_ttys;       /*  tty structure   */
    int (*d_select)();        /*  routine to call to select the device   */
    int (*d_mmap)();          /*  routine to call to mmap the device   */
    struct streamtab *d_str;  /* future support for System V streams  */
};
```

Only teletype-like devices (such as the the console driver, the mti driver, and the zs driver) use the tty structure. All other devices set it to zero.

Routines in the kernel call specific driver routines indirectly by way of the table with the major device number. A typical kernel call to a driver routine will look something like:

```
(*cdevsw[major(dev)].d_open) (params...);
```

And here is a typical line from `/usr/sys/sun/conf.c`, which initializes the requisite pointers in the `cdevsw` structure:

```
                .
                .
                .
                All the other cdevsw entries between 0 and 13 appear first
        {
cgoneopen,  cgoneclose, nodev,  nodev,  /*14*/
cgoneioctl, nodev,  nodev,  0,
seltrue,    cgonemmap,
        },

                Then all the other cdevsw entries from 15 up
                .
                .
                .
```

In the Sun system, a number of devices in `cdevsw` are preassigned. The table below shows the assignments at the time of this writing, though new devices are always being added. In allocating a major number to the new device which you're installing, you must make sure that you don't choose one that's already been allocated. `/sys/sun/conf.c` will contain the major device numbers as currently allocated. Choose yours so it'll go at the end.

Table 3-2 *Current Major Device Number Assignments*

Major Device Number	*Device Abbreviation*	*Device Description*
0	`cn`	Sun Console
1	*not available*	no device
2	`sy`	Indirect TTY
3	*Memory special files*	
4	`ip`	Raw Interphase Disk Device
5	`tm`	Raw Tapemaster Tape Device
6	`vp`	Ikon Versatec Parallel Controller
7	*not available*	
8	`ar`	Archive Tape Controller
9	`xy`	Raw Xylogics Disk Device
10	`mti`	Systech MTI
11	`des`	DES Chip
12	`zs`	UARTS
13	`ms`	Mouse
14	`cgone`	Sun-1 Color Graphics Board
15	`win`	Window Pseudo Device
16	`ii`	INGRES lock device
17	`sd`	Raw SCSI disk
18	`st`	Raw SCSI tape
19	`nd`	Raw Network Disk Device
20	`pts`	Pseudo TTY
21	`ptc`	Pseudo TTY
22	`fb`	Sun console frame buffer
23	`ropc`	RasterOp Chip
24	`sky`	SKY Floating Point Board
25	`pi`	Parallel input device
26	`bwone`	Sun1 Monochrome frame buffer
27	`bwtwo`	Sun-2 Monochrome frame buffer
28	`vpc`	Parallel driver for Versatec printer
29	`kbd`	Sun console keyboard driver
30	`xt`	Raw Xylogics 472 Tape Controller
31	`cgtwo`	Sun-2 Color Frame Buffer
32	`gpone`	Graphics Processor
33	`sf`	Raw SCSI Floppy
34	`fpa`	Floating-Point Accelerator
35	*not available*	System V Stream Support
36	`xd`	Xylogics 751 SMD Disk Driver
37	*not available*	System V Stream Support
38	`pc`	Sun PC Driver
39	`cgfour`	Sun-3/110 Color Frame Buffer

3.3. Run-Time Data Structures

If you skip ahead and read the chapter on *Configuring the Kernel* you'll see a discussion of the procedures by which Sun systems are reconfigured to include new devices and drivers. There are two major programs involved in this process. The first is `config`, which reads the kernel config file and generates the data-structure tables which specify the configuration of the new kernel. You'll also note, in that chapter, references to the kernel's autoconfiguration process (sometimes called `autoconf`). The autoconfiguration process verifies that the devices specified in the config file are actually installed and working, and adjusts the kernel data structures accordingly.

The autoconfiguration approach was first introduced in 4.1BSD as part of a larger kernel rationalization, and it significantly increases the flexibility of the kernel configuration process, for example, by allowing multiple device controllers to be driven by the same instance of a driver.

The autoconfiguration process is called by the kernel during its boot-time initialization. It does several things:

□ It verifies that the information in the kernel config file is correct; that is to say, it verifies that the devices which the kernel thinks are installed are actually installed. It does this by calling device-specific *xx*`probe` routines that are supplied by the driver.

□ It completes the initialization of the kernel data structures that were declared by `config` and linked into the kernel by way of `ioconf.c` (a file which `config` creates but cannot fully initialize). These structures, which are defined in `<sundev/mbvar.h>` and shall hereafter be known as the *mbvar structures*, form a good part of the run-time environment of the driver routines.

□ It maps the device registers (or memory) into kernel virtual space.

□ It sets up polling interrupts on Multibus systems.

The autoconfiguration code does its work, as its name indicates, without worrying the driver developer too much. It's only necessary for the developer to know what conventions to follow and what options exist. The rest will take care of itself.

Note: readers who have written only System V drivers will perhaps find this all a bit mysterious. In System V, as in 4.2BSD, the driver interface to the kernel is defined primarily by the function switch (either `cdevsw` *or* `bdevsw`*) by which driver routines are called, by the parameters these routines are passed and by the values they return. So far so good, but then there are the differences. In System V drivers, nothing like the mbvar structures exists, and generic kernel structures (like the* `user` *structure) are used far more heavily than in 4.2BSD, where mbvar-like structures are consulted by preference. Sun's operating system is, of course, derived from 4.2BSD, and its driver interface is quite similar.*

The *"mb"* in the name of the *mbvar* structures clearly recalls the primary motivation of the kernel rewrite in which they were introduced — to improve the management of bus resources. The *"mb"* is derived from the initials of the *Multibus*, around which older generation Sun machines were built. Newer machines, while built around the VMEbus, nevertheless continue to bear the traces of the past in these *mbvar* structure names, names which are now taken to stand for "Main Bus" rather than for "Multibus".

During the configuration of the kernel, an edifice is built of the *mbvar* structures and its initialization is begun. The edifice consists of a structure which represents the bus itself, two arrays of structures (one representing system controllers; the other, devices) and a number of inter-structure field-to-field links of various kinds.[3] The details of the edifice depend upon the information in the kernel config file, and upon the compile-time declarations made by the individual drivers. During boot time, the initialization that `config` began is completed by the autoconfiguration process.

Then, at run time, the *mbvar* structures are used by both the drivers and the kernel to manage the bus and its interaction with the devices. The *mbvar* structures are linked to each other in quite a complex fashion, for device characteristics and thus device driver structures vary greatly, and these structures are intended to support a great variety of access paths: device to controller, device to driver, controller to driver, and so on. Driver developers do not, however, need to concern themselves with the details of the inter-structure links and access paths. Driver routines will be called by the kernel with pointers to the *mbvar* structures of interest to them. They are then free to build that information into whatever local structures they find most convenient for the representation of whatever access paths are of interest to them.

So, to sum up, the Sun kernel/driver runtime interface can be seen as being built in two different sections. One of these sections is composed of the *mbvar* structures, constructed into interlinked arrays to represent specific kernel configurations on specific machines. The other is similar to the generic UNIX kernel/driver interface, consisting as it does of the two device switches, the `user` and `proc` structures, parameter conventions and a few miscellaneous variables. We'll now discuss the details to these two interfaces.

The Bus-Resource Interface

All controllers are installed on the main system bus, and all slave devices (like disks and tape drivers) are attached to their controllers.[4] Additionally, each controller is associated with a device driver, which is really a controller driver. The *mbvar* data structures reflect these relationships, not only in terms of the fields that they contain but in terms of the ways these fields are linked together.

[3] It's not always clear just when a device is a "controller", and when it's a "device". The extreme cases are clear: if a device attaches to the bus, fields interrupts and has other, so-called "slave" devices, then it's a controller. Likewise, if a device attaches to a controller rather than to the bus, it's a slave device. The confusion surrounds devices which attach to the device and field interrupts, but which do *not* have slave devices. Such "devices" would, in many ways, be better thought of as "controllers" which control only themselves.

[4] Sometimes, in this manual, the word "device" will be used in a generic sense to denote either a "free" device that attaches directly to the system bus rather than to a separate controller, or a regular slave device. This generic usage occurs, for example, whenever the term "device driver" is used — such programs would more accurately be described as "controller drivers". In this section, however, we're being extremely precise — free devices attach to the system bus, and so they're called "controllers", not "devices".

The following *mbvar* structure fields are the ones most relevant to driver developers.

mb_hd The first data structure, mb_hd, is the Main Bus header data structure. There is only one such structure, for Sun systems have only one Main Bus. It contains a queue of mb_ctlr structures, each one representing a controller waiting for DVMA space. The queue only contains entries when DVMA space is full. It also contains other bus-status information. For example, if a driver has exclusive access to the bus, this is noted in mb_hd. Device drivers never directly access the fields in mb_hd.

mb_ctlr Each slave-device controller on the Main Bus has a mb_ctlr structure associated with it. (This structure contains all of the configuration-dependent information which the kernel needs in interactions with the controller's driver, as well as some status information. It is mb_ctlr that is queued onto mb_hd during a wait for DVMA space. The following fields within mb_ctlr are of interest even for character devices (there are others that are used only by block devices):

mc_ctlr
The controller index for the corresponding controller, for example, the '0' in sc0. Used to index into arrays of driver-specific controller status and control structures.

mc_addr
The address of the controller (control and status registers and RAM) in bus space.

mc_space
A bit pattern which identifies the address space within which the controller is installed.

mc_intpri
The interrupt priority level of the controller. This is to be given in the config file and should *never* be hardwired in the driver source code.

mc_intr
Pointer to the vec structure that specifies vectored interrupt behavior (or NULL if vectored interrupts are not used). If mc_intr is set, then the fields within the vec structure become significant:

v_func
Pointer to the vector-interrupt function.

v_vec
Vector number associated with the function in v_func.

v_vptr
The 32-bit argument to be passed to the driver vector-interrupt routine. Defaults to the controller number of the

interrupting device, though it can be reset within the driver. It's often set by the driver *xx*attach routine to contain a local structure pointer.

mc_alive
Set to one by the autoconfiguration process if the controller is determined to be present. Otherwise left at 0.

mc_mbinfo
Main Bus resource allocation information (Used by MBI_ADDR, mbsetup and mbrelse).

mb_device "Free" devices (devices with no separate controllers) as well as "slave" devices, are represented to the kernel bus-management routines by an instance of the mb_device structure. (This is as it has been since 4.1BSD, but it's not ideal — if free devices were taken as controllers and represented by an mb_ctlr structure, then mb_device would only be for slave devices and would contain fewer fields). mb_ctlr contains all of the configuration-related data for the free or slave device. If a controller has multiple slave devices attached to it, there will be as many mb_device structures associated with its mb_ctlr structure. The following fields within mb_device (which are set by the configuration system and are not normally reset by the driver) are of interest:

md_driver
A pointer to the mb_driver structure associated with this device.

md_unit
The device index for the corresponding device, for example, the '0' in nd10. Used to index into arrays of driver-specific device status and control structures.

md_slave
The slave number of the device on its controller.

md_addr
The base address of the device (its control/status registers and perhaps some RAM). For most Multibus devices, this will be an address in I/O space, though for memory-mapped devices this will be an address in Memory space. For VMEbus machines, it's the particular address space within which the device is attached. Unused for devices on controllers.

md_intpri
The Main Bus priority level of the device (the priority that is passed to pritospl). Used to parameterize the setting of hardware priorities. Unused for devices on controllers.

md_intr
The interrupt vector (if vectored interrupts are used). Unused

for devices on controllers.

md_flags
The optional flags parameter from the system config file is copied to this field, to be interpreted by the driver. *Only the driver uses the information in this field.* If flags was not specified in the config file, then this field will contain a 0.

md_alive
Set by the autoconfiguration process to 1 if *xx*probe finds the device, otherwise it's left at 0. Incidently, if *xx*probe fails to find the device, the autoconfiguration process will also leave the device position in the *xx*dinit array (if the driver has one) at 0. The driver is free to test either variable (in its *xx*open routine) to determine *xx*probe's verdict.

mb_driver The system assumes that the source code of your driver declares a mb_driver structure named *xx*driver. This structure contains information relevant to the device driver *as a whole*, as opposed to information about individual devices or controllers. It differs in several important manners from the device and controller structures. For one thing, it contains a number of pointers to driver functions. These pointers, like those in cdevsw and bdevsw, are used by the kernel as entry points into the driver. For another, it's initialized not by the configuration system, but within the driver source code itself — if fact, several of the routines in *xx*driver are actually called by the kernel autoconfiguration process to complete the driver-related kernel initialization. *(Note: while the driver has responsibility for initializing the fields in xxdriver, it is still limited, at run time, to reading these fields — it cannot ever change them).*

*xx*driver must be known more intimately by the driver developer than either the driver md_ctlr structure or the driver md_device structure. We'll therefore give its complete definition:

```
struct mb_driver {
    int       (*mdr_probe)();            /* check device/controller installation */
    int       (*mdr_slave)();            /* check slave device installation */
    int       (*mdr_attach)();           /* boot-time device initialization */
    int       (*mdr_go)();               /* routine to start transfer */
    int       (*mdr_done)();             /* routine to finish transfer */
    int       (*mdr_intr)();             /* polling interrupt routine */
    int       mdr_size;                  /* amount of memory space needed */
    char      *mdr_dname;                /* name of a device */
    struct    mb_device **mdr_dinfo;     /* backpointers to mbdinit structs */
    char      *mdr_cname;                /* name of a controller */
    struct    mb_ctlr **mdr_cinfo;       /* backpointers to mbcinit structs */
    short     mdr_flags;                 /* want exclusive use of Main Bus */
    struct    mb_driver *mdr_link;       /* interrupt routine linked list */
};
```

Here is a brief discussion of the fields in the `mb_driver` structure that you'll need to initialize when declaring *xx*`driver`. Note that many of the fields in `mb_driver` are for the use of block drivers only — they're presented here as useful background information.

`mdr_probe`
is a pointer to the driver *xx*`probe` routine. *xx*`probe` is called for every controller and every independent device (with no separate controller) given in the kernel config file. *xx*`probe` determines if the device/controller is actually installed. If it is, it returns the amount of bus space consumed by the device/controller to the autoconfiguration process, where this space is then mapped into system address space. When *xx*`probe` fails, it returns 0.

`mdr_slave`
is a pointer to a *xx*`slave` function within your driver. *xx*`slave` is analogous to *xx*`probe`, and serves the same function for devices which are driven by separate controllers. Unlike *xx*`probe`, however, *xx*`slave` exists only for controllers that may have multiple devices — it's therefore quite rare in character device drivers.

`mdr_attach`
is a pointer to an *xx*`attach` function within your driver. *xx*`attach` is called during the autoconfiguration process, where it does preliminary setup and initialization for a device or controller. It's commonly used within disk and tape drivers to perform setup tasks like the reading of labels, and in character drivers for tasks like initializing interrupt vectors and reserving blocks of memory. Initialize this field only if there's an *xx*`attach` routine in your driver.

`mdr_go`
`mdr_done`
are pointers to *xx*`go` and *xx*`done` functions within the driver. These functions usually don't exist for character drivers, and these fields are consequently 0.

`mdr_intr`
is a pointer to a polling interrupt routine within your driver. Such a polling routine is used for the "auto-vectoring" of interrupts in systems where the interrupt "vector" can only be based on the interrupt priority. This is the case on all Multibus machines, and if there's any chance that your driver will someday be run on a Multibus machine you should include a polling interrupt routine and plug it in here.

If you have a Sun source license, and take the opportunity it affords to examine a number of drivers, you may note an inconsistency in the naming of interrupt routines:

- Some drivers have two interrupt routines: a polling interrupt routine named *xx*`poll` and a vector interrupt routine, named *xx*`intr`. In such cases *xx*`poll` determines the unit number of the interrupting device and then calls the *xx*`intr` to actually handle the interrupt.

- Other drivers have only one interrupt routine. The routine is named *xx*`intr` and called from `mdr_intr`, but it nevertheless contains polling code. This, like the naming of the field `mdr_intr` (which really should be `mdr_poll`) is an artifact of early Sun systems, in which drivers were written for the Multibus only — in these systems *xx*`intr` was *the* interrupt routine, and it always contained polling code.

In any case, remember that any routine called from `mdr_intr` must check the polling chain, regardless of its name. If you'll not support Multibus machines, and thus need no polling interrupt routine, put a zero in this field.

`mdr_size`
is the size — in bytes — of the memory required for the device. This field is initialized with a value identical to that which *xx*`probe` returns upon success. This is the amount of space that needs to be mapped into system memory by the autoconfiguration code.

`mdr_dname`
is the name of the device for which this driver is written. This field, is an array of pointers to `mb_device` structures, one for each of the installed devices. These pointers are filled in during autoconfiguration (see section below on *Autoconfiguration-Related Declarations)* and the driver is then free to use them to access the structures.

`mdr_dinfo`
is pointer to a pointer to a `mb_device` structure. This pointer is filled in during autoconfiguration (see section below on *Autoconfiguration-Related Declarations* and is necessary to work back from the device unit number to the correct `mb_device` structure by way of an index operation.

`mdr_cname`
is the name of a device supported by this driver (for example, *sc* supports the devices `sc0, sc1,` etc). This field takes the form of a regular null-terminated C string. Fill in this field if you actually have a controller.

`mdr_cinfo`
is pointer to a pointer to a `mb_ctlr` structure. This pointer is filled in during autoconfiguration (see section below on *Autoconfiguration-Related Declarations* and is necessary to work back from the device unit number to the correct `mb_ctlr` structure by way of an index operation.

`mdr_flags`
consists of some flags, as follows:

`MDR_XCLU`
The device needs exclusive use of Main Bus while running.

`MDR_BIODMA`
For block devices that do DMA on the Main Bus (the driver calls `mbgo`). The kernel needs this information in case it must lock other DMA devices off the bus.

`MDR_DMA`
For (character) devices which, while not transferring large block of data, do use DMA to transfer blocks of control information. Such drivers call

mbsetup.

MDR_SWAB

I/O buffers are to be swab'ed — that is, pairs of data bytes are to be exchanged. (This flag is used to push the swab out of mbgo and mbdone and down into the Main Bus driver).

MDR_OBIO

The device is installed in on-board I/O space.

Of these, MDR_XCLU, MDR_SWAB and MDR_OBIO are potentially to be used for user character devices. These flags must be OR'ed together if you wish to place any of that information there. Place a zero (0) in this field if none of the flags apply to your driver.

mdr_link

This field is used by the autoconfiguration routines and is not for the driver's use.

Autoconfiguration-Related Declarations

At the top of each driver, after the include statements, is a group of declarations that are used by the autoconfiguration process to finish the initialization of the *mbvar* structures. Here, as an example, are the relevant declarations from the Sky Floating-Point Driver:

```
/* Driver Declarations for Autoconfiguration   */
int skyprobe(), skyattach(), skyintr();
struct  mb_device *skyinfo[1];  /* Only Supports One Board */
struct  mb_driver skydriver = {
    skyprobe, 0, skyattach, 0, 0, skyintr,
    2 * SKYPGSIZE, "sky", skyinfo, 0, 0, 0,
};
```

The first line declares the names of the autoconfiguration-related entry point routines for the driver. In this case there are only three — skyprobe, skyattach and skyintr. These declarations are necessary because, in a few lines, we will use the names to initialize the driver's mb_driver structure.

The second line declares an array (in this case of dimension one) of pointers to mb_device structures. By the time the driver is linked into the kernel, config will have already declared an array of mb_device structures that contains an entry for each of the devices named in the kernel config file. When the kernel is booted, the autoconfiguration process initializes each driver's *xx*info array to indicate the mb_device structures corresponding to its devices, with each device's unit number being used as its subscript into the *xx*info array. The Sky driver is slightly atypical in that it only supports one device; normally the device count is provided by config in a macro "NXX" (which is set to the number of devices noted in the config file) would be the subscript in this declaration.

If this was a driver for a controller with slave devices, the second line would be followed by an analogous one that declared an array of pointers to mb_ctlr structures.

The third line both declares and initializes the mb_driver structure that represents this driver. The fields within the structure are described in detail in the previous section.

Other Kernel/Driver Interfaces

The kernel/driver interface is almost entirely contained within the *mbvar* structures and the parameter conventions of the driver routines. There are, however, a few other common kernel/driver interface points, which are given in this section.

The kernel `user` structure contains a few fields of interest to drivers. This structure, which maintains status information for the current user process (and which is swapped in and out with the process it describes), is used far less by Sun drivers than it is by System V drivers. This is because, in the Sun operating system, the `user` structure does not define the characters to be written (or the place for characters to be read to). The Sun kernel uses `uio` structures for this purpose, and passes them as parameters to the driver *xx*`read` and *xx*`write` routines. (See *Some Notes About the UIO Structure* in the *The ''Skeleton'' Character Device Driver* section of this manual).

Still, two fields within the `user` structure remain of interest to device drivers. They are:

`u.u_qsav`
Is a `setjmp` environment buffer that can be used by drivers that wish to save the current stack in preparation for a possible `longjmp` return. `setjmp` and `longjmp` are useful in drivers that need to intercept signals being sent to the process, as well as in error handling. For more information, see the setjmp(3) man page.

`u.u_error`
If an I/O operation is not successful, the driver must return an error code (defined in `<errno.h>`), which is plugged into `u.u_error`. From here it's normally stored in the per-process global variable `errno` in the user context.

Note that the `user` structure contains a pointer, `u.u_procp`, to the `proc` structure for the current process. The `proc` structure contains the information that the system needs about a process even when it is swapped out.

Drivers may occasionally need to know what kind of machine they're running on. They can find out by querying a variable, `cpu`, which, while not in the `user` structure, is available to them by including `../machine/cpu.h`. This variable is initialized by the kernel on the basis of information in the ID PROM, and is set to `CPU_SUN2_50`, `CPU_SUN2_120`, `CPU_SUN3_50`, `CPU_SUN3_110`, `CPU_SUN3_160` or `CPU_SUN3_260`. Note that when compiling for a Sun-2 system, only the Sun-2 names are available; likewise for a Sun-3.

DVMA drivers will often need to know the size of kernel DVMA space on the host machine (See *Sun Main-Bus DVMA)* so that they can subtract it from system virtual addresses to get addresses relative to the start of DVMA space. The external variable `DVMA`, declared as an array of characters, is available for this purpose.

The external variable `hz` gives the number of clock ticks per second on the host system.

Related to the `CPU_SUNX_XX` names are the `SUNX_XX` ifdefs. These are set at compile time on the basis of information in the config file, and can be used to eliminate code or data that is unnecessary for machines of any particular type. In general, it's possible (and advised) to write drivers that can compile and run on a variety of Sun machines with no changes.

4

Kernel Topics and Device Drivers

Kernel Topics and Device Drivers **53**

4.1. Overall Layout of a Character Device Driver 53

4.2. User Space versus Kernel Space 55

4.3. User Context and Interrupt Context 55

4.4. Device Interrupts 56

4.5. Interrupt Levels 57

4.6. Vectored Interrupts and Polling Interrupts 58

4.7. Some Common Service Routines 60

Timeout Mechanisms 61

Sleep and Wakeup Mechanism 61

Raising and Lowering Processor Priorities 62

Main Bus Resource Management Routines 62

Data-Transfer Functions 62

Kernel printf Function 63

Macros to Manipulate Device Numbers 63

4

Kernel Topics and Device Drivers

A first step in writing a device driver is deciding what sort of interface the device should provide to the system. The way in which `read` and `write` operations should occur, the kinds of control operations provided via `ioctl`, and whether the device can be mapped into the user's address space using the `mmap` system call, should be decided early in the process of designing the driver. (For simple memory devices that require neither DMA nor an `ioctl` routine, and that don't interrupt, it's possible to use the `mmap` system call to avoid writing a driver altogether. See the *Mapping Devices Without Device Drivers* section of this manual for more details).

Device drivers have access to the memory management and interrupt handling facilities of the UNIX system. The device driver is called each time the user program issues an `open`, `close read`, `write`, `mmap`, or `ioctl` system call, but only the *last* time the file is closed. The device driver can arrange for I/O to happen synchronously, or it can set things up so that I/O proceeds while the user process continues to run.

4.1. Overall Layout of a Character Device Driver

Here's a brief summary of the parts that comprise a typical device driver. In any given driver, some routines may be missing. In a complex driver, all of these routines may well be present. A typical device driver consists of a number of major sections, containing the routines introduced below.

Initial Declarations

Device drivers, like all C programs, begin with global declarations of various sorts. These declarations include the structures that the driver will overlay on the device registers. (These structures are often conveniently declared to contain unsigned integers and bit fields chosen to access various parts of the device registers). They also *must include* the declarations discussed in the *Autoconfiguration-Related Declarations* section of the *Overall Kernel Context* chapter of this manual.

Autoconfiguration Support

Then come the *xx*`probe`, *xx*`attach` and, perhaps, *xx*`slave` routines. These are called at kernel boot time to determine if devices noted as being present in the config file are actually installed, and to initialize them if they are. This initialization may include the resetting of the interrupt vector.

Opening and Closing the Device

xx`open` is called each time the device is opened at the user level; if multiple user processes open the device, *xx*`open` is called multiple times. *xx*`close`, in contrast, is called only when the *last* user process which is using the device closes it.

Reading to and Writing from the Device

xx`read` and *xx*`write` are called to get data from the device, or to send data to the device. Drivers for tty-like devices will probably structure *xx*`read` and *xx*`write` in the terminal-driver style (not described in this manual), while devices that deal simultaneously with groups of characters will probably have their *xx*`read` and *xx*`write` routines implemented in terms of a *xx*`strategy` routine. Such *xx*`strategy` routines are in ever way subsets of block-driver *xx*`strategy` routines — they are integrated with `physio` and they use `buf` structures but they don't have anything to do with the kernel buffer cache. Character drivers for DMA device are likely to have `strategy` routines, but they can be useful for non-DMA devices as well — as long as the devices do I/O in chunks.

Start Routine

xx`start` is needed in drivers that queue requests; it's called from *xx*`read`, *xx*`write` or *xx*`strategy` to start the queue and is also called from *xx*`intr` to send off the next request in the queue.

Mmap Routine

The `mmap` routine is present in drivers for devices which are operated by being mapped into user memory — for example, frame buffers.

Interrupt Routines

There are two kinds of interrupt routines: polling (or auto-vectored) routines and vectored routines. Polling routines are necessary when the host system doesn't allow unambiguous means of mapping hardware interrupts to devices, as is the case with Multibus-based machines. Vectored-interrupt routines are used on VMEbus-based systems, which can map hardware interrupts immediately to devices. Drivers for VMEbus devices that are never run on Multibus-based systems need only vector interrupt routines, while drivers for devices which will be run on both Multibus and VMEbus machines need both types of interrupt routines. In this case the polling routine can determine the interrupting device and then call the vectored routine to do the rest.

Ioctl Routine

The *xx*`ioctl` routine is called when the user process does an `ioctl` system call. These calls are the great escape hatches in the otherwise generally uniform UNIX I/O architecture. They are not, however, panaceas, and you should not overuse them to solve problems in driver design. Terminals have many *xx*`ioctl` calls, but they're a special case. They have many *xx*`ioctl` calls because they're inherently quite complex and yet UNIX still insists that they look like files.

4.2. User Space versus Kernel Space

The UNIX operating system, being a multi-tasking OS, provides for multiple threads of control *at the user level.* (These multiple threads are the various user processes). At the kernel level, however, things are different. The UNIX kernel is *monolithic monitor* type of operating system, and, as such, it cannot be interrupted by user processes. Instead, it contains code which allocates its time (and other resources) among the various user processes, as well as to itself. *The kernel can be interrupted by hardware, but when handling interrupts it doesn't run on behalf of any individual user process.*

Device driver functions are invoked by kernel routines after user processes make system calls. These functions must be able to move data to or from user virtual space quickly and easily. Kernel functions are provided to do help it do so, and to redundantly map memory so that it can be shared by user programs and the kernel.

Device drivers are parts of the kernel, and they inhabit kernel space:

- In the Sun-3 the kernel uses the top 16 Megabytes of the current 256 Megabyte context, starting at `0xF000000`.
- In the Sun-2 the kernel virtual address space is 16 Megabytes, and is completely separate from the individual user virtual address spaces.

In general, drivers don't need to consider the details of kernel address-space implementation. Routines (like `copyin` and `copyout`) which deal in multiple address spaces will manage the details internally, as will programs like `kadb`.

4.3. User Context and Interrupt Context

A device driver can usefully be thought of as having a *top half* and a *bottom half.* The top half, consisting of the `read`, `write`, and `ioctl` routines, and of any other routines which run on behalf of the user process making requests on the driver, is run at I/O request time. The routines in the top half make device requests that can cause long delays during which the system will schedule a new user process so that it can continue doing useful work. The bottom half, consisting of *xx*`intr` and any routines that it may call, is run at hardware interrupt time.

Memory-mapped device are usually not interrupt driven. Their drivers, thus, do not typically need to include interrupt interrupt routines. Memory-mapped devices operate by being written and read as system memory, and make no split-second demands (interrupt-time demands) upon their users.

After starting an I/O request, the top half calls `sleep` to wait for the *bottom half* to indicate (by way of a call to `wakeup`) that the request has been serviced. Thus, when a user program issues a read on (say) an A/D converter, it is normally suspended when the top half of the corresponding driver calls `sleep` to wait until some input arrives. Alternatively, the top half of the driver can call `iowait` and be put to sleep awaiting the completion of a buffer-oriented I/O transfer.

The top half contains not only all the non-interrupt time driver routines, but (for all practical purposes) the kernel routines above the driver as well. In particular, it contains the kernel `physio` routine, which manages the decomposition of large I/O requests into a series of smaller ones that can be handled by the device.

The *bottom half* may include a *xx*`start` routine, which can be called internally to start I/O. This allows the device-specific code to be isolated in one routine. *xx*`start` is not a driver entry point. It's called from either *xx*`strategy` or *xx*`intr`, depending on whether the device is busy or not.

Consider an A/D converter driver that expected the converter to interrupt when a sample was available. The kernel interrupt handler would detect the device interrupt and dispatch *xx*`intr`, which would then store the sample data in a buffer and `wakeup` the user process sleeping in the top half so it process could retrieve the data. If there was no user process sleeping in the top half, the `wakeup` would have no effect, but the next process to read the A/D driver would find the data already there and wouldn't have to `sleep`.

It must be stressed that, in general, *xx*`intr` doesn't run on behalf of the current user process — this is, in fact, why it's distinguished so clearly from the top half. This means is that no information about the current user process is available, in any way, to *xx*`intr`. It shouldn't examine, let alone change, any of the variables in the kernel `user` structure.

4.4. Device Interrupts

In general, the driver developer has limited control over the interrupt characteristics of the device. However, it should be said that some device-interrupt characteristics are better than others. In particular, interrupt-processing takes lots of time, and it's important that devices interrupt as seldom as possible. If, for example, a device can be made to handle multiple characters for each interrupt it processes, it should be. It's also preferable that a device not interrupt until its driver has enabled its interrupts, that it hold its interrupt line high until the driver asks that it be cleared, and that it remain quiescent after a bus reset (system boot).

Most hardware devices interrupt, and all interrupts occur at some given *priority level.* When an interrupt occurs, the system traps it, suspends the in-process operation (which may be a process entirely unrelated to the interrupting device or even the kernel) and resumes execution in the bottom half of the driver associated with the interrupting device. This means that the *top half* of a device driver can be interrupted *at any time* by its bottom half. If they wish to share data, they must do so in shared data structures, and they must take special provision to see that those structures remain consistent. An example of such a data structure is a pointer to a current buffer and a character counter. The top half of the driver must protect itself so that data structures can be updated as atomic actions, that is, the bottom half must not be allowed to interrupt during the time that the top half is updating some shared data structure. This protection is achieved by bracketing critical sections of code (sections that update or examine shared data structures) with subroutine calls that raise the processor priority to levels which can't be interrupted by the bottom half. Such a section of code looks like:

```
s = spln();
        . . .
         critical section of code that can't be interrupted
        . . .
(void)splx(s);
```

Here we've first raised the hardware priority level and then restored it after the protected section of code. (Determining the correct hardware priority will be discussed later). One section of code that almost always needs to be protected is the section where the top half checks to see if there is any data ready for it to read, or whether it can write data or start the device. Since the device can interrupt at any time, the section of code that checks for input in this fashion is wrong:

```
if (no input ready)
        sleep (awaiting input, software_priority)
```

because the device might well interrupt while the `if` condition is being tested, or while the preamble code for the `sleep` function is being executed.

The above section of code must be rewritten to look like this:

```
s = spln();
while (no input ready)
        sleep (awaiting input, software_priority)
(void)splx(s);
```

If the top half executes the `sleep` system call, the bottom half will be allowed to interrupt, because the hardware priority level is reset to 0 as soon as the `sleep` context switches away from this process.

4.5. Interrupt Levels

In many cases it is possible to set the device interrupt level by setting switches on the board. If so, you must decide what level this device is going to interrupt at. At first it may seem that your device is very high priority, but you must consider the consequences of locking out other devices:

- If you lock out the on-board UARTs (level 6) characters may be lost.
- If you lock out the clock (level 5) time will not be accurate, and the UNIX scheduler will be suspended.
- If you lock out the Ethernet (level 3), packets may be lost and retransmissions needed.
- If you lock out the disks (level 2), disk rotations may be missed.
- Level 1 is used for software interrupts and cannot be used for real devices.

In general, it's best to use the lowest level that will provide you with the response that you need.

4.6. Vectored Interrupts and Polling Interrupts

In Multibus-based Sun-2 machines, the kernel uses only auto-vectored (polling) interrupts. With auto-vectoring, the interrupt vector associated with a given device is based solely on the device interrupt priority level. Since many system configurations will contain more devices than there are interrupt levels, multiple devices may share the same interrupt level. Still, when processing an interrupt, the kernel must have a way of determining which device interrupted, and which driver should process the interrupt. In such configurations, the kernel proceeds by *polling* all the drivers at the given interrupt level (in the order that they are given in the config file), calling each of their polling interrupt routines in turn. These routines then proceed to interrogate their corresponding devices looking for the device that has an "attention bit" set, thus indicating that it issued the interrupt. Devices that don't indicate that they've interrupted can still be installed — one per system — by putting them at the end of the config file and thus at the end of the polling chain. Unclaimed interrupts can then be assumed to be from the last device.

After determining that one of its devices issued an interrupt, the polling routine services it and returns a non-zero to indicate that it did so (or a 0 to indicate that no device was found to originate the interrupt).

Polling only works if devices which share interrupt levels continue to interrupt until the driver tells them to stop. This is because the driver polling-interrupt routine returns to the kernel with an indication of which of the devices it has serviced. If two devices (A & B) are polling at the same interrupt level and *both issue an interrupt*, device A will always get serviced first. The kernel will then go on its merry way unless device B continues to interrupt. If it does, then when device A has been serviced, device B will be serviced. Fortunately, most Multibus boards continue to interrupt until told to stop. VMEbus boards typically do not, so it's important that they use vectored interrupts.

Sun VMEbus machines, (even those with Multibus devices installed by way of adapters) can take advantage of vectored interrupts. When handling a vectored interrupt, the kernel calls the appropriate driver's vector interrupt routine directly, passing it an argument to identify which of its devices (or controllers) interrupted.

It's important to realize that a driver can support both vectored interrupts *and* polling interrupts. Such a driver can be run on either type of machine, its polling interrupt routine will determine which device, if any, originated the interrupt, and then call the vectored interrupt routine to actually service it.

VMEbus devices — *if they interrupt* — are assigned unique identifying numbers in the range `0x40` to `0xFF` when they are described in the `config` file. It is these *vector numbers* that are used by the kernel to directly identify the interrupting device.

There are cases where no separate polling routine is needed. The first is where a driver *knows* that it supports only one device, and that no other device will share its device's interrupt level. In this case only a *xx*`intr` routine need exist. It can then be specified in `mb_driver->mdr_intr` for use in the auto-vectored case *and* in the `config` file for the vectored interrupt case. Thus, all configurations will use the same interrupt routine. *Remember, this will only work* *if there are no other devices of any sort installed at the same interrupt level.*

The other case where *xx*poll is not needed is when a driver will *never* support polling — presumably because it will never be run on a Multibus machine. In this case *xx*intr should be specified in the config file for use as the vectored interrupt routine, and the auto-vectored (polling) interrupt routine specified in mb_driver->mdr_intr should be 0.

Note that in the first case above, where the device will have an interrupt level to itself, little need be done to make the driver work with vectored interrupts. One may simply take a polling interrupt routine, (perhaps renaming it *xx*intr to avoid confusion) and install it as the vector interrupt routine by giving its name in the appropriate place in the config file. This isn't the most efficient thing to do, for when the routine is called through the kernel's vectoring mechanism, it will waste the information in its argument (which identifies the device originating the interrupt) and go on to poll its devices. Nevertheless it will work. It's better, however, if drivers contain both *xx*intr and *xx*poll routines, so that they may be easily transported to a variety of systems.

Another issue of concern only to drivers running on VMEbus machines is related to setting up the interrupt-vector number. When using the VMEbus-Multibus adapter or certain VMEbus devices, the vector number is set by switches on the circuit board. But some devices require that software initialize the device by telling it which vector number to use on interrupts. Presently, the only place where this can be done is in *xx*attach. The vector number that *xx*attach communicates to the device is in the md_intr->v_vec field of the mb_device structure — a NULL value in this field indicates that the host machine is Multibus based and does not support vectored interrupts.

A skeleton for a "typical" driver, one supporting both vectored and polling interrupts and using software to set interrupt vectors might look like:

```
/*
 * NXX is computed by config for each device type.
 * It can then be used within the driver source code to
 * declare arrays of device specific data structures.
 */

struct xx_device xxdevice[NXX];

/*
 * Attach routine for a device xx that must be notified of its
 * interrupt vector.
 */

xxattach(md)
    struct mb_device *md;
{
    register struct xx_device *xx = &xxdevice[md->md_unit];

/*
 * Vector number given in kernel config file and passed by the
 * autoconfiguration process during boot.
*/
    if (md->md_intr) {

        /* so we'll be using vectored interrupts */

        /* WRITE interrupt number TO THE DEVICE */
        xx->c_addr->intvec = md->md_intr->v_vec;

        /* Setup argument to be passed to xxattach */
        *(md->md_intr->v_vptr) = (int)xx;

    } else { /* WRITE auto-vector code TO THE DEVICE */
        xx->c_addr->intvec = AUTOBASE + md->md_intpri;
    }
    /* any other attach code */
}

/*
 * Handle interrupt - called from xxpoll and for vectored interrupts.
 */
xxintr(xx)
    struct xx_device *xx;
{
    /* handle the interrupt here */
}

/*
 * Polling (auto-vectored) interrupt routine
 */
xxpoll()
{
    register struct xx_device *xx;
    int serviced = 0;

    /* loop through the device descriptor array */
    for (xx = xxdevice; xx < &xxdevice[NXX]; xx++) {
        if (!xx->c_present ||
            (xx->c_iobp->status & XX_INTR) == 0)
            continue;
        serviced = 1;
        xxintr(xx);
    }
    return (serviced);
}
```

4.7. Some Common Service Routines

The kernel provides numerous service routines that device drivers can take advantage of. The most important of these routines can be clustered into the functional groups given here. These routines, as well as many others, are described more completely in the *Kernel Support Routines* appendix to this manual:

Timeout Mechanisms

If a device needs to know about real-time intervals,

```
timeout(func, arg, interval)
    int (*func)();
    caddr_t arg;
    int interval;
```

is useful. `timeout` arranges that after *interval* clock-ticks, the *func* is called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read a device if there is no response within a specified number of seconds. Also, the specified *func* is called at "software" interrupt priority from the lower half of the clock routine, so it should conform to the requirements of interrupt routines in general — you can't, for example, call `sleep` from within *func*, although you can call `wakeup`. (See also `untimeout`).

Sleep and Wakeup Mechanism

Another key set of kernel routines is `sleep` and `wakeup`. The call

```
sleep(event, software_priority)
    caddr_t address;
    int priority;
```

makes the process wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run. When the process resumes execution, it has the priority specified by *software_priority*.

The call

```
wakeup(event)
    caddr_t address;
```

indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker — it must uniquely identify the device. By convention, *event* is the address of some data area used by the driver (or by a specific minor device if there's more than one).

Processes sleeping on an event should not assume that the event has really happened when they are awakened, for `wakeup` wakes *all* processes which are asleep waiting for the *event* to happen. Processes which are awakened should check that the conditions that caused them to go to sleep are no longer true.

Software priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than `PZERO` and those at numerically larger priorities. The former cannot be interrupted by signals. Thus it is a bad idea to sleep with priority less than `PZERO` on an event that might never occur. On the other hand, calls to `sleep` with larger priority may never return if the process is terminated by some signal in the meantime. In general, sleeps at less than `PZERO` should only be waiting for fast events like disk and tape I/O completion. Waiting for human activities like typing characters should be done at priorities greater than `PZERO`. Incidentally, it is a gross error to call `sleep` in a routine called at interrupt time, since the process that is running is almost certainly not the process that should go to sleep.

Raising and Lowering Processor Priorities

At certain places in a device driver it is necessary to raise the processor priority so that a section of critical code cannot be interrupted, for example, while adding or removing entries from a queue, or modifying a data structure common to both halves of a driver.

The splx function changes the interrupt priority to a specified level, and then returns the old value.

For configuration reasons, the pritospl macro is necessary to convert a Main Bus priority level to a processor priority level. The Main Bus priority level can be found in either md->md_intpri or mc->mc_intpri, where it is put by the autoconfiguration process.

Here's how you normally use the pritospl and splx functions in a hypothetical strategy routine:

```
hypo_strategy(bp)
    register struct buf *bp;
{
    register struct mb_ctlr *mc = hypoinfo[minor(bp->b_dev)];
    int s;

    s = splx(pritospl(mc->mc_intpri));
    while (bp->b_flags & B_BUSY)
        sleep((caddr_t)bp, PRIBIO);
    . . .
        here is some critical code section
    . . .
     (void)splx(s);        /* Set priority to what it was previously */
    . . .
}
```

Alternatively, spl*n* can be used to set the processor to a certain fixed priority level.

Main Bus Resource Management Routines

The routine mbsetup is called when the device driver wants to start up a DMA transfer to the device, for DMA transfers require Main Bus resources. The MBI_ADDR macro can then be used to get the DVMA transfer address.

At some later time, when the transfer is complete, the device driver calls the mbrelse routine to inform the Main Bus resource manager that the transfer is complete and the resources are no longer required.

Data-Transfer Functions

The kernel provides a number of routines designed to transfer data between the user and kernel address spaces. These include copyin and copyout, general routines designed to move blocks of bytes back and forth. They also include uiomove, ureadc and uwritec, routines which are designed to transfer data to or from a uio structure (see *Some Notes About the UIO Structure* for more details about this structure).

Kernel printf Function

The kernel provides a `printf` function analogous to the `printf` function supplied by the C library for user programs. The kernel `printf`, however, is more limited. It writes directly to the console, and it doesn't support `printf`'s full set of formatting conversions. See the *Debugging with* **`printf`** section of this manual for more details on the use of the kernel `printf`.

Macros to Manipulate Device Numbers

A device number (in this system) is a 16-bit number `(typedef short dev_t)` divided into two parts called the *major* device number and the *minor* device number. There are macros provided for the purpose of isolating the major and minor numbers from the whole device number. The macro

```
major(dev)
```

returns the major portion of the device number *dev*, and the macro

```
minor(dev)
```

returns the minor portion of the device number. Finally, given a major and a minor number *x* and *y*, the macro

```
dev_t makedev(x,y)
```

returns a device number constructed from its two arguments.

5

Driver Development Topics

Driver Development Topics **67**

5.1. Installing and Checking the Device 67

Setting the Memory Management Unit 67

Selecting a Virtual Address 68

Finding a Physical Address 70

Selecting a Virtual to Physical Mapping 70

Sun-2 Address Mapping 72

Sun-3 Address Mapping 75

A Few Example PTE Calculations 77

Getting the Device Working and in a Known State 78

A Warning about Monitor Usage 79

5.2. Installation Options for Memory-Mapped Devices 80

Memory-Mapped Device Drivers 80

Mapping Devices Without Device Drivers 81

Direct Opening of Memory Devices 85

5.3. Debugging Techniques 86

Debugging with **printf** 87

Event-Triggered Printing 89

Asynchronous Tracing 90

kadb — A Kernel Debugger 91

5.4. Device Driver Error Handling 92

Error-Handling Mechanisms 92

Error Recovery 92

Error Returns 93

Error Signals 93

Error Logging 93

Kernel Panics 93

5.5. System Upgrades 94

5

Driver Development Topics

5.1. Installing and Checking the Device

The central processor board (CPU) of the Sun Workstation has a set of PROMs containing a program generally known as the "Monitor". (See the *Using the Sun CPU PROM Monitor* appendix for detailed descriptions of the monitor commands and their syntax). The monitor has three basic purposes:

1) To bring the machine up from power on, or from a hard reset (monitor 'k2' command).

2) To provide an interactive tool for examining and setting memory, device registers, page tables and segment tables.

3) To boot UNIX, stand-alone programs, or the kernel debugger `kadb`.

If you simply power up your computer and attempt to use its monitor to examine your device's registers, you will likely fail. This is because, while you may have correctly installed your device (a process that includes specifying its virtual memory mapping in the config file) those mappings are UNIX specific, and don't become active until UNIX is booted. The PROM will, upon power up, map in a set of essential system devices — like the keyboard — but your device is almost certainly not among them.

When installing a new device, you'll use the monitor primarily as a means of examining and setting device registers. But before even beginning the development of your driver, it's a good idea to attach your device to the system bus and use the monitor to manually probe and test it. This will give you a chance to become familiar with the details of its operation, and to ensure that it works as you expect it to.

Setting the Memory Management Unit

Upon power-up, the PROM monitor:

□ Maps the beginning of on-board memory, up to 6 Megabytes, to low virtual addresses starting at virtual `0x0`.

□ *Sun-2 machines only.* Maps the bus spaces into virtual address space, for the purpose of supporting Multibus devices. Multibus IO space is mapped from `0xEB0000` to `0xEBFFFF` on Sun-2 Multibus machines. On Sun-2 VMEbus machines, `vme16d16` is mapped from `0xEB0000` to `0xEBFFFF` so that Multibus cards attached by way of VMEbus adapter cards can be accessed. These two address spaces, Multibus I/O and `vme16d16`, are *not* remapped by the UNIX kernel. This means that, for example, that kernel virtual address `0xEBEE40` can be used to talk to a device at `0xEE40` in Multibus IO space without setting up a mapping. (This shortcut is *only* possible for the two 16-bit Sun-2 spaces).

Later, using the autoconfiguration process, UNIX makes a pass through the config file (actually, through the `ioconf` file that was produced as output by `config` when it processed the config file. For each device, UNIX selects an unused virtual address (using an algorithm that doesn't presently concern us) and maps it into the device's physical address as specified in the config file.

UNIX then calls the *xx*`probe` routine for each device, passing it the chosen virtual address. In this way, *xx*`probe` is kept from having any knowledge of the physical address to which the device is mapped. *xx*`probe` then determines whether or not the device is present. If it isn't, the virtual address can be reused.

To test a device, ignore the UNIX mappings and use the monitor to manually set the MMU to map your device registers to a known address in physical memory. Then you can use the monitor to verify its proper operation. This verification process will consist primarily of using the monitor's `'O'` (open a byte), `'E'` (open a word) and `'L'` (open a long word) commands to examine and modify the device's registers.

The process of setting up the device for initial testing consists of three discrete steps.

□ The selection of an appropriate virtual address for the testing of the device.

□ The determination of the physical address of the device, as well as the address space that it occupies.

□ The use of the monitor to map the system's virtual address to the device's physical address. Detailed discussion of these three steps follow.

Since UNIX initializes the MMU in the course of its autoconfiguration process, it's possible to test a device by actually installing it, and then booting and halting UNIX. (You can halt UNIX by pressing the 'L1' and 'A' keys simultaneously, or, on a terminal console, by hitting the <BREAK> key). Having gotten to the monitor by this route, the MMU will be initialized to its UNIX run-time state. You can then use the monitor to test the device, or, if you wish, boot `kadb`. *(A hard reset — the monitor's 'k2' command — will set the to MMU to its pre- UNIX power-up state). But while using the UNIX memory maps may occasionally be useful, it's not what you want to do during the first stages of device integration.*

Selecting a Virtual Address

First, it's necessary to understand that the MMU, when mapping a virtual address to a physical address, is actually mapping to a page of physical memory and an offset within that page. The low-order bits of a virtual address, those that specify the offset, *do not get mapped* — an address that is X bytes from the beginning of its virtual page will be X bytes from the beginning of whatever physical page it gets mapped into.

The mapping mechanism is the essentially the same for Sun-2 and Sun-3 systems, although the details of address size and page mapping differ. This can be seen in the following two diagrams:

Figure 5-1 *Sun-2 Address Mapping*

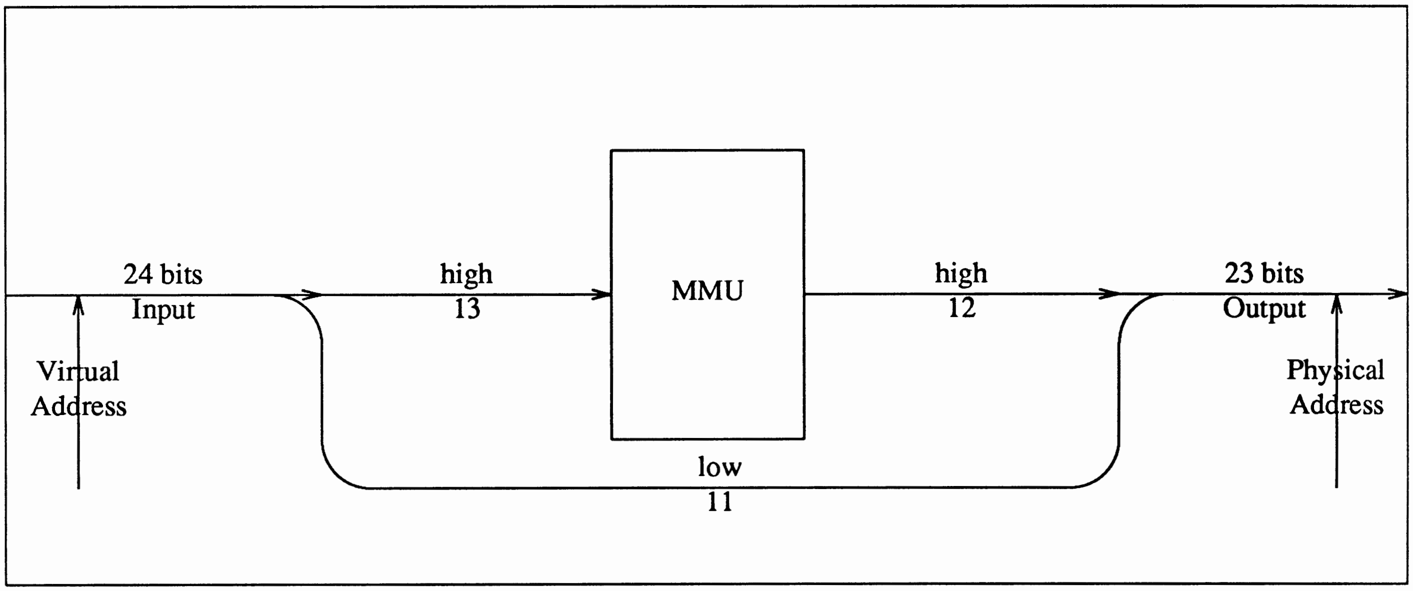
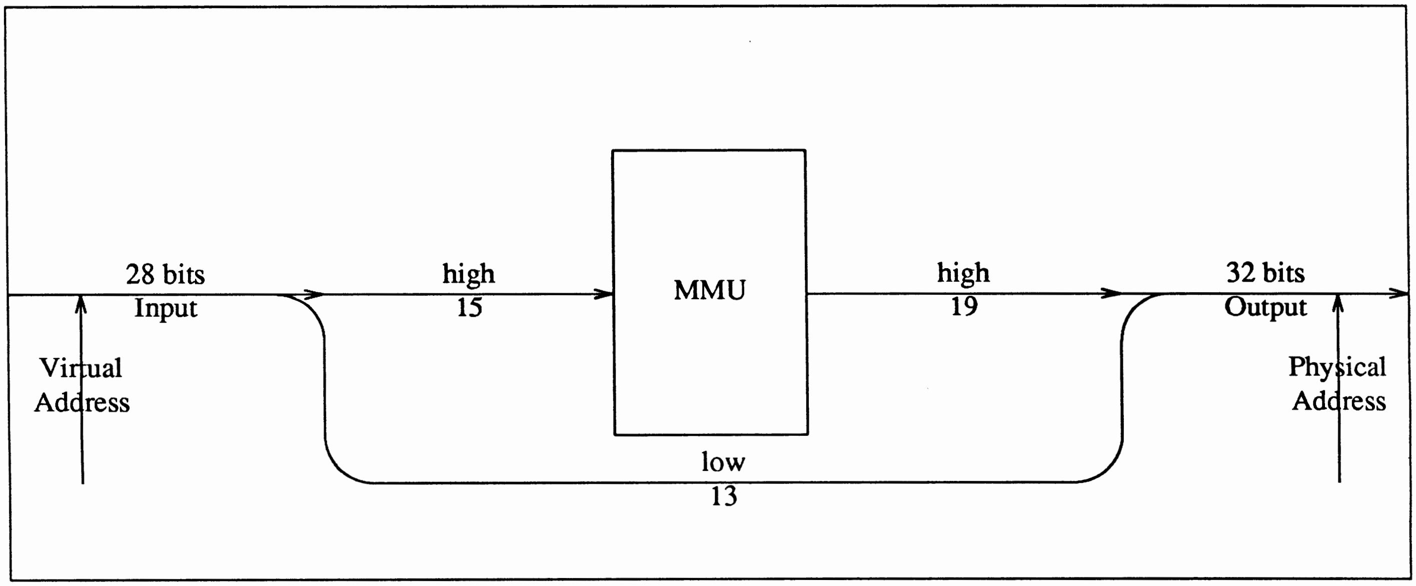


Figure 5-2 *Sun-3 Address Mapping*



The easiest way to select a virtual address for testing is to use `0xE00000` (Sun-2) or `0xE000000` (Sun-3). These addresses are unused by the monitor in both the Sun-2 and the Sun-3, and are thus always available for testing. Furthermore, if you use them, or another address in the same Megabyte of memory, then it won't be necessary to set the *segment register*. Since doing so is a fairly involved process, and since there's no need to go through it, we will skip it and assume that you use these addresses. (These addresses, while convenient for testing, are not those that the kernel will chose when your device is finally installed).

In selecting `0xE00000` (or `0xE000000`) as your virtual address, you're only selecting a virtual page, that is, a page of virtual memory that contains a page of physical memory. The low-order bits in the address you chose will remain unchanged. With `'X'` representing the unmapped low-order bits, then the test address `0xE00000` (Sun-2) or `0xE000000` (Sun-3), is really (in binary):

```
Sun-2: 1110 0000 0000 XXX XXXX XXXX  (23 bits)
Sun-3: 1110 0000 0000 000X XXXX XXXX XXXX (28 bits)
```

Finding a Physical Address

Your board may be preconfigured to some address. If it is, then use that address unless it conflicts with the address of an already installed device. If it does, you'll have to find an unused physical address at which you can install your device. To do so, examine the kernel config file for the system upon which you are working. Tables earlier in this chapter show memory layouts corresponding to typical configurations, but if your system has departed at all from the norm, you'll have to consult your kernel's config file (to determine where devices have been installed) and the header files for the corresponding device drivers (to determine how much space they consume on the bus).

Selecting a Virtual to Physical Mapping

When selecting a virtual to physical mapping, it's best if you understand a bit about the internals of the Memory Management Unit. To this point we've only stressed that the MMU maps the top bits of the virtual address, leaving the offset bits unchanged. Now it will be necessary to explain the mapping process in more detail.

Some new concepts are necessary to discuss the details of virtual to physical memory mapping.

- The *context register (of real significance only on the Sun-2*) is a three-bit register specifying which of eight memory *contexts* should be used when mapping virtual addresses to physical addresses. Each UNIX *process segment* (containing either code, data or stack) is kept within a single memory context.
    - Sun-3s have user and kernel address spaces in the same hardware context. That is to say, there is only one virtual address space, a portion of which is used by the kernel and the rest by user processes.
    - Sun-2s, on the other hand, segregate kernel and user processes into separate hardware contexts with separate address maps. Kernel processes are run in the supervisor context (context 0) and only processes in context 0 have access to the I/O devices.
- The *segment map* is used in conjunction with the *context register* to select the *page map entry group* (PMEG) corresponding to the virtual address being mapped. The eight bits in the segment register specify one of a group of 256 PMEGs.

□ Within each *page map entry group* there are 16 *page table entries.*

□ The *page map* maps the PMEG returned from the segment mapping with a second subfield of the incoming virtual address to exactly specify a single *page table entry* describing the physical page within which the virtual address is mapped.

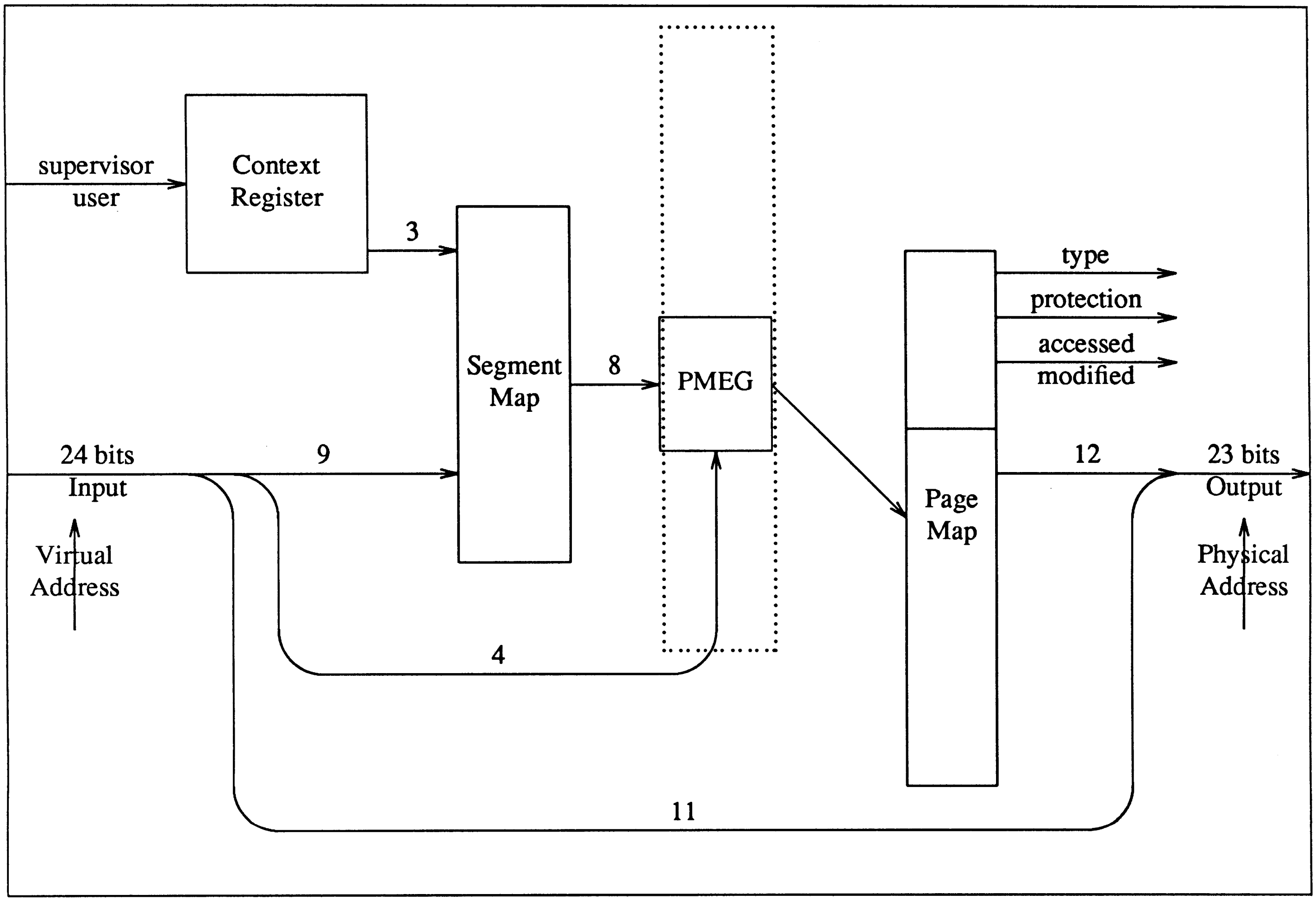
□ The *page table entry* (PTE) is the content of the page map location selected by the previous mappings. It is the number that, when plugged into the MMU's internal RAM, causes the MMU to select a given physical page. A PTE specifies the physical address of a page, as well as its type (e.g., on-board memory space), protection, and the state of its *access* and *modified* flags.

Note (for Sun-2 machines only): when testing your device, it's necessary to ensure both that you are in supervisor state and that you are in context zero (the kernel context). The monitor normally initializes to supervisor state, but if you enter it by way of an abort from UNIX, you'll remain in whatever context you were in at the time of the abort. To be on the safe side, begin all of your monitor sessions with the command 'S5'. This will put you into supervisor data state and context zero, just where you want to be. Note one important exception to this rule: if you've mmap*'ed the device into your (user) program's address space and want to check that this worked, you must use the 'S1' command instead of the 'S5' command. This will put you into context zero in user data state.*

Sun-2 Address Mapping

Note the following diagram of the Sun-2 MMU:

Figure 5-3 *Sun-2 MMU*



Note that:

- The lower 11 bits of the incoming virtual address are passed through the MMU without being mapped — these are the bits that specify the position within the page, regardless of whether that page is physical or virtual.
- Multiple segment maps can specify the same PMEG, and often do.
- The PTE, on the output side of the MMU, specifies a variety of kinds of status information for the specified page, as well as the top bits of its physical address.

The process of mapping a virtual to a physical address consists, in practice, of plugging the right number into the right PTE. The monitor provides a simple means of addressing the right PTE, but you will have to calculate the right value to plug into it.

On Sun-2 systems, hardware PTEs are 32-bit numbers with the following structure (the UNIX PTE used by software is different on the Sun-2):

V	r w x r w x	Type	a m	Unused (8)	Physical Page # (12)

Most of the PTEs that we'll deal with will have similar structures, and so we can begin by making a "template" bit mask that we can use to construct our standard PTEs. One acceptable mask will assume values as follows:

```
V (valid) = 1
rwxrwx = 111111
(a/m) accessed/modified = 00
unused = 00000000
```

Thus, we can see that our template will be:

1	1 1 1 1 1 1	Type	1 1	0 0 0 0 0 0 0 0	Physical Page # (12)

This gives us a mask of `0xFE000000` (if we assume that the type field is `0000`). Now, as already mentioned, there are four types of memory, represented in the PTE by values of 0, 1, 2 and 3 in the type field indicated above. (Types 0 and 1 have the same meaning in both Multibus and VMEbus machines, but types 2 and 3 do not. Type 2 is used, on Sun-2 VMEbus machines, to designate the first 8 Megabytes of the 24-bit VMEbus space — `0x0` to `0x7FFFFF` — and type 3 is used to designate the second 8 Megabytes — `0x800000` to `0xFFFFFF`. (But remember that the top 64K of the 24-bit space is stolen for the 16-bits space). This use of two memory types to designate physical memory is necessary because the Sun-2 physical address size, 23 bits, is not sufficient to address all 16 Megabytes of `vme24d16`.

Table 5-1 *Sun-2 PTE Masks*

Type	Description	Mask
0	On Board Memory	0xFE000000
1	On Board I/O Space	0xFE400000
2	(Multibus) Memory Space	0xFE800000
3	(Multibus) I/O Space	0xFEC00000
2	(VMEbus) VMEbus Low	0xFE800000
3	(VMEbus) VMEbus High	0xFEC00000

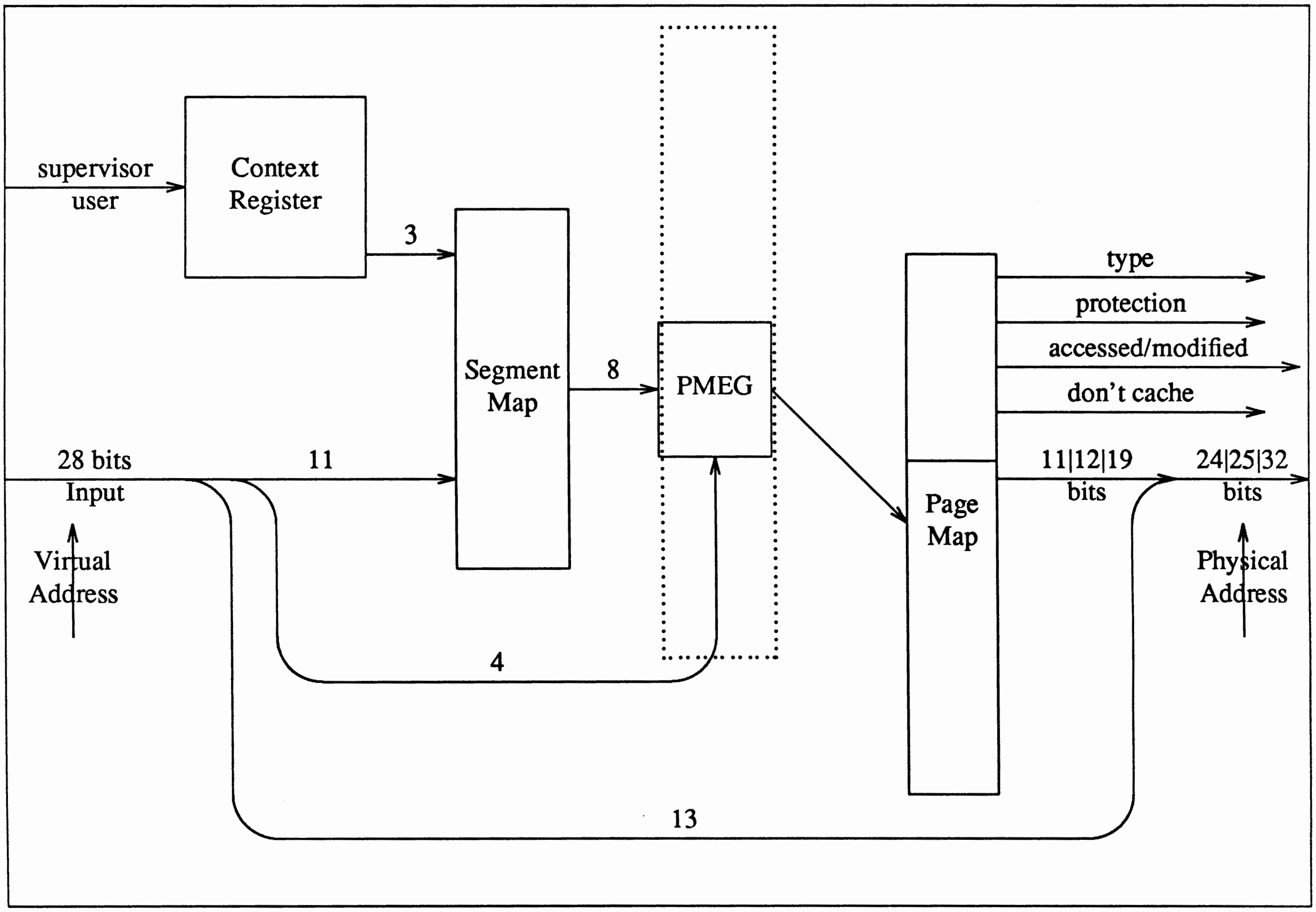
To determine the value which we need to plug into the PTE, we must add the appropriate mask to the appropriate physical page number, thus giving us the full 32-bit number that we need. Here, we will cease to explain details and simply give a series of rules for calculating physical page numbers.

```
If Sun-2 Multibus:

    If Multibus I/O Space, use Type-3 Template
    If Multibus Memory Space, use Type-2 Template

    Physical Page Number = Physical Address >> 11

If Sun-2 vme24d16:

    If Physical Address >= 0x800000
        Use Type-3 Template
        Physical Page Number =
            (Physical Address - 0x800000) >> 11

    If Physical Address < 0x800000
        Use Type-2 Template
        Physical Page Number = Physical Address >> 11

If Sun-2 vme16d16

    Use Type-3 Template
    Physical Page Number =
        (Physical Address + 0x7F0000) >> 11
```

Sun-3 Address Mapping

Consider the following diagram of address mapping on the Sun-3.

Figure 5-4 *Sun-3 MMU*



As you can see, the general scheme is the same as it was in the Sun-2, but the details have changed:

- The MMU is getting a 28-bit virtual address as its input, as opposed to a 24-bit address in the Sun-2.
- The number of mode and permission bits in the PTE has been reduced.
- The number of high-order bits reported out of the MMU, and thus the size of the physical address, is variable. The address size is fixed for any given Sun-3 machine, and varies only with the model — there are different kinds of Sun-3 machines and they have different physical address sizes.

On Sun-3 systems, PTEs are 32-bit numbers with the following structure.

V	w s c	Type	a m	Unused (5)	Physical Page Number (19)

as we did with Sun-2 PTEs, we will make a "template" bit mask that we can use to construct our standard PTEs. One acceptable mask assumes values as follows:

```
V (valid) = 1
w/s (write ok/supervisor only) = 11
c (cache/don't cache) = 1
unused = 00000
```

Thus, we can see that our template will be:

1	1 1 1	Type	0 0	0 0 0 0 0	Physical Page Number (19)

This gives us a mask of `0xF0000000` (if we assume that the type field is `00`). Thus, the four masks for the four types of memory are:

Table 5-2 *Sun-3 PTE Masks*

Type	Description	Mask
0	On Board Memory	0xF0000000
1	On Board I/O Space	0xF4000000
2	`vme16d16`	0xF8000000
2	`vme24d16`	0xF8000000
2	`vme32d16`	0xF8000000
3	`vme16d32`	0xFC000000
3	`vme24d32`	0xFC000000
3	`vme32d32`	0xFC000000

To determine the value to be plugged into the PTE, we must add the appropriate mask to the appropriate physical page number, thus giving us the full 32-bit number that we need. Here, again, we'll give rules instead of details.

```
If vme16d16
   or vme24d16
   or vme32d16

    Use Type-2 Template

If vme16d32
   or vme24d32
   or vme32d32
```

```
    Use Type-3 Template

If vme32d16
   or vme32d32

    Physical Page Number = Physical Address >> 13

If vme24d16
   or vme24d32

    Physical Page Number =
        (Physical Address +0xFF000000) >> 13

If vme16d16
   or vme16d32

    Physical Page Number =
        (Physical Address +0xFFFF0000) >> 13
```

A Few Example PTE Calculations

Example One: You wish to map a device which you have attached at physical 0x280008 onto bus type vme24d16 on a Sun-3. You will map it at virtual 0xE000000. What is the corresponding PTE?

Well, since we are mapping the device into vme24d16, we will use 0xF8000000 as the template. Then, following the Sun-3 rules, as given above, we add the physical address to 0xFF000000. This yields 0xFF280008. In binary, this is:

```
1111 1111 0010 1000 0000 0000 0000 1000
```

Shifting this right by 13 yields:

```
XXXX XXXX XXXX X111 1111 1001 0100 0000
```

Adding the template, 0xF8000000, we get values for the 13 bits that are undefined from the shift. Thus the PTE is:

```
1111 0100 0000 0111 1111 1001 0100 0000
```

Which is 0xF407F940.

A final note: we've now calculated the PTE that maps the virtual page beginning at 0xE000000 to the physical page containing 0x280008. To get the virtual address by which to access the device it's necessary to take the lower 13 bits of the physical installation address — the bits that are just passed through the MMU — and add them to virtual 0xE000000. The lower 13 bits of physical 0x280008 are 0008, and adding them to 0xE000000 yields 0xE000008, the virtual address by which the device can be accessed.

Example Two: You wish to map physical 0xEE48 on bus type vme16d32 on a Sun-3. Using virtual address 0xE000000, what is the PTE?

Since we are mapping the device into `vme16d32`, we will use `0xFC000000` as the template. Then, following the Sun-3 rules, as given above, we add the physical address to `0xFFFF0000`. This yields `0xFFFFEE48`. In binary, this is:

```
1111 1111 1111 1111 1110 1110 0100 1000
```

Shifting this right by 13 yields:

```
XXXX XXXX XXXX X111 1111 1111 1111 1111
```

Adding the template, `0xFC000000`, we get values for the 13 bits that are undefined from the shift. Thus the PTE is:

```
1111 1100 0000 0111 1111 1111 1111 1111
```

Which is `0xFC07FFFF`. To get the virtual address by which to access the device at physical `0xEE48`, add its lower 13 bits, `0xE48`, to `0xE000000` — this yields `0xE000E48`.

Getting the Device Working and in a Known State

Before you even *think* about writing any code you should check out your device. You must get to know it, finding out early if it has any peculiarities that'll affect its driver. It may, for example, have addressing and data-bandwidth limitations. Or, if it's a bus master, it may not implement the *release on request* bus-arbitration scheme the Sun supports. *Know the peculiarities of your device early, and then test it to verify that it's working before proceeding further with driver development.*

Make sure that the board is set up as specified in the vendor's manual. Device characteristics which, in general, have to be set properly before the device can successfully be used include:

- I/O register addresses for I/O mapped Multibus boards,
- Memory base addresses for Multibus boards that use Multibus memory space,
- Address and data widths,
- Interrupt levels,
- Interrupt vector numbers for VMEbus device.

Then, take down your system and power it off. Plug the device into the card cage and attempt to bring the system back up. If you can't boot the system, then there's a problem. Perhaps the board isn't really working, or perhaps it's responding to addresses used by other system devices. You must resolve this problem before proceeding further.

Take down UNIX again and attempt to contact the device using the PROM monitor. To do so, you'll need to set up a PTE mapping to device physical installation address. Use the procedures given above to calculate a PTE, then:

- Select the function code that will put you into *supervisor data* state. To do so, give the monitor's 's5' command:

```
>s5
```

- Calculate, using the procedures given above, the PTE appropriate to the physical address you've chosen.
- Set the position in the kernel page map that corresponds to your physical address to contain the calculated PTE. This will map your chosen physical address, thus putting you in contact with your device. You may then use the monitor's 'P' command to perform this mapping. The 'P' command takes a virtual address as its argument, displays the PTE that corresponds to that virtual address, and gives you the option of modifying the PTE. For example:

  ```
  >pF32000
  ```

  selects the page map entry that corresponds to the virtual address of `0xF32000` and displays it. It also displays a '?', which indicates that you may type in a new value to replace the one displayed. (See the *Using the Sun CPU PROM Monitor* appendix for more details). Note that all virtual addresses within a page select the same PTE.

Having contacted the device from the monitor, try some of the following:

- Try reading from the device status register(s), if there are any.
- Try writing to the device control and data registers(s), if there are any. Then try reading the data back to see if it got written properly (this assumes, of course, that the device allows the reading of these register(s).
- Try actually getting the device to do something by sending it data.
- If the device is a controller with separate slave devices, then switch a slave on and off and watch for changes in the controller status bits.

Your goal is to try to actually operate the device, for a moment, from the monitor. For example, if you have a line printer, try to print a line with a few characters. Be aware that bit and byte ordering issues are critical in this process. The reason you're doing this is to ensure that the device works and that you understand the way it works. When you understand the device's peculiarities, you can proceed to write a driver for it.

A Warning about Monitor Usage

When you use the monitor's 'o', 'e' or 'l' commands to open a location, the monitor *reads* the present contents of that location and displays them before giving you the option to rewrite them. In the best of all possible worlds, this would present no problems, *but many devices don't respond to reads and writes in as straightforward a fashion as does normal memory.*

For example, the Intel 8251A and the Signetics 2651 ues the same externally addressable register to access *two* separate internal mode registers, and they have internal state logic that alternates accesses to the external register between the two internal registers. So suppose that you want to put something in mode register 1 of the 8251. You open the external register, the monitor displays its contents, and you then do your write. If, being cautious, you then read the external register to check that the data you wrote is there, you'll find that it's not —

because the read will sequence you on to the second register.

To deal correctly with such devices, it's necessary to use the monitor's "write without looking" facility and then read the locations back later to check them. You can write without looking with any of the monitor commands that "open" an area of memory; all that's necessary is that you enter a `value` after the `address` argument. For example:

```
>l [address] [value]
```

This will cause `value` to be written into `address` without first reading its current contents. For more information on hardware peculiarities and the problems that they can cause for the monitor, the *Hardware Peculiarities to Watch Out For* section of the *Hardware Context* chapter.

5.2. Installation Options for Memory-Mapped Devices

Memory-Mapped Device Drivers

Memory-mapped devices are the simplest types of devices to write drivers for. Frequently, however, their essential simplicity isn't obvious from a quick glance at their source code. This is because many memory-mapped devices are frame buffers, and frame-buffer drivers must set up and manage the low-level interface for the Sun window system as well as the standard device interface. Consequently, they tend to be littered with declarations and manipulations related to the "pixrect" (pixel rectangle) system.

Memory-mapped device are most frequently installed into Sun systems with simple drivers that map them into user address space (there are sometimes alternatives to such drivers, as you'll see below). Such memory-mapped drivers don't really do much. Obviously, *xx*`probe` and `mmap` must be real, for the kernel must be able to check the device installation and perform the actual device mapping. And, in addition, *xx*`intr` must be real if the device is interrupt driven. But *xx*`open` and *xx*`close` are usually stubs, and *xx*`read` and *xx*`write` can be calls to `nulldev`.

Keep in mind that the major purpose of a memory-mapped driver is to support the `mmap` system call. This is very important because user processes which call window code must first map the frame buffer into their address space. They do so with the `mmap` system call, which is translated by the kernel into a series of calls to the driver's `mmap` routine. Each of these calls returns page table entry information which the kernel needs to map a single page (the next page) of frame-buffer memory into a virtual address space. Here's the *xx*`mmap` *code for the very simple case of the Sun-1 Color Graphics Board.*

```
/*ARGSUSED*/
cgonemmap(dev,off,prot)
    dev_t dev;
    off_t off;
    int prot;
{
    return (fbmmap(dev,off,prot,NCGONE,cgoneinfo,CG1SIZE));
}

/*ARGSUSED*/
int fbmmap(dev,off,prot,numdevs,mb_devs,size)
    dev_t dev;
    off_t off;
    int prot;
    int numdevs;
    struct mb_device **mb_devs;
    int size;
{
    struct mb_device *mb_dev = *(mb_devs+minor(dev));
    register int page;

    if (off >= size) return (-1);
    page = getkpgmap(mb_dev->md_addr + off) & PG_PFNUM;
    return (page);
}
```

dev is, of course, the device major and minor number, and *off* is the offset into the frame buffer (passed down from the user's mmap system call). *prot* is also passed down from the user's call, but it is not currently used. As you can see, there's bit of shuffling around and then a call to getkpgmap. The PTE returned by getkpgmap is masked with PG_PFNUM to get rid of extraneous bits and just leave the Page Frame Number and type fields, which is what *xx*mmap is expected to return.

The utility routine gets the address of the frame buffer from the Main Bus device structure. This is the device installation address as given in the kernel config file. Next the offset is checked to be sure the user isn't mapping beyond the end of the frame buffer. Then comes a call to getkpgmap to get a PTE, which is passed up the stack by fbmmap and cgonemmap.

Mapping Devices Without Device Drivers

Under a restricted set of circumstances, it's possible to avoid writing a device driver altogether by using the mmap system call to overlay the device's registers and memory onto user memory. Having done this, you can read and write the registers — as if they were normal user memory — from a user program.

What this really amounts to is piggybacking the new device onto an another, system standard, virtual memory device (and its driver). The mmap routine of a system virtual memory device is then used to do the user-device mapping, and the "installation" is accomplished without the development of a driver specific to the user device. Instead, a user level program is written, one that calls the mmap system call.

The restrictions on this shortcut are, however, fairly severe.

□ The device must not require any special handling of the type that would go into *xx*`ioctl`.

□ The device (including all control registers) must work with user function codes, since that's what it'll get when mapped into and then accessed from user space.

□ The device must not require any other sort of special handling — it cannot, for example, be multiplexed, interrupt driven, or do DMA.

□ Finally, there are security problems associated with this sort of installation. Since the system virtual-memory devices are normally owned by and restricted to the superuser, your programs will either have to change their permissions to allow normal users to access them, or will have to run with superuser privileges. This latter strategy is usually not acceptable in the long run, because it creates a gaping hole in the security of the system. And it's far from clear that the first alternative is desirable either.

The virtual-memory devices of interest here are those that support mapping over the entire range of a virtual address space. They are:

Table 5-3 *Virtual Memory Devices*

Machine Type	Memory Device Name
Multibus	`mbmem`
Multibus	`mbio`
VMEbus	`vme16d16`
VMEbus	`vme24d16`
VMEbus (Sun-3 only)	`vme32d16`
VMEbus (Sun-3 only)	`vme16d32`
VMEbus (Sun-3 only)	`vme24d32`
VMEbus (Sun-3 only)	`vme32d32`

In addition, `/dev/fb`, a system memory device which, on any given system, is set up as the local frame-buffer device, can be used as if it were a system memory devices. On any given system, `/dev/fb` can be `mmap`'ed into user memory and then written, with the effect of writing the local frame buffer memory.

To use `mmap` with one of the system memory devices, you must do three things:

□ Open the appropriate device.

□ Call `valloc` to get enough *page aligned* virtual space to store the device's registers.

□ Call `mmap` to map that virtual space to the physical bus address of your device, which you must know. (See the *Hardware Context* chapter for a discussion on how to pick a good physical address from the information in the system config file).

The following program uses /dev/fb rather than one of the virtual memory devices. This makes a good example because it maps the system frame buffer into user memory so that it can then be written from a user program. It uses the mmap system call to set things up, but doesn't bother with calling munmap, because unmapping occurs automatically when the memory device is closed. This close occurs implicitly when the program ceases execution. (Special care should, however, be taken when mapping more that 128K of memory — see the discussion of mmap in the *Summary of Device Driver Routines* appendix).

Once the device has been mapped into user space it can be treated as a piece of local user memory. (Remember that memory accesses performed by way of this mechanism will be reflected — at the device level — as non-privileged (user) accesses. This is because mmap accesses inherit the privilege of the process that calls mmap. Thus, if mmap is called from a driver, subsequent memory accesses will have the standard supervisor data access privilege, but if it's called from a user process, as described here, subsequent accesses will be non-privileged. Attempts to access supervisor-only device registers without supervisor privilege might produce a bus error).

```
#include <stdio.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/types.h>

/* Width and Height of Frame Buffer in Bytes */
#define WIDTH 1152
#define HEIGHT 900

main()
{
    int fd, prot, share, offset, pagesize;
    unsigned len;
    char *addr, *valloc();

    /* Open the frame-buffer device */
    if ((fd = open("/dev/fb",O_RDWR)) < 0)
         syserr("open");

    /* Get page size (addresses must be multiples of the page size) */
    pagesize = getpagesize();

    /* Compute total number of bytes */
    len = ((WIDTH * HEIGHT)/8);

    /* Adjust len to multiple of page size */
    len = pagesize + len - (len % pagesize);

    /* Allocate len bytes of page-aligned memory */
    if ((addr = valloc(len)) == 0)
        syserr("valloc failed");

    prot = PROT_READ|PROT_WRITE;
```

```
        share = MAP_SHARED;
        offset = 0;

        /* Map device memory to user space */
        if (mmap(addr, len, prot, share, fd, offset) != 0)
            syserr("mmap failed");

        writeFB()
    }

    void writeFB() /* Write to frame buffer */
    {
        char color, *cptr;
        int i,j;

        color = 0xFF;
        cptr = addr;
        for (i = 0; i < HEIGHT; i++) {
            color = ~color;
            for (j = 0; j < WIDTH/8; j++)
                *cptr++ = color;
        }
    }

    void syserr(msg) /* print system call error message and terminate */
        char *msg;
    {
        extern int errno, sys_nerr;
        extern char *sys_errlist[];

        fprintf(stderr,"ERROR: %s (%d", msg, errno);
        if (errno > 0 && errno < sys_nerr)
            fprintf(stderr, "; %s)0, sys_errlist[errno]);
        else
            fprintf(stderr,")0);
        exit(1);
    }
```

The memory into which the device has been mapped is allocated with valloc — this is because valloc guarantees that the memory it returns is aligned on a page boundary, as mmap requires. len and offset must also be page aligned, and len must be adjusted to the byte count of the lowest number of pages that will completely contain the area being mapped.

This program can easily be rewritten to use a system virtual memory device rather than /dev/fb. Since the standard Sun-2 frame buffer is installed in on-board memory, the appropriate memory device is /dev/obmem. (/dev/mem could be used for frame buffers installed in main memory). /dev/obmem isn't included in the above table since no user devices can ever appear in it. Nevertheless, it's use involves almost no changes to the code: /dev/obmem is opened instead of /dev/fb, and offset is initialized to (long) BW2MB_FB (for a Multibus machine). This offset it defined in <sundev/bw2reg.h>.

So, despite the plethora of limitations on the sorts of devices that can be installed by way of mapping them into user space, it's quite an easy thing to do. If your device characteristics are such that this is an option, you may well wish to take it. And even if such an installation isn't an attractive long-term option (for example, because of unacceptable security problems) it may still be attractive as a short-term alternative to driver development. Even in environments where security considerations make it unacceptable in the long term, it can allow you to get your device up and running very quickly. Sometimes this counts for a lot.

Direct Opening of Memory Devices

It should be noted, for the purpose of completeness, that there's another approach to avoiding driver development, one that's even easier than the use of mmap described here. That is, it's possible to simply open the virtual memory device that contains your board, to seek to the location of its registers, and then to read and write those registers as if they were regular memory.

This approach has most of the same problems as does the use of mmap, *though it will at least insure that the device receives supervisor function codes. It does, however, introduce a few new problems.* It doesn't give you the same degree of control as does mmap, and you often need that control when dealing with devices. When you use mmap, the device actually becomes part of your user memory space, and it's left to the compiler to generate exactly the I/O accesses which you implicitly specify in your structure and variable declarations. You can always access exactly what you want, and the accesses occur directly as *move byte* and *move word* operations. They are thus very fast.

When, however, you simply open a system memory device as a file and then read and write to it, your communication with your board is mediated by the I/O system. The I/O system will always try and do the "right thing" (if you request I/O at an odd address or for an odd number of bytes it will perform byte accesses as appropriate; otherwise it will use short integers), but nevertheless it doesn't give you the kind of control that can be had using mmap. Furthermore, I/O operations involve lots of code, and take *hundreds* of times as long as mmap, which uses the MMU to treat device registers and memory as physical memory directly accessible by low-level store and move instructions.

So the bottom line is that, if you just need to access the device a few times, or if performance isn't critical, you can do your installation by opening a system memory device and then seeking to your device registers and memory space. Otherwise, use mmap or write a driver. If you do decide to use the open/lseek method, do so directly rather than with the standard I/O library. The standard I/O library implements a buffered I/O scheme which will add considerably to your problems.

The following user program writes the same pattern on system frame-buffer memory as does the mmap-based routine given above, but it does so by directly opening the system memory device within which the frame buffer is installed. The syserr routine is the same as in the above example, but writeFB now uses the I/O system; it's thus much slower than the version above.

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

```
#include <sys/param.h>
#include <sys/buf.h>
#include <sundev/mbvar.h>
#include <sundev/bw2reg.h>
#include <sys/file.h>

void syserr();
long lseek();

#define WIDTH 1152
#define HEIGHT 900

main()
{
    int fd;


    /* Open the system memory device containing the frame buffer */
    if ((fd = open("/dev/obmem",O_RDWR)) < 0)
        syserr("open");

    /* Seek to the frame buffer memory */
    if (lseek(fd, (long)BW2MB_FB, L_SET) == -1L)
        syserr("lseek");

    writeFB(fd);
}

void writeFB(fd) /* Write to frame buffer */
    int fd;
{
    char color;
    int i,j;

    color = 0xFF;
    for (i = 0; i < HEIGHT; i++) {
        color = ~color;
        for (j = 0; j < WIDTH/8; j++) {
            if (write(fd, &color, 1) == -1)
                syserr("write");
        }
    }
}
```

5.3. Debugging Techniques

As described above, it's a good idea to begin debugging by using the monitor to check that the device has been installed at the intended address, and that it works, before proceeding to debug your device driver. This allows you to avoid debugging the device simultaneously with the driver, an experience that you'd like to avoid for as long as possible. Alternatively, if you're confident in both your device and the correctness of your installation, you can simply make a new kernel, boot it and proceed with debugging. In this case you should put some `printf` messages — see below — into the *xx*probe routine. Then you can at least see the device get contacted and initialized.

Debugging drivers is significantly more difficult than debugging regular user programs, for a number of reasons:

□ In the first place, device drivers are part of the system kernel. This means that the system is not protected from their errors. Addressing errors, for example, will frequently produce hardware traps and a system crash.

□ As mentioned above, there's the possibility that the device hardware will be buggy. For this reason, you can't really trust your environment in the same way as you can when writing a user program on a mature computer system.

□ Some devices behave in rather peculiar ways. (See *A Warning about Monitor Usage* above).

□ Finally, the debugging environment in the kernel is thinner than it is in user space. Beginning with Sun Release 3.2, there is a kernel debugger, kadb, and this is certainly a big step towards making life easier for driver developers. Still, life remains more difficult when debugging in kernel space. *It's possible to prototype drivers in user address space by using techniques similar to those described in the Mapping Devices Without Device Drivers section of this chapter. The same constraints given there apply to prototyping. In particular, it's not possible to run an interrupt routine, or to xx*probe *for non-existent devices without generating bus errors from prototype drivers in user space. If the device generates no interrupts, and if it doesn't do DMA, the entire driver might be able to be run in user space.*

For all of these reasons, you should give extra care to desk-checking your code, and check a reference manual when not absolutely sure of the meaning of a given construction. Don't take chances.

Also, make changes incrementally. Don't try to save time be making many changes at once. You will save time in the long run if you take the time to add and test a few parts at a time. Keep your feet on solid ground.

Use trace output from printf, as described below. Drivers can act in surprising ways, and the best way to proceed is by making the flow of operations highly visible.

Debugging with printf

With the introduction, in Sun release 3.2, of the kernel debugger kadb, the importance of printf in the debugging of device drivers has been significantly reduced. Still, even with kabd available, printf statements remain useful as means of providing synchronous tracing of overall driver flow and structure. kadb can be made to provide a similar sort of tracing (by tying print commands to strategically chosen breakpoints) but this probably won't altogether eliminate the printf statement. The printf has long found application in driver debugging, and, as a matter of taste and experience, some programmers will continue to use it. For this reason, we'll discuss its use in some detail.

The kernel printf sends its message directly to the system console, without going through the tty driver. As a consequence, the printing is uninterruptible —

the characters aren't buffered. Furthermore, printf runs at high priority, and no other kernel or user process activity takes place while its output is being produced. printf thus radically limits overall system performance (though this is usually ok while device drivers are being debugged).

There is a second kernel print statement, uprintf. uprintf, however, is of little use to driver developers. It attempts to print to the current user tty as identified in the user structure, and prints to the console only if there's no current user tty (at which point it becomes identical to printf). uprintf cannot be called from lower-half routines, which run in interrupt context and cannot make any assumptions about the user structure (where uprintf looks to determine the current user tty). uprintf is most useful for production drivers, like tape drivers that encounter media errors, which want to report errors not to a programmer but to the user.

There are occasions in which the use of printf *(or* uprintf*) statements will change the behavior of your driver.* printf *statements, for example, can affect the timing of operations in the driver being tested as well as in other drivers. The output may be so slow relative to other device operations that interrupts are lost and system failures are introduced; thus, it is frequently impossible to synchronously trace a device interrupt routine. Driver code may begin to fail only when* printf*s are introduced, or, even worse, only when* printf*s are disabled. If you're debugging a tty driver, you may even face a situation where* printf*-based tracing generates new calls to the driver being debugged. Thus, there are situations in which it cannot be used. In such situations, you should use* kadb *or the techniques suggested below in the section on Asynchronous Tracing.*

The best way to use printf statements for tracing driver execution is by setting things up so that you can toggle printing by using the kernel debugger, kadb (see below) to set and reset print-control variables. Doing so is very simple. At the top of the driver source file, include statements like:

```
#ifdef XXDEBUG
int xxdebug = 0;
#define XXDPRINT if (xxdebug > 0) printf
#endif
```

(It's important that the variables like xxdebug be global, so that you can later access them freely from the debugger — remember that all drivers are part of one program, the kernel, and name your print-control variables so as to avoid naming conflicts).

Then, instead of calling printf inside the driver routines, call XXDPRINT. Each call should be in the form:

```
#ifdef XXDEBUG
XXDPRINT("driver name...",...);
#endif
```

which will only call printf if XXDEBUG is defined and xxdebug is set to a value greater than 0.

Make sure that each call to `XXDPRINT` identifies the driver, for it's possible that you, or some other programmer, will want to see debugging output from several drivers at once. And leave the debugging code in for a while after you're done — bugs may surface later.

Having set things up like this, you can turn the `printf`'s on or off at any time by using `kadb` to set unset or change the print-control variable `xxdebug`. Or you can use `adb` if you wish, running it at user level in a separate window:

```
adb -w /vmunix /dev/kmem
```

Where `/vmunix` is an *unstripped* version of the kernel. (`adb` won't allow you to set breakpoints in the kernel, but it will allow you to set and unset variables — you can change the value of `xxdebug`, or even reset a variable which has caused your driver to hang). *Remember that you're in the kernel and BE CAREFUL.*

Incidently, `/dev/kmem` represents the kernel *virtual* address space, which is why it's used here. `adb -k /vmunix /dev/mem`, in contrast, generates a view of the *physical* address space, because `/dev/mem` represents the physical memory. This latter command is useful for examining core files.

Good places to put `printf` statements include:

□ driver routine entry points

□ before critical subroutine calls

□ upon reading status information from the device

□ before writing of commands or data to the device

□ at intermediate points in complex routines

□ at routine exit points

Note again that you don't have to restrict yourself to a single `xxdebug` variable, or to binary tests that check to see if a variable is on or off. You can use as many variables, and as many values for each variable, as necessary to reflect the functional divisions most appropriate to your driver. It might even be useful to get truly esoteric, and send certain trace statements directly to the user tty (by calling `uprintf`) while the rest use `printf` and go to the console.

Event-Triggered Printing

In the above discussion, the `xxdebug` variable was initialized by the compiler, and toggled with a debugger. However, it's just as easy to have the driver routines themselves set a trigger variable under pre-chosen conditions.

For example, if you wanted to enable tracing after a given *condition* had occurred, you could declare `xxdebug`, just as was shown above, but define `XXDPRINT` somewhat differently:

```
#ifdef XXDEBUG
int xxdebug = 0;
#define XXDPRINT(v,msg,a1,a2) \
    if (xxdebug > (v)) printf(msg,a1,a2);
#endif
```

and then, in the code that checks for the condition:

```
#ifdef XXDEBUG
if (condition) xxdebug = 1;
#endif
```

Then to call XXDPRINT:

```
#ifdef XXDEBUG
XXDPRINT(0,"driver name...\n",a,b);
#endif
```

One major disadvantage of using the kernel printf is that its output doesn't go through a device driver, and thus can't be paused with Control-S or redirected to a file. It's possible, then, that printf will overwhelm you with output. There are a number of things that you can do if you run into this problem:

- If you haven't used multivalued print-control variables, then do so. This gives you more control than you have with simple on/off print control, and will allow you to reduce the amount to trace noise.
- You can use a debugger to set the global variable noprintf. This will keep printf's output from being sent to the console, but that output will still go to a buffer where kernel error messages are kept before being transferred to /usr/adm/messages. You can examine the message buffer at your leisure, in one of two different ways:
  - From a user window, you can use dmesg.
  - From kadb (or adb on /dev/kmem) you can type msgbuf+8/s.
- It's also possible to reconfigure your system so that it uses a hardcopy terminal as its console over a RS-232 line. Then, you won't lose any of the printf output.
- Best of all, you can get another machine and connect it to your machine over a RS-232 line. Having done so, use tip to open a window on the second machine *as the console of the test machine*. You can then use tip's record feature (see the tip man page) to make a record of all the stuff that printf is sending to the test machine's console.

Asynchronous Tracing

As mentioned above, there are occasions when timing problems forbid the use of the printf statement. In these cases, its a good idea to give up any attachment that you might have to printf statements and use kadb.

Or, if you prefer, it's possible to deal with timing problems by using kadb to patch the noprintf variable, and then to check the message buffer to see what's going on. Doing so:

□ allows you to continue using the debugging code that you installed before encountering the timing problem, and

□ presents you with a sequential list of the events in your driver, a list spelled out in English phrases and including interrupt-level events.

Or, you can simply use kadb for everything.

kadb — A Kernel Debugger

kadb is an interactive debugger similar in operation to adb. It's included in Sun release 3.2 and subsequent releases, but it'll not not work with versions of the kernel earlier than 3.2. kadb differs in several key respects from adb. It runs as a standalone program under the PROM monitor, rather than as a user process in user address space. And it allows you to set breakpoints and single step in the kernel!

Thus, running a kernel under kadb is significantly different than running it under adb -k. The 'k' option to adb merely makes it simulate the kernel memory mappings while kadb actually runs in the kernel address space. And unlike adb, which runs at user level and as a separate process from the process being debugged, kadb runs in system space as a *coprocess*. It shares not only the kernel address space but its CPU supervisor mode as well.

kadb, for all intents and purposes, is a version of adb. It has the same command syntax and almost the same command set. Thus, you can see the kadb and adb manual pages, as well as *Debugging Tools for the Sun Workstation*, for more details on its use. Note, however, the following points of special interest to driver developers:

□ All interrupts are disabled while interacting with kadb (except non-maskable interrupts). Thus, when using kadb to examine memory, the kernel remains stable. However, while single stepped instructions are being executed, the actual standing priority of the kernel is temporarily restored, and interrupts can get dispatched, run and return. You won't notice unless you have a break point set in the interrupt routine, which works just fine.

□ kadb is installed so that, when a program is being run under it, an abort sequence (L1-A) will transfer control not to the PROM monitor but to kadb itself. Once in kadb, you can abort again and be transferred to the monitor. The transfer is direct and immediate, so you can use the monitor to examine control spaces (e.g. page and segment maps) which are not accessible from kadb. The monitor 'c' command will return you to kadb.

□ kadb runs in the same virtual memory space as the kernel itself, and with the CPU in supervisor mode. This means that kadb uses the kernel memory maps when calculating virtual addresses, and that it can directly examine kernel structures. This is in contrast to the situation with adb -k, where software copies of the page table entries are used to map virtual addresses to physical pages.

□ kadb's memory view is almost the same as that resulting from adb /vmunix /dev/kmem. In other ways, however, kadb is much different. To give just one example: on Sun-3 machines, where users and supervisors share the virtual address space, kadb allows the user to examine the user virtual address space (this is *not* true with `adb -k`).

- Finally, be aware that `kadb` — as a consequence of the way that `adb` works — always does 32-bit memory reads. Even if you tell `kadb` to read a byte it will read a long. This leads to a lack of control that can cause problems when reading device registers.

5.4. Device Driver Error Handling

There are various types of errors: "expected" errors like those generated by *xx*`probe` routines, transient errors in operations that can reasonably be retried, fatal errors that require controlled shutdowns, and others. The kinds of errors that you'll face depends upon the kinds of drivers that you write and the peculiarities of your devices; few generalizations can usefully be made.

To further complicate matters, the detection and treatment of errors varies greatly from device to device. You should begin by carefully reading your device specification manual to determine the error indications that can arise and the responses that should be made when they do. Most devices have at least an error bit in the control/status register, and usually more detailed error information is available. Ideally, you should understand all potential errors, avoid those that you can and recover from the rest. This ideal isn't always achievable, but try not to leave any obvious holes. *If you do nothing else, check for device errors and use the kernel* `printf` *function to report them to the system console.*

Error-Handling Mechanisms

There are various error reporting and management mechanisms available to the driver developer. Most of them have already been mentioned as they've become relevant; here they are collected and summerized:

Error Recovery

It's difficult to generalize about error-recovery mechanisms, for they are largely device specific. It's worth noting, however, that:

- Some errors are worth retrying and some aren't; the matter is entirely device specific.
- Error-recovery routines should be able to run at the interrupt level. This is because errors can occur either synchronously or asynchronously with respect to the dispatch of device commands, and, therefore, recovery routines must be callable from interrupt routines.
- If you do implement error recovery logic, you must do so consistently. The data structure that contains retry-status information must be global, and must be protected by critical sections. Error-recovery routines, like interrupt routines in general, must take special pains to protect data-structure integrity; indeed, they must *restore* such integrity upon encountering errors they can't recover from.

Error Returns

There are three mechanisms by which driver routines can report errors up to their calling routines. The first, of course, is by way of the values that the driver routines return to their callers. The second is by way of the system error codes they return in u.u_error and the third — useful when returning errors from *xx*strategy, *xx*start, and *xx*intr — is the error-reporting mechanism built into the buffer-header.

Error Signals

It is sometimes desirable to have a driver send a software interrupt to the process or processes. It's possible, for example, that a device will fail in a unrecoverable fashion — in this case it's perhaps a good idea to signal the user processes, rather than merely returning an extraordinary error code. It's also possible (though rare) for a driver to encounter serious errors from which it can recover by restarting the device — user processes may also need to be notified in this case. The kernel psignal and gsignal routines can signal either a single process or all the processes in a given process group.

Error Logging

When you use the kernel printf statement to report errors to the console, those errors are also placed into a system error-message buffer accessible to the dmesg daemon. dmesg can be, and typically is, run every 30 minutes by the crontab daemon, for the purpose of appending the messages in the buffer to /usr/adm/messages. Note that the message buffer is small, and that if a lot of entries are being written into it, some of them will get lost before being transferred into /usr/adm/messages.

Kernel Panics

The most unequivocal way of dealing with an error is to panic when you get it. The panic routine is provided to help you do so in a somewhat controlled fashion — panic is called only on unresolvable fatal errors. It prints "panic: mesg" on the console, and then reboots. (Or, if you're running under the debugger, it transfers control to kadb). panic also keeps track of whether it's already been called, and avoids attempts to sync the disks (by flushing all pending write buffers) if it has, since this can lead to recursive panics.

The final production version of a driver should call panic only when "impossible" situations are encountered; lesser errors should be recovered from. During debugging, though, panic can be used to implement a passable assert mechanism.

```
#ifdef XXDEBUG
if (inconsistent condition)
    panic("Assertion Failed: ...");
#endif
```

(It's possible to write a fancier assert mechanism, for example by having an ASSERT macro which calls an assert routine which prints error context information and then calls panic, but this minimal hack will perhaps do).

Finally, note that in cases where it's *very* important to halt the system *immediately* after detecting an inconsistent condition, kadb can be used. The driver code can test for the inconsistent condition, and then set a debugging variable:

```
if (inconsistent condition)
    junk = 1;
```

kabd can then be used to set a breakpoint at the machine instruction generated from the assignment to junk.

5.5. System Upgrades

System upgrades generally have minimal effects on user-written device drivers. The changes that are necessary are rare and release specific.

Some changes must be made if user-written drivers are to work with new release software. In Release 2.0, for example, there was a minor change in one of the bus-interface structures. There wasn't much involved in adapting user-written drivers, but it had to be done.

In other cases, changes are optional. When VMEbus machines were introduced, for example, drivers had to be adapted to run on them; however, it was possible to upgrade Multibus machines without rewriting user-written drivers.

In any case, any release upgrades that imply changes — either optional or mandatory — to user-written device drivers will be documented in the *System Summary and Change Notes* for the release in question.

6

The ''Skeleton'' Character Device Driver

The ''Skeleton'' Character Device Driver 97

6.1. General Declarations in Driver 100

6.2. Autoconfiguration Procedures 101

Probe Routine 101

Attach Routine 103

6.3. Open and Close Routines 103

6.4. Read and Write Routines 105

Some Notes About the UIO Structure 106

6.5. Skeleton Strategy Routine 107

6.6. Skeleton Start Routine — Initiate Data Transfers 108

6.7. Interrupt Routines 110

6.8. Ioctl Routine 112

6.9. DMA Variations 112

Multibus or VMEbus DVMA 112

A DMA Skeleton Driver 112

6

The ''Skeleton'' Character Device Driver

This chapter presents one of the simplest drivers you could ever hope to encounter, a driver for an imaginary Multibus character device known as the "Skeleton" device. Both programmed I/O and DMA versions of the driver will be discussed. There is a complete version of this driver in the *Sample Driver Listings* appendix to this manual — the parts are presented piecemeal here with some discussion of their functions.

What we're doing here is inventing the very simple, I/O mapped, Skeleton controller. It's actually a "free device" with no separate controller and no separate slaves. It has a single-byte command/status register, and a single-byte data register. It's a write-only device. It's not a slow tty-type device — you can provide vast blocks of data and the Skeleton board gets it all out very fast. It interrupts when it's ready for a data transfer, and comes up in the power-on state with interrupts disabled and everything else in neutral.

Note: the Skeleton device is capable, in both its simple and its DMA variants, of writing chunks (not to say "blocks") of data in a single operation. It is, therefore, a character device that can make good use of *xx*`strategy` routines, `physio`, `buf` structures and other block-I/O mechanisms. As explained in *Kernel Topics and Device Drivers*, its use of these mechanisms does *not* make it a block driver. Rather, its simple needs are a subset of the needs of block drivers, and it's convenient here for form to follow function.

Let us assume that we've installed the Skeleton board with its control/status register at `0x600` in Multibus I/O space — this puts its data register at `0x601`. The control/status register is both a read and a write register, with bit assignments as shown in the tables below.

BIT	7	6	5	4	3	2	1	0
Read	Interrupt				Device Ready	Interface Ready	Error	Interrupt Enabled

BIT	7	6	5	4	3	2	1	0
Write						Reset		Enable Interrupt

Here is a brief description of what the bits mean:

When *reading* from the status register

bit 7 is a 1 when the board is interrupting, 0 otherwise.

bit 3 is a 1 when the device that the board controls is ready for data transfers.

bit 2 is a 1 when the Skeleton board itself is ready for data transfers.

bit 0 is a 1 when interrupts are enabled, 0 when interrupts are disabled.

When *writing* to the status register

bit 2 resets the Skeleton board to its startup state — interrupts are disabled and the board should indicate that it is ready for data transfers.

bit 0 enables interrupts by writing a 1 to this bit, disables interrupts by writing a 0.

The header file for this interface is in `skreg.h`. By convention, we put the register and control information for a given device (say `xy`) in a file called `xyreg.h`. The actual C code for the `xy` driver would by convention be placed in a file called `xy.c`. The header file for the Skeleton board looks like this:

```
/*
 * Registers for Skeleton Multibus I/O Interface -- note the byte swap
 */
struct sk_reg {
    char sk_data;      /* 01: Data Register */
    char sk_csr;       /* 00: command(w) and status(r) */
};

/* sk_csr bits (read) */
#define SK_INTR         0x80      /* Device is Interrupting */
#define SK_DEVREADY     0x08      /* Device is Ready */
#define SK_INTREADY     0x04      /* Interface is Ready */
#define SK_ERROR        0x02      /* Device Error */
#define SK_INTENAB      0x01      /* Interrupts are Enabled */

#define SK_ISTHERE      0x0C      /* Existence Check;
              Device and Interface Ready */

/* sk_csr bits (write) */
#define SK_RESET        0x04      /* Reset Device and Interface */
#define SK_ENABLE       0x01      /* Enable Interrupts */
```

The complete device driver for the Skeleton board consists of the following parts:

`skprobe`
: is the autoconfiguration routine called at system startup time to determine if the `sk` board is actually in the system, and to notify the kernel of its memory requirements.

`skopen` and `skclose`
: routines for opening the device for each time the file corresponding to that device is opened, and for closing down after the last time the file has been closed.

`skwrite`
: routine that is called to send data to the device.

`skstrategy`
: routine that is called from `skwrite` via `physio` to control the actual transfer of data.

`skstart`
: routine that is called for every byte to be transferred.

`skpoll`
: the polling interrupt routine that services interrupts and arranges to transfer the next byte of data to the device.

The subsections to follow describe these routines in more detail.

6.1. General Declarations in Driver

In addition to including a bunch of system header files, there are some data structures that the driver must define.

```
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/file.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/uio.h"
#include "../machine/psl.h"
#include "../sundev/mbvar.h"

#include "sk.h"          /* file generated by config;
                            contains the definition of NSK */

#include "skreg.h"       /* register definitions */

#define SKPRI (PZERO-1) /* software sleep priority for sk */

#define SKUNIT(dev)  (minor(dev))

struct buf skbufs[NSK];    /* static buffer headers for physio */

/* autoconfiguration-related declarations */
int skprobe(), skpoll();      /* kernel interface routines */
struct mb_device *skdinfo[NSK];
struct mb_driver skdriver = { skprobe, 0, 0, 0, 0, skpoll,
    sizeof(struct sk_reg), "sk", skdinfo, 0, 0, 0, 0,
    };

/* device state information -- global to driver */
struct sk_device {
    char soft_csr;          /* software copy of csr */
    struct buf *sk_bp;      /* current buf */
    int sk_count;           /* number of bytes to send */
    char *sk_cp;            /* next byte to send */
    char sk_busy;           /* true if device is busy */
} skdevice[NSK];
```

Here's a brief discussion on the declarations in the above example.

`sk.h` file is automatically generated by `config`. It contains the definition of NSK, the number of sk devices configured into the system.

`SKPRI` declaration declares the software priority level at which this device driver will sleep.

`SKUNIT` macro is a common way of obtaining the minor device number in a driver. Study just about any device driver and you will find a declaration like this — it is a stylized way of referring to the minor device number. One reason for this is that sometimes a driver will encode the bits of the minor device number to mean things other than just the device number, so using the SKUNIT convention is an easy way to make sure that if things change, the code will not be affected.

skbufs array is necessary so that the driver will have its own `buf` headers to pass to the `physio` routine. Character drivers should *never* use `buf` headers from the kernel's I/O queue. `physio` will fill in certain fields (only a few, really) before calling *xx*`strategy` with the `buf` structure as the argument.

There then follows a series of declarations, one for each of the autoconfiguration-related entry points into the device driver. In this driver, the only such entry points we use are `skprobe` (which probes the Main Bus during system configuration) and `skpoll` (the polling interrupt routine).

`skdinfo` is an array of pointers to the `mb_device` structures that correspond to the driver's devices. The autoconfiguration process will initialize it during kernel boot time.

`skdriver`

is a definition of the `mb_driver` structure for this driver. An explanation of the fields in this structure and how they are initialized appears in the *Autoconfiguration-Related Declarations* section of this manual.

This data structure is the major linkage to the kernel. It *must* be called *driver-name*`driver` where *driver-name* is the name of the device driver. `config` assumes that all device-driver structures have names in the form *driver-name*`driver`.

`sk_device`

is a definition of a structure, global to the driver, that holds driver-specific state information.

6.2. Autoconfiguration Procedures

Sun device drivers are tightly bound to the Sun autoconfiguration system. They assume, at compile time, that certain services have been provided for them by `config`, and they, in turn, provide boot-time hooks by which the kernel can determine if the actual system configuration matches that given in its `config` file.

There are, essentially, two autoconfiguration routines provided by the driver. The first is *xx*`probe`, the second *xx*`attach`. For more information, see the *Overall Kernel Context* section of this manual.

Probe Routine

There should be a *xx*`probe` function in every driver. During the system boot each device entry in the config file generates a call to the *xx*`probe` routine in the corresponding driver. *xx*`probe` has three functions:

1. To determine if a device is present at the address indicated in the config file.
2 To determine if its the expected type of device.
3. To notify the kernel of the system resources required for the device.

Under normal circumstances, addressing non-existent memory or I/O space on the Multibus or the VMEbus generates a bus error in the CPU. The kernel, however, supports checking for device existence with a set of functions designed to probe the address space, recover from possible bus errors, and return an indication as to whether the probe generated a bus error.

These functions are peek, peekc, poke, and pokec. They provide for accessing possibly non-existent addresses on the bus without generating the bus errors that would otherwise terminate the process trying to access such addresses. peek and poke read and write, respectively, 16-bit words (shorts in the Sun system). peekc and pokec read and write 8-bit characters. In general, you will use the character routines for probing single-byte I/O registers. See the *Kernel Support Routines* appendix for details on these routines.

Having determined whether the device exists in the system, the *xx*probe function returns either:

- the size (in bytes) of the device structure if it does exist. The kernel uses the value returned from probe to reserve memory resources for that device. For both I/O-mapped and memory-mapped devices, probe returns the total amount of space consumed by the device registers and memory.
- a value of 0 (zero) if the device does not exist.

Now we can write skprobe:

```
/*ARGSUSED*/
skprobe(reg, unit)
    caddr_t reg;
    int unit;
{
    register struct sk_reg *sk_reg;
    register int c;

    sk_reg = (struct sk_reg *)reg;

    /* contact the device */
    c = peekc((char *)&sk_reg->sk_csr);
    if (c == -1 || (c != SK_ISTHERE))
        return (0);

    /* contact the device */
    if (pokec((char *)&sk_reg->sk_csr, SK_RESET))
        return (0);

    return (sizeof (struct sk_reg));
}
```

The *reg* argument is the purported address of the device, as given in the config file. The *unit* argument is only needed for controller drivers that must distinguish among multiple slave devices.

The *xx*probe routine determines that the device actually exists, resets it to make sure that it's ready to go, and then returns the amount of bus space that it uses to the kernel autoconfiguration process. That value is plugged into the md_alive field in the device structure. md_alive is subsequently used by other driver (and kernel) functions to check that the device was probed successfully at startup time. (These routines can also check the device's position in the driver's *xx*dinit array (if it has one) to see if it's been initialized).

Attach Routine

The second autoconfiguration routine is *xx*attach. The purpose of *xx*attach is to do device-specific initialization. Such initialization may include the issuing of commands to the actual device hardware, for example, the disabling of its interrupts, or it may be entirely confined to the initialization of local device-specific structures. It's up to the driver what kind of initialization is done in *xx*attach.

The Skeleton device is artificially simple, and it requires no initialization besides the assignment of SK_RESET into its control/status register. This assignment, as you'll note, has already been done in skprobe, where it serves as a doublecheck on the correct installation of the device. Since no further initialization is necessary, the Skeleton driver needs no attach routine.

6.3. Open and Close Routines

During the processing of an open call for a special file, the system always calls the device's *xx*open routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, and so on). However, the *xx*close routine is called only when the last process closes a file, that is, when the i-node table entry for that file is being deallocated. Thus it is not feasible for a device driver to maintain, or depend on, a count of its users, although it is quite simple to implement an exclusive-use device that can't be reopened until it has been closed.

skopen is quite straightforward. It's called with two arguments, namely, the device to be opened, and a flag indicating whether the device should be opened for reading, writing, or both. The first task is to check whether the device number to be opened actually exists — skopen returns an error indication if not. The second check is whether the open is for writing only. Since the Skeleton device is write only, it's an error to open it for reading. If all the checks succeed, skopen enables interrupts from the device, and then returns zero as an indication of success. Here's the code for skopen:

```
skopen(dev, flags)
    dev_t dev;
    int flags;
{
    register int unit = SKUNIT(dev);
    register struct mb_device *md;
    register struct sk_reg *sk_reg;

    md = skdinfo[unit];

    if (unit >= NSK || md->md_alive == 0)
        return (ENXIO);
    if (flags & FREAD)
        return (ENODEV);

    sk_reg = (struct sk_reg *)md->md_addr;

    /* enable interrupts */
    skdevice[unit].soft_csr = SK_ENABLE;

    /* contact the device */
    sk_reg->sk_csr = skdevice[unit].soft_csr;

    return (0);
}
```

The first `if` statement checks if the device actually exists. The first clause

```
(unit >= NSK)
```

is necessary because, as root, someone could make a special file that has a minor device number greater than NSK then try to open it. This actually isn't unusual, many /dev directories have entries for devices that are not really installed. The second clause tests to see if the *probe* routine found the device. Note the use of the SKUNIT macro to obtain the minor device number — we discussed this earlier on. Also note that we're maintaining a copy

```
(skdevice[unit].soft_csr) of the
```

control/status register in memory. Each time we write the register we'll do so first it in memory and then in the actual hardware register. We'll do this doggedly, to make the point that we must protect ourselves from the potential side effects of inadvertent calculations within registers. For example

```
csr &= ~SK_ENABLE
```

has the side effect of reading the csr register — and patterns read from this register are *not* always identical to those written into it. (For more information, see the *Hardware Peculiarities to Watch Out For* section of the *Hardware Context* chapter).

`skclose` is quite straightforward, since all it does is disable interrupts:

```
/*ARGSUSED*/
skclose(dev, flags)
    dev_t dev;
    int flags;
{
    register int unit = SKUNIT(dev);
    register struct mb_device *md;
    register struct sk_reg *sk_reg;

    md = skdinfo[unit];

    /* disable interrupts */
    sk_reg = (struct sk_reg *)md->md_addr;
    skdevice[unit].soft_csr &= ~SK_ENABLE;

    /* contact the device */
    sk_reg->sk_csr = skdevice[unit].soft_csr;
}
```

`skclose` could in fact be more complicated than this. It could, for example:

□ deallocate resources that were allocated for the device being closed, or

□ shut down the device itself, for example by signaling a port to hang up.

6.4. Read and Write Routines

The Skeleton device is write-only, but this discussion would apply equally to reading in such a non-tty oriented character device.

When a *read* or *write* takes place, the user's arguments — as well as some system-maintained information about the file to which the I/O operation is to be performed — are used to initialize two structures — `uio` and `iovec` — that are used for character I/O. The fields of greatest interest within these structures are `iovec.iov_base`, `iovec.iov_len`, and `uio.uio_offset` which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate *xx*`read` or *xx*`write` routine is called — this routine is responsible for transferring data and updating the count and current location appropriately as discussed below.

For most non-tty devices, *xx*`read` and *xx*`write` call *xx*`strategy` through the system `physio` routine. `physio` ensures that the user's memory space is locked into core (not paged out) for the duration of the data transfer. It also provides an automated way of breaking a large transfer into a series of smaller, more manageable ones. Note that character drivers that use `physio` must declare an array of `buf` structures, one for each of their devices (here the array is named `skbufs`). By doing so they avoid any need to use the kernel's buffer cache, which is provided for the use of system block-structured devices.

xx`write` differs from *xx*`read` only in the value of the flag it passes to `physio`. `skwrite` looks like this:

```
skwrite(dev, uio)
    dev_t dev;
    struct uio *uio; See note on the uio structure below
{
    int unit = SKUNIT(dev);

    if (unit >= NSK)
        return (ENXIO);
    return (physio(skstrategy, &skbufs[unit], dev,
        B_WRITE, skminphys, uio));
}
```

The skminphys routine is called by physio to determine the largest reasonable block size to transfer at once. If the user requests a larger transfer, physio will call skstrategy repeatedly, requesting no more than this block size each time. This is important when DVMA transfers are done. (DVMA is covered in more detail below). The reasoning is that only a finite amount of address space is available for DVMA transfers and it is not reasonable for any device to tie up too much of it. A disk or a tape might reasonably ask for as much as 63 Kilobytes; slow devices like printers should only ask for one to four Kilobytes since they will tie up the resource for a relatively long time. Here's the skminphys routine.

```
skminphys(bp)
    struct buf *bp;
{

    if (bp->b_bcount > MAX_SK_BSIZE)
        bp->b_count = MAX_SK_BSIZE;
}
```

Note that if you don't supply your own minphys routine, you place the name of the system supplied minphys routine, whose name is minphys, as the argument to physio in its place, and the system supplied minphys routine gets used instead. This is not always a good thing, however, for the system routine divides an I/O operation into 63K chunks, and this can be too large for optimum system performance when the device in question is slow (like a printer).

Some Notes About the UIO Structure

When the system is reading and writing data from or to a device, the uio structure is used extensively (see /usr/include/sys/uio.h for more information). The uio structure is generalized to support what is called *gather-write* and *scatter-read*. That is, when writing to a device, the blocks of data to be written don't have to be contiguous in the user's memory but can be in physically discontiguous areas. Similarly, when reading from a device into memory, the data comes off the device in a continuous stream but can go into physically discontiguous areas of the user's memory. Each discontiguous area of memory is described by a structure called an iovec (I/O vector). Each iovec contains a pointer to the data area to be transferred, and a count of the number of bytes in that area. The uio structure describes the complete data transfer. uio contains a pointer to an array of these iovec structures. Thus when you want to write a number of physically discontiguous blocks of memory to a device, you can set up an array of iovec structures, and place a pointer to the start of the array in the uio structure. In the simplest case, there's just one block of data to be transferred, and the uio structure is quite simple. Note that physio will call the *strategy* routine at least once for each iovec contained by the uio structure.

6.5. Skeleton Strategy Routine

*xx*strategy is called by physio after it's locked the user's buffer into memory. The name *strategy* originated in the world of disk drivers, and implied that the routine could be clever about queuing I/O requests (for example, by disk address) so as to minimize time wasted by the disk. The skstrategy routine has no such problems, since it doesn't queue I/O requests for a random-access device. Still, a number of tasks remain — skstrategy must check that the device is ready, initiate the data transfer, and wait for its completion to be signaled by the interrupt routine. Note that skstrategy can safely assume that physio has properly initialized a number of variables — here we'll assume that the b_dev field in the buf has been set to contain the device number.

```
skstrategy(bp)
    register struct buf *bp;
{
    register struct mb_device *md;
    register struct sk_device *sk;
    int s;

    md = skdinfo[SKUNIT(bp->b_dev)];
    sk = &skdevice[SKUNIT(bp->b_dev)];

    s = splx(pritospl(md->md_intpri)); /* begin critical section */
    while (sk->sk_busy)
        sleep((caddr_t) sk, SKPRI);

    /* set up for first I/O operation */
    sk->sk_busy = 1;
    sk->sk_bp = bp;
    sk->sk_cp = bp->b_un.b_addr;
    sk->sc_count = bp->b_bcount;
    skstart(sk, (struct sk_reg *)md->md_addr);

    (void) splx(s); /* end critical section */
}
```

*xx*strategy doesn't actually do any I/O. It just insures that the device is not busy (by sleeping on the address of a data structure that is global to the driver) sets up for the first I/O operation and then calls *xx*skstart to get things rolling. The critical section is necessary because *xx*strategy is trying to acquire the device on behalf of one, and only one, user process.

6.6. Skeleton Start Routine — Initiate Data Transfers

*xx*start is actually responsible for getting the data to or from the device. skstart is called once directly from skstrategy to get the very first byte out to the device. After that, it is assumed that the device will interrupt every time it is ready for a new data byte, and so skstart is thereafter called from skintr. Here is one possible skstart routine:

```
skstart(sk, sk_reg)
    struct sk_device *sk;
    struct sk_reg *sk_reg;
{

    sk_reg->sk_data = *sk->sk_cp++;

    if (--sk->sc_count > 0) {
        sk->soft_csr = SK_ENABLE;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
    }
}
```

This routine will work, but not very efficiently. There's a lot of overhead in taking an device interrupt on every character. Since we know that the device can accept characters very quickly, it would be much more efficient to give the characters quickly, and thus avoid generating unnecessary interrupts. *xx*start should take advantage of device-specific characteristics to win efficiency enhancements of this type. It can wait for characters, check for ready, etc — here, we'll just check after each character and give another one if the device is ready for it. Here's the new, more efficient skstart routine.

```
skstart(sk, sk_reg)
    struct sk_device *sk;
    struct sk_reg *sk_reg;
    {

    while(sk->sc_count > 0) {   /* still more characters */
        sk_reg->sk_data = *sk->sk_cp++;
        sk->sc_count--;

        /* stop giving characters if device not ready */
        /* Note: the softcopy isn't needed for reads */
        /* contact the device */
        if (!(sk_reg->sk_csr & SK_DEVREADY))
            break;
    }

    /* still more characters */
    if (sk->sc_count > 0) {
        sk->soft_csr = SK_ENABLE;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
    } else {
        /* special case: finished command without taking any interrupts! */

        /* disable interrupts */
        sk->soft_csr = 0;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
        sk->sk_busy = 0;

        /* free device to sleeping strategy routine */
        wakeup((caddr_t) sk);

        /* free buffer to waiting physio */
        iodone(sk->sk_bp);
    }
}
```

We give characters to the device as long as there are more characters and the device is ready to receive them. If we run out of characters, we disable interrupts to keep the device from bothering us and call `iodone` to mark the buffer as done.

It may be that the device is not quite quick enough to take a character and raise the `SK_DEVREADY` bit in the time we can decrement and test the counter. If so, it would be very worthwhile to busy wait for a short time. The reasoning is that while busy waiting is a waste, servicing an interrupt costs lots more CPU time, and if busy waiting works fairly often it is a big win. There is a macro `DELAY` that takes an integer argument which is approximately the number of microseconds to delay, so we could add

```
    DELAY(10);
```

just before the while. Clearly this is an area where experimentation with the real device is called for.

6.7. Interrupt Routines

Each device should have appropriate interrupt-time routines. When an interrupt occurs, it is transformed into a C-compatible call on the device's interrupt routine. After the interrupt has been processed, a return from the interrupt handler returns from the interrupt itself.

The address of the polling interrupt routine for a particular device driver is contained in the per-driver (that is, a mb_driver) data structure for that device driver. It is installed there during the kernel configuration process based upon information in the config file.

Since (on Multibus machines) devices typically need to share interrupt levels, it's the specific driver's responsibility to determine if the interrupt is intended for it or not. The driver does so by providing a polling interrupt routine that queries the interrupt state of each of its devices in turn — if a driver doesn't provide such a routine, it can only run on VMEbus machines. Polling interrupt routines that determine that an interrupt belongs to one of their devices must notify the kernel to that effect (after servicing the interrupt) by returning a non-zero value. If a polling interrupt routine determines that an interrupt is *not* from one of its devices, it must return a zero value.

It's expected that the device actually indicates when it's interrupting. If there are any more bytes to transfer, the interrupt routine calls *xx*start to transfer the next byte. If there are no more bytes to transfer, the interrupt routine disables the interrupt (so that the device won't keep interrupting when there's nothing to do), and finishes up by calling iodone. (iodone, incidently, is another of the mechanisms provided primarily for block drivers). Here are the interrupt routines for the Skeleton driver:

```
skpoll()
{
    register struct sk_reg *sk_reg;
    int serviced, i;

    serviced = 0;
    for (i = 0; i < NSK; i++) { /* try each one */
        sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;

        /* contact the device */
        if (sk_reg->sk_csr & SK_INTR) {
            serviced = 1;
            skintr(i);
        }
    }
    return (serviced);
}
```

```
skintr(i)
    int i;
{
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;

    sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;
    sk = &skdevice[i];

    /* check for an I/O error */

    /* contact the device */
    if (sk_reg->sk_csr & SK_ERROR) {

        /* error-retry logic would go here */

        printf("skintr: I/O error0);
        sk->sk_bp->b_flags |= B_ERROR;
        goto error_return;
    }

    /* I/O transfer completed */
    if (sk->sc_count == 0) {
error_return:

        /* clear interrupt */
        sk->soft_csr = 0;

        /* contact the device */
        sk_reg->sk_csr = sk->soft_csr;
        sk->sk_busy = 0;

        /* free device to sleeping strategy routine */
        wakeup((caddr_t) sk);

        /* free buffer to waiting physio */
        iodone(sk->sk_bp);
    } else
        skstart(sk, sk_reg);
}
```

`skintr` checks the hardware for an error every time it's called, and upon finding an error, calls `printf`, flags the error in the I/O buffer and then returns. Note that:

- `skintr` needs the buffer associated with the failed transfer so that it indicate the error in its `b_flags` field.
- A retry attempt could be made before giving up and taking the error return. Whether or not this is advisable is entirely dependent on the specific device and error characteristics.

- The error return aborts the I/O request that produced the error and then places both the device and the driver in their normal idle states.

6.8. Ioctl Routine

*xx*ioctl is used to perform any tasks that can't be done by *xx*open, *xx*close, *xx*read, or *xx*write. Typical applications are: "what is the status of this device", or "go into mode X". This device is modeless and has no such special functions so we don't have an *xx*ioctl routine.

6.9. DMA Variations

Devices that are capable of doing DMA are treated differently than the Skeleton device we've been working with so far. Let's assume that we have a new version of the Skeleton board; call it the Skeleton II. It can do DMA transfers and we want to use this feature since it is much more efficient.

Multibus or VMEbus DVMA

The Sun processor board is always listening to the Multibus or VMEbus for memory references. When there is a request to read or write any address in the DVMA space (see the *Sun Main-Bus DVMA* section of the *Hardware Context* chapter for more information) the DVMA hardware adds a machine-specific offset to the address to find the location in kernel virtual memory that contains the device RAM being used in the transfer.

On Sun-2 Multibus machines, DVMA space consists of all addresses between 0 and 0x3FFFF. On Sun-2 VMEbus machines, it consists of all addresses between 0x0 and 0xFFFFF. Upon encountering one of these addresses, the DVMA hardware adds 0xF00000 to get the system virtual address of the device RAM.

On the Sun-3, the DVMA space is defined by the address range 0x0 to 0xFFFFF for 24-bit or 32-bit addressing; its system virtual address is 0xFF00000.

If you wish to do DMA over the Main Bus, you must make entries in the kernel memory map to map your device's RAM into the appropriate DVMA space. As you might expect, there are subroutines to help with this chore. mbsetup sets up the kernel memory map and mbrelse clears entries in it to release DVMA space. Note that all Sun DMA occurs between the bus and kernel virtual address space — if you wish to do DMA directly into a user buffer, you'll have to first map that buffer into kernel space, then pass it to mbsetup to map it into DVMA space.

A DMA Skeleton Driver

The addition of DMA to the capabilities of the device opens up several new options. For the moment, consider only the changes necessary to switch the driver over to DMA-style I/O. These changes turn out to be surprisingly straightforward. First we'll extend the sk_reg structure which defines the device registers. We'll assume that the Skeleton II board is a bus-master which supports 20-bit transfers, and that the following structure overlays its registers.

```
struct sk_reg {
    char sk_data;         /* 01: Data Register */
    char sk_csr;          /* 00: command(w) and status(r) */
    short sk_count;       /* bytes to be transferred */
    caddr_t sk_addr;      /* 20-bit DMA address */
};
```

Next we assume that bit 5 in the csr is set to initiate a DMA transfer.

```
#define SK_DMA  0x10     /* Do DMA transfer */
```

and a definition of the maximum DMA transfer for skminphys.

```
#define MAX_SK_BSIZE 4096    /* DMA transfer block */
```

And we must add another element to the sk_device structure for use by mbsetup and mbrelse. (The alternative would be to use the mc_mbinfo structure in the mb_ctlr structure, but since we don't use that structure for anything else, this seems more reasonable):

```
int sk_mbinfo;
```

Now we change skstrategy to use the DMA feature.

```
skstrategy(bp)
    register struct buf *bp;
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;
    int s;

    md = skdinfo[SKUNIT(bp->b_dev)];
    sk_reg = (struct sk_reg *)md->md_addr;
    sk = &skdevice[SKUNIT(bp->b_dev)];

    s = splx(pritospl(md->md_intpri));   /* begin critical section */
    while (sk->sk_busy)
        sleep((caddr_t) sk, SKPRI);
    sk->sk_busy = 1;
    sk->sk_bp = bp;

    /* this is the part that is changed */

    /* grab bus resources */
    sk->sk_mbinfo = mbsetup(md->md_hd, bp, 0);

    /* plug the remainder */
    sk_reg->sk_count = bp->b_bcount;

    /* plug bus transfer address */
    sk_reg->sk_addr = (caddr_t)MBI_ADDR(sk->sk_mbinfo);

    /* make sure we didn't overrun the address space limit */
    if (sk_reg->sk_addr > (caddr_t) 0x000FFFFF) {
        printf("sk%d: ", sk_reg->sk_addr);
        panic("exceeded 20 bit address");
    }

    sk->soft_csr = SK_ENABLE | SK_DMA;
    sk_reg->sk_csr = sk->soft_csr; /* contact the device */

    /* end of DMA-related changes */

    sk->sk_busy = 0;
    wakeup((caddr_t)sk); /* free device to sleeping strategy routine */
    (void) splx(s);       /* end critical section */
}
```

There are a number of details here that are worth noting:

□ `skstart` is no longer needed and may be completely eliminated.

□ The return value from `mbsetup` is being saved for use in calls to `MBI_ADDR` and `mbrelse`.

□ The 32-bit address returned by `MBI_ADDR` is being tested to ensure that it doesn't exceed the 20-bits limits of the device.

- □ All the I/O now is started by skstrategy and continues until skpoll is called — thus we can delete the sk_cp and sc_count fields from the sk_device structure.
- □ skintr has been simplified. There's no longer any need to check the count and sometimes call skstart. Instead, iodone is always called and physio is relied upon to proceed with the transfer.
- □ Finally, skintr needs to free up the Main Bus resources, so it'll call mbrelse.

Here are the new skintr and skpoll routines:

```
skintr(i)
    int i;
{
    register struct mb_device *md;
    register struct sk_reg2 *sk_reg;
    register struct sk_device2 *sk;

    md = (struct mb_device *)skdinfo[i];
    sk_reg = (struct sk_reg2 *)md->md_addr;
    sk = &skdevice2[i];

    /* check for an I/O error */
    if (sk_reg->sk_csr & SK_ERROR) { /* contact the device */

        /* error-retry logic would go here */

        printf("skintr: I/O error\n");
        sk->sk_bp->b_flags |= B_ERROR;
    }

    /* this is the part that changed */
    sk->soft_csr = 0;    /* clear interrupt */
    sk_reg->sk_csr = sk->soft_csr;
    mbrelse(md->md_hd, &sk->sk_mbinfo);
    iodone(sk->sk_bp);   /* free buffer to waiting physio */
}
```

```
skpoll()
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    int serviced, i;

    serviced = 0;
    for (i = 0; i < NSK; i++) {
        md = (struct mb_device *)skdinfo[i];
        sk_reg = (struct sk_reg *)md->md_addr;
        if (sk_reg->sk_csr & SK_INTR) {
            serviced = 1;
            skintr(i);
        }
    }
    return (serviced);
}
```

7

Configuring the Kernel

Configuring the Kernel **119**

7.1. Background Information 119

7.2. An Example 121

7.3. Devices that use Two Address Spaces 125

7.4. Booting Kernels on Diskless Workstations 126

7

Configuring the Kernel

7.1. Background Information

In this chapter, we'll assume that you've written and debugged your driver. The next step, obviously, is to build a kernel that includes your new driver. This process isn't difficult, Sun systems are support easy kernel configuration, even without access to system source code.

In heterogeneous server/client environments, kernels must be configured in fairly general ways. For one thing, they must work on both Multibus and VMEbus machines, for another, they have to tolerate normal variations among system devices (e.g. client Ethernet boards may be made by either 3COM or Sun). The `GENERIC` *config file thus contains configuration lines for all common devices for both bus types. However, if you're configuring a kernel for a known system, you need not carry around extraneous options — you can tailor your configuration file as appropriate and thus get a smaller (by 100 kilobytes or more!) and more efficient kernel.*

For additional information on kernel configuration, see the *Adding Hardware to Your System* section of the *System Administration Manual* and the config(8) man page. (Incidently, `config` is found in the `/usr/etc/` directory — so make sure that your path includes `/usr/etc` before proceeding).

First, a simple distinction. If your kernel already contains a certain driver, and you're simply installing a corresponding device, you'll only need to edit the kernel config file — all of the installation specific information about devices themselves is contained in this file. If, however, you'll be adding a new driver to the kernel, you'll need to edit some additional files:

- □ The first of these is `/usr/sys/sun/conf.c`, a C-language source-code file which contains the definitions of the switches `cdevsw` and `bdevsw`, as well as a bit of initialization infrastructure for the installed devices.

- □ The second is either `files.sun2` or `files.sun3` (depending upon whether your machine is a Sun-2 or a Sun-3). This file tells `config` where to find the source code for the kernel and its drivers. The pathnames (e.g. `sundev/sk.c`) in `files.sun[23]` are relative to `/sys`. (It's ok to keep driver source files in directories other than `sundev`, but if you do, put symbolic links in `sundev` so that `files.sun[23]` need contain no references to directories outside of the kernel source subtree).

The discussion in this chapter concerns `config`, *a utility program that is used in configuring kernels and initializing the kernel/driver interface structures.*

`config` *is altogether different from the autoconfiguration process, sometimes called* `autoconfig`, *which is built into the initialization pass of the UNIX kernel, and thus run at system boot time. Autoconfiguration completes the run-time driver environment initialization that* `config` *begins, for example by checking that the devices indicated as present in the kernel config file are actually present in the running system. Autoconfiguration is discussed in much greater detail in the Overall Kernel Context section of this manual.*

`config`'s goal is to output a set of files that can be directly used to configure a new kernel. The purpose of the configuration may simply be to install a device (for which the kernel already contains a driver) or it may be to integrate a new device and its driver. The kernel configuration system learns of new devices by way entries in the config file, whereas new drivers are indicated by editing one or all of the files `conf.c`, `files` and `files.sun[23]`. The files output by `config` are used in the construction of the new kernel, but so are others, notably `conf.c` itself.

- `ioconf.c` — the major input to the autoconfiguration process. It contains arrays of *mbvar* structures — `struct mb_ctlr mbcinit[]` and `struct mb_device mbdinit[]` — that have been initialized on the basis of the device and controller information in the config file. (Incidently, the order of the device declarations in the config file will determine the order of the structures in `ioconf.c`, and thus the order in which devices are polled). The autoconfiguration process assumes that `ioconf.c` exists and will complete the initialization of its structures by calling *xx*`probe`, *xx*`attach`, and *xx*`slave`. See the *Overall Kernel Context* chapter for more information.

- *xx*.h — a set of header files, one for each driver. These header files define macros (e.g. `#define NSK 2`) that tell the drivers how many devices they'll be managing. The drivers will use these macros *at compile time* to control conditional compilation and to size device tables. *When developing and testing a driver, you should make yourself a fake header file;* `config` *won't have a chance to make one for you until you're actually installing your driver.*

- *mbglue.s* — contains assembly-level code that translates from the hardware interrupt mechanisms to the device-interrupt routines for the installed devices.

- *makefile* — a makefile that, when executed, will actually make the new kernel, compiling and linking files as necessary. Note that the entries in `files.sun[23]` refer to source files (i.e. `sundev/sk.c`), but that if `config` fails to find a named source file it will set up to use the corresponding object file (from the OBJ subdirectory of the configuration directory) instead. Thus, `config` works on both source licensed and object licensed machines.

7.2. An Example

The example that follows assumes that you're adding a driver for the Skeleton board (`sk.c`) to your system. To proceed, you'll need a configuration directory and a config file for your new kernel. Let's say they're both named `SKELETON`. So the first step is to create the configuration file and directory:

```
example#  cd /sys/conf
example#  cp GENERIC SKELETON
example#  mkdir ../SKELETON
```

Then edit the `SKELETON` config file to reflect the presence, in your system, of the Skeleton board. As you can see by checking config(8), each line in the file describes a different device — thus, you'll simply need to add lines that describe the installation of the Skeleton board. The exact format of those lines will depend upon the address space within which the board is to be installed.

The address space that's given in the kernel config file will determine the address-space mappings that are set up by the MMU — the virtual addresses that the driver receives from the kernel, and then treats as pointers to the device's registers, will be within the address space given here. What's important is that the driver writer know and specify, at this point, the number of bits in the device address, and the number of bits in its data-access length.

The Skeleton board, as we've defined it, is an I/O-mapped Multibus device with an eight-bit status and an eight-bit data register. Thus, in a Sun-2 Multibus machine, it would be installed in I/O space; if we put it at offset `0x600` within that space, we'd add the following line to `SKELETON`:

```
device sk0 at mbio ? csr 0x600 priority 2
```

This says that we have a `sk` device (the first device is always, by convention, number 0) on the Multibus, which the control/status register (device register) is at Multibus I/O address 0x600 (this is passed to *xx*`probe` at boot time), and that the device will interrupt at level 2.

If our machine is a Sun-2 VMEbus machine, we'll install the Skeleton device within `vme16d16` by way of a Multibus-VMEbus adapter. We choose `vme16d16` because it's the smallest address space:

```
device sk0 at vme16d16 ? csr 0x600 priority 2 vector skintr 0xC8
```

This says that, when plugged into an adapter board, the vector number `0xC8` is set up to route to the `skintr` routine. (Vector numbers `0xC8` through `0xFF` are reserved for user devices). Notice that `0x600` within `mbio` maps directly to `0x600` within `vme16d16`.

On a Sun-3, it would likewise be reasonable to choose the smallest of the available address spaces:

Each of these config-file entries specify the installation of the Skeleton device for either a Multibus or a VMEbus systems. It's fine for one config file to contain both entries — `config` will know the type of system that it is running on, and automatically use the right entry.

Only very rudimentary error checking is done on the config file. For example, if you declare a device attached to a controller, you must declare the controller as well. Also, a sanity check is done on the timezone and date entries. The checking, however, is not comprehensive.

One more point about the config file. During the development process, you probably maintained a file named `sk.h`, which indicated to your driver how many devices it had to manage. Now, the number of installed devices will be determined, for each driver, by `config`, and it will generate the header file for you.

Now, you can go on with the process of building the new kernel. The next step is to edit `conf.c`, adding to it the names of the entry point routines for the Skeleton driver, and then installing those routines into the kernel's character device switch `cdevsw`. The following code accomplishes these two purposes:

```
#include "sk.h"
#if NSK > 0
int skopen(), skclose(), skread(), skwrite(), skmmap();
#else
#define skopen  nodev
#define skclose nodev
#define skread  nodev
#define skwrite nodev
#define skmmap  nodev
#endif
    .
    .
    .
struct cdevsw cdevsw[] =
{
    .
    .
    {
skopen, skclose, skread, skwrite,
nodev, nodev, nodev, 0,
seltrue, skmmap,
    },
    .
    .
}
```

This will add the driver's routines to `cdevsw` if `NSK` is greater than 0 (`NSK` is, as already explained, calculated by `config`). Note well that the position in the `cdevsw` where we've installed our routines (the exact position depends, of course, upon how many device are already installed) is the same as the major device number which we'll later assign to all devices driven by this driver — the major number is an index into `cdevsw`.

The entries in `cdevsw` are, in order, *xx*`open`, *xx*`close`, *xx*`read`, *xx*`write`, *xx*`ioctl`, *xx*`stop` and *xx*`reset`, a `tty` structure pointer, and finally, *xx*`select` and *xx*`mmap`. The Skeleton driver doesn't have an *xx*`ioctl` routine so this entry is set to `nodev`, the special routine that always returns an error. Since our device is not a tty it doesn't have a *xx*`stop` routine (used for flow flow control) nor does it have a tty structure. *xx*reset is ***never*** used so all devices set its entry to nodev. *xx*select is called when a user process does a select(2) system call; it returns 1 if the device can be immediately selected. Since the Skeleton device is write only and arbitrarily fast, it's always selectable — so we'll use the default seltrue routine that always returns 1.

The next step is to edit the file that tells config how to locate the driver source code. This source code will *not* be common to all Sun systems, and thus its pathname will go not into files but into files.sun[23]. Assuming that the driver source has been moved into /sys/sundev, here's the line you must add to files.sun[23]:

```
    sundev/sk.c optional sk device-driver
```

This says that the file sundev/sk.c contains the source code for the optional *sk* device and that it is a device driver.

After adding these lines to your configuration file, you can run config:

```
    example#  config SKELETON
```

config uses SKELETON, files, and files.sun as input, and generates a number of files in the ../SKELETON directory. One of these files is the makefile that contains a dependency tree for any new C source files you created during the process of adding new drivers (or whatever) to the kernel. make will use this as its command file when it is actually executed to produce the new kernel. When config starts generating the makefile it will notify you with the message:

```
 Doing a "make depend"
```

Now you can change directory to the new configuration directory,

```
    example#  cd ../SKELETON
    example#  make
```

Now you must add a new device entry to the /dev directory. The connections between the UNIX operating system kernel and the device driver are established through the entries in the /dev directory. Using the example above as our model, we want to install the device for the Skeleton driver.

Device entries are made with one of two shell scripts in the /dev directory. The first, MAKEDEV, is for standard system devices and should be left as is. The second, MAKEDEV.local, differs only in that it contains entries for user devices, and it is here that entries for new devices should be placed.

It's worth looking inside MAKEDEV to see the kinds of things it does. The lines of shell script below reflect what you'd add to MAKEDEV.local for the new Skeleton device. First, there are some lines of commentary:

```
#! /bin/sh
#   MAKEDEV.local   4.45    86/04/15
# Graphics
#   sk* Skeleton Board
```

Then there's the actual shell code that makes the device entries:

```
sk*)
    unit=`expr $i : 'sk)'`
    /etc/mknod sk$unit c 40 $unit
    chmod 222 sk$unit
    ;;
```

This code extracts the numeric portion of `MAKEDEV.local`'s argument and passes it on to `mknod` and `chmod`. In the simplest case, we simply say:

```
example#  MAKEDEV.local sk0
```

`MAKEDEV.local` then makes the special inode `/dev/sk0` for a character special device with major device number 40 and minor device number 0, and then sets the mode of the file so that anyone can write to the device.

Having added the new device entry, you can install the new system and try it out.

```
example#  cp /sys/SKELETON/vmunix /vmunix+
example#  halt
              The system here goes through the halt sequence, then
              the monitor displays its prompt, at which point you can
              boot the system in single-user state
>   b vmunix+ -s
              The system boots up in single user state and
              then you can try things out
example#
```

If the system appears to work, save the old kernel under a different name and install the new one in /vmunix:

```
example#  cd /
example#  mv vmunix vmunix-
example#  mv vmunix+ vmunix
example#
```

Make sure that the new version of the kernel is actually called `vmunix` because programs like `ps` and `netstat` use that exact name in collecting information they need from runtime tables. If the running version of the kernel is called something other than `vmunix` the results from such programs will be wrong.

7.3. Devices that use Two Address Spaces

Normally, devices interface to the system by way of a single address space. However, there are exceptions. Some Multibus devices have registers in Multibus I/O space *and* memory in Multibus memory space. And there are any number of VMEbus devices coming on the market that have memory in 24 or 32-bit VME space while keeping their control and status registers in 16, or even 8-bit, VME space.

Unfortunately, such situations can't currently be handled in a clean fashion because the kernel configuration program (config) can't cope with dual-space devices. The *xx*probe routine is the core of the problem, since it deals with only a single space.

There are, fortunately, two ways to work around the problem:

- The first is easier, but rather inelegant. It consists of treating the device as if it were two devices, and of writing two separate "drivers" for it. So, for example, if we were to have an new, dual-space, VMEbus version of the Skeleton device, we'd add the following *two* lines to the config file:

```
# Skeleton Memory Space
device skm0 at vme32d32 ? csr 0xD0000000 priority 3
# Skeleton Register Space
device skr0 at vme16d16 ? csr 0xD000 priority 3 vector skintr 0x88
```

It's also necessary to have two entries in files.sun:

```
sundev/skm.c        optional skm device-driver
sundev/skr.c        optional skr device-driver
```

And it's necessary to have a second "driver". Actually, all of the real driver code goes into skr.c, which manipulates the device registers. The second driver, skm.c, consists entirely of a probe routine — all its other routines are null.

Both sides of the driver, skr.c and skm.c, include the same register header file skreg.h. skreg.h contains an *external* declaration for an array of structures (one for each instance of the device) that contain whatever information skr.c needs from the memory-side probe routine:

```
    extern struct sk_devinfo sk_devinfo[NSK];
```

All that remains is for the memory-side probe routine to initialize sk_devinfo.

- There's a second procedure for installing dual-space devices. It's a bit harder to use, but it doesn't require a stub driver containing only a probe routine.

  Pick one of the two device installation addresses for normal treatment in the config file. It doesn't matter which one you pick, unless the device is a memory-mapped Multibus device, in which case you must pick the address in Multibus Memory space. Otherwise just pick the one that's most convenient for your *xx*probe routine to use to test the device installation. The registers and memory in this first space will then be automatically mapped into kernel virtual space (as usual) by the autoconfiguration process.

Then use the config file `flags` word to communicate the second space installation address to your driver. The driver will then find that address in `md–>md_flags` and be able to access it from either the *xx*`attach` or *xx*`slave` routine; it's best (for most character devices) to pick it up at *xx*`attach` time. It'll then be necessary to use `rmalloc` to allocate (from `kernelmap`) virtual space for the second-space registers/memory and to call `mapin` to map them into kernel space. (It's the details of calling `mapin` that make this approach to dual-space installation more difficult than the first. These details are *not* covered in this manual, though you will find an example call to `mapin` in the Sun-2 Color Graphics Driver listed in the appendix).

7.4. Booting Kernels on Diskless Workstations

If you're working on a diskless workstation, it's still possible (though awkward) to develop a driver, link it into a kernel and then boot that kernel on your server.

□ On Sun-2 machines, this is possible because you can boot a program (in this case the new kernel) which is on your private root `nd` partition. To do so, give the following command:

```
> b xx(0,0,40) program_name
```

□ Where **xx** indicates either `ie`, `ec`, or (on a Sun-3) `le`, depending upon which of the Ethernet devices is installed on the diskless workstation.

On Sun-3 machines you can do the same boot by using `tftp`. See the boot(8s) man page for more details.

A

Using the Sun CPU PROM Monitor

Using the Sun CPU PROM Monitor **129**

A.1. PROM Monitor Command Syntax 129

A.2. PROM Monitor Syntax for Memory and Register Access 129

A.3. PROM Monitor Command Descriptions 131

A

Using the Sun CPU PROM Monitor

A.1. PROM Monitor Command Syntax

The monitor understands commands in quite a simple format. The format is:

< *verb* >< *space* >*[< *argument* >]< *return* >

< *verb* > is always one alphabetic character; case does not matter.

< *space* >* means that any number of spaces are skipped here.

< *argument* > is normally a hexadecimal number or a single letter; again, case does not matter. Square brackets '[ ]' indicate that the argument portion is optional.

< *return* > means that you should press the carriage-return key.

When typing commands, < *backspace* > and < *delete* > (also called < *rubout* >, generated by the key labelled < *backtab* > on the non-VT100 Sun keyboard) erase one character; control-U erases the entire line.

A.2. PROM Monitor Syntax for Memory and Register Access

Several of the commands *open* a memory location, map register, or processor register, so that you can examine and/or modify the contents of the specified location. These commands include `a`, `d`, `e`, `l`, `m`, `o`, `p`, and `r`.

Each of these commands takes the form of a command letter, possibly followed by a hexadecimal memory address or register number, followed by a sequence of zero or more 'action specifier' arguments. The various options are illustrated below, using the `e` command as an example. You type the parts as shown in **`bold typewriter font`**, with a < *return* > at the end of each command.

If no action specifier arguments are present, the address or register name is displayed along with its current contents. You may then type a new hexadecimal value, or simply < *return* > to go on the next address or register. Typing any non-hex character and < *return* > gets you back to command level. For registers, 'next' means within the sequence of registers:

D0-D7	the data registers
A0-A6	the address registers
SS	the system stack pointer (Note that Sun-3 machines, being based upon MC68020 processors, actually have three stack pointers: USP is the user stack pointer, ISP is the interrupt stack pointer and MSP is the master stack pointer. In general, SS is equivalent to ISP, though this can change as SR changes).

US	the user stack pointer
SF	the source function code register
DF	the destination function code register
VB	the vector base register
SC	the system context register
UC	the user context register
SR	the status register
PC	the program counter

For example, the following command sets consecutive locations 0x1234 and 0x1236 to the values 0x5678 and 0x0000 respectively:

```
> e1234
001234: 007F? 5678
001236: 51A4? 0
001238: C022? q
>
```

A non-hex character (such as question mark) on the command line means read-only:

```
> e1000 ?
001000: 007F
>
```

Multiple nonhex characters read multiple locations:

```
> e1000 ???
001000: 007F
001002: 0064
001004: 1234
>
```

A hex number on the command line does store-only:

```
> e1000 4567
001000 -> 4567
>
```

Multiple hex writes multiple locations:

```
> e1000 1 2 3
001000 -> 0001
001002 -> 0002
001004 -> 0003
>
```

Nonhex followed by hex reads, then stores.

```
> e1000 ? 346
001000: 007F -> 0346
>
```

Finally, reads and writes can be interspersed:

```
> e1000 ? 1 ? ? 3 4
001000: 007F -> 0001
001002: 0064
001004: 1234 -> 0003
001006 -> 0004
>
```

Spaces are optional except between two consecutive numbers. When actions are specified on the command line after the address, no further input is taken from the keyboard for that command; after executing the specified actions, a new command is prompted for. Note especially that these commands provide the ability to write to a location (such as an I/O register) without first reading from it.

A.3. PROM Monitor Command Descriptions

In the descriptions listed below, the command letters in `typewriter text` are the commands, and things in *italic font* represent things that you substitute. Things in brackets are optional.

`A` [*n*][*actions*] Open A-register *n* (0≤*n*≤7, default zero). A7 is the System Stack Pointer; to see the User Stack Pointer, use the **`r`** command. For further explanation, see *PROM Monitor Syntax for Memory and Register Access* above.

`B` [!][*args*] Boot. Resets appropriate parts of the system, then bootstraps the system. This allows bootstrap loading of programs from various devices such as disk, tape, or Ethernet. Typing 'b?' lists all possible boot devices. Simply typing 'b' gives you a default boot, which is configuration dependent. For an explanation of the booting options, see *System Administration for the Sun Workstation.*

If the first character of the argument is a '!', the system is not reset, and the bootstrapped program is not automatically executed. To execute it, use the 'C' command described below.

`C` [*addr*] Continue a program. The address *addr*, if given, is the address at which execution will begin; default is the current PC. The registers will be restored to the values shown by the A, D, and R commands.

`D` [*n*][*actions*] Open D-register *n* (0≤*n*≤7, default zero). For a detailed explanation, see *PROM Monitor Syntax for Memory and Register Access* above.

`E` [*addr*][*actions*] Open the word at memory address *addr* (default zero) in the address space defined by the 'S' command. For a detailed explanation, see *PROM Monitor Syntax for Memory and Register Access* above.

`F` [*address_1*][*address_2*][*data*][*size*]
Sun-3 only. Fill address space from the lower address specified by *address_1*, up to and including the higher address specified by *address_2*, with the constant *data*, of

size b, w, or l, where b specifies a byte pattern, w specifies a word (16-bit) pattern, and l specifies a long word (32-bit) pattern. l (for long word) size is used if the *size* parameter is not specified.

G [*addr*][*param*] Start the program by executing a subroutine call to the address *addr* if given, or else to the current PC. The values of the address and data registers are undefined; the status register will contain 0x2700. One parameter is passed to the subroutine on the stack; it is the address of the remainder of the command line following the last digit of *addr* (and possible blanks).

Here are some notes on the monitor's g command when UNIX is in memory:

> Typing g0 gets a
>
> ```
> panic: zero dump.
> ```
>
> Typing g4 dumps the kernel stack, giving the saved program counter and framepointer for each procedure on the kernel stack as well as the first four arguments passed to these procedures. This stack trace looks very similar to the the one you get from UNIX when certain nasty conditions rear their ugly head. This tracing facility can be used at any time without destroying any system state, so UNIX can be continued.

H *Sun-3 only*. Display a menu of monitor commands and their descriptions.

K [*number*] If *number* is 0 (or not given), this does a 'Reset Instruction': it resets the system without affecting main memory or maps. If *number* is 1, this does a 'Medium Reset', which re-initializes most of the system without clearing memory. If *number* is 2, a hard reset is performed and memory is cleared. This causes the PROM-based diagnostics to be run, a process which can take several minutes.

L [*addr*][*actions*] Open the longword at memory address *addr* (default zero) in the address space defined by the 'S' command. For a detailed explanation, see *PROM Monitor Syntax for Memory and Register Access* above.

M [*addr*] [*actions*] Opens the Segment Map entry which maps virtual address *addr* (default zero) in the current context. The choice of supervisor or user context is determined by the 'S' command setting (0–3 = user; 4–7 = supervisor). See *PROM Monitor Syntax for Memory and Register Access* above.

O [*addr*][*actions*] Opens the byte location specified (default zero) in the address space defined by the 'S' command. See *PROM Monitor Syntax for Memory and Register Access* above.

`P` [*addr*] [*actions*] Opens the Page Map entry which maps virtual address *addr* (default zero) in the current context. The choice of supervisor or user context is determined by the 'S' command setting (0–3 = user; 4–7 = supervisor). With each page map entry, the relevant segment map entry is displayed in brackets. See *PROM Monitor Syntax for Memory and Register Access* above.

`Q` [*addr*] [*actions*] *Sun-3 only*. Opens the EEPROM address specified by *addr* (default zero) in the EEPROM address space. All addresses are referenced relative to the base of the EEPROM in physical address space. The monitor checks to ensure that specified addresses are within the bounds of the EEPROM physical address space. This command is used to examine or modify configuration parameters specifying things such as the amount of memory to test during self-test, whether to display a standard banner or a custom banner, if a serial port (A or B) is to be used for the system console, and so on. See *PROM Monitor Syntax for Memory and Register Access* above.

`R` [*actions*] Opens the miscellaneous registers (in order): SS (Supervisor Stack Pointer), US (User Stack Pointer), SF (Source Function Code), DF (Destination Function Code), VB (Vector Base), SC (System Context), UC (User Context), SR (Status Register), and PC (Program Counter). Alterations made to these registers (except SC and UC) do not take effect until the next 'C' command. For further explanation, see *PROM Monitor Syntax for Memory and Register Access* above.

`S` [*number*] Sets or queries the address space to be used by subsequent memory access commands by setting the function code register in the CPU. The value in this register determines what address space is to be used in the subsequent decoding of addresses. *number* is the function code to be used, ranging from 1 to 7. The most useful values are 1 (user data), 2 (user program), 3 (memory maps), 5 (supervisor data), and 6 (supervisor program). If no *number* is supplied, the current setting is printed. Upon entry into the monitor, this is set to 5 if the program was in supervisor state, or to 1 if the program was in user state.

`U` [*arg*]

The `U` command manipulates the serial ports and switches the current input or output device. The argument may have the following values ('`{AB}`' means that either 'A' or 'B' is specified):

`{AB}`	Select serial port A (or B) as input and output device
`{AB}io`	Select serial port A (or B) as input and output device
`{AB}i`	Select serial port A (or B) for input only
`{AB}o`	Select serial port A (or B) for output only
`k`	Select keyboard for input
`ki`	Select keyboard for input
`s`	Select screen for output
`so`	Select screen for output
`ks, sk`	Select keyboard for input and screen for output
`{AB}#`	Set speed of serial port A (or B) to # (such as 1200, 9600, ...)
`e`	Echo input to output
`ne`	Don't echo input to output
`u` *addr*	Set virtual serial port address

If no argument is specified, the `U` command reports the current values of the settings. If no serial port is specified when changing speeds, the 'current' input device is changed.

At power-up, the following default settings are used: The default console input device is the Sun keyboard or, if the keyboard is unavailable, serial port A. The default console output device is the Sun screen or, if the graphics board is unavailable, serial port A. All serial ports are set to 9600 baud.

`V` [*address_1*][*address_2*][*size*]

Sun-3 only. Display the contents of addresses from the lower address specified by *address_1*, up to and including the higher address specified by *address_2*, in a format specified by *size*, where *size* is one of `b`, `w`, or `l`, where `b` specifies byte format, `w` specifies word (16-bit) format, and `l` specifies long word (32-bit) format. `l` (for long word) size is used if the *size* parameter is not specified. Type a RETURN character to suspend the display for viewing; type another RETURN character to restart the display. Press the SPACE bar to quit displaying and return to the monitor command mode.

`W`

Sun-3 only. Allows a subroutine to be called from the monitor. The first argument is the address of the routine, while subsequent arguments are interpreted as parameters.

`X`

Sun-3 only. Present a menu of extended boot-path tests.

B

Summary of Device Driver Routines

Summary of Device Driver Routines **137**

B.1. Standard Error Numbers 137

B.2. Device Driver Routines 137

*xx***attach** — Attach a Slave Device 137

*xx***close** — Close a Device 138

*xx***intr** — Handle Vectored Interrupts 138

*xx***ioctl** — Miscellaneous I/O Control 139

*xx***mmap** — Mmap a Page of Memory 140

*xx***minphys** — Determine Maximum Block Size 141

*xx***open** — Open a Device for Data Transfers 141

*xx***poll** — Handle Polling Interrupts 142

*xx***probe** — Determine if Hardware is There 142

*xx***read** — Read Data from Device 143

*xx***strategy** Routine 143

*xx***write** — Write Data to Device 144

B

Summary of Device Driver Routines

B.1. Standard Error Numbers

The system has a collection of standard error numbers that a driver can return to its callers. These numbers are described in detail in intro(2), the introductory pages of the *System Interface Manual.* A complete listing of the error numbers appears in `<sys/errno.h>`.

B.2. Device Driver Routines

These routines actually compose the bulk of the device driver. Some of them, like *xx*`ioctl`, are optional. Others, like *xx*`probe`, must appear in every driver. Omitted from this section is the *xx*`slave` routine, which appears only in block-device drivers.

When a user program makes a system call that involves I/O devices, it's translated by the kernel into a call to the appropriate driver routine. However, when that driver routine is called, its parameters are no longer the same as the parameters that the user program passed to the system call — they will have been translated into parameters reflecting the actual run-time environment of the drivers, an environment set up and initialized by `config` and the autoconfiguration process and then maintained by the kernel and the drivers themselves. For example, a user program will call

```
write (fileno, address, nbytes)
    int fileno;
    char *address;
    int nbytes;
```

but the kernel will translate this into

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

by the time it calls the driver's *xx*`write` routine.

xx`attach` — Attach a Slave Device

```
xxattach(md)
    struct mb_device *md;
```

xx`attach` does boot-time, device-specific setup and initialization. It's commonly used in disk and tape drivers for setup tasks like reading labels, and in character drivers for the initialization of interrupt vectors and the reserving of blocks of memory. Its proper tasks are not limited to the initialization of actual hardware devices — *xx*`attach` is also used to set up and initialize local data structures.

When it needs to set a device interrupt-vector number, *xx*`attach` finds it in the `md_intr->v_vec` field of the `mb_device` structure. A `NULL` value in this field indicates that the host machine is Multibus based and does not support vectored interrupts. On VMEbus machines `md_intr->v_vec` is the interrupt-vector number given for the device in the kernel config file and *must* be present.

xx`attach` can also be used to set the 32-bit argument that's subsequently passed to *xx*`intr`. This argument (contained in `md->v_vptr`) is initially set to the vector number of the interrupting device, but it's often convenient to reset it to contain a pointer to a local structure.

xx`close` — Close a Device

```
xxclose(dev, flags)
    dev_t dev;
    int flags;
```

xx`close` does whatever it has to do to indicate that data transfers can't be made on the device until it's been reopened. This may involve nothing at all, or it may include resetting and quieting the device, flushing data buffers, and releasing or unlocking resources (or unlocking the device itself if it's opened exclusively). Since *xx*`close` is called only when the *last* user process which is using the device closes it, *xx*`close` must clean up for *all* user processes which have had the device open. *xx*`close` doesn't need to report an error, although it can. *flags*, incidently, is the same as it is for *xx*`open`.

xx`intr` — Handle Vectored Interrupts

```
xxintr(ctrl_num)
    int ctrl_num;
```

xx`intr` is responsible for fielding vectored interrupts from the device. As such, it is specified (with its interrupt vector) in the kernel config file. As an interrupt routine, *xx*`intr` (and any routines that it calls) is prohibited from calling `sleep`.

xx`intr` receives one 32-bit parameter, which is, by default, the vector number of the device that interrupted. However, you can arrange for it to receive something else by changing the value in `md->v_vptr`. (See *xx*`attach`, above).

In character drivers which, like block drivers, make use of `physio` and its associated structures, mechanisms and routines, *xx*`intr` is used to indicate when the device is finished with one chunk and ready for the next. *xx*`intr` is also responsible for error handling and reporting. More specifically:

- *xx*`intr` should check the device for an error every time it's called. It can also check the driver state against the device state to ensure that the device is, in fact, doing what the driver expects it to be doing. Upon finding an "impossible" or unrecoverable error, *xx*`intr` should `panic`. But for regular errors it should call `printf` (or `uprintf`), flag the error in the I/O buffer, and then return.

- The error is flagged by setting the B_ERROR bits in the the buffer b_flags field (and, if an error code other that EIO is desired, by assigning that error code into the buffer b_error field). The error code will then be propagated up the the user by way of physio. physio checks to see if the error flag has been set in the buffer, and if it has, passes the error code up to the user program, which usually plugs it into the global error register errno. *intr* doesn't itself return anything.
- A retry attempt can be made before giving up and taking the error return. Whether or not this is advisable is entirely dependent on the specific device and error characteristics.
- The error return should abort the I/O request that produced the error and then place both the device its driver in their normal idle states.

*xx*ioctl — Miscellaneous I/O Control

```
xxioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
```

The device-driver entry routines, taken as a set, are intended to constitute a uniform abstract interface capable of accommodating all possible I/O devices. Obviously, such devices differ greatly, and thus the need for this routine — *xx*ioctl is the escape mechanism by which miscellaneous operations are accommodated.

These functions vary greatly — almost anything is possible. The range of possibilities requires a very general interface, and *xx*ioctl has one. The *cmd* variable identifies a specific device control operation, and is typically used by *xx*ioctl as the index into a switch statement. The *data* parameter is the real escape hatch, a pointer to an array up to 127 bytes in length. This array, over which the driver and its users will overlay a driver-specific structure, can be treated as both an input parameter by which user programs send data to the driver and as an output parameter by which the driver returns data to its users. *flag* is set to the f_flags field of the file structure. The file structure, together with the file-mode flags to which its f_flags field can be set (FREAD, FWRITE, and so on) is defined in <sys/file.h>. The driver is free to use *flag* to make its operation sensitive to the manner in which the file was opened by the user.

In <sys/ioctl.h> will be found a collection of macros which encode parameter size and read/write control information into ioctl command codes. These macros tell the kernel, on a command by command basis:

- How many of the maximum of 127 bytes in the ioctl parameter are significant when that parameter is read.
- How many of these bytes are significant when the parameter is written.
- If the parameter bytes should be read into kernel space before calling *xx*ioctl.

- □ If they should be read into user space after calling *xx*ioctl.

The Versatec Interface driver in the *Sample Driver Listings* appendix of this manual contains some simple examples of the use of these ioctl macros. (More complex examples can be found in <sys/ioctl.h>). The Versatec Interface driver defines two ioctl command codes:

```
#define    VGETSTATE _IOR(v, 0, int)
#define    VSETSTATE _IOW(v, 1, int)
```

The first parameter of the ioctl macros is an ASCII character that serves to group together each driver's command codes. It must be different for each device — in this case, it's "v" for "Versatec". The second parameter is the command code itself. The third is the size of the ioctl argument, which cannot exceed 127 bytes. Note that the size is given as the name of the structure which will be used to interpret the parameter array. The macros _IOR, _IOW and _IOWR then use the sizeof operator to determine the number of bytes consumed by the structure.

The definitions of such ioctl-related structures, together with the command-code definitions themselves, must be collected into a user accessible include file. Such include files are usually, though not necessarily, kept in /usr/include/sys.

When the kernel processes the ioctl system call, translating its parameters into the terms appropriate to a *xx*ioctl driver routine, it consults the read/write encode bits in the command code. If the read bit is set, then the argument is read into an buffer in kernel space, and a pointer to that buffer is passed to the driver ioctl routine. Likewise, if the write bit is set, the argument is copied back into user space after command execution is completed.

*xx*ioctl does whatever it has to do, then returns 0 if there were no errors, an error code if there were. ENOTTY is the code used if the requested command did not apply to the device. The kernel passes error codes up to the user program, which usually plugs them into errno.

*xx*mmap — Mmap a Page of Memory

```
xxmmap(dev, off, protection)
    dev_t dev;
    off_t off;
    int protection;
```

*xx*mmap is called for PTE information about the page (at offset off) of *dev*'s memory. (This information is what the kernel needs to map the page to a virtual address). *xx*mmap should first check that off doesn't exceed the device-memory size:

```
if (off >= XXSIZE)
    return (-1);
```

for this would cause the mapping of an area greater than the device memory. *xx*mmap returns a subset of the page table entry (PTE) containing the page frame number and the page type to its caller in the kernel (it selects out these two fields by masking the PTE with PG_PFNUM). *xx*mmap is called iteratively to perform a mapping requested by a call to mmap — the looping and all of its bookkeeping, as well as the actual mapping, is performed by the kernel in a way that's transparent to the driver.

On Sun-3/260 and Sun-3/280 systems, *xx*mmap should remove kernel pages from the virtual-address cache. It does this by calling vac_disable_kpage for every address it's called to remap. For example:

```
. . .
page = getkpgmap(addr + off) & PG_PFNUM;
vac_disable_kpage(addr + off);
. . .
```

*xx*mmap returns -1 to the kernel if it can't do the mapping, otherwise it returns its PTE subset. Upon receipt of a -1, the kernel returns the error code EINVAL (Illegal argument) to the user program, where it's usually plugged into the global error variable errno.

*xx***minphys** — Determine Maximum Block Size

```
unsigned xxminphys(bp)
    register struct buf *bp;
```

*xx*minphys determines a "reasonable" block size for transfers, so as to avoid tying up too many resources. *xx*minphys is passed as an argument to physio. The system version of the *xx*minphys function, minphys, may be used by any driver. *xx*minphys should perform the calculation:

```
int block; /* some reasonable block sixe for transfers */

if (bp->b_bcount > block)
    bp->b_bcount = block;
```

*xx***open** — Open a Device for Data Transfers

```
xxopen(dev, flags)
    dev_t dev;
    int flags;
```

*xx*open is called each time the device is opened, and may include any device-specific initialization. Typically, it will:

- begin by validating the minor device number and doing other device-specific error checking.
- Then if everything is ok, it will initialize the device (for example by clearing registers, enabling interrupts or checking for power-up errors) and possibly the local data structures. This structure initialization may include locking the device if it's exclusive use, or allocating driver resources — for example allocating dynamic buffers that'll be needed later).
- Finally, *xx*open will typically wait for the device to come on-line, and return an error if it doesn't.

The integer argument *flags* indicates if the open is for reading, writing, or for both. The constants FREAD and FWRITE (from <sys/file.h>) are available to be AND'ed with *flags*.

The minor device number encoded in *dev* is of concern only to the device driver itself. It can itself be encoded to contain various kinds of information, as needed by the driver. The driver developer will want to provide macros to break out encoded subfields. *dev* may encode a unit or driver number, a special feature, or an operating mode.

xx`open` returns `ENXIO` (No such device or address) if the minor device number is out of range, `ENODEV` (No such device) if an attempt was made to open the device with an inappropriate mode or `EIO` (I/O Error) to indicate an I/O error in the course of an attempted initialization. If the open is successful, *xx*`open` returns 0. The kernel will return the error code to the user program, where it is usually plugged into the global error variable `errno`.

xx`poll` — Handle Polling Interrupts

```
xxpoll()
```

xx`poll` is responsible for fielding non-vectored interrupts from the device. In situations where multiple devices share the same interrupt level, *xx*`poll` must determine if the interrupt was actually destined for this driver or not. *xx*`poll` returns 0 to indicate that the interrupt was not serviced by this driver, and non-zero to indicate that the interrupt was serviced. It is a gross error for *xx*`poll` to say that it serviced an interrupt when it did not.

If a device driver handles both vectored interrupts and polling interrupts, *xx*`poll` typically calls the *xx*`intr` routine with the proper arguments, normally the unit number of the device that interrupted. `sleep` may never be called from *xx*`poll`, or, for that matter, from any of the lower-half routines.

xx`probe` — Determine if Hardware is There

```
xxprobe(reg, unit)
    caddr_t reg;
    int unit;
```

xx`probe` determines whether the device at the kernel virtual address *reg* actually exists and is the correct device for this driver. The method by which it accomplishes this is impossible to standardize, for devices provide no uniform means of identification. Indeed, some devices fail to provide even reasonable non-standard means of identification.

The kernel provides a set of function to help with probing. These functions can probe an address, recover from the bus error that will occur if no device is installed at that address, and return with an indication as to whether such a bus error occurred. These functions are `peek`, `peekc`, `poke`, and `pokec`.

It's possible for `probe` to check the value of the *reg* parameter to ensure that the device isn't installed at an address that it can't itself address. The device's entry in the kernel config file determines which address space it's mapped into, but it's sometimes possible for the device itself to be configured differently. The driver can check, for example, that *reg* doesn't contain an address greater that `0xFFFFF` (that is, an address with more than 20 significant bits) if the device is configured for 20-bit references.

It's also possible for *xx*probe to do some device initialization, even though such initialization is properly the job of *xx*attach. This can make sense if such initialization allows *xx*probe to identify and verify the device, but it should only do the amount of initialization necessary to determine if the device is really there. It definitely should not allocate any memory that won't be used if the device isn't found, and it should not assume that just because it found a device that the system will choose to include that device in its configuration

If the correct device is found at the probed location, *xx*probe returns (sizeof (struct *xx*device)). (This is the size of the device registers in I/O space if the device is an I/O mapped Multibus device; otherwise it's the size of the device registers in memory space). If no device is found at the expected location, or if the device found is not the one that was expected, *xx*probe returns a 0. If it doesn't, the kernel will be incorrectly led to believe that a device is present, and future attempts to contact it will cause the kernel to panic with a bus error.

*xx***read** — Read Data from Device

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

*xx*read is the high-level routine called (in character device drivers) to perform data transfers from the device. *xx*read must check that the minor device number passed to it is in range. If the minor device number is out of range, *xx*read returns like so:

```
    if (XXUNIT(dev) >= NXX)
        return (ENXIO);
```

Subsequent actions of *xx*read differ depending on whether the device is a tty-style character-at-a-time device or a device that buffers its I/O into blocks.

For block transfers, *xx*read uses physio, its associated mechanisms, and the *xx*strategy. buf is here an array of locally declared buffers:

```
    return (physio(xxstrategy, &buf[minor(dev)],
        dev, B_READ, minphys, uio));
```

If *xx*strategy fails, *xx*read passes its error code up to the kernel. The kernel then passes it on to the user program, which usually plugs it into the global error variable errno.

*xx***strategy** Routine

```
xxstrategy(bp)
    register struct buf *bp;
```

*xx*strategy is a high-level I/O routine designed to be called from physio. Its name derives from its role in block-device drivers, where *xx*strategy has responsibility for reordering the I/O request queue so as to increase the overall I/O bandwidth. In character devices (even those which queue I/O) such reordering is to no advantage, and *xx*strategy's major function is structural. It allows the *xx*read and *xx*write routines to share their common code in a routine designed to be called from physio. *xx*strategy returns no error code to its caller in the kernel. Instead, errors that occur in the course of the I/O operation are reported by *xx*`intr` by way of the buffer header and passed along by *xx*`strategy`.

xx`write` — Write Data to Device

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

xx`write` is the high-level routine called (in character device drivers) to perform data transfers to the device. *xx*`write` must check that the minor device number passed to it is in range. If the minor device number is out of range, *xx*`write` returns like so:

```
if (XXUNIT(dev) >= NXX)
    return (ENXIO);
```

Subsequent actions of *xx*`write` differ depending on whether the device is a tty-style character-at-a-time device or a device that buffers its I/O into blocks.

For block transfers, *xx*`write` uses `physio`, its associated mechanisms, and the *xx*`strategy`. `buf` is here an array of locally declared buffers:

```
return (physio(xxstrategy, &buf[minor(dev)],
    dev, B_WRITE, minphys, uio));
```

If *xx*`strategy` fails, *xx*`write` passes its error code up to the kernel. The kernel then passes it on to the user program, which usually plugs it into the global error variable `errno`.

C

Kernel Support Routines

Kernel Support Routines 147

copyin — Move Data From User to Kernel Space 147

copyout — Move Data From Kernel to User Space 147

CDELAY — Conditional Busy Wait 147

DELAY — Busy Wait for a Given Period 148

iodone — Indicate I/O Complete 148

iowait — Wait for I/O to Complete 148

getkpgmap — get PTE for Virtual Address 148

gsignal — Send Signal to Process Group 148

kmem_alloc — Allocate Space from Kernel Heap 149

kmem_free — Return Space to Kernel Heap 149

MBI_ADDR — Get Address in DVMA Space 149

mbrelse — Free Main Bus Resources 149

mbsetup — Set Up to Use Main Bus Resources 149

panic — Reboot at Fatal Error 150

peek, peekc — Check and Read an Address 150

physio — Block I/O Service Routine 150

poke, pokec — Check and Write an Address 152

printf — Kernel Printf Function 152

pritospl — Convert Priority Level 153

psignal — Send Signal to Process 153

rmalloc — General-Purpose Resource Allocator 153

rmfree — Recycle Map Resource 154

sleep — Sleep on an Event 154

spl*n* — Set CPU Priority Level 155

splx — Reset Priority Level 155

swab — Swap Bytes 155

timeout — Wait for an Interval 155

uiomove — Move Data To or From an **uio** Structure 156

untimeout — Cancel **timeout** Request 156

uprintf — Nonsleeping Kernel Printf Function 156

ureadc and **uwritec** — uio Structure Read and Write 157

vac_disable_kpage — Stop Caching of a Kernel Page 157

wakeup — Wake Up a Process Sleeping on an Event 157

C

Kernel Support Routines

These routines are in alphabetical order, on the assumption that this will make them easier to find than any "logical" order.

copyin — Move Data From User to Kernel Space

copyin moves data from the user address space to the kernel address space. It is commonly used when writing *xx*ioctl routines. See copyout.

```
copyin(udaddr, kaddr, n)
    caddr_t udaddr, kaddr;
    u_int n;
```

where *kaddr* is a kernel virtual address, *udaddr* is a user virtual address, and *n* is the number of bytes to copy in. Returns 0 if no error occurs and EFAULT on a memory error.

copyout — Move Data From Kernel to User Space

copyout moves data from the kernel address space to the user address space. It is commonly used when writing *xx*ioctl routines. See copyin.

```
copyout(kaddr, udaddr, n)
    caddr_t kaddr, udaddr;
    u_int n;
```

where *kaddr* is a kernel virtual address, *udaddr* is a user virtual address, and *n* is the number of bytes to copy out. Returns 0 if no error occurs and EFAULT on a memory error.

CDELAY — Conditional Busy Wait

```
CDELAY(condition, time)
    int condition, time;
```

CDELAY is like DELAY (see below) in that it busy waits for a specified number of microseconds. It differs, however, in that it has a second argument *condition*. Each time it goes through its busy-wait loop, CDELAY checks *condition*, and, if it's true, it immediately returns. In typical usage, *condition* is a masked subset of the bits in a device register.

DELAY — Busy Wait for a Given Period

```
DELAY(time)
    int time;
```

DELAY busy waits for a specified minimum number of microseconds. That is, it just spins around using CPU time. It can be useful in situations where a device is not quite slow enough to justify having its driver go to sleep. In such cases, it's useful to busy wait for a short time. The reasoning is that while busy waiting is a waste, servicing an interrupt costs a lot more CPU time.

DELAY is also useful in introducing pauses between accesses to a device with write latency. A device register may, for example, require multiple sequential writes, and yet also require delays between the writes. See vpprobe in the *Sample Driver Listings* appendix for an example. See CDELAY.

iodone — Indicate I/O Complete

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

iodone is called to indicate that I/O associated with the buffer header *bp* is complete. iodone sets the DONE flag in the buffer header, then does a wakeup call with the buffer pointer as argument. It's called from the bottom half in place of wakeup. See iowait.

iowait — Wait for I/O to Complete

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

iowait waits on the buffer header addressed by *bp* for the DONE flag to be set. iowait actually does a sleep on the buffer header and is called from the top half in place of sleep. See iodone.

getkpgmap — get PTE for Virtual Address

```
getkpgmap(addr)
    caddr_t addr;
```

getkpgmap takes a kernel virtual address and returns the page table entry of the physical page that's mapped to it. getkpgmap can do much of the work for the driver *xx*mmap routine, though that routine must still mask the value returned from getkpgmap with PG_PFNUM to select out only the Page Frame Number and its memory type.

gsignal — Send Signal to Process Group

```
gsignal(pgrp, sig)
    int pgrp;
    int sig;
```

Sends signal *sig* to all of the processes in the process group *pgrp*. See psignal.

`kmem_alloc` — Allocate Space from Kernel Heap

```
caddr_t kmem_alloc(nbytes)
    u_int nbytes;
```

Allocates *nbytes* of contiguous kernel memory and returns a pointer to it. Calls `panic` is the request can't be satisfied. Note that `kmem_alloc` takes a while, and shouldn't be used frivolously. Also note that it can't, in system releases prior to 3.2, be called by `probe` or `attach`, since the kernel heap from which it allocates is not yet initialized. Memory allocated with `kmem_alloc` can be recycled with `kmem_free`.

`kmem_free` — Return Space to Kernel Heap

```
kmem_free(ptr, nbytes)
    caddr_t ptr;
    u_int nbytes;
```

Returns the block (allocated by `kmem_alloc`) at *ptr* to the kernel heap. If the block has already been freed, or if *ptr* doesn't indicate an address within the heap, `kmem_free` panics. When the block is freed, it is coalesced with adjacent free blocks to ensure that the free blocks in the heap are as large as possible.

`MBI_ADDR` — Get Address in DVMA Space

```
MBI_ADDR(mb_cookie)
    int mb_cookie;
```

`MBI_ADDR` is a macro that takes the integer that is returned by `mbsetup`. It returns a 32-bit virtual address, which may be either in the DVMA space or a VMEbus address space. This is the transfer address that is then given to the bus-master device, though it may first need to be checked to ensure that it is not larger than the capacity of the device. See `mbsetup` and `mbrelse`.

`mbrelse` — Free Main Bus Resources

```
mbrelse(md_hd, mbinfop)
    struct mb_hd *mb_hd;
    int *mbinfop;
```

`mbrelse` releases the Main Bus DVMA resources allocated by `mbsetup`. Note that the second parameter is a *pointer* to the integer returned by `mbsetup`.

`mbsetup` — Set Up to Use Main Bus Resources

```
mbsetup(md_hd, bp, flag)
    struct mb_hd *mb_hd;
    struct buf  *bp;
    int flag;
```

`mbsetup` is called to set up the memory map for a single Main Bus DVMA transfer. It assumes that *bp*'s fields have been set up to define the transfer, which is generally true, since `physio` sets them up before calling the driver *xx*`strategy` routine. *flag* is `MB_CANTWAIT` if the caller desires not to wait for map resources (slots in the map or DVMA space) if none are available — it's highly unlikely that this will ever happen, but if it does `mbsetup` will return immediately with a 0. In this case its caller can, presumably, wait before trying again.

`mbsetup` is typically called from the driver `strategy` routine, so when `physio` breaks up a large I/O request, one result is the generation of a series of calls to `mbsetup`. (`mbrelse` is then called from the driver *xx*`intr` routine). `mbsetup`, like `physio`, is intended primarily for the use of block drivers, though character drivers can use it as long as they don't use buffer headers from the kernel cache. The buffer is *double mapped* so that the system will consider it as being in kernel DVMA space as well as in the address space of the process being serviced.

Upon success, `mbsetup` returns an integer which must be saved for the call to `mbrelse`. This integer can also be passed to `MBI_ADDR`.

panic — Reboot at Fatal Error

```
panic(message)
    char *message;
```

`panic` can be called upon encountering an unresolvable fatal error. It prints its *message* to the system console, and then reboots the system, so don't take its use lightly. (It does have the sense to avoid the reboot if it's already been called — thus preventing recursive calls to `panic`). A kernel core image is dumped.

peek, **peekc** — Check and Read an Address

```
peek(address)
    short *address;

peekc(address)
    char *address;
```

`peek` and `peekc` are called with an address from which that want to read. Both `peek` and `peekc` return –1 if the addressed location doesn't exist, otherwise they return the value that was fetched from that location. See `poke` and `pokec`.

physio — Block I/O Service Routine

```
physio(strategy, buf, dev, flag, minphys, uio)
    void   (*strategy) ();
    struct  buf  *buf;
    dev_t  dev;
    int  flag;
    void   (*minphys) ();
    struct  uio  *uio;
```

Character drivers sometimes do block I/O, and when they do it's convenient for them to use `physio`. Such drivers resemble simple block drivers in that they have *xx*`read` and/or *xx*`write` and *xx*`strategy` routines, call those *xx*`strategy` routines indirectly through `physio`, and use `buf` structures. Too much, however, should not be made of the similarity. Character-driver *xx*`strategy` routines typically implement no strategy, and they are not driver entry points. And while character drivers can use `physio` (and `mbsetup` and `iowait` and the few other kernel support routines that manipulate buffer headers) they do not use buffers from the kernel buffer cache.

physio serves two major purposes:

- It ensures that pages of user memory are locked down (physically available and not paged out) during the duration of a data transfer.
- It breaks large transfers (those greater than the value returned by minphys) into smaller pieces, thus keeping slow devices from monopolizing the bus.

If the size of the transfer is greater than the system determined maximum, physio calls the driver *xx*strategy routine repeatedly, making sure that all relevant pointers and counters are updated correctly. Basically, physio looks like:

```
loop:
    error and termination checking;
    s = spl6();
    while (buf->b_flags & B_BUSY) {
        buf->b_flags |= B_WANTED;
        sleep(buf);
    }
    (void) splx();
    set up buffer for I/O;
    while (more data) {
        more buffer I/O set up;
        (*minphys) (buf);
        lock buffer into memory;
        (*strategy) ();
        spl6();
        unlock buffer;
        if (buf->b_flags & B_WANTED)
            wakeup(buf);
        (void) splx(s);
        bookkeeping;
    }
    buf->b_flags &= ~(B_BUSY|B_WANTED);
    error checking and bookkeeping;
    goto loop:
```

buf is a buffer header for this device. physio wants exclusive use of this buffer header and its associated buffer, and when called it checks to see if it has it. If it doesn't, it will sleep until it gets it. dev is the device to which the transfer is taking place. *flag* is B_READ or B_WRITE to indicate the direction of the transfer. minphys is a function that determines the amount of data to be transferred in one call to the *xx*strategy routine. *uio* is a pointer to the uio structure. physio returns an error code if an I/O error occurs, a 0 upon success.

poke, pokec — Check and Write an Address

```
poke(address, value)
    short *address;
    short value;

pokec(address, value)
    char *address;
    char value;
```

`poke` and `pokec` are called with an *address* you want to store into, and *value* is the value you want to store there. Both `poke` and `pokec` return 1 if the addressed location doesn't exist, and 0 if the addressed location does exist. See `peek` and `peekc`.

printf — Kernel Printf Function

The kernel provides a `printf` function analogous to the `printf` function supplied with C library for user programs. The kernel `printf`, however, is more limited than is the version in the C library. It writes directly to the console tty, its output cannot be easily redirected, and it supports only a subset of `printf`'s formatting conversions. Furthermore, it's not interrupt driven, and thus causes all system activities to be suspended while it outputs its message. Nevertheless, `printf` is useful as a debugging tool, and for reporting error messages. See `uprintf`.

The formatting conversions supported by the kernel `printf` are:

`%x, %X`	- Hexadecimal numbers
`%d, %D`	- Decimal numbers
`%o, %O`	- Octal numbers
`%c`	- Single characters
`%s`	- Strings
`%b`	- Bit values

Note that floating-point conversions are *not* supported. Also note that a special format %b is provided to decode error registers. Its usage is:

```
printf("reg=%b\n", regval, "<base><arg>*");
```

Where `<base>` is the output base expressed as a control character. For example, `\10` gives octal and `\20` gives hex. Each `arg` is a sequence of characters, the first of which gives the bit number to be inspected (counting from 1), and the rest of which (up to a control character, that is, a character <= 32), give the name of the register. Thus:

```
printf("reg=%b\n", 3, "\10\2BITTWO\1BITONE\n");
```

would produce the output:

```
reg=3<BITTWO,BITONE>
```

Also note that no conversion modifiers (field widths and so on) are supported — only a single character can follow the `%`.

The kernel `printf` function raises the priority level and therefore locks out interrupts while it is sending data to the console. And it displays its messages directly on the console, unless specifically redirected by the `TIOCCONS` ioctl.

pritospl — Convert Priority Level

```
pritospl(value)
    int  value;
```

pritospl is a macro that converts the hardware priority level given by *value*, which is a Main Bus priority level, to the processor priority level that splx expects. The Main Bus priority level can be found in either md->md_intpri or mc->mc_intpri, where it is put by the autoconfiguration process. pritospl is used to parameterize the setting of priority levels. See spl*n* and splx.

psignal — Send Signal to Process

```
psignal(p, sig)
    struct proc *p;
    int sig;
```

Sends signal *sig* to the process specified by the proc structure. See gsignal.

rmalloc — General-Purpose Resource Allocator

```
long rmalloc(mp, size)
    struct map *mp;
     long size;
```

rmalloc (for resource map allocator) is a rather specialized sort of resource allocator. In fact, it doesn't really allocate resources at all, but rather names of resources (that is, lists of numbers). Such lists are initialized by rminit and are called resource "maps". Given such a map, rmalloc can parcel out the names in it. The relationship of such names to real resources (virtual address space, physical memory, and so on) is entirely a matter of usage conventions. Names allocated with rmalloc are recycled with rmfree.

rmalloc is a low-level routine, and shouldn't be used casually. If you just want some kernel virtual memory, use kmem_alloc. rmalloc is called by drivers that need to allocate kernel virtual address space during their *xx*probe and *xx*attach routines. They call it, rather than kmem_alloc, because they want an address space without physical memory mapped to it.

rminit is *not* documented here, for device drivers only have occasion to use two pre-initialized rmalloc maps:

- The map kernelmap (in <sys/map.h>) is used to allocate chunks of generic kernel virtual address space.
- The map iopbmap (in <sundev/mbvar.h>) contains addresses that are guaranteed to be in the high Megabyte and thus suitable for use as DVMA buffer addresses. iopbmap is quite small, and should be used only for temporary or very small buffers.

`rmfree` — Recycle Map Resource

```
rmfree(mp, size, addr)
    struct map *mp;
    long size, addr;
```

`rmfree` recycles the map resource allocated with `rmalloc`.

`sleep` — Sleep on an Event

```
sleep(address, priority)
    caddr_t address;
    int priority;
```

`sleep` is called to put the calling process to sleep, typically while it awaits the availability of some system resource. *address* is the address of a location in memory, usually a field in some global driver structure that is being used as a "semaphore" (such fields are not true semaphores, see below). *priority* is the software priority the calling process will have after being awakened.

`sleep` must *never* be called from the interrupt-level side of a driver. This is because `sleep` is always executed on behalf of a specific process. It suspends that process while the scheduler picks and executes another waiting process. And since, when handling an interrupt, the kernel isn't running on behalf on any process, it makes no sense to call `sleep`. Incidently, the kernel will `panic` if sleep is called while it's running on the interrupt stack.

A process that has called `sleep` will be reawakened by any `wakeup` call issued with the same *address*. However *it's not guaranteed that, upon waking, the process will find the resource that it was waiting for to be available.* It must, therefore, check again before proceeding, and go back to sleep if necessary. This is because the UNIX `sleep` and `wakeup` facilities do not constitute true semaphore primitives in the usual P/V sense. `wakeup` will wakeup *every process* that is sleeping on that event, where a true 'V' semaphore will wake only one sleeper (the highest priority one or whichever).

Thus in UNIX you always do:

```
s = spln(high_priority);
while (resource_busy)
    sleep(resource, high_priority);
make_resource_busy;
(void) splx(s);
. . .
<critical section>
. . .
wakeup(resource);
```

whereas with real semaphores you would simply do:

```
P(resource);
. . .
<critical section>
. . .
V(resource);
```

which is a much simpler and cleaner design.

However, semaphores are not easy to use to implement lockouts around hardware interrupts so UNIX just uses the `sleep/wakeup` mechanism for both situations.

spl*n* — Set CPU Priority Level

The `spl`*n* functions are available for setting the CPU priority level to *n*, where *n* ranges from 0 to 7. Note that `spl6` actually gets you `spl5` on Sun systems to avoid lockout of the level 6 on-board UART interrupts. When you allocate a CPU priority level to your device, choose one that's high enough to give you the performance you need, but don't overdo it or you'll interfere with the operation of the system:

- □ If you lock out the on-board UARTS (level 6) characters may be lost.
- □ If you lock out the clock (level 5) time will not be accurate, and the UNIX scheduler will be suspended.
- □ If you lock out the Ethernet (level 3), packets may be lost and retransmissions needed.
- □ And if you lock out the disks (level 2), disk rotations may be missed.

The `spl`*n* functions return the previous priority level.

splx — Reset Priority Level

```
splx(s)
    int  s;
```

`splx` called with an argument *s* sets the priority level to *s*, which was returned from a previous call to `spl`*n*, `pritospl`, or `splx`. `splx` is typically used to restore the priority level to a previously stored level. `splx` returns the previous level.

swab — Swap Bytes

```
swab(from, to, nbytes)
    caddr_t  from;
    caddr_t  to;
    int  nbytes;
```

`swab` swaps bytes within words. *nbytes* is the number of bytes to swap, and is rounded up to a multiple of two. No checking is done to ensure that the *from* and *to* areas do not overlap each other.

timeout — Wait for an Interval

```
timeout(func, arg, interval)
    int (*func)();
    caddr_t arg;
    int interval;
```

`timeout` arranges that after *interval* clock-ticks, *func* will be called with *arg* as its argument, in the style *(*func)(arg)*. A clock tick is about a fiftieth of a second; the precise number of clock ticks per second is given in the external variable `hz`. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to cancel read or write requests that have received no response within a specified amount of time (if there's a lost interrupt or if the device otherwise flakes out). The specified *func* is eventually called from the lower half of the clock-interrupt routine, so it must conform to the requirements of interrupt routines in general. In particular, it can't call sleep. See untimeout.

uiomove — Move Data To or From an **uio** Structure

```
uiomove(cp, n, rw, uio)
    caddr_t  cp;
    int n;
    enum uio_rw  rw;
    struct *uio;
```

uiomove is the most common way for device drivers to move a specified number of bytes between a byte array in kernel address space and an area defined by a uio structure (which may or may not be in kernel address space). If the uio_seg field in the uio structure is set to UIOSEG_USER, uiomove will assume the *uio* pointer to be in user space; if it is UIOSEG_KERNEL, it will assume it to be in kernel space (see <sys/uio.h>). uiomove moves *n* bytes between the uio structure and the area defined by the *cp* parameter. The read/write flag are interpreted as follows: — UIO_READ indicates a transfer from kernel to user space (a call to copyout), and UIO_WRITE a transfer from user to kernel space (a call to copyin). uiomove returns 0 upon success, –1 upon failure.

For more information about the uio structure, see *Some Notes About the UIO Structure* in the *The ''Skeleton'' Character Device Driver* section of this manual.

untimeout — Cancel **timeout** Request

```
untimeout(func, arg)
    int (*func)();
    caddr_t arg;
```

untimeout is called to cancel a prior timeout request. *func* and *arg* are the same as in timeout.

uprintf — Nonsleeping Kernel Printf Function

uprintf is like printf, with two important differences. The first is that it checks to see if the process' "controlling terminal" is open, and if it is the message is sent to it rather than to the system console (uprintf consults the user structure, so it must not be called from the lower-half routines). If there's no controlling terminal, uprintf executes as would printf. The second difference is that uprintf is interruptible, and thus reasonably efficient.

uprintf is often called from open routines to report errors to the user. It's used for errors which, like tape-read errors, are likely to indicate operator error rather than system failure. See printf.

ureadc and **uwritec** — uio Structure Read and Write

```
ureadc(c, uio)
    int c;
    struct *uio;
```

ureadc transfers the character *c* into the uio structure (which is normally passed to the driver when it is called). ureadc is normally used when "reading" a character in from a device.

```
uwritec(uio)
    struct *uio;
```

uwritec returns the next character in the uio structure (which is normally passed to the driver when it is called), or returns –1 on error. uwritec is normally used when "writing" a character to a device.

Note that "read" and "write" are slightly confusing in the above contexts, since ureadc actually obtains a character from somewhere and places it *into* the uio structure, whereas uwritec obtains a character from the uio structure and "writes" it somewhere else. The "read" and the "write," then, are from the perspective of the user program.

ureadc and uwritec replace the routines cpass and passc, which are no longer supported.

vac_disable_kpage — Stop Caching of a Kernel Page

```
vac_disable_kpage(vaddr)
    caddr_t vaddr;
```

Note: this routine is for use on Sun-3/260 and Sun-3/280 machines only. On all other machines it does nothing.

Sun-3/260 and Sun-3/280 machines have a write-back virtual address cache. When more than one virtual page maps to the same physical page, the cache can cause a data inconsistency unless such pages are removed from it. Therefore, when drivers remap a kernel page, they must call vac_disable_kpage to remove it from the virtual-address cache and thus maintain the cache's consistency. The kernel page isn't returned to the cache until the kernel is rebooted. vac_disable_kpage should be called in all driver mmap routines.

wakeup — Wake Up a Process Sleeping on an Event

```
wakeup(address)
    caddr_t address;
```

wakeup is called when a process waiting on an event must be awakened. *address* is typically the address of a location in memory. wakeup is typically called from the low level side of a driver when (for instance) all data has been transferred to or from the user's buffer and the process waiting for the transfer to complete must be awakened. See sleep.

D

User Support Routines

User Support Routines **161**

free — Free Allocated Memory 161

getpagesize — Return Pagesize 161

mmap — Map Memory from One Space to Another 161

munmap — Unmap Pages of Memory 162

valloc — Allocate Virtual Memory 162

D

User Support Routines

free — Free Allocated Memory

```
free(ptr)
    char *ptr;
```

free is used to recycle the virtual memory allocated with valloc. (Actually, it can be used with a variety of memory allocators, including malloc, the most general purpose of the allocators). See valloc and malloc(3).

getpagesize — Return Pagesize

```
int getpagesize()
```

getpagesize returns the number of bytes in a page. The page size is the system page size and may not be identical with the page size in the underlying hardware — it is, however, the pagesize of interest in all of the memory management functions. See getpagesize(2).

mmap — Map Memory from One Space to Another

```
mmap(addr, len, protection, share, fd, off)
    caddr_t addr;
    int len, protection, share, fd;
    off_t off;
```

mmap maps pages of memory space from the memory device associated with the file *fd* into the address space of the calling process. The mapping is performed one page at a time, by iteratively calling the memory device's mmap routine.

The memory is mapped from the memory device, beginning at *off*, into the callers address space beginning at *addr* and continuing for *len* bytes. *fd* is a file descriptor obtained by opening the character special device to be mmap'ed. (In the future, regular files will be allowed). *share* specifies whether modifications made to this mapped copy of the page are to be kept private or are to be shared with other references to the same page — it must currently be set to MAP_SHARED. *addr*, *len* and *off* must be multiples of the page size (which can be found by using *getpagesize*). For this reason, the local memory space beginning at *addr* should be allocated by using valloc, which returns a properly aligned buffer. *protection* specifies the read/write accessibility of the mapped pages.

Pages are automatically unmapped when *fd* is closed and can be explicitly unmapped with munmap. Special care must be taken when unmapping an area of 128K or more, for when an area that large is mapped in the first place, the kernel releases the swap area associated with it. Consequently, when this area is unmapped, the pages that make it up are marked invalid, and the next call to `valloc` will return invalid pages — any attempt to reference those pages will then produce a segmentation violation. To avoid this problem, do not `free` such large chunks; instead call `valloc` again without calling `free`. `mmap` returns a -1 on error, 0 on success. See mmap(2).

munmap — Unmap Pages of Memory

```
munmap(addr, len)
    caddr_t addr;
    int len;
```

`munmap` causes the pages starting at *addr* and continuing for *len* bytes to be unmapped, that is, to refer only to private pages within the callers address space. The pages are initialized to zero, unless `len` is 128K or more, in which case the pages are marked invalid. Returns a -1 on error, 0 on success. See mmap(2).

valloc — Allocate Virtual Memory

```
char *valloc(size)
    unsigned size
```

`valloc` is commonly called from user programs that need it to allocate page-aligned virtual memory to pass to `mmap`. No other allocator can be used with `mmap`, since it requires that the space passed to it start on a page boundary. The memory it allocates is recycled with a call to `free`. See malloc(2).

E

Sample Driver Listings

Sample Driver Listings **165**

E.1. Skeleton Board Driver 166

E.2. Sun-2 Color Graphics Driver 174

E.3. Sky Floating-Point Driver 186

E.4. Versatec Interface Driver 194

E

Sample Driver Listings

The following C-code listings are for sample Sun device drivers. There are four drivers listed here; the first being the skeleton driver and the other three being real production drivers. (These three drivers, it should be mentioned, have been chosen as relatively simple illustrations of the three major types of drivers — not as software ideals to be closely emulated).

SKELETON
is the driver for the "skeleton board" discussed earlier in this manual.

CGTWO
is a device driver for the Sun-2 Color Graphics board. It is one of the simplest drivers around, being memory mapped.

SKY
is a programmed I/O driver for the Sky floating-point board, with both polling interrupts and vectored interrupts. However, the interrupt routines don't do a whole lot.

VP is a driver for the Versatec Printer Interface. It's a fairly good example of a DMA device driver.

E.1. Skeleton Board Driver

```
/*
 * (skreg.h) Registers for Skeleton Board -- note the byte swap
 */

struct sk_reg {
     char sk_data;   /* 01: Data Register */
     char sk_csr;    /* 00: command(w) and status(r) */
};

/* sk_csr bits (read) */
#define   SK_INTR          0x80  /* Device is Interrupting*/
#define   SK_DEVREADY      0x08  /* Device is Ready */
#define   SK_INTREADY      0x04  /* Interface is Ready */
#define   SK_ERROR         0x02  /*  Device Error  */
#define   SK_INTENAB       0x01  /* Interrupts are Enabled */

#define   SK_ISTHERE       0x0C  /*  Existance Check; Device and Interface Ready  */

/* sk_csr bits (write) */
#define   SK_RESET  0x04         /* Reset Device and Interface */
#define   SK_ENABLE 0x01         /* Enable Interrupts */


/*
 * Further definitions for DMA skeleton board
 */

#define   SK_DMA     0x10        /* Do DMA transfer */
#define   MAX_SK_BSIZE 4096      /* DMA tranfer block */

struct sk_reg2 {
     char sk_data;          /* 01: Data Register */
     char sk_csr;           /* 00: command(w) and status(r) */
     short sk_count;        /* bytes to be transferred */
     caddr_t sk_addr;       /* DMA address */
};
```

```
/*
 * (sk.c) The "Skeleton Board" Driver"
 */

/* This listing is not heavily annotated. This is because it's identical to
 * the Skeleton driver discussed at length in the main body of the manual.
 * It appears here for purposes of completeness.
 */

#include "../h/param.h"
#include "../h/buf.h"
#include "../h/file.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/uio.h"
#include "../machine/psl.h"
#include "../sundev/mbvar.h"

#include "sk.h"       /* file generated by config (defines NSK) */
#include "skreg.h"    /* register definitions */

#define SKPRI (PZERO-1)    /* software sleep priority for sk */

#define SKUNIT(dev) (minor(dev))

struct buf skbufs[NSK];

int skprobe(), skpoll();

struct mb_device *skdinfo[NSK];
struct mb_driver skdriver = { skprobe, 0, 0, 0, 0, skpoll,
     sizeof(struct sk_reg), "sk", skdinfo, 0, 0, 0, 0,
};

struct sk_device {
     char soft_csr;         /* software copy of control/status register */
     struct buf *sk_bp;     /* current buf */
     int sc_count;          /* number of bytes to send */
     char *sk_cp;           /* next byte to send */
     char sk_busy;          /* true if device is busy */
} skdevice[NSK];

/*ARGSUSED*/
skprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     register struct sk_reg *sk_reg;
     register int c;

     sk_reg = (struct sk_reg *)reg;

     c = peekc((char *)&sk_reg->sk_csr);    /*  contact the device */
```

```
        if (c == -1 || (c != SK_ISTHERE))
                return (0);
        if (pokec((char *)&sk_reg->sk_csr, SK_RESET)) /* contact the device */
                return (0);

        return (sizeof (struct sk_reg));
}

skopen(dev, flags)
        dev_t dev;
        int flags;
{
        register int unit = SKUNIT(dev);
        register struct mb_device *md;
        register struct sk_reg *sk_reg;

        md = skdinfo[unit];

        if (unit >= NSK || md->md_alive == 0)
                return (ENXIO);
        if (flags & FREAD)
                return (ENODEV);

        sk_reg = (struct sk_reg *)md->md_addr;

        /* enable interrupts */
        skdevice[unit].soft_csr = SK_ENABLE;

        /* contact the device */
        sk_reg->sk_csr = skdevice[unit].soft_csr;

        return (0);
}

/*ARGSUSED*/
skclose(dev, flags)
        dev_t dev;
        int flags;
{
        register int unit = SKUNIT(dev);
        register struct mb_device *md;
        register struct sk_reg *sk_reg;

        md = skdinfo[unit];

        /* disable interrupts */
        sk_reg = (struct sk_reg *)md->md_addr;
        skdevice[unit].soft_csr &= ~SK_ENABLE;

        /* contact device */
        sk_reg->sk_csr = skdevice[unit].soft_csr;
}
```

```
skminphys(bp)
     struct buf *bp;
{

     if (bp->b_bcount > MAX_SK_BSIZE)
          bp->b_bcount = MAX_SK_BSIZE;
}

skstrategy(bp)
     register struct buf *bp;
{
     register struct mb_device *md;
     register struct sk_device *sk;
     int s;

     md = skdinfo[SKUNIT(bp->b_dev)];    /* physio put the device number into bp */
     sk = &skdevice[SKUNIT(bp->b_dev)];

     s = splx(pritospl(md->md_intpri)); /* begin critical section */
     while (sk->sk_busy)
          sleep((caddr_t) sk, SKPRI);

     /* set up for first write */
     sk->sk_busy = 1;
     sk->sk_bp = bp;
     sk->sk_cp = bp->b_un.b_addr;
     sk->sc_count = bp->b_bcount;
     skstart(sk, (struct sk_reg *)md->md_addr);

     (void) splx(s);       /* end critical section */
}

skwrite(dev, uio)
     dev_t dev;
     struct uio *uio;
{
     register int unit = SKUNIT(dev);

     if (unit >= NSK)
          return (ENXIO);
     return (physio(skstrategy, &skbufs[unit],
          dev, B_WRITE, skminphys, uio));
}

skstart(sk, sk_reg)
     struct sk_device *sk;
     struct sk_reg *sk_reg;
{

     while(sk->sc_count > 0) {   /* still more characters */
          sk_reg->sk_data = *sk->sk_cp++;
          sk->sc_count--;
```

```
        /* stop giving characters if device not ready */
        /* Note: the softcopy isn't needed for reads */
        if (!(sk_reg->sk_csr & SK_DEVREADY))   /* contact the device */
            break;
    }

    if (sk->sc_count > 0) {   /* still more characters */
        sk->soft_csr = SK_ENABLE;
        sk_reg->sk_csr = sk->soft_csr;  /* contact the device */
    } else {
        /* special case: finished the command without taking any interrupts! */
        sk->soft_csr = 0;     /* disable interrupts */
        sk_reg->sk_csr = sk->soft_csr;       /* contact the device */
        sk->sk_busy = 0;
        wakeup((caddr_t) sk);      /*free device to sleeping strategy routine */
        iodone(sk->sk_bp);    /*free buffer to waiting physio */
    }
}

skpoll()
{
    register struct sk_reg *sk_reg;
    int serviced, i;

    serviced = 0;
    for (i = 0; i < NSK; i++) {     /* try each one */
        sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;
        if (sk_reg->sk_csr & SK_INTR) { /* contact the device */
            serviced = 1;
            skintr(i);
        }
    }
    return (serviced);
}

skintr(i)
    int  i;
{
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;

    sk_reg = (struct sk_reg *)skdinfo[i]->md_addr;
    sk = &skdevice[i];

    /* check for an I/O error */
    if (sk_reg->sk_csr & SK_ERROR) { /* contact the device */

        /* error-retry logic would go here */

        printf("skintr: I/O error\n");
        sk->sk_bp->b_flags |= B_ERROR;
        goto error_return;
    }
```

```
     if (sk->sc_count == 0) { /* I/O transfer completed */
error_return:
          sk->soft_csr = 0;  /* clear interrupt */
          sk_reg->sk_csr = sk->soft_csr; /* contact the device */
          sk->sk_busy = 0;
          wakeup((caddr_t) sk); /* free device to sleeping strategy routine */
          iodone(sk->sk_bp); /* free buffer to waiting physio */
     } else skstart(sk, sk_reg);
}


/* DMA VARIATIONS FOLLOW */

struct sk_device {
     char soft_csr;           /* software copy of control/status register */
     struct buf *sk_bp;       /* current buf */
     char sk_busy;            /* true if device is busy */
     int sk_mbinfo;           /* Information stash for DMA */
} skdevice[NSK];

skstrategy(bp)
     register struct buf *bp;
{
     register struct mb_device *md;
     register struct sk_reg *sk_reg;
     register struct sk_device *sk;
     int s;

     md = skdinfo[SKUNIT(bp->b_dev)];
     sk_reg = (struct sk_reg *)md->md_addr;
     sk = &skdevice[SKUNIT(bp->b_dev)];

     s = splx(pritospl(md->md_intpri));          /* begin critical section */
     while (sk->sk_busy)
          sleep((caddr_t) sk, SKPRI);
     sk->sk_busy = 1;
     sk->sk_bp = bp;

     /* this is the part that is changed */

     /* grab bus resources */
     sk->sk_mbinfo = mbsetup(md->md_hd, bp, 0);

     /* the remainder */
     sk_reg->sk_count = bp->b_bcount;

     /* plug bus transfer address */
     sk_reg->sk_addr = (caddr_t)MBI_ADDR(sk->sk_mbinfo);

     /* make sure we didn't overrun the address space limit */
     if (sk_reg->sk_addr > (caddr_t) 0x000FFFFF) {
           printf("sk%d: ", sk_reg->sk_addr);
           panic("exceeded 20 bit address");
```

```
        }

        sk->soft_csr = SK_ENABLE | SK_DMA;
        sk_reg->sk_csr = sk->soft_csr;          /* contact the device */

        /* end of DMA-related changes */

        sk->sk_busy = 0;
        wakeup((caddr_t) sk); /* free device to sleeping strategy routine */
        (void) splx(s);         /* end critical section */
}

skpoll()
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        int serviced, i;

        serviced = 0;
        for (i = 0; i < NSK; i++) {
                md = (struct mb_device *)skdinfo[i];
                sk_reg = (struct sk_reg *)md->md_addr;
                if (sk_reg->sk_csr & SK_INTR) {
                        serviced = 1;
                        skintr(i);
                }
        }
        return (serviced);
}

skintr(i)
        int  i;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;

        md = (struct mb_device *)skdinfo[i];
        sk_reg = (struct sk_reg *)md->md_addr;
        sk = &skdevice[i];

        /* check for an I/O error */
        if (sk_reg->sk_csr & SK_ERROR) { /* contact the device */

                /* error-retry logic would go here */

                printf("skintr: I/O error0);
                sk->sk_bp->b_flags |= B_ERROR;
        }

        /* this is the part that changed */
        sk->soft_csr = 0;     /* clear interrupt */
        sk_reg->sk_csr = sk->soft_csr;
```

```
	mbrelse(md->md_hd, &sk->sk_mbinfo);
	iodone(sk->sk_bp);    /* free buffer to waiting physio */
}
```

E.2. Sun-2 Color Graphics Driver

```
/*
 * (cgreg.h) Register definitions for Sun2 (Memory Mapped) Color Board
 * Copyright (c) 1983 by Sun Microsystems, Inc
 */

#define CGSIZE (16*1024)        /* 16K of address space */
#define GR_bd_sel   CGXBase     /* Select Color Board */

#define GR_x_select  0x0800     /* Access a column in the frame buffer */
#define GR_y_select  0x0000     /* Access a row in the frame buffer */
#define GR_y_fudge   0x0200     /* Bit 9 not used at all */
#define GR_update    0x2000     /* Update frame buffer if this bit set */
#define GR_x_rhaddr  0x1b80     /* Location to read X address bits A9-A8
                                     Data put into D1-D0 */
#define GR_x_rladdr  0x1b00     /* Location to read X address bits A7-A0
                                     Data put into D7-D0 */
#define GR_y_rhaddr  0x1bc0     /* Location to read Y address bits A9-A8 */
#define GR_y_rladdr  0x1b40     /* Location to read Y address bits A7-A0 */

#define GR_set0      0x0000     /* Address Register pair 0 */
#define GR_set1      0x0400     /* Address Register pair 1 */

#define GR_red_cmap  0x1000     /* Address to select Red Color Map */
#define GR_grn_cmap  0x1100     /* Addr for Green Color Map */
#define GR_blu_cmap  0x1200     /* Addr for Blue Color Map */

#define GR_sr_select 0x1800     /* Addr to select status register */
#define GR_cr_select 0x1900     /* Addr to select mask (color) register */
#define GR_fr_select 0x1a00     /* Addr to select function register */

/* Pointers to the mask(color), status, and function regs */
#define GR_creg      (u_char *)(GR_bd_sel + GR_cr_select)
#define GR_mask      (u_char *)(GR_bd_sel + GR_cr_select)
#define GR_sreg      (u_char *)(GR_bd_sel + GR_sr_select)
#define GR_freg      (u_char *)(GR_bd_sel + GR_fr_select)

/* Bit assignments in the Status Register */
#define GRW0_cplane 0x00       /* Select CMap Plane number zero for R/W */
#define GRW1_cplane 0x01       /* Select CMap Plane number one for R/W */
#define GRW2_cplane 0x02       /* Select CMap Plane number two for R/W */
#define GRW3_cplane 0x03       /* Select CMap Plane number three for R/W */

#define GRV0_cplane 0x04       /* Select CMap Plane number zero for video */
#define GRV1_cplane 0x05       /* Select CMap Plane number one for video */
#define GRV2_cplane 0x06       /* Select CMap Plane number two for video */
#define GRV3_cplane 0x07       /* Select CMap Plane number three for video */

#define GR_inten    0x10       /* Enable Interrupts at start of next vertical
                                  retrace Must clear bit to disable */
```

```
#define GR_paint     0x20          /* Enable Writing five pixels in parallel */
#define GR_disp_on   0x40          /* Enable Video Display */
#define GR_vretrace  0x80          /* Unused on write On read, true if monitor in
                                      vertical retrace */

/* Returns true if the board is in vertical retrace */
#define GR_retrace   (*GR_sreg & GR_vretrace)

/* Function register encodings */
#define GR_copy              0xCC   /* Copy data reg to Frame buffer */
#define GR_copy_invert       0x33   /* Copy inverted data reg to FB  */
#define GR_wr_creg           0xF0   /* Copy color reg to Frame buffer */
#define GR_wr_mask           0xF0   /* Copy mask to Frame buffer */
#define GRinv_wr_creg        0x0F   /* Copy inverted Creg to FB */
#define GRinv_wr_mask        0x0F   /* Copy inverted Mask to FB */
#define GR_ram_invert        0x55   /* 'Invert' color in Frame buffer */
#define GR_cr_and_dr         0xC0   /* Bitwise and of color and data regs */
#define GR_clear             0x00   /* Clear frame buffer */
#define GR_cr_xor_fb         0x5A   /* Xor frame buffer data and Creg */
```

```
/*
 * (cgtwo.c) Sun2 (Memory Mapped) Color Board Driver
 * Copyright (c) 1984 by Sun Microsystems, Inc.
 */

/*
 * As a driver for a frame-buffer device, cgtwo.c must provide not only the
 * standard device-driver functionality, but also low-level support for the
 * Sun virtual desktop. That is to say, frame-buffer drivers not only
 * implement the standard device-driver hardware interface, but also declare,
 * initialize and export the pixrect structures which allow the kernel to
 * view the frame-buffer memory as a large rectangle within which it can
 * rapidly paint a cursor. As a consequence, much of the code here is pixrect
 * related, though among the muck you'll find the operations common to all
 * memory-mapped drivers.
 *
 * The kernel does not context switch frame buffers, despite the fact that some
 * of them (including the Sun2 Color Board which this driver controls) do have
 * context. In general, the kernel assumes that frame buffers either have no
 * context that needs to be switched, or are used in a manner that doesn't
 * require them to be context switched. SunWindows takes the second of these
 * tacts, arbitrating frame-buffer access (with pixwin locking) so that no
 * process can use the frame buffer while another process has "context" in it.
 *
 */

#include "cgtwo.h"          /* installed device count -- from config */
#include "win.h"
#if NCGTWO > 0

#include "../h/param.h"          /* general kernel parameters */
#include "../h/systm.h"          /* miscellaneous kernel variables */
#include "../h/map.h"            /* resource allocation maps */
#include "../h/buf.h"            /* I/O buffers */
#include "../h/dk.h"             /* system instrumentation */
#include "../h/vm.h"             /* virtual machine */
#include "../h/dir.h"            /* directories */
#include "../h/user.h"           /* kernel per-process status */
#include "../h/proc.h"           /* unswapped kernel per-process status */
#include "../h/conf.h"           /* kernel entry-point switches */
#include "../h/file.h"           /* open file information */
#include "../h/uio.h"            /* uio structures */
#include "../h/ioctl.h"          /* ioctl definitions */
#include "../h/kernel.h"         /* kernel global variables */

/* ../machine is a symbolic link set to either ../sun2 or ../sun3 */
#include "../machine/mmu.h"      /* memory-management unit */
#include "../machine/pte.h"      /* page table entries */

#include "../sun/fbio.h"         /* frame buffer definitions */

/* ../sundev is the device driver source directory */
#include "../sundev/mbvar.h"     /* bus-interface definitions */
```

```
/* ../pixrect contains pixrect-related source */
#include "../pixrect/pixrect.h"        /* basic pixrect definitions */
#include "../pixrect/pr_util.h"        /* pixrect utilities */
#include "../pixrect/memreg.h"         /* rasterop hardware registers */
#include "../pixrect/cg2reg.h"         /* Sun2 color frame buffer definitions */
#include "../pixrect/cg2var.h"         /* more Sun2 color frame buffer */


#if NWIN > 0
#define CG2_OPS &cg2_ops
struct pixrectops cg2_ops = {
     cg2_rop,
     cg2_putcolormap,
     cg2_putattributes,
};
#else
#define   CG2_OPS   (struct pixrectops *)0
#endif

struct cg2pr cgtwoprdatadefault =
    {0/*cgpr_va set in ioctl*/, 0, 255, 0, 0 };

struct pixrect cgtwopixrectdefault =
    { &cg2_ops, { 0/* set in ioctl */, 0/* set in ioctl */ }, CG2_DEPTH,
     0/* set in ioctl */};

/* Bytes - size of a single page */
#define CG2PROBESIZE (NBPG)

/* Bytes - size of the entire device */
#define CG2TOTALSIZE (sizeof(struct cg2fb)+sizeof(struct cg2memfb))

/* Bytes - size that the kernel needs to address */
#define CG2EXTRAOFF (sizeof(struct cg2memfb))

/* Bytes - size that the kernel needs to address */
#define CG2EXTRASIZE (sizeof(struct cg2fb))

int cg2extraclicks = btoc(CG2EXTRASIZE);

struct cgtwoextra {
     unsigned char *cg2x_ropaddr;   /* addr of cg2fb (set in cgtwoprobe) */
     int  cg2x_hwwidth;             /* device width (from resolution) */
     int  cg2x_hwheight;            /* device height (from resolution) */
     unsigned int cg2x_physaddr;    /* color board VME address */
};
struct cgtwoextra cgtwoextra[NCGTWO];
struct pixrect cgtwopixrect[NCGTWO];
struct cg2pr cgtwoprdata[NCGTWO];

/* Driver information for auto-configuration */
int cgtwoprobe(), cgtwointr();
struct mb_device *cgtwoinfo[NCGTWO];
struct mb_driver cgtwodriver = {
     cgtwoprobe, 0, 0, 0, 0, cgtwointr,
     CG2PROBESIZE, "cgtwo", cgtwoinfo, 0, 0, 0,
};

/*
 * Prove that the device is a cgtwo board by accessing the rop chip, and do an unusually
 * complex mapping. Since the kernel pixrect only uses 1.3Meg of the 3.3Meg device
 * address space, we save 2Megs of virtual address space in the kernel by only mapping
 * the first page and the last 1.3 Meg of the device address space. We map the 1.3Meg
 * chunk (CGEXTRAOFF) here in cgtwoprobe, and return the number of bytes in a page
 * to the kernel autocongiguration process so that it can finish the job.
 */
cgtwoprobe(reg, unit)
     caddr_t reg;
     int  unit;
{
     int  a = 0, page;
     register struct cg2fb *cg2fb;
     struct cg2statusreg *status;
     u_char testpixel;
     short shrt;
     int  opcount = 0; /* Used only for debugging */

     /* Get physical page number and type of first device page */
     page = getkpgmap(reg) & PG_PFNUM;

     /* Allocate virtual memory that device will access */
     if ((a = (int)rmalloc(kernelmap, (long)cg2extraclicks)) == 0)
          panic("out of kernelmap for cg2");
     cg2fb = (struct cg2fb *)kmxtob(a);
     cgtwoextra[unit].cg2x_ropaddr = (unsigned char *)cg2fb;

     /* Mapin only the virtual memory that the kernel will use */
     mapin(&Usrptmap[a], btop(cgtwoextra[unit].cg2x_ropaddr),
         page+btop(CG2EXTRAOFF), cg2extraclicks, PG_V|PG_KW);

     /* ACTUALLY DO THE PROBE: Enable writing to all rop chips */
     if (poke((short *)&cg2fb->ppmask.reg, 0xFF))
          goto failure;
     opcount++;

     /* Set SRC op in all chips: cg2_setfunction(cg2fb, CG2_ALLROP, 0xCC) */
     if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].ropregs.mrc_op, 0xCC))
          goto failure;
     opcount++;

     /* Set pixel mode access: cg2fb->status.reg.ropmode = SWWPIX */
     if ((shrt = peek((short *)&cg2fb->status.reg)) == -1)
          goto failure;
     opcount++;
     status = (struct cg2statusreg *)&shrt;
     status->ropmode = SWWPIX;
```

```
if (poke((short *)&cg2fb->status.reg, shrt))
     goto failure;
opcount++;

/* Zero the end masks */
if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].ropregs.mrc_mask1, 0))
     goto failure;
opcount++;
if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].ropregs.mrc_mask2, 0))
     goto failure;
opcount++;

/* Zero the width and opcount */
if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].ropregs.mrc_width, 0))
     goto failure;
opcount++;
if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].ropregs.mrc_opcount,0))
     goto failure;
opcount++;

/*
 * Set source alignment shift to zero, set fifo direction to one,
 * ie. bus -> src1 -> src2 -> ROPC function unit
 */
if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].ropregs.mrc_shift, 1<<8))
     goto failure;
opcount++;

/* Load the fifo src1 */
if (poke((short *)&cg2fb->ropcontrol[CG2_ALLROP].prime.ropregs.
     mrc_source2, 0))
     goto failure;
opcount++;

/* Pump src2 into a pixel and load src1 with 0xFFFF */
if (pokec((char *)&cg2fb->ropio.roppixel.pixel[0][0], 0xFF))
     goto failure;
opcount++;

/* mask every other pixel bit */
if (poke((short *)&cg2fb->ppmask.reg, 0xAA))
     goto failure;
opcount++;

/* pump src1 (0xF) into the pixel through the ppmask 0xAA */
if (pokec((char *)&cg2fb->ropio.roppixel.pixel[0][0], 0))
     goto failure;
opcount++;

/* read it back */
if ((testpixel =
     (u_char)peekc((char *)&cg2fb->ropio.roppixel.pixel[0][0])) == 0xFF)
     goto failure;
     opcount++;
     if (testpixel == 0xAA) {
          switch (cg2fb->status.reg.resolution) {
          case 0:
               cgtwoextra[unit].cg2x_hwwidth = CG2_WIDTH;
               cgtwoextra[unit].cg2x_hwheight = CG2_HEIGHT;
               break;
          case 1:
               cgtwoextra[unit].cg2x_hwwidth = CG2_SQUARE;
               cgtwoextra[unit].cg2x_hwheight = CG2_SQUARE;
               break;
          default:
               printf("CGTWO unknown resolution (%D)\n",
                   cg2fb->status.reg.resolution);
               opcount++;
               goto failure;
          }

          /* save the physical address of the color board (for the gp later) */
          cgtwoextra[unit].cg2x_physaddr =
               (unsigned int)( (page << PGSHIFT)  & 0xFFFFFF);
          return (CG2PROBESIZE);
}

#ifdef DEBUG
     printf("CGTWO testpixel was %X instead of 0xAA\n", testpixel);
     memropc_print(&cg2fb->ropcontrol[CG2_ALLROP].ropregs);
#endif

failure:
#if defined(DEBUG) || defined(lint)
     printf("CGTWO probe failure (opcount == %D)\n", opcount);
#endif
     rmfree(kernelmap, (long)cg2extraclicks, (long)a);
     mapout(&Usrptmap[a], cg2extraclicks);
     return (0);
}

#ifdef DEBUG
memropc_print(rop)
     struct memropc *rop;
{

     printf("ROP CONTROL: dest=%x, src1=%x, src2=%x, pat=%x,\n",
        rop->mrc_dest, rop->mrc_source1, rop->mrc_source2, rop->mrc_pattern);
     printf("mask1=%x, mask2=%x, shift=%x, op=%x\n",
        rop->mrc_mask1, rop->mrc_mask2, rop->mrc_shift, rop->mrc_op);
     printf("width=%x, opcnt=%x, decout=%x, 11=%x,\n",
        rop->mrc_width, rop->mrc_opcount, rop->mrc_decoderout, rop->mrc_x11);
     printf("12=%x, 13=%x, 14=%x, 15=%x\n",
          rop->mrc_x12, rop->mrc_x13, rop->mrc_x14, rop->mrc_x15);
}
#endif
```

```
/*ARGSUSED*/
cgtwoopen(dev, flag)
     dev_t dev;
{
     return (fbopen(dev, flag, NCGTWO, cgtwoinfo));
}

/* When driver is closed destroy the pixrect */
/*ARGSUSED*/
cgtwoclose(dev, flag)
     dev_t dev;
{
     register int unit = minor(dev);

     if ((caddr_t)&cgtwoprdata[unit] == cgtwopixrect[unix].pr_data) {
          bzero((caddr_t)&cgtwoprdata[unit], sizeof (struct cg2pr));
          bzero((caddr_t)&cgtwopixrect[unit], sizeof (struct pixrect));
     }
}

/*
 * cgtwoioctl contains the only pixrect-related code in the whole driver that isn't
 * directly related to providing cursor-painting support to the kernel. The command
 * codes are defined in fbio.h, normally found in /usr/include/sun.
 */
/*ARGSUSED*/
cgtwoioctl(dev, cmd, data, flag)
     dev_t dev;
     caddr_t data;
{
     register int unit = minor(dev);
     register struct cg2fb *cg2fb =
         (struct cg2fb *)cgtwoextra[unit].cg2x_ropaddr;

     switch (cmd) {

     case FBIOGTYPE: {
          register struct fbtype *fb = (struct fbtype *)data;

          fb->fb_type = FBTYPE_SUN2COLOR;
          fb->fb_height = cgtwoextra[unit].cg2x_hwheight;
          fb->fb_width = cgtwoextra[unit].cg2x_hwwidth;
          fb->fb_depth = CG2_DEPTH;
          fb->fb_cmsize = 256;
          fb->fb_size = CG2TOTALSIZE;
          break;
     }
     case FBIOGPIXRECT: {
          register struct fbpixrect *fbpr = (struct fbpixrect *)data;

          /* "Allocate" and initialize pixrect data with default */
          fbpr->fbpr_pixrect = &cgtwopixrect[unit];
          cgtwopixrect[unit] = cgtwopixrectdefault;
```

```
        fbpr->fbpr_pixrect->pr_height = cgtwoextra[unit].cg2x_hwheight;
        fbpr->fbpr_pixrect->pr_width = cgtwoextra[unit].cg2x_hwwidth;
        fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgtwoprdata[unit];
        cgtwoprdata[unit] = cgtwoprdatadefault;
        cgtwoprdata[unit].cgpr_va = cg2fb; /* Fixup pixrect data */
        cg2fb->status.reg.video_enab = 1;  /* Enable video */
        cgtwointclear(cg2fb); /* Clear interrupt */
        break;
    }
    case FBIOGINFO: {
        register struct fbinfo *fbinfo = (struct fbinfo *)data;

        /* Hands down color board info to be passed to the GP */
        fbinfo->fb_physaddr  = cgtwoextra[unit].cg2x_physaddr;
        fbinfo->fb_hwwidth  = cgtwoextra[unit].cg2x_hwwidth;
        fbinfo->fb_hwheight = cgtwoextra[unit].cg2x_hwheight;
        fbinfo->fb_ropaddr = cgtwoextra[unit].cg2x_ropaddr;
        break;
    }
    default:
        return (ENOTTY);
    }
    return (0);
}

/*
 * Just turn off an interrupting cgtwo board. (Don't need to load color map
 * at vertical retrace. Use cg2fb->status.reg.update_cmap instead).
 */
cgtwointclear(cg2fb)
    struct    cg2fb *cg2fb;
{
    cg2fb->status.reg.inten = 0;
}

/*
 * Like most memory mapped devices, the Sun2 color-graphics board is not
 * interrupt driven. cgtwointr, however, does clear interrupts, since
 * they can be accidently set by any user process which has access to the
 * device registers (that is, any process which uses the device by mapping it
 * into user space.) The driver must do this to prevent the kernel from
 * crashing upon receiving an interrupt it can't handle.
 */
int cgtwointr()
{
    return (fbintr(NCGTWO, cgtwoinfo, cgtwointclear));
}

/*ARGSUSED*/
cgtwommap(dev, off, prot)
    dev_t dev;
    off_t off;
    int prot;
```

```
{
     register int unit = minor(dev);

     if (off >= CG2EXTRAOFF) {
          /*
           * If the offset puts us into the last 1.3 Mb of frame-buffer memory (the part
           * that's already been mapped into kernel address space by cgtwoprobe)
           * then fbmmap can be used.
           */
          caddr_t addrsaved = cgtwoinfo[unit]->md_addr;
          int  result;

          cgtwoinfo[unit]->md_addr = (caddr_t)cgtwoextra[unit].cg2x_ropaddr;
          result = fbmmap(dev, off-CG2EXTRAOFF, prot, NCGTWO, cgtwoinfo,
              CG2EXTRASIZE);
          cgtwoinfo[unit]->md_addr = addrsaved;
          return (result);
     } else {
          /*
           * Here we can't call fbmmap since only the first page and the last 1.3
           * Mb of frame buffer memory was mapped into kernel memory space by
           * cgtwoprobe. So we get the PTE for the first page (which is mapped)
           * and then adjust up for the actual offset.
           */
          int page;

          page = getkpgmap(cgtwoinfo[unit]->md_addr) & PG_PFNUM;
          page += btop(off);
          return (page);
     }
}
#endif
```

```
/*
 * (fbutil.c) Frame Buffer Driver Support Utilities
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

/*
 * The routines in this file, called from all of the Sun frame buffer drivers,
 * perform the essential operations necessary for all memory-mapped drivers.
 */

#include "../h/param.h"          /* machine dependent kernel parameters */
#include "../h/systm.h"          /* miscellaneous kernel variables */
#include "../h/dir.h"            /* directories */
#include "../h/user.h"           /* kernel per-process status */
#include "../h/proc.h"           /* unswapped kernel per-process status */
#include "../h/buf.h"            /* I/O buffers */
#include "../h/conf.h"           /* kernel entry-point switches */
#include "../h/file.h"           /* regular files */
#include "../h/uio.h"            /* uio structures */
#include "../h/ioctl.h"          /* ioctl definitions */

/* ../machine is a symbolic link set to either ../sun2 or ../sun3 */
#include "../machine/mmu.h"      /* memory-management unit */
#include "../machine/pte.h"      /* page table entries */

/* ../sundev is the device driver source directory */
#include "../sundev/mbvar.h"     /* bus-interface definitions */

/*
 * Makes the necessary error checks and returns. Everything is ok if the
 * device predefined in the config file and if the probe routine found it as
 * expected.
 */
int fbopen(dev, flag, numdevs, mb_devs)
     dev_t dev;
     int flag, numdevs;
     struct mb_device **mb_devs;
{
     register int unit = minor(dev);
     struct mb_device *mb_dev = *(mb_devs+unit);

     if (unit >= numdevs || mb_dev == 0 || mb_dev->md_alive == 0)
          return (ENXIO);
     return (0);
}

/*
 * There'll almost always be only one installed device, and more likely than
 * not, it won't be interrupt driven, but fbintr is written to be general,
 * so it loops. If fbintr finds an interrupting device, it returns with
 * a 1 after calling intclear to turn off the device's interrupt.
 */
int fbintr(numdevs, mb_devs, intclear)
```

```
        int numdevs;
        struct mb_device **mb_devs;
        int (*intclear)();
{
        register int i;
        register struct mb_device *md;

        for (i = 0; i < numdevs; i++) {

                /*
                 * These two tests amount to the same thing, and using them both is
                 * a bit paranoid. In general, if md->md_alive isn't
                 * set, then the device isn't there.
                 */
                if ((md = *(mb_devs+i)) == NULL)
                        continue;
                if (!md->md_alive)
                        continue;

                if ((*intclear)(md->md_addr)) return (1);
        }
        return (0);
}

/*
 * The per-page memory mapping operation involves adding an offset to the device address
 * to specify a page address in the virtual space occupied by the device, getting the
 * page table entry of that page, and then masking off the unwanted bits. fbmmap
 * can only be used if the page has already been mapped (by the kernel autoconfiguration
 * process) into kernel virtual space.
 */
/*ARGSUSED*/
int fbmmap(dev, off, prot, numdevs, mb_devs, size)
        dev_t dev;
        off_t off;
        int rot;
        int numdevs;
        struct mb_device **mb_devs;
        int size;
{
        struct mb_device *mb_dev = *(mb_devs+minor(dev));
        register int page;

        if (off >= size)
                return (-1);
        page = getkpgmap(mb_dev->md_addr + off) & PG_PFNUM;
        return (page);
}
```

E.3. Sky Floating-Point Driver

```
/*
 * (skyreg.h) Sky Floating Point Processor Registers
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

struct    skyreg {
     u_short   sky_command;
     u_short   sky_status;
     union {
          short     skyu_dword[2];
          long skyu_dlong;
     } skyu;
#define   sky_data  skyu.skyu_dlong
#define   sky_dlreg skyu.skyu_dword[0]
     long sky_ucode;
     u_short   sky_vector;     /* VME interrupt vector number */
};

/* command masks */
#define   SKY_SAVE       0x1040
#define   SKY_RESTOR     0x1041
#define   SKY_NOP        0x1063
#define   SKY_START0     0x1000
#define   SKY_START1     0x1001

/* status masks */
#define   SKY_IHALT      0x0000
#define   SKY_INTRPT     0x0003
#define   SKY_INTENB     0x0010
#define   SKY_RUNENB     0x0040
#define   SKY_SNGRUN     0x0060
#define   SKY_RESET      0x0080
#define   SKY_IODIR      0x2000
#define   SKY_IDLE       0x4000
#define   SKY_IORDY      0x8000
```

```
/*
 * (sky.c) SKY Floating-point Processor Driver
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

/*
 * The Sky driver is quite unusual in that maintains some state information
 * in the kernel user structure. This is because the kernel must context
 * switch the Sky board among the processes that wish to use it. This is not
 * typical, and, in fact, there is currently no way for users to add new
 * devices which, like the Sky board, must be context switched by the kernel.
 *
 * The Sky board is used only with Sun2 machines, and machines with Sky boards
 * are known to have only one installed.
 */

/*
 * Most device drivers include about the same set of system header files,
 * with variation reflecting driver differences in functionality. The system
 * include files are located in directories whose location is fixed relative
 * to the configuration directories (for both source and object distributions.)
 * Note that there is not a sky.h file included here; the sky board is a
 * special case and we know that there' s only one installed.
 */

#include "../h/param.h"          /* general kernel parameters */
#include "../h/buf.h"            /* I/O buffers */
#include "../h/file.h"           /* open file information */
#include "../h/dir.h"            /* file system directories */
#include "../h/user.h"           /* kernel per-process status */

/* ../machine is a symbolic link set to either ../sun2 or ../sun3 */
#include "../machine/pte.h"      /* page table entries */
#include "../machine/mmu.h"      /* memory management unit */
#include "../machine/cpu.h"      /* architecture-related defs */
#include "../machine/scb.h"      /* system control block */

/* ../sundev is the device driver source directory */
#include "../sundev/mbvar.h"     /* bus interface definitions */
#include "../sundev/skyreg.h"    /* sky register definitions */

/*
 * The "page" size (for the VME sky board only) is an offset which must be
 * added to the device base address to get access to the full set of device
 * registers. The second page (page 1) is taken as the supervisor page and
 * allows access to all the registers; the first (0) page is the user page and
 * does not, thus preventing access to the registers needed to load microcode
 * and context switch the device. In user mode it' s only possible to access the
 * registers needed to control floating-point operations.
 */
#define    SKYPGSIZE 0x800

/* auto-configuration information */
```

```
int       skyprobe(), skyattach(), skyintr();
struct    mb_device *skyinfo[1];    /* Only one Sky board */
struct    mb_driver skydriver = {
     skyprobe, 0, skyattach, 0, 0, skyintr,
     2 * SKYPGSIZE, "sky", skyinfo, 0, 0, 0,
};

/*
 * The global variable skyaddr is set in skyprobe to contain the
 * base address of the "supervisor page" (page 1) of the Sky board (the base
 * address of the device registers.)
 */
struct    skyreg *skyaddr;

/*
 * These two global variables are used to control extraordinary aspects of the
 * Sky driver logic:
 *      skyinit is set to 1 when the device (during system initialization)
 * is opened for microcode loading. When the microcode loader closes the
 * device, skyinit is set to 2, indicating that the device is available
 * for general use. This mechanism is necessary to handle the special open
 * state needed for microcode loading.
 *      skyisnew is even more peculiar, being necessary only to distinguish
 * two slightly different versions of the Sky board.
 */
int skyinit = 0, skyisnew = 0;

/*ARGSUSED*/
skyprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     register struct skyreg *skybase = (struct skyreg *)reg;

     /* Is something there? */
     if (peek((short *)skybase) == -1)
          return (0);

     /* If so, is it a Sky board? */
     if (poke((short *)&skybase->sky_status, SKY_IHALT))
          return (0);

     skyaddr = (struct skyreg *)(SKYPGSIZE + reg);
     if (cpu == CPU_SUN2_120 ||
         poke((short *)&skyaddr->sky_status, SKY_IHALT)) {

          /* old VMEbus or Multibus version of the Sky board */
          skyaddr = (struct skyreg *)reg;
          skyisnew = 0;
     } else
          skyisnew = 1;

     return (sizeof (struct skyreg));
```

```
}

/*
 * If it's the new version of the board, then it has to be told what interrupt
 * to respond to. This is true for both vectored and auto-vectored interrupts.
 * In the auto-vectored case, the VME interrupt vector is set to be identical
 * to the 68000 auto-vector for the appropriate interrupt level. For the old
 * version of the Sky board, skyattach does nothing.
 */
skyattach(md)
     struct mb_device *md;
{

     if (skyisnew) {
          if (!md->md_intr) {
               /* auto-vectored interrupts */
               (void) poke((short *)&skyaddr->sky_vector,
                   AUTOBASE + md->md_intpri);
          } else {
               /* vectored interrupts */
               (void) poke((short *)&skyaddr->sky_vector,
                   md->md_intr->v_vec);
          }
     }
}

/*ARGSUSED*/
skyopen(dev, flag)
     dev_t dev;
     int flag;
{
     int i;
     register struct skyreg *s = skyaddr;

     if (skyaddr == 0)  /* skyprobe didn't find the device */
          return (ENXIO);

     if (skyinit == 2) {
          /*
           * skyinit is 2 only when skyclose has previously been
           * called. This is true only in the case where skyclose was
           * called by the microcode loader, and so it's used here to recognize
           * the first time that the device is opened for use by a user
           * process. Thus, it's here that the device (and its related
           * bookkeeping fields) need to be initialized.
           */
          s->sky_status = SKY_RESET;
          s->sky_command = SKY_START0;
          s->sky_command = SKY_START0;
          s->sky_command = SKY_START1;
          s->sky_status = SKY_RUNENB;
          u.u_skyctx.usc_used = 1;
          u.u_skyctx.usc_cmd = SKY_NOP;
          for (i=0; i<8; i++)
               u.u_skyctx.usc_regs[i] = 0;
          skyrestore();

     } else if (flag & FNDELAY)
          /*
           * This open is for the the user program that loads the microcode.
           * This is a special case that allows it to open the device, even
           * though it isn't initialized.
           */
          skyinit = 1;

     else
          return (ENXIO);
     return (0);
}

/*ARGSUSED*/
skyclose(dev, flag)
     dev_t dev;
     int flag;
{

     /*
      * Call skysave in case a user aborted and left the board in an
      * unclean state.  We're really not saving the device state here, but
      * rather calling skysave to ensure that the state is safe for the
      * next user.
      */
     if (skyinit == 2)
          skysave();

     /*
      * This is not the normal case. skyinit is being set to 2 to indicate to
      * skyopen that the device has been initialized.
      */
     if (skyinit == 1)
          skyinit = 2;
     u.u_skyctx.usc_used = 0;
     return (0);
}

/*ARGSUSED*/
skymmap(dev, off, prot)
     dev_t dev;
     off_t off;
     int prot;
{

     if (off)
          return (-1);

     /*
```

```
         * If this is a VME Sky board, and the board has been initialized (its
         * microcode loaded), then allow the user process to have access only to
         * the "user" page. This allows users to do floating-point operations,
         * but not to load microcode. The Multibus Sky board doesn't offer such
         * protection, so any process can load microcode and screw up other users
         * of the board. If this is a VME board, but we've still in the
         * microcode-loading state, allow access to the "supervisor" version of
         * the registers so we can load the microcode.
         */
        off = (off_t)skyaddr;
        if (skyisnew && skyinit == 2)
             off -= SKYPGSIZE;

        off =  getkpgmap((caddr_t)off) & PG_PFNUM;
        return (off);
}

/*
 * skyintr is also quite atypical, being used only for error reporting
 * and to disable interrupts. It must disable interrupts because they may (on
 * the Multibus version for sure) have been accidently set by a user process
 * with access to the device registers. The kernel must be able to handle
 * all the interrupts which can be generated by all the devices, even if it
 * doesn't use them for anything.
 */
/*ARGSUSED*/
skyintr(n)
        int n;
{
            static u_short   skybooboo = 0;

            if (skyaddr && (skyaddr->sky_status & (SKY_INTENB|SKY_INTRPT))) {
                    if (skyaddr->sky_status & SKY_INTENB) {
                         printf("skyintr: sky board interrupt enabled, status = 0x%x\n",
                                      skyaddr->sky_status);
                              skyaddr->sky_status &= ~(SKY_INTENB|SKY_INTRPT);
                              return (1);
                    }
                    if (!skybooboo && (skyaddr->sky_status & SKY_INTRPT)) {
                         printf("skyintr: sky board unrecognized status, status = 0x%x\n",
                                      skybooboo = skyaddr->sky_status);
                              return (0);
                    }
        }
        return (0);
}

/*
 * skysave does the actual work of saving the device state. It has to
 * jump through some hoops to do so, but these hoops are completely device
 * specific.
 */
skysave()
```

```
{
        register short i;
        register struct skyreg *s = skyaddr;
        register u_short stat;

        for (i = 0; i < 100; i++) {
                stat = s->sky_status;
                if (stat & SKY_IDLE) {
                        u.u_skyctx.usc_cmd = SKY_NOP;
                        goto sky_save;
                }
                if (stat & SKY_IORDY)
                        goto sky_ioready;
        }
        printf("sky0: hung\n");
        skyinit = 0;
        u.u_skyctx.usc_used = 0;
        return;

        /* I/O is ready, is it a read or write? */
sky_ioready:
        s->sky_status = SKY_SNGRUN;     /* set single step mode */
        if (stat & SKY_IODIR)
                i = s->sky_d1reg;
        else
                s->sky_d1reg = i;

        /*
         * Check again since data may have been in a long word.
         */
        stat = s->sky_status;
        if (stat & SKY_IORDY)
                if (stat & SKY_IODIR)
                        i = s->sky_d1reg;
                else
                        s->sky_d1reg = i;

        /*
         * Read and save the command register. Decrement it by 1 since it's
         * actually Sky program counter and must be backed up.
         */
        u.u_skyctx.usc_cmd = s->sky_command - 1;

        /*
         * Reinitialize the board.
         */
        s->sky_status = SKY_RESET;
        s->sky_command = SKY_START0;
        s->sky_command = SKY_START0;
        s->sky_command = SKY_START1;
        s->sky_status = SKY_RUNENB;

        /*
```

```
         * Do the actual context save. (Unrolled loop for efficiency.)
         */
sky_save:
        s->sky_command = SKY_NOP;       /* set device to a clean mode */
        s->sky_command = SKY_SAVE;
        u.u_skyctx.usc_regs[0] = s->sky_data;
        u.u_skyctx.usc_regs[1] = s->sky_data;
        u.u_skyctx.usc_regs[2] = s->sky_data;
        u.u_skyctx.usc_regs[3] = s->sky_data;
        u.u_skyctx.usc_regs[4] = s->sky_data;
        u.u_skyctx.usc_regs[5] = s->sky_data;
        u.u_skyctx.usc_regs[6] = s->sky_data;
        u.u_skyctx.usc_regs[7] = s->sky_data;
}

skyrestore()
{
        register struct skyreg *s = skyaddr;

        if (skyinit != 2) {
                u.u_skyctx.usc_used = 0;
                return;
        }
        s->sky_command = SKY_NOP;       /* set device to a clean mode */

        /*
         * Do the actual context restore.
         */
        s->sky_command = SKY_RESTOR;
        s->sky_data = u.u_skyctx.usc_regs[0];
        s->sky_data = u.u_skyctx.usc_regs[1];
        s->sky_data = u.u_skyctx.usc_regs[2];
        s->sky_data = u.u_skyctx.usc_regs[3];
        s->sky_data = u.u_skyctx.usc_regs[4];
        s->sky_data = u.u_skyctx.usc_regs[5];
        s->sky_data = u.u_skyctx.usc_regs[6];
        s->sky_data = u.u_skyctx.usc_regs[7];
        s->sky_command = u.u_skyctx.usc_cmd;
}
```

E.4. Versatec Interface Driver

```
/*
 * (vcmd.h) Include file for user programs that'll give ioctl commands to the
 * Ikon 10071-5 Multibus/Versatec interface.
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

#ifndef _IOCTL_
#include <sys/ioctl.h>
#endif

#define   VPRINT          0100
#define   VPLOT           0200
#define   VPRINTPLOT      0400
#define   VPC_TERMCOM     0040
#define   VPC_FFCOM       0020
#define   VPC_EOTCOM      0010
#define   VPC_CLRCOM      0004
#define   VPC_RESET       0002

/*
 * _IOR and _IOW encode read/write instructions to the kernel within the ioctl
 * command code. These instructions cause the kernel to read the ioctl
 * command argument into user space (_IOR), or to write it into kernel space (_IOW).
 */
#define   VGETSTATE _IOR(v, 0, int)
#define   VSETSTATE _IOW(v, 1, int)
```

```
/*
 * (vpreg.h) Registers for Ikon 10071-5 Multibus/Versatec interface.
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Note that the vpdevice structure actually spans the registers of several
 * contiguous IC devices (a 8259 and a 8237.)  Only the low byte of each
 * (short) word is used.
 */

struct vpdevice {
     u_short   vp_status;     /* 00: mode(w) and status(r) */
     u_short   vp_cmd;        /* 02: special command bits(w) */
     u_short   vp_pioout;     /* 04: PIO output data(w) (unused) */
     u_short   vp_hiaddr;     /* 06: hi word of Multibus DMA address(w) */
     u_short   vp_icad0;      /* 08: ad0 of 8259 interrupt controller */
     u_short   vp_icad1;      /* 0A: ad1 of 8259 interrupt controller */

     /* The rest of the fields are for the 8237 DMA controller */
     u_short   vp_addr;       /* 0C: DMA word address */
     u_short   vp_wc;         /* 0E: DMA word count */
     u_short   vp_dmacsr;     /* 10: command and status (unused) */
     u_short   vp_dmareq;     /* 12: request (unused) */
     u_short   vp_smb;        /* 14: single mask bit (unused) */
     u_short   vp_mode;       /* 16: dma mode */
     u_short   vp_clrff;      /* 18: clear first/last flip-flop */
     u_short   vp_clear;      /* 1A: DMA master clear */
     u_short   vp_clrmask;    /* 1C: clear mask register */
     u_short   vp_allmask;    /* 1E: all mask bits (unused) */
};

/*
 * Warning - this is one of those devices in which the read bits are not
 * identical to write bits.
 */

/* vp_status bits (read) */
#define   VP_IS8237  0x80     /* 1 if 8237 (sanity checker) */
#define   VP_REDY    0x40     /* printer ready */
#define   VP_DRDY    0x20     /* printer and interface ready */
#define   VP_IRDY    0x10     /* interface ready */
#define   VP_PRINT   0x08     /* print mode */
#define   VP_NOSPP   0x04     /* not in SPP mode */
#define   VP_ONLINE  0x02     /* printer online */
#define   VP_NOPAPER 0x01     /* printer out of paper */

/* vp_status bits (write) */
#define   VP_PLOT    0x02     /* enter plot mode */
#define   VP_SPP     0x01     /* enter SPP mode */

/* vp_cmd bits */
#define   VP_RESET   0x10     /* reset the plotter and interface */
```

```
#define   VP_CLEAR    0x08      /* clear the plotter */
#define   VP_FF       0x04      /* formfeed to plotter */
#define   VP_EOT      0x02      /* EOT to plotter */
#define   VP_TERM     0x01      /* line terminate to plotter */

/* vp_mode bits */
#define   VP_DMAMODE 0x47       /* put interface in DMA mode */

/*
 * These two values are used to set the device (which is reticent to disclose
 * that it has issued an interrupt) so that the polling routine can find out.
 */
#define   VP_ICPOLL 0x0C
#define   VP_ICEOI  0x20
```

```
/*
 * (vp.c) DMA driver for Ikon 10071-5 Versatec matrix printer/plotter driver.
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

/*
 * Most device drivers include about the same set of system header files, with
 * variation reflecting driver differences in functionality. The system include
 * files are located in directories whose location is fixed relative to the
 * configuration directories (for both source and object distributions.) vp.h
 * is presumed to be in the configuration directory, where config will have
 * left it and from which it is assumed that driver source files (like this one)
 * are compiled.
 */

#include "vp.h"                /* installed device count -- from config */
#include "../h/param.h"        /* general kernel parameters */
#include "../h/dir.h"          /* file system directories */
#include "../h/user.h"         /* kernel per-process status */
#include "../h/buf.h"          /* I/O buffers */
#include "../h/systm.h"        /* miscellaneous kernel variables */
#include "../h/kernel.h"       /* kernel global variables */
#include "../h/map.h"          /* resource allocation maps */
#include "../h/ioctl.h"        /* ioctl definitions */
#include "../h/vcmd.h"         /* for all Versatec interface drivers */
#include "../h/uio.h"          /* uio structures */

/* ../machine is a symbolic link set to either ../sun2 or ../sun3 */
#include "../machine/psl.h"    /* processor status codes */
#include "../machine/mmu.h"    /* memory-management unit */

/* ../sundev is the device driver source directory */
#include "../sundev/vpreg.h"   /* vp register definitions */
#include "../sundev/mbvar.h"   /* bus-interface definitions */

/*
 * Define the Versatec sleeping priority to be lower than PZERO, that is, make
 * its sleep be uninterruptible by signals. This is appropriate because the
 * events which we'll be waiting for, slow as they may be, are relatively fast
 * and sure (unlike user input) to occur.
 */
#define   VPPRI     (PZERO-1)

/*
 * Define an array of vp_softc structures, one for each of the NVP
 * installed devices. By convention, the names xx_softc and
 * xx_device are used for the private, per-device software state
 * structure.
 */
struct vp_softc {
     int  sc_state;          /* current device state */
     struct buf *sc_bp;      /* buffer mapped to device */
     int  sc_mbinfo;         /* stash for mbsetup's return code */
```

```
} vp_softc[NVP];

/*
 * sc_state bits - passed in VGETSTATE and VSETSTATE ioctl calls.
 * The user-level ioctl command codes are in vcmd.h, normally found
 * in /usr/include/sys
 */
#define    VPSC_BUSY       0400000
#define    VPSC_MODE       0000700
#define    VPSC_SPP        0000400
#define    VPSC_PLOT       0000200
#define    VPSC_PRINT      0000100
#define    VPSC_CMNDS      0000076
#define    VPSC_OPEN       0000001

/* no special encoding in minor device number */
#define    VPUNIT(dev)     (minor(dev))

/*
 * Declare an array of private buf headers, by convention named rvpbuf, one for
 * each of the NVP installed devices.
 */
struct     buf   rvpbuf[NVP];

/* The autoconfig-related declarations. */
int vpprobe(), vpintr();
struct     mb_device *vpdinfo[NVP];
struct     mb_driver vpdriver = {
     vpprobe, 0, 0, 0, 0, vpintr,
     sizeof (struct vpdevice), "vp", vpdinfo, 0, 0, 0,
};

/*
 * vpprobe already indicates the persnickety nature of the device, a
 * nature that will become more clear as we proceed.
 */
vpprobe(reg)
     caddr_t reg;
{
     register struct vpdevice *vpaddr = (struct vpdevice *)reg;
     register int x;

     x = peek((short *)&vpaddr->vp_status);

     /*
      * Note that the device provides a sanity check bit, which
      * we can use to ensure that vpprobe is accurate
      */
     if (x == -1 || (x & VP_IS8237) == 0)
          return (0);

     /* Now reset the 8259; also return 0 if reset fails */
     if (poke((short *)&vpaddr->vp_cmd, VP_RESET))
```

```
          return (0);

     /*
      * Device-specific magic to shut up the device, by setting the 8259 -- it
      * doesn't have enough sense to wait for the driver's instructions, and
      * starts interrupting after being reset. Note that even this isn't
      * straightforward because of register write latency.
      */
     vpaddr->vp_icad0 = 0x12;  /* ICW1, edge-trigger */
     DELAY(1);
     vpaddr->vp_icad1 = 0xFF;  /* ICW2 - don't care (non-zero) */
     DELAY(1);
     vpaddr->vp_icad1 = 0xFE;  /* IR0 - interrupt on DRDY edge */

     /* Also reset the 8237 */
     vpaddr->vp_clear = 1;

     return (sizeof (struct vpdevice));
}

vpopen(dev)
     dev_t dev;
{
     register struct vp_softc *sc;
     register struct mb_device *md;
     register int s;
     static int vpwatch = 0;

     /* Do a variety of error checks upon opening the device. Fail if dev
      * is greater than the configured number of devices, or if the device
      * (which is exclusive open) has already been opened, or if vpprobe
      * failed to find the device as expected.
      *
      * Note that, if the device wasn't found by the probe routine, both
      * vpdinfo[VPUNIT(dev)] and md->md_alive will be 0. Any given
      * driver may chose, for its convenience, to make either test, but it's
      * paranoid to -- as is done here -- make both. (All drivers have
      * access to md->md_alive; this isn't the case with xxdinfo).
      */
     if (VPUNIT(dev) >= NVP ||
         ((sc = &vp_softc[minor(dev)])->sc_state&VPSC_OPEN) ||
         (md = vpdinfo[VPUNIT(dev)]) == 0 || md->md_alive == 0)
          return (ENXIO);

     /*
      * vpwatch is a static local which is set to 0 the first time
      * vpopen is called. This code sets vpwatch to one and then
      * calls vptimo -- the effect is that vptimo gets called only once,
      * the first time a user process calls vpopen. But if you examine
      * vptimo, you'll see that it arranges matters so that it's called
      * repeatedly. This helps to keep the device from locking up.
      */
     if (!vpwatch) {
```

```
                vpwatch = 1;
                vptimo();
        }

        /*
         * Initialize softc state variable. Here we are, among other things, setting
         * sc->sc_state = VPSC_OPEN, which indicates that the device (which is
         * exclusive use) is tied up, and that no one else can open it. We are also
         * dispatching two commands, CLRCOM and VPC_RESET.
         */
        sc->sc_state = VPSC_OPEN|VPSC_PRINT | VPC_CLRCOM|VPC_RESET;

        /* Loop while any command is in process */
        while (sc->sc_state & VPSC_CMNDS) {
                /*
                 * This critical section ensures that only one instance of the driver can
                 * vpwait/vpcmd at any time. vpcmd clears command request
                 * bits as it processes commands. This is absolutely necessary, since
                 * vpcmd intends to actually dispatch a command (posted in
                 * sc->sc_state) to the hardware.
                 */
                s = splx(pritospl(md->md_intpri));
                vpwait(dev);
                vpcmd(dev);
                (void) splx(s);
        }
        return (0);
}

vpclose(dev)
        dev_t dev;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

        sc->sc_state = 0;
}

vpstrategy(bp)
        register struct buf *bp;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(bp->b_dev)];
        register struct mb_device *md = vpdinfo[VPUNIT(bp->b_dev)];
        register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
        int s;
        int pa, wc;

        /*
         * The hardware doesn't support writes to odd addresses or DMA requests
         * of less than two bytes in length.
         */
        if (((int)bp->b_un.b_addr & 1) || bp->b_bcount < 2) {
                bp->b_flags |= B_ERROR;
                iodone(bp);
                return;
        }

        s = splx(pritospl(md->md_intpri));
        while (sc->sc_bp != NULL)
                sleep((caddr_t)sc, VPPRI);

        sc->sc_bp = bp;

        vpwait(bp->b_dev);
        /*Map next request for the now idle device onto the bus for a DMA transfer*/
        sc->sc_mbinfo = mbsetup(md->md_hd, bp, 0);

        vpaddr->vp_clear = 1;

        /* Get the address in DVMA space */
        pa = MBI_ADDR(sc->sc_mbinfo);

        /*
         * Now comes some VERY device-specific code, as we set the DMA transfer
         * address on the device.
         */
        vpaddr->vp_hiaddr = (pa >> 16) & 0xF;
        pa = (pa >> 1) & 0x7FFF;
        wc = (bp->b_bcount >> 1) - 1;
        bp->b_resid = 0;

        /*
         * Note the 2 sequential 8-bit writes into the same address to indicate
         * a 16-bit address!
         */
        vpaddr->vp_addr = pa & 0xFF;
        vpaddr->vp_addr = pa >> 8;

        vpaddr->vp_wc = wc & 0xFF;
        vpaddr->vp_wc = wc >> 8;
        vpaddr->vp_mode = VP_DMAMODE;
        vpaddr->vp_clrmask = 1;

        /*
         * By setting the VPSC_BUSY bit in sc->sc_state, we indicate that the device
         * is to sleep, and that vpwait is to loop. This is because we want to insure
         * that another command doesn't get issued until this DMA transfer is completed.
         */
        sc->sc_state |= VPSC_BUSY;

        (void) splx(s);         /* end of critical section */
}

/*
 * There is no read routine, as this is a write-only device.
 */
/*ARGSUSED*/
```

```
vpwrite(dev, uio)
     dev_t dev;
     struct uio *uio;
{

     if (VPUNIT(dev) >= NVP)
          return (ENXIO);
     return (physio(vpstrategy, &rvpbuf[VPUNIT(dev)], dev, B_WRITE,
              minphys, uio));
}

/*
 * vpwait kills time, but not by busy waiting.  Instead, it relies on the
 * fact that sleep and wakeup aren't proper semaphores, and that ALL
 * processes which are sleeping on a channel wake when a wakeup is issued
 * on that channel.  vpwait's sleep, then, is awaken by vpintr.
 */
vpwait(dev)
     dev_t dev;
{
     register struct vpdevice *vpaddr =
         (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
     register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

     for (;;) {
          if ((sc->sc_state & VPSC_BUSY) == 0 &&
              vpaddr->vp_status & VP_DRDY)
               break;
          sleep((caddr_t)sc, VPPRI);
     }
     return;
}

struct pair {
     char soft;               /* software bit */
     char hard;               /* hardware bit */
} vpbits[] = {
     VPC_RESET,          VP_RESET,
     VPC_CLRCOM,         VP_CLEAR,
     VPC_EOTCOM,         VP_EOT,
     VPC_FFCOM,          VP_FF,
     VPC_TERMCOM,        VP_TERM,
     0,                  0,
};

/*
 * vpcmd is designed to be called after vpwait has returned, thus
 * indicating that the hardware is quiet and ready to receive a new command.
 * When it's called, it runs through the possible command bits in
 * sc->sc_state, and, finding one set, issues the corresponding hardware
 * command to the actual device.  At the same time it clears the command from
 * sc->sc_state, so that the next time vpcmd is called another
 * command will be issued to the hardware.  Note that vpcmd waits a long
 * time, probably too long, for the device to recover before it returns.
 */
vpcmd(dev)
     dev_t;
{
     register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
     register struct vpdevice *vpaddr =
         (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
     register struct pair *bit;

     for (bit = vpbits; bit->soft != 0; bit++) {
          if (sc->sc_state & bit->soft) {
               vpaddr->vp_cmd = bit->hard;
               sc->sc_state &= ~bit->soft;
               DELAY(100);     /* time for DRDY to drop */
               return;
          }
     }
}

/*ARGSUSED*/
vpioctl(dev, cmd, data, flag)
     dev_t dev;
     int cmd;
     caddr_t data;
     int flag;
{
     register int m;
     register struct mb_device *md = vpdinfo[VPUNIT(dev)];
     register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
     register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
     int s;

     switch (cmd) {

     case VGETSTATE:
          *(int *)data = sc->sc_state;
          break;

     /*
      * Turn off VPSC_MODE; restrict the user to resetting it and setting
      * VPSC_CMNDS
      */
     case VSETSTATE:
          m = *(int *)data;
          sc->sc_state =
              (sc->sc_state & ~VPSC_MODE) | (m&(VPSC_MODE|VPSC_CMNDS));
          break;

     default:
          return (ENOTTY);     /* "Not a typewriter" */
     }
```

```
        /*
         * More careful handling to make sure that one command doesn't get issued until the
         * last one has completed. Wait, then post some state information from
         * sc->sc_softc to the hardware, then wait again, then call vpcmd to
         * fire off the next command. And all in a critical section!
         */
        s = splx(pritospl(md->md_intpri));
        vpwait(dev);
        if (sc->sc_state&VPSC_SPP)
                vpaddr->vp_status = VP_SPP|VP_PLOT;
        else if (sc->sc_state&VPSC_PLOT)
                vpaddr->vp_status = VP_PLOT;
        else
                vpaddr->vp_status = 0;
        while (sc->sc_state & VPSC_CMNDS) {
                vpwait(dev);
                vpcmd(dev);
        }
        (void) splx(s);
        return (0);
}

/*
 * This is really a polling interrupt routine. The code at the top that checks
 * the polling chain should really be broken out into a vppoll routine
 * that gets plugged into the mb_device structure. The rest of the code
 * would then be where it properly belongs, in a vpintr routine that can
 * be named in the config file.
 */
vpintr()
{
        register int dev;
        register struct mb_device *md;
        register struct vpdevice *vpaddr;
        register struct vp_softc *sc;
        register int found = 0;

        for (dev = 0; dev < NVP; dev++) {
                if ((md = vpdinfo[dev]) == NULL)
                        continue;
                vpaddr = (struct vpdevice *)md->md_addr;

                /*
                 * It's not easy to find out if an interrupt has occurred.
                 */
                vpaddr->vp_icad0 = VP_ICPOLL;
                DELAY(1);
                if (vpaddr->vp_icad0 & 0x80) {
                        found = 1;

                        /* Wake up the guilty device */
                        DELAY(1);
                        vpaddr->vp_icad0 = VP_ICEOI;
                }

                sc = &vp_softc[dev];

                /* Is there a command currently dispatched and does the hardware
                 * say it's done with it?
                 */
                if ((sc->sc_state&VPSC_BUSY) && (vpaddr->vp_status & VP_DRDY)) {
                        sc->sc_state &= ~VPSC_BUSY;    /* clear busy indicator */
                        if (sc->sc_state & VPSC_SPP) {

                                /*device-specific mode toggle */
                                sc->sc_state &= ~VPSC_SPP;
                                sc->sc_state |=  VPSC_PLOT;
                                vpaddr->vp_status = VP_PLOT;
                        }
                        iodone(sc->sc_bp); /* break wait in physio */
                        sc->sc_bp = NULL;

                        /*
                         * Note that the resources being deallocated here were allocated
                         * in vpstrategy, in the top half of the driver.  This is
                         * standard form for DMA drivers.
                         */
                        mbrelse(md->md_hd, &sc->sc_mbinfo);
                }
                wakeup((caddr_t)sc); /* break loops in vpstrategy AND vpwait */
        }
        return (found);
}

/*
 * vptimo is used to repeatedly kickstart the device, which has a tendency
 * to freeze up if left alone too long.  It calls vpintr, and then it sets
 * up a timer to call vptimo again (and again, and again...) to make sure
 * that a call to vpintr is always pending.  The kernel global hz is set
 * to reflect the clock rate of the system processor chip (it's 50 for a Sun3).
 */
vptimo()
{
        int s;
        register struct mb_device *md = vpdinfo[0];

        s = splx(pritospl(md->md_intpri));
        (void) vpintr();
        (void) splx(s);
        timeout(vptimo, (caddr_t)0, hz);
}
```

Index

6

- 680X0, 11

A

- **adb**, 89
- addresses
  - convenient testing, 69
  - DVMA virtual, 17
  - finding physical, 70
  - kernel space, 55
  - mapping of, 68
  - mapping Sun-2, 69, 72
  - mapping Sun-3, 69, 75
  - selection of virtual, 68
  - space terminology, 6
  - user space, 55
  - virtual space warning, 6
  - virtual to physical mapping, 70
- assert mechanism, 93
- asynchronous tracing, 90
- **attach** routine, 45, 59, 103, 143
- autoconfiguration, 53
  - and initialization, 40
  - Skeleton example, 101
- autoconfiguration-related declarations, 47

B

- **bdevsw**, 35
  - definition, 119
- block driver mechanisms, 5
- bottom half of driver, 55, 56
- 4.2BSD, 40
- building a kernel, 119
- byte order, 22

C

- **cdevsw**, 35, 122
  - definition, 119
- character driver overview, 53
- **close** routine, 103
- commands to PROM monitor
  - **a** — open A register, 131
  - **b** — boot, 131
  - **c** — continue, 131
  - **d** — open D register, 131
  - **e** — open memory, 131
- commands to PROM monitor, *continued*
  - **f** — fill address space, 131
  - **g** — go to address, 132
  - **h** — display monitor commands menu, 132
  - **k** — reset system, 132
  - **l** — open longword, 132
  - **m** — open segment map, 132
  - **o** — open byte location, 132
  - **p** — open page map, 133
  - **r** — open registers, 133
  - **s** — set/query address space, 133
  - **u** — handle serial ports, 134
  - **v** — view memory blocks, 134
  - **w** — vector command, 134
  - **x** — extended boot-path tests, 134
- computer architecture, 11
- **config**, 119
- config file, 121
- configuration, **119**, 120
  - autoconfiguration, 119
  - **conf.c**, 122
  - config file, 121
  - configuration makefile, 120
  - device installation, 120
  - dual address-space devices, 125
  - example, 121
  - MAKEDEV shell script, 123
  - mknod, 124
- context registers, 70
- controllers, 41
- CPU PROM monitor, 67 *thru* 80
  - warning, 79
- CPU PROM monitor commands
  - **a** — open A register, 131
  - **b** — boot, 131
  - **c** — continue, 131
  - **d** — open D register, 131
  - **e** — open memory, 131
  - **f** — fill address space, 131
  - **g** — go to address, 132
  - **h** — display monitor commands menu, 132
  - **k** — reset system, 132
  - **l** — open longword, 132
  - **m** — open segment map, 132
  - **o** — open byte location, 132
  - **p** — open page map, 133
  - **r** — open registers, 133
  - **s** — set/query address space, 133

CPU PROM monitor commands, *continued*
- **u** — handle serial ports, 134
- **v** — view memory blocks, 134
- **w** — vector command, 134
- **x** — extended boot-path tests, 134

CPU state, 71
critical sections, 56

D

data structures, kernel, 41
debugging techniques, 86
/dev directory, 34
device
- as special files, 34
- block devices, 34
- character devices, 34
- checkout, 87
- classes, 34
- devices and controllers, 41
- independence, 3
- initial checkout, 78
- installation, 124
- major numbers, 34
- major types, 3
- memory-mapped installation, 80
- minor numbers, 34
- names, 34
- number macros, 63
- numbers, 34
- peculiarities, 22
- preassigned devices, 38
- slave vrs free devices, 41
- testing, 68
- tty-like devices, 37
- virtual-memory, 82
- warnings, 22

diskless booting, 126
DMA, 27
- devices, 27
- DVMA hardware, 27
- DVMA space, 28
- **DVMA** variable, 29
- Multibus, 112
- no user-level DVMA, 29
- **rmalloc**, 29
- Skeleton Board DVMA, 112
- Sun Main Bus DVMA, 27
- VMEbus, 112

DMA devices, 112
dmesg, 90
driver debugging, 87
driver example, 97
driver listings, 165
driver overview, 53
driver routines, 137
- *xx***attach**", 137
- *xx***close**", 138
- *xx***intr**", 138
- *xx***ioctl**", 139
- *xx***minphys**", 141
- *xx***mmap**", 140
- *xx***open**", 141

driver routines, *continued*
- *xx***poll**", 142
- *xx***probe**", 142
- *xx***read**", 143
- *xx***strategy**", 143
- *xx***write**", 144

driver source code, 123
drivers
- and the kernel, 33
- and user processes, 33
- kernel interface, 48

dual address-space devices, 125

E

error
- handling, 92
- logging, 93
- numbers, 137
- recovery, 92
- returns, 93
- signals, 93

error-handling mechanisms, 92
example driver, 97
Example PTE calculations, 77

F

filesystems, 5
frame buffers, 80
- mapping without drivers, 83

G

getkpgmap, 81

H

hardware peculiarities, 22
heterogeneous networks, 119

I

I/O paths, 35
initial checkout, 78
initial declarations, 53
initial device tests, 79
installation of device, 124
interrupt
- context, 55
- levels, 57
- routines, 45, 54

interrupt number, setting, 59
interrupt-related problems, 26
interrupt-vector assignments, 21
interrupts, 56
- polling, 58
- vectored, 58

intr routine, 54, 110
ioctl macros, 139
ioctl routine, 54, 112

K

kadb, 91

kadb — the kernel debugger, 91
- and virtual spaces, 91

kernel, 87
- booting, 126
- buffer cache, 5
- config file, 121
- configuration, 119
- data structures, 41
- interface, 40
- interface points, 101
- kernel/driver interface, 48
- memory context, 33
- panics, 93
- run-time data structures, 40
- space, 55

L

limitations of this manual, 4

listings, 165

M

Main Bus, 41

Main Bus resource management, 40

major macro, 63

makedev macro, 63

MAKEDEV shell script, 123

manual overview, 7

mapping without drivers, examples, 83

mb_ctlr structure, 42, 59

mb_device structure, 43, 59

mb_driver structure, 44

mb_hd structure, 42

mbglue.s, 120

mbvar structures, 40

MC680X0, 11

memory contexts, 70

Memory Management Unit, 13

memory mapping, 11, 80

memory-mapped device drivers, 80

memory-mapped devices, 55
- installation options, 80

minor macro, 63

minphys routine, 106

mknod, 124

mmap, **80**
- direct opening of devices, 85
- **mmap**, 80
- installation options, 85
- without drivers, 81

mmap routine, 54

MMU
- setting the, 67
- Sun-2, 72
- Sun-3, 75

monitor, 67 *thru* 80
- warning, 79

monitor commands
- **a** — open A register, 131

monitor commands, *continued*
- **b** — boot, 131
- **c** — continue, 131
- **d** — open D register, 131
- **e** — open memory, 131
- **f** — fill address space, 131
- **g** — go to address, 132
- **h** — display monitor commands menu, 132
- **k** — reset system, 132
- **l** — open longword, 132
- **m** — open segment map, 132
- **o** — open byte location, 132
- **p** — open page map, 133
- **r** — open registers, 133
- **s** — set/query address space, 133
- **u** — handle serial ports, 134
- **v** — view memory blocks, 134
- **w** — vector command, 134
- **x** — extended boot-path tests, 134

Multibus, **11**
- 3.0 changes, 15
- adapter, 21
- adapter warning, 25
- byte-ordering issues, 22
- device peculiarities, 22
- DMA, 112
- I/O mapped devices, 12
- I/O space, 11
- I/O Space allocation, 15
- memory allocation, 14
- memory mapped devices, 12
- memory space, 11
- memory types, 12
- MMU, 13
- multibus resource management, 62
- other peculiarities, 24
- Sun-2 Multibus, 13
- Sun-2 Multibus memory map, 14

multiple address-space devices, 125

N

noprintf variable, 90

O

open routine, 103

P

P2 bus, 25

Page Map Entry Groups, PMEGs, 71

page maps, 71

Page Table Entries, PTEs, 71

pixrects, 80

PMEGS, 70

poll routine, 54, 59, 110

polling
- restrictions on, 58

polling chain, 46

polling interrupts, 58

printf
- debugging with, 87
- event triggered, 89
- kernel, 87

- `printf`, *continued*
  - restrictions on, 88
  - `uprintf`, 88
  - usage hints, 89
  - with debuggers, 88
- `probe` routine, 45
- `probe` routine, 101
- `proc` structure, 48
- processes, 70
- processor priorities
  - raising and lowering, 62
- processor priority, 56
- processor state, 71
- PROM monitor, 67 *thru* 80
  - warning, 79
- PROM monitor commands
  - `a` — open A register, 131
  - `b` — boot, 131
  - `c` — continue, 131
  - `d` — open D register, 131
  - `e` — open memory, 131
  - `f` — fill address space, 131
  - `g` — go to address, 132
  - `h` — display monitor commands menu, 132
  - `k` — reset system, 132
  - `l` — open longword, 132
  - `m` — open segment map, 132
  - `o` — open byte location, 132
  - `p` — open page map, 133
  - `r` — open registers, 133
  - `s` — set/query address space, 133
  - `u` — handle serial ports, 134
  - `v` — view memory blocks, 134
  - `w` — vector command, 134
  - `x` — extended boot-path tests, 134
- PTE, 71
  - calculations, 77
  - Sun-2 masks, 73
  - Sun-2 PTE, 73
  - Sun-3 masks, 76
  - templates, 73, 76

R

- raising and lowering processor priorities, 62
- `read` routine, 54, 105
- register peculiarities, 22
- register sequencing logic, 25
- register warnings, 22
- run-time data structures, 40

S

- sample listings, 165
- segment maps, 70
- semaphores, 154
- service functions, 60
  - data-transfer functions, 62
  - multibus resource management, 62
  - `printf`, 63
  - raise and lower processor priorities, 62
  - sleep and wakeup, 61
  - timeout, 61
  - untimeout, 61
- Skeleton driver, 97
- Skeleton Driver declarations, 100
- `sleep` system call, 55, 57
- sleep and wakeup mechanism, 61
- software priorities, 61
- `start` routine, 54, 108
- `strategy` routine, 107
- support routines
  - `CDELAY`, 147
  - `copyin`, 147
  - `copyout`, 147
  - `DELAY`, 148
  - `getkpgmap`, 148
  - `gsignal`, 148
  - `iodone`, 148
  - `iowait`, 148
  - `kmem_alloc`, 149
  - `kmem_free`, 149
  - `MBI_ADDR`, 149
  - `mbrelse`, 149
  - `mbsetup`, 149
  - `panic`, 150
  - `peek`, 150
  - `peekc`, 150
  - `physio`, 150
  - `poke`, 151
  - `pokec`, 151
  - `printf`, 152
  - `pritospl`, 153
  - `psignal`, 153
  - `rmalloc`, 153
  - `rmfree`, 153
  - `sleep`, 154
  - `spl`*n*, 155
  - `splx`, 155
  - `swab`, 155
  - `timeout`, 155
  - `uiomove`, 156
  - `untimeout`, 156
  - `uprintf`, 156
  - `ureadc`, 156
  - `uwritec`, 157
  - `vac_disable_kpage`, 157
  - `wakeup`, 157
- system calls, 34, 55
- system configuration, 119
- System DVMA, 28
- system memory devices, 82
- system reset, 67
- system upgrades, 94
- System V compatibility, 5
- System V differences, 40

T

- timeout mechanisms, 61
- timing problems, 26
- top half of driver, 55
- tracing, 90

U

uio structure, 106
UNIX I/O paths, 35
UNIX source license, 4
upgrades, 94
user context, 55
user space, 55
user structure, 48
user-level routines
- **free**, 161
- **getpagesize**, 161
- **mmap**, 161
- **munmap**, 162
- **valloc**, 162

V

vac_disable_kpage, 141
valloc routine, 84
vector numbers, 58
vectored interrupts, 58
virtual memory devices, 82
virtual to physical mapping, 70
VMEbus, **16**
- 16-bit allocation, 20
- 24-bit allocation, 20
- 32-bit allocation, 20
- allocation of VMEbus memory, 19
- device address assignments, 20
- DMA, 112
- generic, 18
- Multibus Adapter, 21
- Sun-2 VMEbus, 16
- Sun-2 VMEbus address spaces, 16
- Sun-2 VMEbus memory types, 16
- Sun-3 address spaces, 18
- Sun-3 VMEbus, 19
- Sun-3 VMEbus address types, 18

VMEbus machines, 16

W

write routine, 54, 105

Revision History

Rev	Date	Comments
A	15 July 1983	First release of this Manual as part of the *System Internals Manual for the Sun Workstation.*
B	15 August 1983	Minor corrections.
C	15 November 1983	Minor corrections.
D	19 November 1984	Minor corrections.
E	15 May 1985	Separated out of the *System Internals Manual for the Sun Workstation* to form a standalone manual. Added narrative to deal with VMEbus and support for vectored interrupts.
F	17 February 1986	Upgrade for release 3.0. Included details of Sun-3 architecture and 32-bit VMEbus. *Using the CPU PROM Monitor*, much more on address spaces, `mmap`, and DVMA. Many bug fixes.
50	15 June 1986	Most of the overall organizational changes necessary for the final version, as well as the bulk of the small changes.
A	15 October 1986	Upgrade for release 3.2. Major expansion of run-time environment, development and debugging sections. General restructuring, expansion and improvement.

Notes