sun®
microsystems

Writing Device Drivers

Part Number: 800-3851-10
Revision A of 27 March, 1990

Contents

Tables

Figures

1 Introduction

This manual is a guide to adding drivers for new devices to the SunOS kernel. It comes in two parts.

- Part One, *Regular Device Drivers*, discusses a variety of issues relevant to standard (non-STREAMS) device drivers. It is intended to be self-contained, and to include all necessary discussion of hardware and kernel topics.
- Part Two, *Non-STREAMS Appendices*, includes reference material related to regular (non-STREAMS) drivers.

Throughout the manual, statements that apply only to specific machines, e.g. Sun-3's, Sun-3x's, Sun-4's, SPARCstations or Sun386i's, will be clearly flagged to that effect.

1.1. Device Independence

One of SunOS'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 SunOS 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 latter into calls to their corresponding device drivers. These drivers control *all* device operations; devices do nothing until their drivers tell them to.

System calls provide the interface between user processes and the SunOS kernel, while device drivers provide an interface between the kernel itself and its peripheral devices. Device drivers are therefore crucial elements in SunOS'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. The loadable driver capability makes it possible to attach a driver to a system without rebuilding the kernel and rebooting the system. For more information on how to reconfigure your kernel to include new device drivers, see the *Configuring the*

Kernel chapter of this manual, the *SunOS STREAMS Topics* chapter of the *STREAMS Programming* manual, the *Adding Hardware to Your System* chapter of *Network Programming* 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 ATbus devices to their system. It does *not*, however, explain how to write drivers for all possible Sun devices.

For information specific to SPARCstation machines and to writing drivers for SBus-based devices, see *Writing Device Drivers for the SBus*, part number 800-4455-01, which is one of the manuals in the *SBus Developer's Kit*, part number 825-1219-01.

We can classify devices into nine major categories:

1. Co-processors.
2. Disks and tapes.
3. Network interface drivers such as Ethernet or X.25.
4. SCSI devices. For more information see your Sun Representative for information on the two documents, "SCSI Implementation Guide" Part Number 800-4700-10, Rev A of 15 November, 1989 and "SCSA: SUN Common SCSI Architecture" Part Number 800-4701-10, Rev A of 15 November, 1989.
5. Serial communications multiplexors.
6. 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.
7. Programmed I/O devices, that is, devices which send and receive data on the main system bus under direct control of the system CPU.
8. Frame buffers and other memory-mapped devices. Such devices are typically mapped into user-process memory and then accessed directly.
9. So called *pseudo devices*, which are actually drivers without associated hardware devices.

This manual does *not* cover driver development for devices in categories 1, 2, 3, 4 and 5. Part one does discuss drivers for the devices in categories 6, 7, 8 and 9. STREAMS-related information of interest to programmers planning drivers for serial communications devices should see the *STREAMS Programming* manual. The majority of the devices which users will want to add to their systems, from categories 6 through 9, include:

- input devices like mice, digital tablets and analog-to-digital converters, (though these are usually implemented as streams drivers independent of Sun View).
- output and display devices like frame buffers, printers, and plotters,

- 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 disk drivers or drivers for any *structured* or *block I/O* devices, for such drivers are quite difficult to write well. Besides, most customers will find that the structured-device drivers provided with the standard system software fill their needs quite adequately. The extensive use of standards within the Sun product line will allow them to use hardware interfaces already provided by Sun to drive whatever 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 will also want to start with an existing driver, and will thus need a source-code license). For consultation, please see your Sun Representative who will put you in contact with Consulting Services.

Finally, this manual doesn't really discuss the issues relevant to serial communications and local network interface driver development. 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 will need a source license to go it alone).

This manual is primarily 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 SunOS filesystems can be mounted. Such devices (almost always disks) 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

[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 `xxstrategy()` 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 make 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.

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

The SunOS applications interface is almost completely compatible with that of AT&T's System V UNIX system. The driver/kernel interface, however, is not. In general, though, drivers that were written for System V (or V7 or 4.1BSD, which have driver interfaces similar to System V) will be easily ported to SunOS, 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.

Sun device drivers differ from typical System V drivers because the Sun operating system has 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-Management Unit (MMU) and the MMUs of other systems. Consequently, drivers which map addresses require a lot of Sun-specific code.

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. If you have written the driver as a loadable driver, then compile the driver and use the `modload(1)` command to load the driver into a running 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 has its own distinct virtual address space. A user process (or the kernel) can make arrangements to share address space 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 will 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 the *Driver Development Topics* chapter for more information.

1.7. Manual Overview

Regular Drivers

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

Chapter 7 explains how to add new drivers to the SunOS kernel.

Chapter 8 explains pseudo-drivers, and provides source and installation instructions for a real ramdisk pseudo-driver.

Appendices

Appendix A summarizes the device driver routines available to all device driver writers.

Appendix B describes all the kernel support routines useful in developing device drivers.

Appendix C describes the user-level routines useful in driver development.

Appendix D contains a number of annotated driver listings to show how sample drivers are written.

Last Word

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 of the *Driver Development Topics* chapter.

And finally, note that if you have no previous experience writing UNIX device drivers, you should expect to seek some help from the Sun Technical Support or Consulting organizations. Contact your Sun Representative for more information.

PART ONE: Regular Device Drivers

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. Fortunately, the Sun kernel provides facilities (described in the *Other Kernel/Driver Interfaces* section of the *Overall Kernel Context* chapter) by which a driver can determine the type of the machine upon which it's running.

2.1. VMEbus Machines

VMEbus machine architecture 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 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-3/Sun-3x/Sun-4 Address Spaces

Sun-3, Sun-3x and Sun-4 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. In all these memory references, we are referring to virtual VMEbus addresses, not Sun physical memory locations.

The SPARCstation line of machines utilizing the SBus is not documented here. Refer to the *SBus Developer's Kit*, part number 825-1219-01. The information that follows pertains to Sun-4 VME; it does not apply to SPARCstations SBus.

Table 2-1 *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 and Sun-4 lines. The following table indicates their sizes and physical address mappings.

Table 2-2 *Sun-3/Sun-4 Page Table 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 - 0xFEFFFF)		
2	`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		
3	`vme24d32` — Stolen from top 16M of `vme32d32` (0x0 - 0xFEFFFF)		
3	`vme16d32` — Stolen from top 64K of `vme24d32` (0x0 - 0xFFFF)		

The Sun-3x is different than the Sun-3 and Sun-4 in that the hardware does not use page table entries (PTE's) with a type identifier to map the devices into physical memory. The Sun-3x uses absolute physical addresses when mapping devices. Therefore the type field is not used as an identifier of physical address mapping. The next two tables show the virtual VME addresses and the corresponding physical addresses for the specific ranges. Note for the Sun-3x there is no `vme32d16` entry and there is a hole in the address space usage from the end of the on-board I/O area to the beginning of the `vme16d16` area.

Table 2-3 *Sun-3x VMEbus Address Types*†

Address-Space Name	*Address Size*	*Offset Address*
`vme24d16`	32 bits	`0x0 – 0x00FFFFFF`
`vme32d32`	32 bits	`0x0 – 0x7FFFFFFF`
`vme16d16`	—	`0x0 – 0xFFFF`
`vme24d32`	—	`0x0 – 0xFFFFFF`
`vme16d32`	—	`0x0 – 0xFFFF`
†*Types are not used with the Sun-3x architecture.*		

Table 2-4 *Sun-3x Physical Address map*

Address-Space Name	*Address Size*	*Address Range*
On-board Memory	32 bits	`0x00000000 – 0x57FFFFFF`
On-board I/O	32 bits	`0x58000000 – 0x6EFFFFFF`
`vme16d16`	32 bits	`0x7C000000 – 0x7C00FFFF`
`vme16d32`	32 bits	`0x7D000000 – 0x7D00FFFF`
`vme24d16`	32 bits	`0x7E000000 – 0x7EFFFFFF`
`vme24d32`	32 bits	`0x7F000000 – 0x7FFFFFFF`
`vme32d32`	32 bits	`0x80000000 – 0xFFFFFFFF`

Sun-3/Sun-3x/Sun-4 space overlays are much more complex than those of earlier Sun machines, as is evident from both the tables above and the diagrams 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 overlaid space. One simply cannot address above `0xFF000000` in 32-bit VMEbus space or above `0x00FF0000` in 24-bit VMEbus space.

As the following diagrams illustrate, Sun-3 and Sun-4 addressing schemes are almost identical. They differ only in the size of the virtual address which — output by the CPU or a DVMA Bus Master — is fed to the MMU.

The Sun-3x, which has the MMU on the CPU chip, is a different hardware architecture than the Sun-3's and Sun-4's. There is a full 32 bit input to the MMU from the CPU, and all 32 bits are used for input to the OnBoard and vme modules. No Sun devices use the `vme32d16` so it is not part of the memory map.

Figure 2-1 *Sun-3 VMEbus Address Spaces*

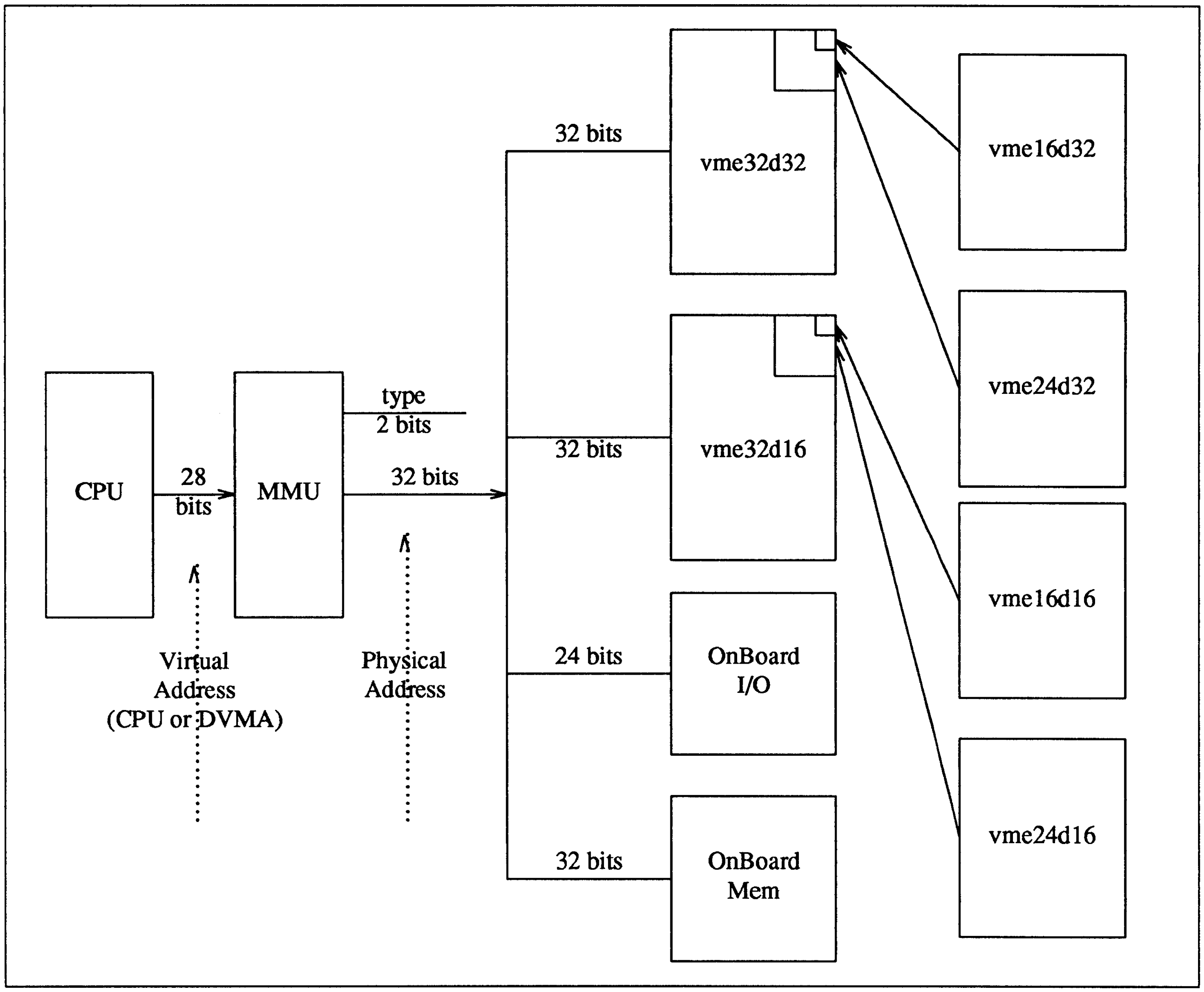


Figure 2-2 *Sun-3x VMEbus Address Spaces*

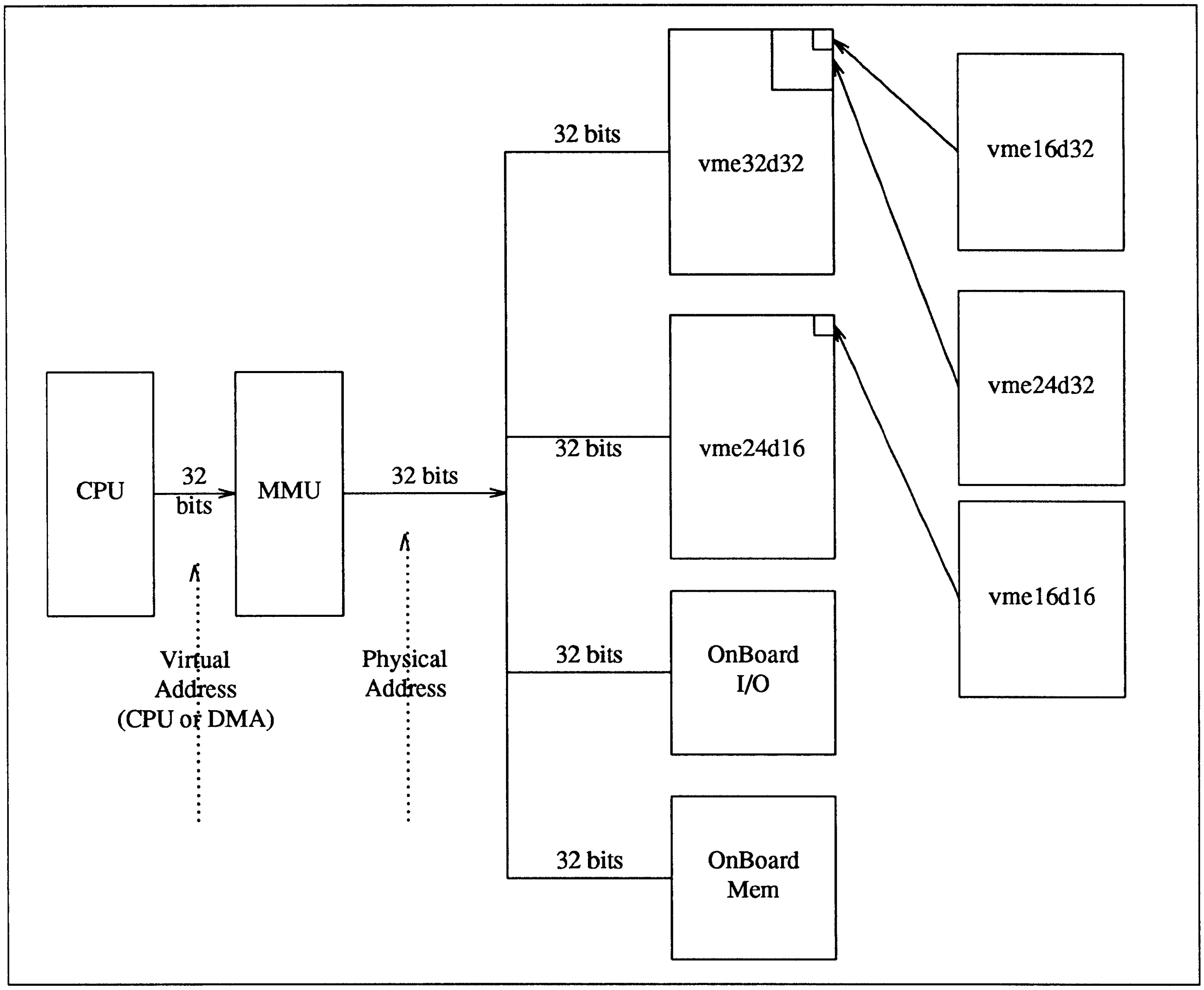
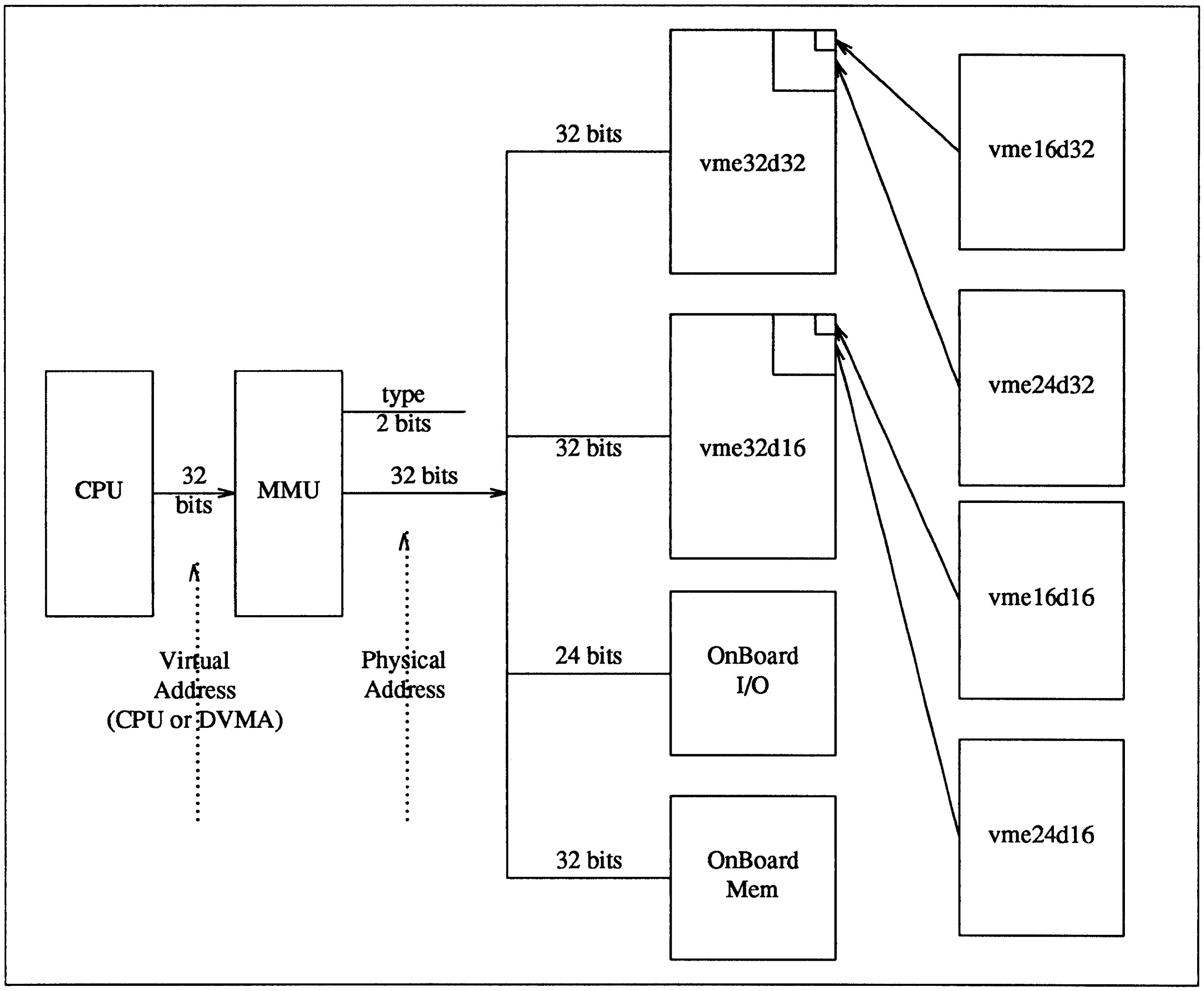


Figure 2-3 *Sun-4 VMEbus Address Spaces*



Allocation of VMEbus Memory

This section summarizes the typical use of the 16, 24 and 32-bit VMEbus address spaces by Sun devices. Note 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 needs to be shifted later.

Table 2-5 *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 `0xFFFF0000` to `0xFFFFFFFF` (on Sun-3's, Sun-3x's, and Sun-4's). 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 2-6 *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`		Reserved for 16-bit VMEbus space

Table 2-7 *32-bit VMEbus Address Space Allocation (Sun-3's, Sun-3x's, Sun-4's)*

Address	*Size*	*Description of Use*
`0x00000000`	1MB	DVMA Space
`0x00100000`	15MB	Reserved
`0x01000000`	112MB	<=2MB Sun Devices
`0x08000000`	128MB	Sun graphic Devices
`0x10000000`	80MB	<=2MB OEM Devices
`0x15000000`	176MB	>2MB OEM Devices
`0x20000000`	1536MB	>2MB Sun Devices
`0x80000000`	1920MB	Reserved
`0xF8000000`	48MB	Sun-4/110 Sun Devices
`0xFB000000`	64MB	Sun-4/110 OEM Devices
`0xFF000000`	16320KB	Reserved for 24 bit addr space
`0xFFFF0000`	64KB	Reserved for 16 bit addr space

These same assignments apply to both 16-bit-data and 32-bit-data VMEbus accesses. Note that, at least in the GENERIC kernel, there are some Sun devices (`vpc0, vpc1 and mti0-4`) installed in the OEM/user area. It's always best to check, when choosing an installation address, that you aren't going to conflict with an already installed device.

Table 2-8 *VMEbus Address Assignments for Some Devices*

Device	*Addressing*	*Addresses Used*
VMEbus SCSI Board	`vme24d16`	0x200000 - 0x2007FF
Graphics Processor	`vme24d16`	0x210000 - 0x210FFF

This table is, of course, not complete. There is always a variety of devices on the bus, 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.

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 will 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 interrupt 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 2-9 *Vectored Interrupt Assignments*

Vector Numbers	*Description*
0x00 *thru* 0x3F	reserved for internal processor traps
0x40 *thru* 0x43	sc0, sc? si0, si? — SCSI Host Adapters
0x44 *thru* 0x47	xdc0, xdc1, xdc2, xdc3 — Xylogics 7053 Disk Controller
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/Sun-3x 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	Reserved — currently unused
0xB4 *thru* 0xB7	SunLink Channel Attach
0xB8 *thru* 0xC7	Reserved for Sun Use
0xC8 *thru* 0xFF	Reserved for Customer Use

2.2. ATbus Machines

The Intel 80386 processor handles I/O devices placed in either memory space or in I/O space. On the 80386, memory-mapped I/O provides additional programming flexibility. Any memory instruction can access any I/O port located in the memory space. For example, the MOV instruction transfers data between any register and any port. The AND, OR, and TEST instructions can be used to manipulate bits in the internal registers of a device.

On some devices, reading a register will not read back what was written. Therefore, instructions such as AND, OR, and TEST can, in some cases, produce unexpected results because the instruction reads a good location, changes it, and writes it back. See the *Other Device Peculiarities* section, ahead.

Memory-mapped I/O can use the full complement of instructions. The 16 MB memory of AT memory exists in the 4 GB physical address space of the Sun386i at 0xE000 0000. For example, a device that, on an AT, shows up in memory at D0 0000 will show up in the Sun386i physical memory at 0xE0D0 0000. Virtual addresses are assigned during the autoconfiguration process.

If an I/O device is mapped into the I/O space then the IN, OUT, INS, and OUTS instructions are used to communicate to and from the device. All I/O transfers are performed via the AL (8-bit), AX (16-bit), or EAX (32-bit) registers. The first 256 bytes of the I/O space are directly addressable. The entire 64 Kbyte I/O space is indirectly addressable through the DX register.

The Sun386i has 21 interrupt channels, but only 11 are available to devices on the AT bus. The following list of interrupt channel assignments shows all of the interrupt channels.

Table 2-10 *Interrupt Channel Assignments*

*AT Channel**	*Assignee*
3	AT Pin B25
4	AT Pin B24
5	AT Pin B23
6	Not available (system diskette)
7	Not available (parallel port)
8	SCSI
9	AT Pin B04
10	AT Pin D03
11	AT Pin D04
12	AT Pin D05
13	Not available (Ethernet)
14	AT Pin D07
15	AT Pin D06

* *Available to AT Cards*

When you add an AT card to the AT bus, you must select one of the values in the Channel column for the AT card's jumpers. For example, if you select channel 10 for a serial card, the "device" line in the config file might look as follows:

```
device  ns0 at atio ? csr 0x3f8 irq 10 priority 6
controller  fdc0 at atmem ? csr 0x001000 irq 6 priority 2
```

The Sun386i does not permit two AT cards to use the same interrupt channel.

Some cards will also use DMA and will have jumpers to select a DMA channel to use. The following list shows that DMA channels 0-3 and channel 5 are available for AT cards. Note that channel 0 and 5 can be used with 16-bit DMA devices; 1, 2, and 3 can be used only with 8-bit DMA devices. Note also that channels 4, 6, and 7 are pre-assigned.

The main difference is that the DMA controller is on the CPU board, not on the device. The AT bus does not support bus master devices, they must allocate a DMA channel from the DMA controller on-board the the CPU. The Sun machine uses an Intel 82380 for the interrupt and DMA controller, instead of the

8259 chip. See the Intel manual for more details.

Table 2-11 *Sun386i DMA Channel Assignments*

Channel	*Assignee*	*Size (bits)*
0	AT Bus	16
1	AT Bus	8
2	AT Bus	8
3	AT Bus	8
4	Software	Not Available
5	AT Bus	16
6	Ethernet	16
7	SCSI	16

For example, you might set up a controller that uses DMA channel 3. For this, the "controller" line in the config file might look like: this:

```
controller wds0 at atio ? csr 0x320 dmachan 3 irq 3 priority 3
```

The Sun386i does not permit two AT cards to use the same DMA channel.

In these examples, ''priority'' refers to the `spl` levels used in the driver. That is, the phrase ''priority 3'' implies that the driver uses `splr(pritospl(3))` to protect its critical regions.

Loadable Drivers

On Sun machines, device drivers can be dynamically loadable. That is, they can be attached to a system without rebuilding its kernel and without having to bring the system down and restart it. See the *Adding and Removing Loadable Drivers* section of the *Configuring the Kernel* chapter for details.

DOS and SunOS Environments

The Sun386i system supports both DOS drivers and SunOS drivers. Only 8086 type devices and their drivers are supported in the DOS environment. Boards which need to use memory above 1 Megabyte or drivers which use 286/386 specific instructions are not supported.

You can attach a DOS device driver in the standard DOS way, but it will be usable only from within the DOS environment. Usually, all you need to do is to first plug in an add-in board. Then you insert an installation diskette (which comes with the board) into Drive A> and re-boot the system. The device driver is already compiled and linked. Generally, the diskette contains programs called "INSTALL" or something similar. You execute this program by typing its name. It copies the driver file from the diskette to the hard disk. At the same time, this procedure will modify the disk's `config.sys` file.

The DOS system must be re-booted. The device driver will automatically be loaded into memory, its options will be parsed, and the driver will be initialized.

NOTE *The DOS driver on the Sun386i is running under SunOS and DOS, but the driver is unaware of this. SunOS might switch control to another task during device operation, so strict timing dependencies could fail. Real time devices, for* *example, may not work properly. If a peripheral and controller have strict timing requirements, their drivers should be written in the standard SunOS style. DOS drivers do not run at the elevated priority of SunOS drivers.*

SunOS drivers, of course, are parts of the system kernel. Thus the timing requirements of most devices can be met under SunOS. SunOS drivers are accessible from the DOS environment via the device nodes /dev in the Unix filesystem.

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.

NOTE *Multibus is NOT supported in SunOS 4.1, but is included here for the benefit of all who are porting from a Multibus environment over to the other busses that Sun supports.*

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 either Motorola MC680X0 family or the Sun SPARC processor (the reduced instruction set CPU upon which Sun-4 machines are built). 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

The Sun-3 and Sun-3x processors are members of the Motorola MC680X0 family, while Sun-4 processors are based on the SPARC CPU. All of these processors address bytes within words by what we shall call *IBM conventions* — the most significant byte of a word is stored at the lowest addressed byte of the word. The Multibus, on the other hand, uses *DEC conventions* — the least significant byte of a 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/SPARC 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 machine *words* with significant internal structure (or structures that contain words) are moved across the bus interface. (If you write only words, and the device uses only words, there's no problem). The Multibus 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 and SPARC CPUs place byte 0 in bits 8 through 15 of the 16-bit word, whereas the Multibus places byte 1 in those bits. If you did everything with the CPU, 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.

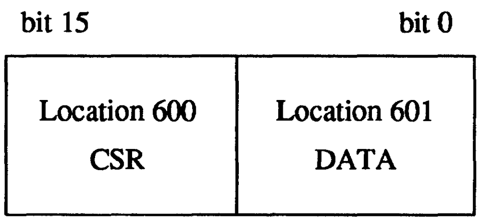
To clarify this, consider an interface for a hypothetical Multibus board containing only two 8-bit I/O registers, namely a control and status register (csr) and a data register (we actually use this design later on in our example of a simple device driver). In this board, we place the command and status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

Hypothetical Board Registers



so that if you were to read location 600 from the point of view of the 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/SPARC CPU and the Multibus.

Other Multibus-related Peculiarities

- Many Multibus device controllers are geared 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 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.
- 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 support only 8-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.

Sun-4/SPARC Peculiarities

There are two peculiarities which are specific to machines built upon the Sun SPARC CPU (currently, just Sun-4's) which can impact device drivers. For more information about the Sun-4 machine architecture, see *Porting Software to SPARC Systems*, part number 800-1796-01.

- The first problem is structure alignment. In MC680X0 family processors, structures are aligned on half-word boundaries, but on Sun-4's, the structure-alignment requirements are imposed by the most strictly-aligned structure components. For example, a structure containing only bytes and characters has no alignment restrictions, while a structure containing a `double` word must be constructed so as to guarantee that this word falls on a 64-bit boundary.

  Programmers must be aware of these rules when writing drivers, for Sun-4 compilers will pad structures to enforce them, and such padding will not always be correct for structures intended to map to device registers. Also, structures must be carefully designed if drivers are to be portable across machine architectures.

- The second problem is data alignment. In MC680X0 family processors, characters are aligned on byte boundaries, while integers of all sizes are aligned on 16-bit boundaries. In Sun-4 machines, in contrast, all quantities must be aligned on their "natural" boundaries: 16-bit half words on 16-bit boundaries, 32-bit words on 32-bit boundaries and 64-bit `double` words on 64-bit boundaries.

  In normal programs, details such as these are handled by the compiler. In drivers, however, more care must be taken. SPARC (unlike the MC68010) doesn't break down 32-bit transactions into successive 16-bit transactions.

Thus, there are times when 32-bit entities have to be broken down by the driver if they are to get across the bus correctly. More specifically, 32-bit or 64-bit alignment is not possible in the 16-bit VMEbus spaces, and thus 32-bit and 64-bit data access does not exist. In the 32-bit VMEbus spaces, all data paths exist.

Other Device Peculiarities

There are other device peculiarities of interest to the driver developer. 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 an 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. This is why compiler optimization can do the wrong thing for kernel code.
- 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 DELAY0) when writing out characters with an 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.

- 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 facility.
  - 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

NOTE *Sun-3 and Sun-4 machines use Direct Virtual Memory Access (DVMA) to allow devices on the Main Bus (a VMEbus) to perform DMA transfers from and to system virtual address space. In the Sun386i system, however, the Memory Management Unit (MMU) is incorporated directly on the Intel 80386 chip itself; devices need to use physical addresses. Sun386i DMA is discussed in the next Section.*

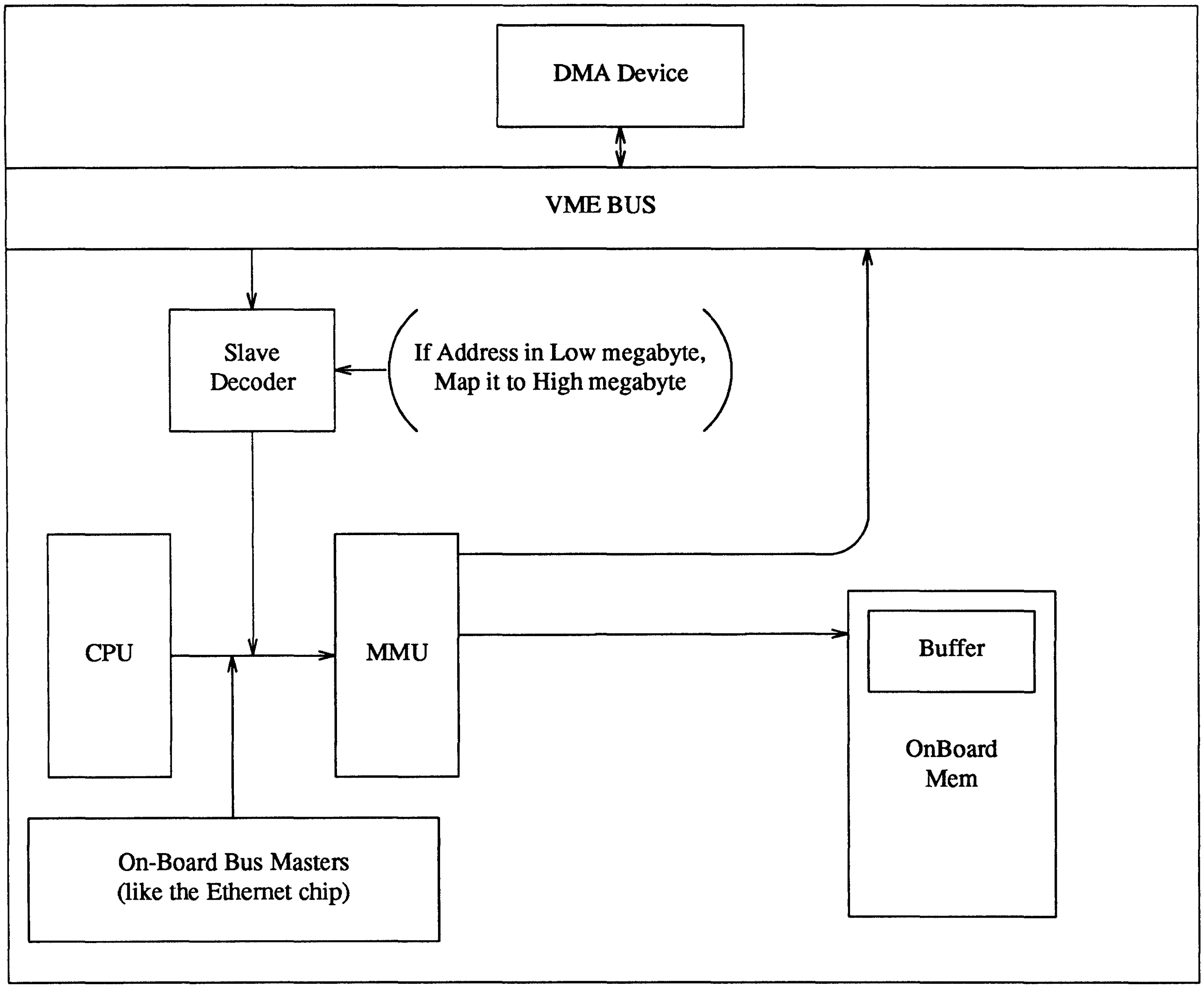
Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun Memory Management Unit to allow devices on the Main Bus (a VMEbus) to perform DMA directly to Sun processor memory. It also allows Main Bus master devices to do DMA directly to Main Bus slaves without the extra step of going through 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.

- On the Sun-3, Sun-3x, and Sun-4 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 (`0x0FF00000` to `0xFFFFFFF` for the Sun-3 and `0xFFF00000` to `0xFFFFFFFF` for the Sun-3x and Sun-4). The Sun-3, Sun-3x, and Sun-4 DVMA hardware use 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)

Devices 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.
- You can also use `kmem_alloc()` to allocate some kernel memory, and then use `mbsetup()` to gain access to it.

When dealing with addresses which are in DVMA space, the driver must strip off the high bits by subtracting the external variable `DVMA`, which contains the address of DVMA (declared as an array of characters). `DVMA` is initialized by the system to `0xFFF00000` for Sun-3's, Sun-3x's and Sun-4's. If the driver fails to make this adjustment, the device will attempt to use a null address — in the high megabyte — and the CPU board will not respond to it.

NOTE *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.

The kernel supports the redundant mapping of physical memory pages into multiple virtual addresses as a means of providing DVMA between devices and user address space. 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.

DMA on ATbus Machines

The Sun386i uses the Intel 80386 chip. This chip has an integrated MMU, so the I/O devices cannot access the Sun MMU address-translation facility and therefore must use physical addresses to access memory directly.

To do DMA on the Sun386i, you must make certain changes in the kernel's memory map (its page tables). Use the `mbsetup()`, `dma_setup()`, `mbrelse()`, and `dma_done()` routines to make these changes. The changes you must make to the kernel memory map are described with these routines in the *Kernel Support Routines* appendix.

3 Overall Kernel Context

3.1. The System Kernel

Device drivers are parts of the SunOS 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 entirely paged. Certain parts of the Sun386i kernel are paged, but drivers can safely assume that their text and data 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 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.

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 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 a 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, these devices are viewed 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 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, character, FIFO, or socket), 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).

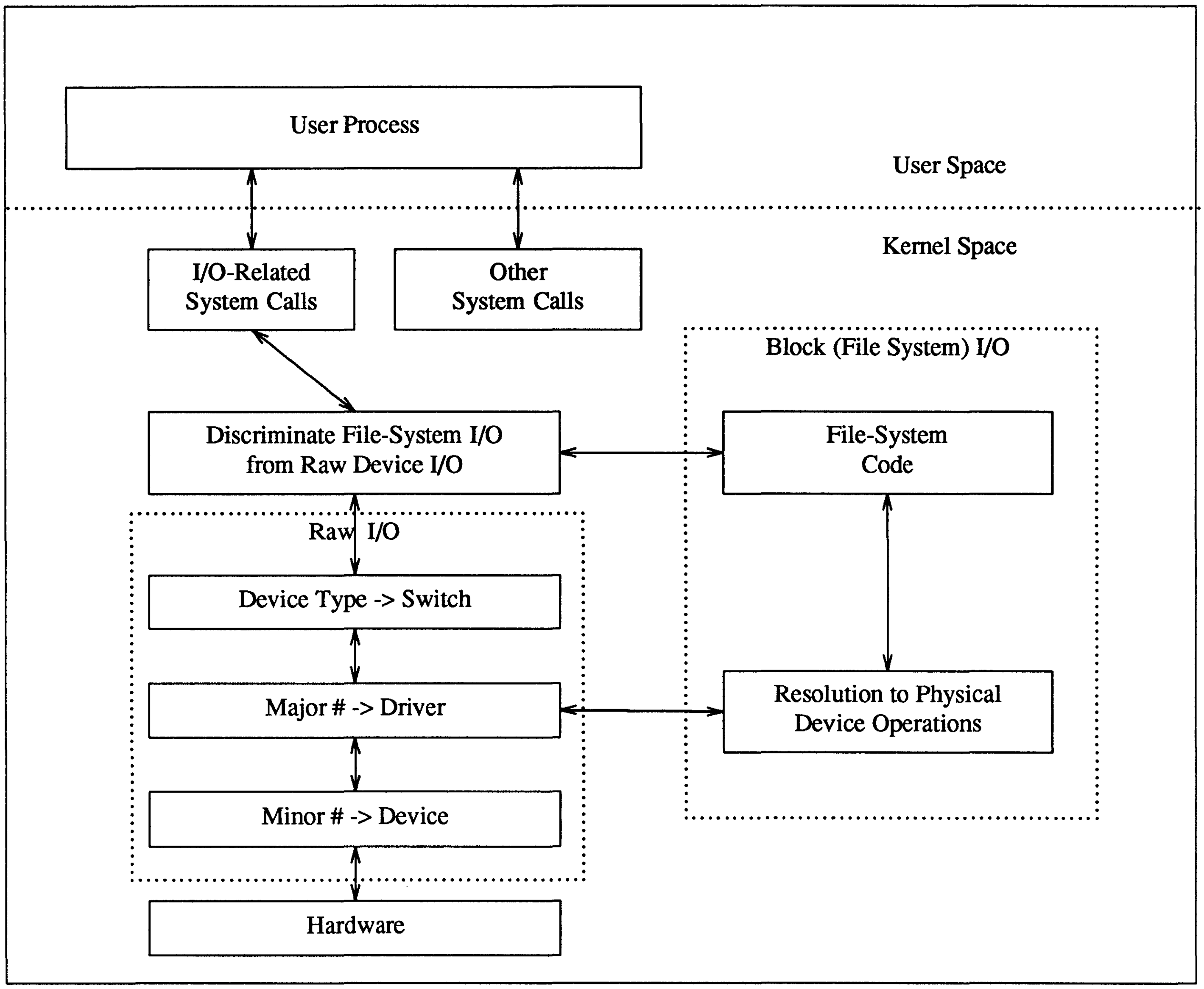
Using the `ls -l` command on the `/dev` directory shows you the i-node information associated with special files:

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

Type	*permissions*	*link*	*owner*	*major #*	*minor #*	*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 *I/O Paths in the UNIX system*



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 will 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 to 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.

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 SunOS code and the driver. The declaration of the device switches is in /usr/share/src/sys/sys/conf.h.

```
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_reset)();     /* reset device and recycle its bus resources */
    int (*d_select)();    /* routine to call to select the device  */
    int (*d_mmap)();      /* routine to call to mmap the device  */
    struct streamtab *d_str; /* support for STREAMS */
    int (*d_segmap)(); /* handles mmap devices that support d_mmap */
*/
};
```

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/share/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,      seltrue,    cgonemmap,
0,      spec_segmap,
},

        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 *some* of these assignments at the time of this writing. It is not complete, and besides, new devices are always being added. In allocating a major number to the new device which you're installing, make sure that you don't choose one that's already been allocated. /usr/sys/sun/conf.c will give the major device numbers as currently allocated on your system. Choose yours so it will 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	mm	Memory special files
4	*Not Available*	No Device
5	tm	Raw Tapemaster Tape Device
6	vp	Ikon Versatec Parallel Controller
7	*Not Available*	No Device
9	xy	Raw Xylogics Disk Device
10	mti	Systech MTI
11	des	DES Chip
12	zs	UARTS
13	ms	Mouse
15	win	Window Pseudo Device
16	*Not Available*	Log Device

Table 3-2 *Current Major Device Number Assignments— Continued*

Major Device Number	*Device Abbreviation*	*Device Description*
17	`sd`	Raw SCSI disk
18	`st`	Raw SCSI tape
19	*Not Available*	No Device
20	`pts`	Pseudo TTY
21	`ptc`	Pseudo TTY
22	`fb`	Sun Console Frame Buffer
25	`pi`	Parallel input device
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
34	`fpa`	Floating-Point Accelerator
35	*Not Available*	STREAMS Support
36	*Not Available*	No Device
37	*Not Available*	STREAMS Clone
38	`pc`	Sun PC Driver
39	`cgfour`	Sun-3/110 Color Frame Buffer
40	*Not Available*	STREAMS NIT
41	*Not Available*	Dump Device
42	`xd`	Xylogics 7053 SMD Disk Driver
	. . .	
	. . .	
	. . .	

3.3. Run-Time Data Structures

If you skip ahead and read the chapter on *Configuring the Kernel* you will 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 will also note, in that chapter, references to the kernel's autoconfiguration process (sometimes called `autoconfig`). 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.

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 BSD UNIX systems, 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.[2] 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.

[2] 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.

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 SunOS 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 will 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.[3] 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.

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 an `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

[3] 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".

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

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

mc_intpri
The interrupt priority level of the controller. This is to be given in the config file and should be used, in the driver source, only as an argument to `splx()` — e.g. `splx(pritospl(mc_intpri))`.

mc_intr
On Sun-3, and Sun-4 systems, base address of array of ,`struct`vec one for each specified in the config file. If `mc_intr` is set, then the fields within the `vec` structure become significant: On the Sun386i system, this field contains the irq (interrupt request channel). The Sun386i system does not support vectored interrupts, so the `v_*` fields are not present.

v_func
Pointer to the vector-interrupt function.

v_vec
Vector number associated with the function in `v_func`.

v_vptr
A pointer to 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_space
A bit pattern which identifies the address space within which the controller is installed.

mc_dmachan

On the Sun386i only, a field containing the DMA channel.

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 `xy0`. 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 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_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_intr
On Sun-3, and Sun-4 systems, base address of array of `struct vec`, one for each specified in the config file. Unused for devices on controllers. On the Sun386i system, this field contains the interrupt channel as an integer.

md_dmachan
On the Sun386i only, a field containing the DMA channel.

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*`dinfo()` 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 xx*`driver`*, 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 will therefore give its complete declaration:

```
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 will need to initialize when defining *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 an *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 can find them in `/usr/share/src/sys/sundev`) 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 *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 will 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, and specifies the amount of space that needs to be mapped into system memory by the autoconfiguration code. The value returned by *xx*`probe()`, while identical, is used only to indicate if the device was found.

`mdr_dname`
is the name of the device for which this driver is written.

`mdr_dinfo`
is a pointer to a pointer to the `mb_device` structure in *xx*`dinfo()`. 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 the controller supported by this driver (for example, *sc* supports the controllers `sc0`, `sc1`, etc). This field takes the form of a regular null-terminated C string. Fill it in if you actually have a controller.

`mdr_cinfo`
is a pointer to a pointer to an `mb_ctlr` structure declared in the driver. This pointer is filled in during autoconfiguration (see the 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. This flag is used only by `mbgo()` and `mbdone()` routines (which are not documented in this manual), and it guarantees exclusive use only among drivers which use it to enforce an exclusive-use protocol. Not all drivers do so.

`MDR_BIODMA`
For block devices that do DMA on the Main Bus (such drivers call `mbgo()` and `mbdone()`). This flag tells the kernel that it must lock other DMA devices off the bus.

MDR_DMA
: For character devices which use DMA, either to transfer large blocks of data or simply to transfer small blocks of control information. The drivers for such devices call `mbsetup()`. This flag tells the kernel that it must lock other DMA devices off the bus, and all DMA drivers should set it.

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_DMA`, `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 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.

WARNING ***The*** `user` ***structure is valid for the current process only while execution is in the top half of the driver. It must never be accessed from the bottom half.***

The kernel `user` structure (`/usr/share/sys/sys/user.h`) 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 SunOS, the `user` structure does not define the address of 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* chapter of this manual).

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

`u.u_qsave`
is a `setjmp()` environment buffer that can be used to save the current stack in preparation for a possible `longjmp()` return. `setjmp()` and `longjmp()` are useful in drivers that need to intercept signals, and then to wake sleeping processes. They can also be used for 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 in most cases the *kernel* plugs the value into `u.u_error`, and it is not necessary for the driver to do so. In fact, a driver cannot access `u.u_error` in its interrupt routine, where transfer errors are normally detected, since the current `user` structure is unlikely to belong to the process for which the failed I/O was being performed).

`u.u_procp`
The `u.u_procp` field in the `user` structure is a pointer to the processs (`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. `u.u_procp` is used by drivers which contain `select()` routines. See the *Variation with "Asynchronous I/O" Support* section of the *The "Skeleton" Character Device Driver* chapter of this manual for details.

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 one of the following values:

```
CPU_SUN3_50
CPU_SUN3_110
CPU_SUN3_60
CPU_SUN3_160
CPU_SUN3_260
CPU_SUN3_E
CPU_SUN3X_80
CPU_SUN3X_460
CPU_SUN4_110
CPU_SUN4_260
CPU_SUN4_330
CPU_I386_AT386
```

Note that when compiling for a Sun-3 system, only the Sun-3 names are available; likewise for Sun-3x's, Sun-4s and Sun386i's.

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.

DVMA drivers will often need to know the address of kernel DVMA space on the host machine (See the *Sun Main-Bus DVMA* section in the *Hardware Context* chapter) 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.

The external variable `KERNELBASE_DEBUG` gives the start of kernel address space in the current memory context.

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 SunOS. The device driver is called each time the user program issues an `open()`, `read()`, `write()`, `mmap()`, `select()` 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 from and Writing to 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 every way subsets of block-driver *xx*`strategy()` routines — they are integrated with `physio()`, and 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.

Select Routine

xx`select` supports the `select()` system call, by which user processes can poll various devices (by way of I/O descriptors which specify them) to see if they are ready for reading, writing, or have an exceptional condition pending on them.

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 I/O architecture. They are not, however, panaceas, and you should not overuse them to solve problems in driver design. Terminals have many `ioctl` calls, but they're a special case. They have many `ioctl` calls because they're inherently quite complex and yet SunOS still insists that they look like files.

4.2. User Space versus Kernel Space

SunOS, being a multi-tasking operating system, 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 SunOS kernel is a *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 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 and Sun-4, the kernel virtual address space is at the top of the current context, starting at `KERNELBASE`.
- In the Sun-4, the kernel uses the top 16 megabytes of the current Gigabyte context, starting at `0xFF000000`.
- In the Sun386i, the kernel uses the top 64 Megabytes. Of these, the kernel has 32 Mbytes reserved for its use; `kadb` has 16 Mbytes reserved, and the EPROM uses 16 Mbytes.

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 devices are usually not interrupt driven. Their drivers, thus, do not typically need to include 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 an *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 upon 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 the 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 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 after the `if` condition is tested, but before the process switch. (The consequences, if this happens, are grave — the call to `wakeup()` will occur before the process has actually gone to sleep, and thus it will never wake up).

The above section of code must thus 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 its board. If so, you must decide what processor-interrupt level the 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 SunOS 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 older Sun 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 an *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];


#ifndef sun386
/*
 * Vector number given in kernel config file and passed by the autoconfiguration
 * process during boot. This code does not apply to the Sun386i, which does not
 * support vectored interrupts.
 */
    if (md->md_intr) {

        /* so we will 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 */
#endif
}

/*
 * 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 use to their advantage. These routines, as well as many others, are described more completely in the *Kernel Support Routines* appendix to this manual. The most important of these routines can be clustered into the functional groups given here:

Timeout Mechanisms

If a device needs to know about clock 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)*. *interval* is often expressed as a multiple of the external variable `hz`, since `hz` gives the number of ticks per second on the host machine. (`10*hz`, then, specifies a timeout of ten seconds). 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 event;
    int software_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 event;
```

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 condition. A distinction is made between processes sleeping at priority less than or equal to the macro `PZERO` and those sleeping at numerically greater priorities.

If a process is blocked in `sleep()` at a priority less than or equal to `PZERO`, it will not be awakened upon receipt of a signal; the signal will not be processed until the process is awakened elsewhere and returns to user mode. (This means that a user cannot interrupt such a process by typing their interrupt character). Thus, it is a bad idea to sleep with priority less than or equal to `PZERO` on an event that may not occur.

On the other hand, if a process is blocked in `sleep()` at a priority greater than `PZERO`, and if a signal is sent to the process, it will be awakened. However, the call to `sleep()` will not return. This means that the routine that called `sleep()` cannot clean up after receiving the signal. If the routine needs to do such clean up, it can arrange for this by ORing the `PCATCH` flag into the priority it passes to `sleep()`. If this is done, and `sleep()` is interrupted by a signal, it will return 1; if the process is awakened normally, `sleep()` will return 0.

In general, sleeping at priorities less than or equal to `PZERO` should only be used to wait for events that occur quickly, such as disk and tape I/O completion. Waiting for events that may not occur quickly—for example, the typing of a particular key by a human at a keyboard—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.

The `splr()` function raises the priority without lowering the current priority level.

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. (These structures are defined in

<sundev/mbvar.h>).

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

On the Sun-3, Sun-3x, and Sun-4, 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 transform the abstract integer returned by `mbsetup()` into a 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.

On the Sun386i, the `mbsetup()` and `dma_setup()` routines are called when the device driver wants to start up a DMA transfer. After the transfer is complete, the driver calls `mbrelse()` and `dma_done()`.

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* in the *The "Skeleton" Character Device Driver* chapter 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

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 appropriate *PROM Commands* chapter of the *PROM User's Manual* 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 SunOS, 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 SunOS specific, and don't become active until SunOS 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 will use the monitor primarily as a means of examining and setting device registers. 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`.

Later, using the autoconfiguration process, SunOS makes a pass through the config file (actually, through the `ioconf.c` file that was produced as output by `config` when it processed the config file). For each device, SunOS 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.

SunOS 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 SunOS 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. Note that, in Sun-4 machines, words and long words are both 32 bits in length.

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 SunOS 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 SunOS. (You can halt SunOS 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 SunOS 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 — sets the to MMU to its pre-SunOS power-up state). But while using the SunOS 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, 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 is X bytes from the beginning of whatever physical page it gets mapped into.

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

Figure 5-1 *Sun-3 Address Mapping*

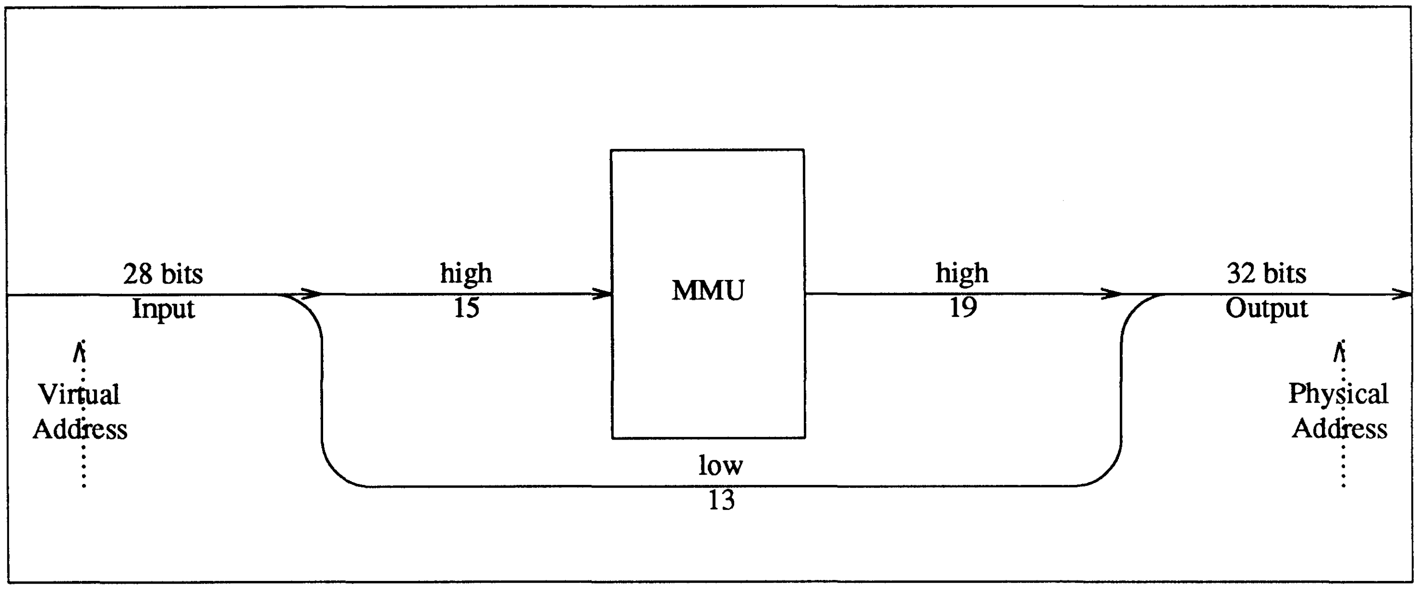


Figure 5-2 *Sun-3x/Sun-4 Address Mapping*

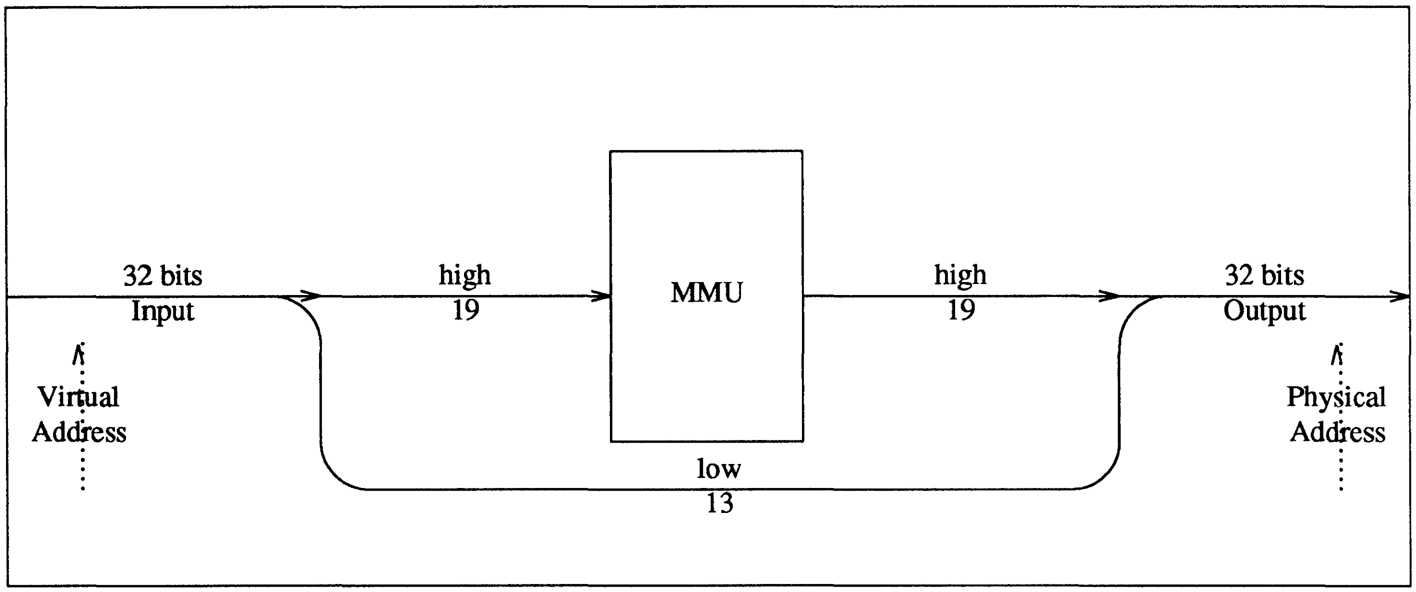
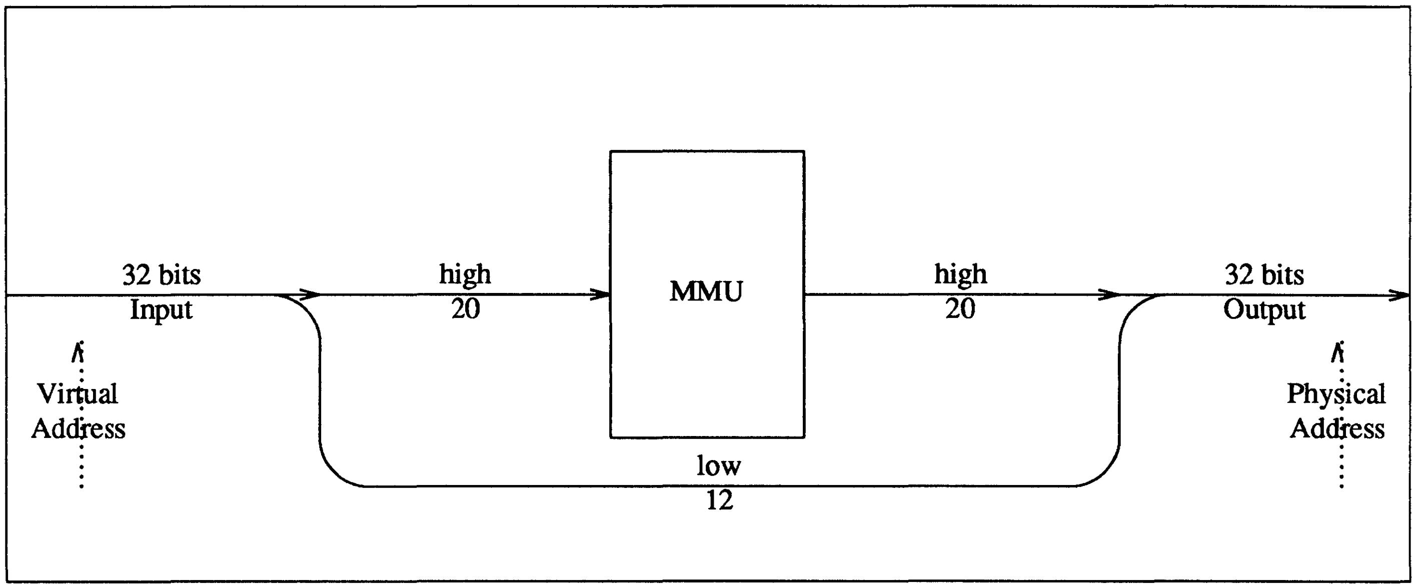


Figure 5-3 *Sun386i Address Mapping*



The easiest way to select a virtual address for PROM-monitor testing is to use one between `0x4000` and `0x100000` on Sun-3, Sun-3x and Sun-4 systems, or `0x20000` and `0x100000` on Sun386i systems. Addresses in these ranges are unused by the monitor in the respective Sun models, and are thus available. (Note that these addresses, while convenient for testing, are not those that the kernel will choose when your device is finally installed).

It's most convenient to select a virtual address which has only zero's in its low-order bits. This way you select the first address in a virtual page. The low-order bits in the address you choose remains unchanged. With `'X'` representing the unmapped low-order bits (13 for a Sun-3, Sun-3x or Sun-4, 12 for a Sun386i) the test address `0x4000` is, in binary:

```
  Sun-3 :       0000 0000 0000 100X XXXX XXXX XXXX  (28 bits)
  Sun-3x:  0000 0000 0000 0000 100X XXXX XXXX XXXX  (32 bits)
  Sun-4 :  0000 0000 0000 0000 100X XXXX XXXX XXXX  (32 bits)
Sun386i :  0000 0000 0000 0000 0100 XXXX XXXX XXXX  (32 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 conflicts, you 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 in the *Hardware Context* chapter show memory layouts corresponding to typical configurations, but if your system has departed at all from the norm, you 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. The Sun-3, and Sun-4 all use the same proprietary MMU architecture. The Sun-3x uses the MMU that is on the same chip as the CPU. This MMU works differently than the Sun MMU.

The following description is about the Sun MMU operation as it pertains to the Sun-3 and Sun-4. There is also an example of how to perform a mappings using sample numbers. The Sun-3x description follows the Sun-3/Sun-4 description and includes a page mapping example.

Sun-3/Sun-4 Virtual to Physical Mapping

Up to this point we've only stressed that the MMU maps the top bits of the virtual address, leaving the offset bits unchanged. Following is the explanation of the mapping process in more detail.

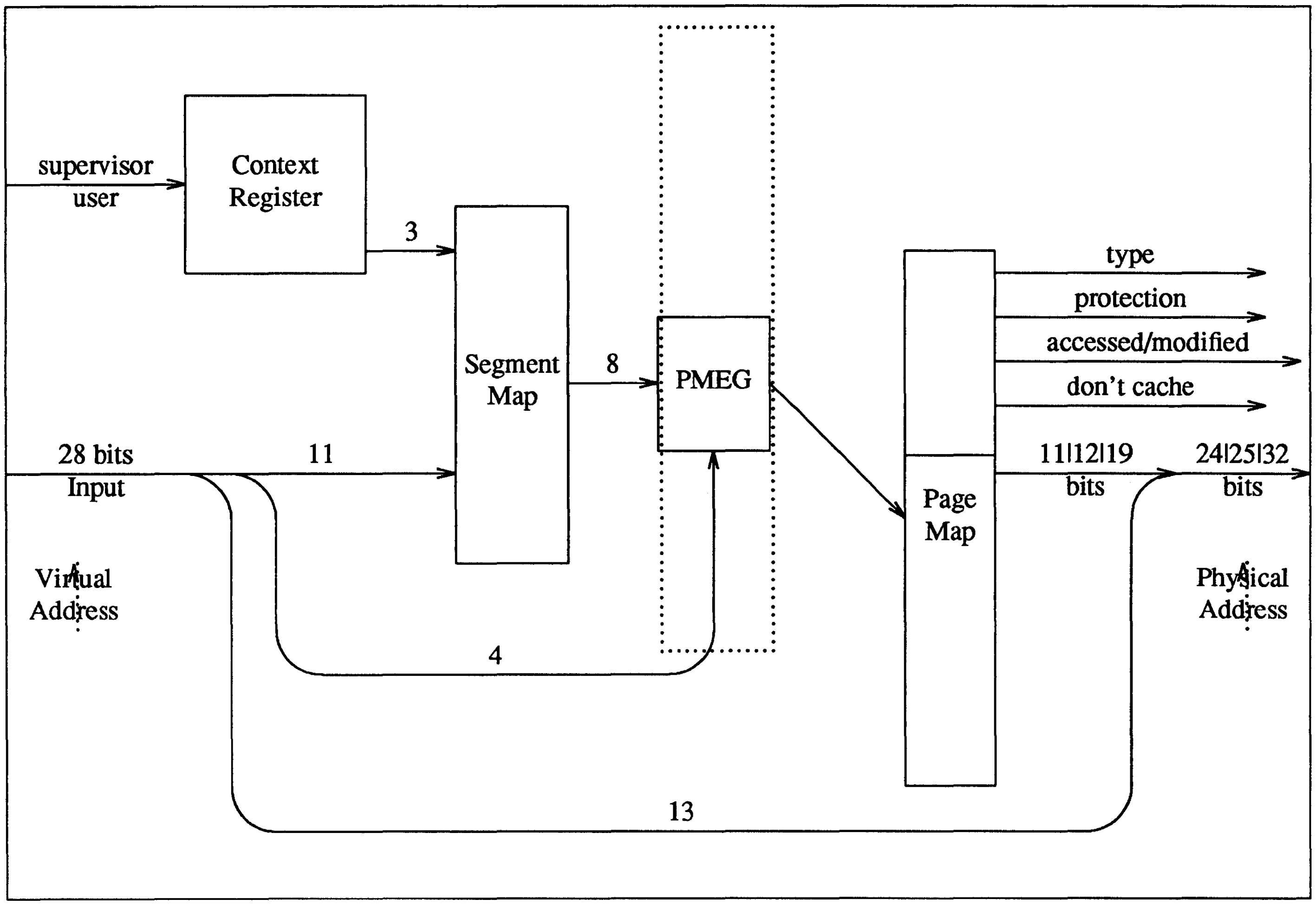
Some new concepts are necessary to discuss the details of virtual to physical memory mapping.

- The *context register is a register specifying which of memory contexts should be used when mapping virtual addresses to physical addresses. Sun-3 Context Registers contain 3 bits, and specify one of eight memory contexts; Sun-4/260 Context Registers contain four bits, and specify one of 16 memory contexts. Each SunOS process segment (containing either code, data or stack) is kept within a single memory context.*
  - Sun-3s and Sun-4s 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-4 virtual address spaces are divided into two chunks. One of them is at the top of the addressable virtual memory space and the other is at the bottom. The size of the unused space between these two spaces varies with the model — in the Sun-4/260 each of the two virtual address spaces is 512 megabytes in size, and the space between them consumes 3 Gigabytes.
- 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 final output of the MMU. 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.

Sun-3 and Sun-4 Address Mapping

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

Figure 5-4 *Sun-3 MMU*

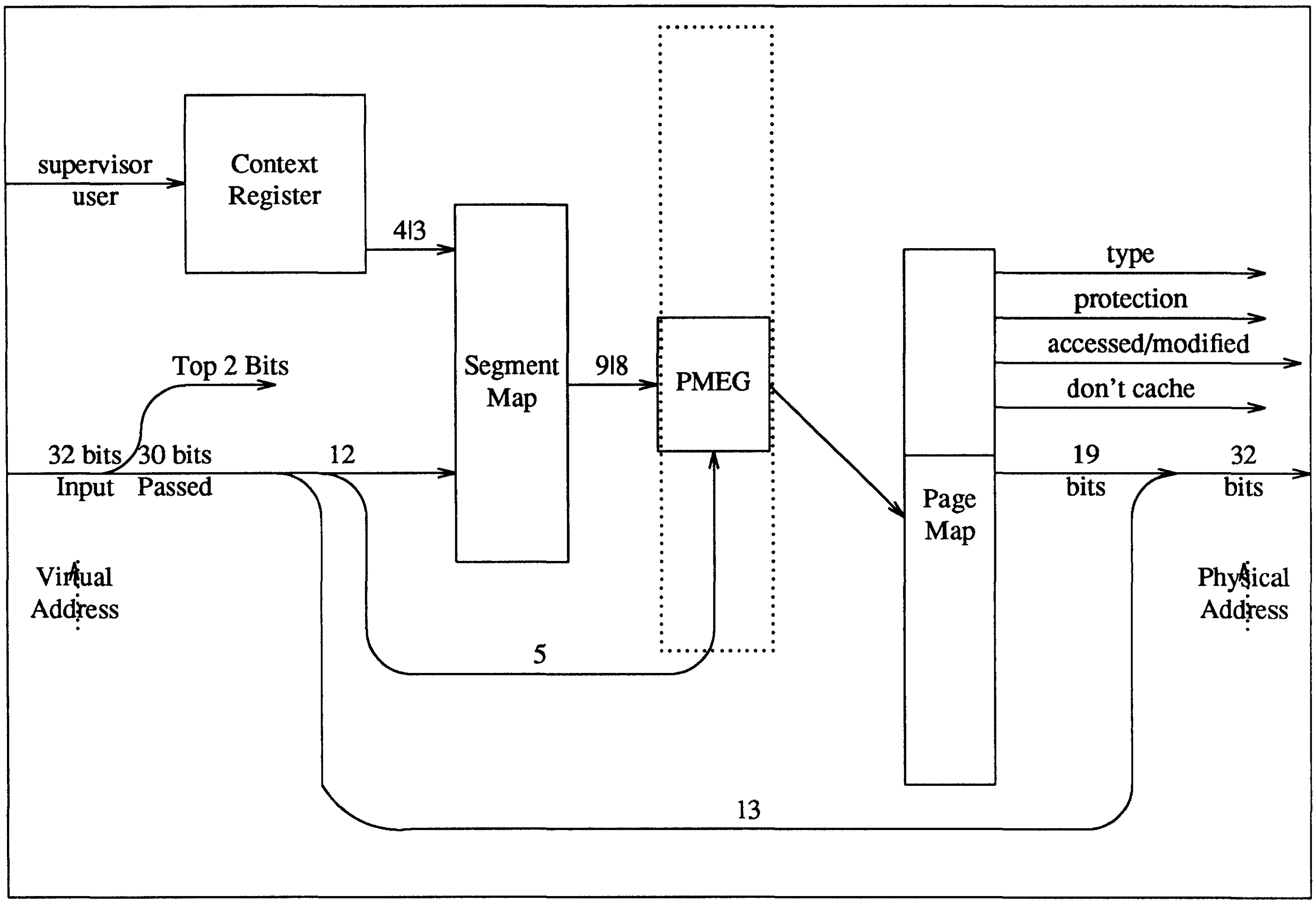


Note that:

- The MMU is getting a 28-bit virtual address as its input.
- 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.

The Sun-4 MMU is almost the same:

Figure 5-5 *Sun-4 MMU*



As you can see, the Sun-4 MMU is largely identical to the Sun-3 MMU. The differences are that:

- The Sun-4 MMU gets a 32-bit virtual address as its input, as opposed to a 28-bit address in the Sun-3. The top two bits are immediately shunted off. They must be either `00` or `11`, and are used to specify one of the two "chunks" in the virtual address space. (See *Selecting a Virtual to Physical Mapping* above).
- The number of bits coming off the Context Register is 4 (to specify one of 16 contexts) on Sun-4/260s and 3 (to specify one of 8 contexts) on Sun-4/110s.
- The number of bits coming off the Segment map is 9 for Sun-4/260s and 8 for Sun-4/110s.

On both Sun-3 and Sun-4 systems, PTEs are 32-bit numbers with the following structure.

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

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 (don't cache) = 1
(a/m) accessed/modified = 00
unused = 00000
```

(A one (1) in the don't cache position only disables caching if the type is zero (0), since other types of pages are never cached). With the above values, our template then looks like this:

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 5-1 *Sun-3/Sun-4 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 will 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
```

Sun-3x Virtual to Physical Mapping

In the previous CPU board designs, such as the Sun-3 architecture, a discrete MMU was designed and implemented to handle Demand Paging (off chip). That MMU was implemented mostly in hardware, with a dedicated register for the Context and separate high speed RAM for the Segment and Page values. In the Sun-3x architecture where the MC68030 is used as the CPU, a fully programmable Memory Management Unit (MMU) integrated into the silicon (on the 68030 chip) is used to handle demand paging. A similar MMU has been offered by Motorola for some time (the MC68851 MMU) but was not used by Sun due to certain architectural incompatibilities.

This Memory Management Unit is drastically different in operation from the popular discrete version of its processors. Some of the MMU's most significant changes involve how the Translation Tables are initialized, accessed, and updated and also the way the Address Translation procedure, or Table Walk, is completed. This next discussion presents the process of how the firmware builds, initializes, and updates the entries in the MMU Translation Tables, how the Table Walk is accomplished, and how the MMU performs Address Translation. An example is shown how to use the monitor to map virtual addresses into physical addresses to access devices through the PROM.

The MMU handles the translation of addresses from virtual to physical using translation tables stored at arbitrary locations in memory. The MMU has an

Address Translation Cache (ATC) that holds recently used virtual to physical address translations. When the CPU passes a virtual address to the MMU for translation, it first searches the ATC for the corresponding physical address. If the requested entry is not in the ATC, the processor searches the translation tables in main memory for the information. An ATC access operates in parallel with the other on-chip caches, namely the CPU's Instruction Cache and Data Cache. In order for the MMU to operate correctly, its internal registers must be initialized to a known state.

The MMU has several internal registers that are initialized to known values before the MMU is Enabled (Address Translation Enabled) and during various Reset (k2 or power-on) operations. These registers include the CPU Root Pointer (CRP), the Supervisor Root Pointer (SRP), and the Translation Control (TC) register, all of which are initialized while the MMU is Disabled (Translation Disabled). The CRP and SRP are discussed in the Motorola 68030 Manual, but for now it is important to say that these registers contain the starting addresses for the MMU's table walk.

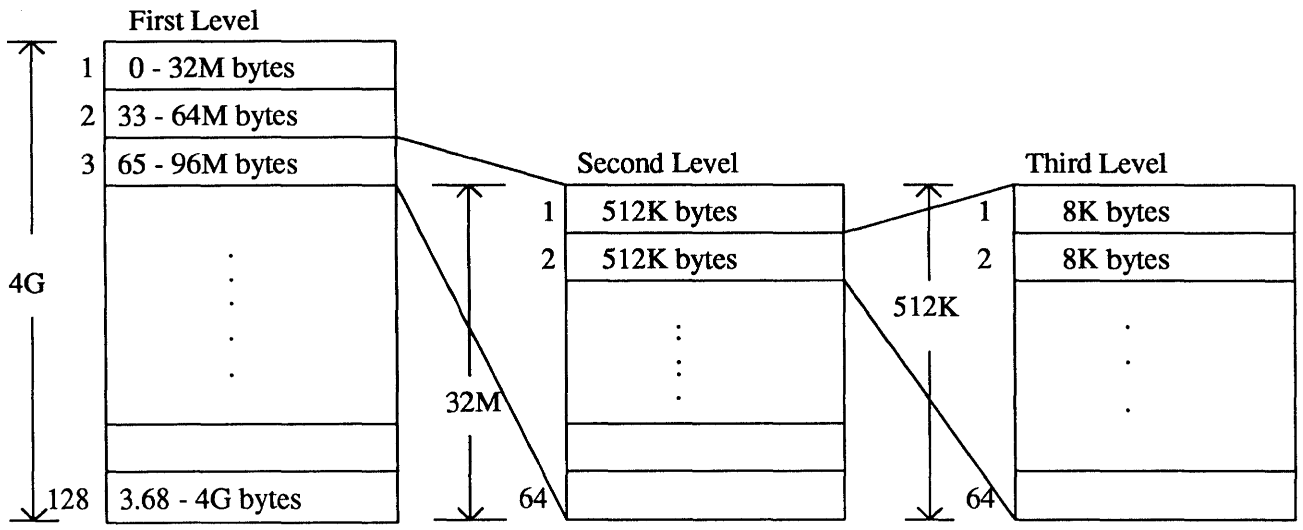
The Table Walk

The MMU's principal function is address translation, which involves converting a virtual or logical address to a physical address. This process is known as a Table Walk. For the Sun-3x architecture a three level MMU has been designed and requires that a three level table walk be initiated to perform address translation. This process terminates when either an INVALID Entry or PAGE Descriptor is encountered. The three levels of address translation are referred to as TIA, TIB, and PAGE respectively.

The three level table walk is needed to evenly divide the four gigabyte addressing range of the MC68030. This could have been accomplished several different ways, but a specified design goal was to have the Firmware, the Executive Diagnostic and the Unix Operating System all use the same Translation Table format.

The first level of lookup, the TIA table entry, must be able to map in the entire four gigabyte addressing range all at once. The largest block of virtual memory that is required at any one time is 32 megabytes. By dividing 4 gigabytes by 32 megabytes we get 128 entries for the first level of address translation. For the second level of translation, the TIB entries take each of the 32 megabyte TIA entries and divide them by 64. This allows each TIA entry to be accessed as 64 separate 512 Kbytes (1/2 megabyte) blocks. Each of the 64 TIB entries are then divided into 64 again which results in 8 Kbyte page sizes.

It is because of this table traverse that the name Table Walk is used. Each virtual address is translated to a physical one by taking parts of the virtual address and using them as indexes into the three tables, the resulting output being a Page Table Entry (PTE) which determines the exact physical address. See the table below for how the entire virtual address range is divided into 8 Kbyte ranges.



The beginning of the table walk starts with a pointer to the location of the MMU tables in main memory. The PMMU has two pointers, one that is used by the CPU (CPU Root Pointer), and one that is used by the CPU while in supervisor state (Supervisor Root Pointer). For the firmware's use, both the CRP and the SRP are initialized to the same value, which means they both point to the base of the MMU tables.

When the MMU is Enabled, the CPU passes virtual addresses to be translated to the MMU. If the requested entry is not in the ATC, a table walk of the translation table is initiated. The table walk sequence is described below.

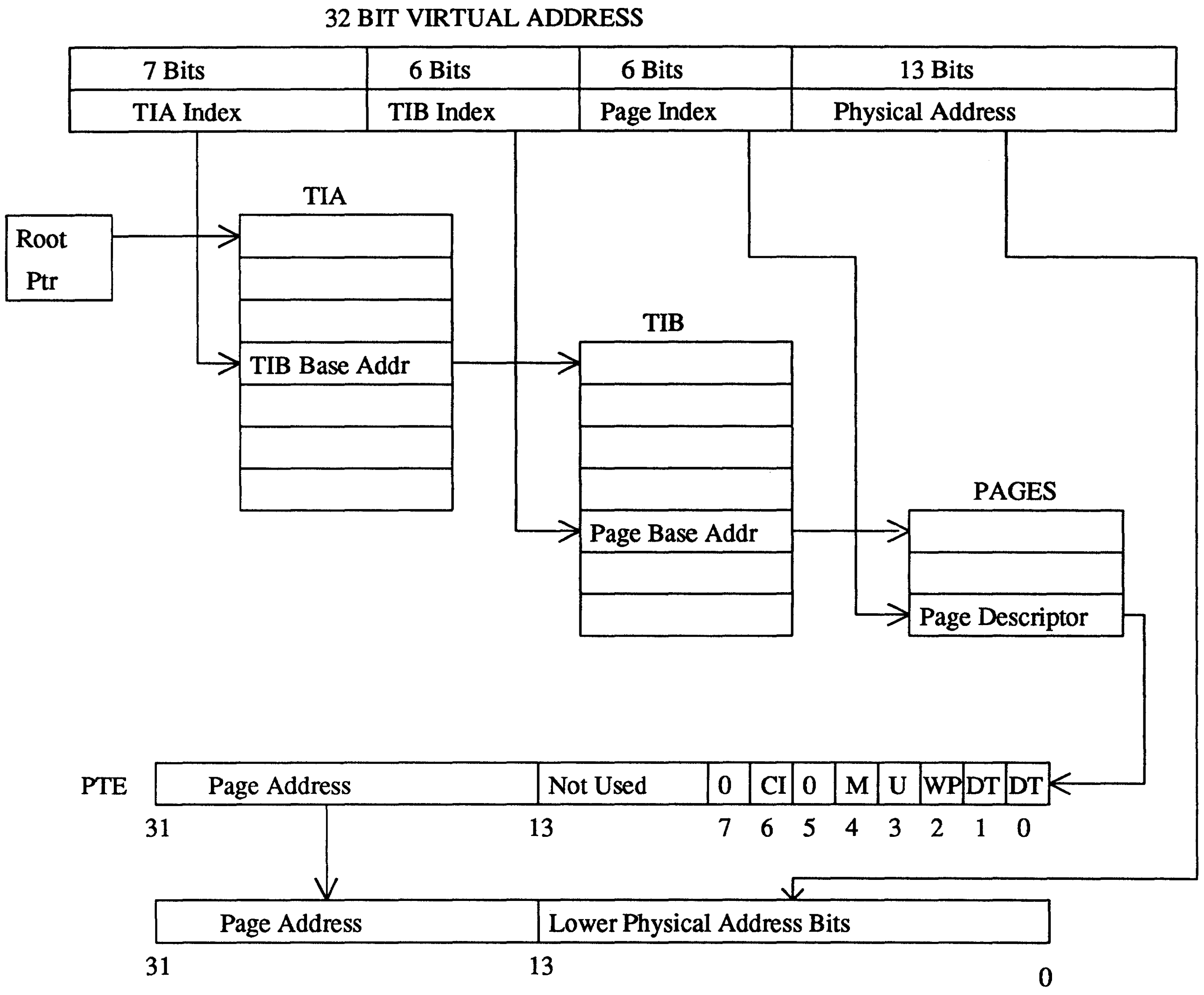
Step One: The CRP contains the base address of the TIA table in memory. The top seven bits of the Virtual Address are used to calculate the index into the TIA table. This index is added to the CRP to generate the specific TIA table entry. The TIA entry contains the base address of the TIB table for the next step.

Step Two: The next six bits of the virtual address are used as an index into the TIB table. When added to the base address from the TIA table the specific TIB table entry is generated. The TIB entry contains the base address of the PAGE Table.

Step Three: The next six bits of the virtual address are used as an index into the PAGE table. The base address from the TIB table plus the index result in the PAGE Table Entry (PTE). The PTE contains a 32 bit PAGE Descriptor of which 19 bits are the Page address, 5 are unused, and the remaining 8 are Status bits.

The Physical address is calculated by taking the top 19 bits from the PTE and the lower 13 bits from the Virtual address. These 13 bits are an offset into the physical memory page that is selected from the 19 bits.

The table walk is completed by passing the physical address back to the CPU. If an INVALID descriptor is ever encountered the table walk terminates.



A Few Example PTE Calculations

Example One: You wish to map a device which you have attached at physical `0x280008` onto bus type `vme24d16` which will be mapped into virtual memory at address `0xE000000`. What is the corresponding PTE?

Sun-3 Solution

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 1000 0000 0111 1111 1001 0100 0000
```

Which is 0xF807F940.

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.

Sun-3x Solution

Our variables are:

```
physical address    280008
virtual address     E000000
bus type            vme24d16
```

The base address for vme24d16 for the Sun-3x, which is in Table 2-8 in Chapter 2, is 0x7e000000 So we add the physical memory address to the vme base pointer which gives us a specific physical address.

```
vme24d16    7E000000
physical      280008
            --------
physical    7E280008
```

Then we take off the top 19 bits to mask out just the vme page, which gives us the physical page of memory. We then need to logically 'or' in some status bits to allow us to write to this page. The value 1 enables the write status.

```
physical    7E280008
and mask    7E280000
            --------
page        7E280000
or flag            1
            --------
PTE         7E280001
```

To use the monitor to perform the mapping, use the 'p' command for displaying and changing the Page Table. The syntax is

```
p[virtual address]
```

where the virtual address is the original virtual memory given in the problem initially. The monitor returns the current PTE and asks you for a new value. The newly calculated PTE is input, which modifies the PTE to map to a new physical memory location

```
monitor cmd      >pE000000<cr>
return value     xxxxxxxx
new PTE          ?7E280001<cr>
exit monitor     .
```

Now every reference to the virtual memory location `E000008` will be mapped to the device. Note that since the original physical address was folded into the virtual address and then was masked, we still have the 8 offset at the end of the memory reference to index into the physical page of memory to access the device.

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

Sun-3 Solution

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`.

The Sun-4/110 MMU does not store bits 28-31. For the VME, which is the only addressing that use 32 bits of physical addressing on the Sun-4/110, bits 28-31 are generated by sign extending bit 27. When the PTE is read back, these upper bits are always set to zero. This essentially creates a hole in the address space that is not addressable.

When entering page table entries on a Sun-4/110 to test hardware from the prom monitor, use a virtual address less than `0x800000`. Virtual addresses from `0x800000` and above are not setup by the prom monitor for use and will result in an invalid PMEG.

If you are mapping the device to `vme16`, `vme24` or the top half of the `vme32` address space, after entering the PTE the top five bits of the physical page number are zero because the Sun-4/110 physical address space is split with 128 megabytes at the bottom and 128 megabytes at the top. Whenever the physical address goes above 128 megabytes, the high bit is sign extended so that the address lies within the top 128 megabytes. If you sign extend the high bit into the next five bits you should come up with your previously calculated physical page number.

In this example, instead of using `0xE000000` as the starting address, the value `0xE0000` has been used successfully.

Sun-3x Solution

Using the same steps above, this is how the solution looks:

```
physical     EE48
virtual      E000000
bus type     vme16d32

vme16d32     7D000000
physical         EE48
             --------
physical     7D00EE48


physical     7D00EE48
and mask     7D00E000     0111 1101 0000 0000 111
             --------
masked page  7D00E000
or flag             1
             --------
PTE          7D00E001
```

This is the new PTE value that can be used in the monitor as shown in the previous example.

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 will 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:

- Address and data widths,
- Interrupt levels,
- Interrupt vector numbers for VMEbus device,
- VMEbus address modifiers,
- The bus grant level for VMEbus devices should be set at 3.

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 SunOS down again and attempt to contact the device using the PROM monitor. To do so, you will need to set up a PTE on the Sun-3 or Sun-4 which maps to the device's physical installation address. Use the procedures given above to calculate a PTE, then:

- Issue the monitor command that puts you into *supervisor data* state. This will be **s B** for Sun-4 machines and **s5** for all others. So, if you have a Sun-3, give the

  ```
  >s5
  ```

  command.

- 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 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 appropriate *PROM Commands* chapter of the *PROM User's Manual* 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 use 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 will 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. Builtin I/O Cache

Using the I/O Cache in the Driver

To use the I/O cache for devices that process buffers, such as disk and tape drivers, the driver needs to mark the buffer with the `B_IOCACHE` flag in the strategy routine and turn off this flag in the interrupt routine when the I/O completes. The buffer must be properly aligned, which is on a 16 byte boundary.

To use the cache with ethernet-like devices, set the `IOCACHE` bit in the page tables and flush the I/O cache after I/O completes. Cached I/O is only valid for 16 byte aligned transfers of a multiple of 16 bytes. On future machines, there may be 32 byte-aligned I/O caches.

Address Mapping

The device driver doesn't have to tell the I/O cache what physical address range matches with a particular DVMA address range. The kernel routines used to allocate and map in DVMA space already handle the physical to virtual mappings. The I/O cache is not concerned about these mappings because it does not see the mechanics of it. The `mb` routines set up the I/O mapper entries that translate DVMA addresses to physical addresses.

5.3. 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. See the *Pixrect Reference Manual* for more details.

Memory-mapped devices 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 will see below). Such memory-mapped drivers don't really do much. Obviously, *xx*`probe()` and *xx*`mmap()` must exist, 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 some very simple driver *xx*`mmap()` code.

```
/*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, numdevs;
    struct mb_device **mb_devs;
    int size;
{
    int kpfnum;

    if ((u_int) off >= size)
        return -1;

    kpfnum =
      hat_getkpfnum(mb_devs[minor(dev)]->md_addr + off);
    return kpfnum;
}
```

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 a bit of shuffling around and then a call to `hat_getkpfnum`, which returns a Page Frame Number which *xx*`mmap()` is expected to return.

Note that `mb_dev->md_addr` is 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. The offset is checked to be sure the user isn't mapping beyond the end of the frame buffer.

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 its control registers) must work with user function codes, since that's what it will get when mapped into and then accessed from user space. To be able to access a board from a user mmap program, the address modifier on the board must be set to non-privileged data access, or user data space. This is so that the board will respond to user function codes in the user data space, such as address modifiers `0x09`, `0x29`, and `0x39` for `vme32, 16, and 24 respectively.`

NOTE *MC680X0 processors, SPARC processors and the Intel 80386 all run in either 'user' or 'supervisor' state. Many devices, in turn, restrict certain of their operations, and will only perform them when the processor is in supervisor state. The Sun CPU is in supervisor state only when executing kernel code. This means that device drivers, which are part of the kernel, can issue device commands which are not available from user processes. Also note that, when the CPU is in supervisor state, as it is when driver code is executing, the device will receive different VMEbus address modifier codes than when the CPU is in user state. For details about these codes see the VMEbus specification.*

- 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. The former 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 second 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 5-2 *Virtual Memory Devices*

Machine Type	Memory Device Name
VMEbus (All Sun's)	`vme16d16`
VMEbus (All Sun's)	`vme24d16`
VMEbus (Sun-3 and Sun-4)	`vme32d16`
VMEbus (Sun-3/Sun-3x/Sun-4)	`vme16d32`
VMEbus (Sun-3/Sun-3x/Sun-4)	`vme24d32`
VMEbus (Sun-3/Sun-3x/Sun-4)	`vme32d32`
ATbus (Sun386i only)	`atmem`

In addition, there are memory pseudo-devices that support access to the on-board devices that users are allowed to access. These are `/dev/fb`, `/dev/mem` and `/dev/kmem` (See the `mem(4)` manual page for details).

/dev/fb is a memory device which, on any given system, is set up to address the local frame-buffer device. It can be used as if it were a system memory device — on any given system, /dev/fb can be mmap()'ed into user memory and then written to, 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 device.
- Calculate the *offset* which you will need to call mmap(). This offset is merely the device address on the appropriate system memory device rounded to a page boundary. That is to say that you get the offset from the device manual and/or the switches on the device itself.
- Call mmap() to allocate virtual space and map in 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 example program uses /dev/fb rather than one of the virtual memory devices. It 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 mmap() 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. (The machine segment size is 128K for the Sun-3; 256K for the Sun-4; and 4Mbytes for the Sun386i. Areas greater than the machine segment size should be mapped only with special care. The Sun-3x has no segment size so any input value will work. For details, see the discussion of mmap() in the *User Support 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 memory is mapped by a driver, subsequent accesses to it 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, i.e., they're inaccessible from a user program, and thus a kernel level driver must be written to manipulate them. The device will also receive different address modifier codes when accessed from a user process than when accessed via a device driver).

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

/* Width and Height of Frame Buffer in Bits */
#define WIDTH 1152
```

```
#define HEIGHT 900

main()
{
    int fd;
    off_t offset;
    unsigned len;
    char *addr;

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

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

    /*
     * offset  must be page aligned.   /dev/fb
     * is already aligned with frame-buffer memory
     */
    offset = (off_t)0;

    /* Map device memory to user space */
    addr = mmap((caddr_t)0, len, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, offset);
    if (addr == (caddr_t)-1)
        syserr("mmap failed");

    writeFB(addr);
    exit(0);
}

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

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

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)\n", sys_errlist[errno]);
    else
        fprintf(stderr,")\n");
    exit(1);
}
```

NOTE *This example uses the special memory device* `/dev/fb`, *since this device is always set up to address the frame buffer memory.*

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, and even more limited. 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()`, *and is notable mainly because, with it, the device receives supervisor function codes. It does, however, introduce 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. Thus they are 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 systems will always try to do the "right thing" (if you request I/O at an odd address or for an odd number of bytes it will perform byte access as appropriate; otherwise it will use short integers), but it still 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 direct references to `mmap()`'ed references, which proceed by way of the MMU and use low-level store and move instructions to directly access device registers and memory as physical memory.

So the bottom line is that, unless you need to access a device only a few times, or if you need to receive supervisor function codes (and the corresponding VMEbus address-modifier codes) and 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 with low-level I/O 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 is similar to the example above, in that it writes the same pattern to the memory of a frame buffer. This time, though, the write is done by way of the I/O system rather than by using mmap(), and the frame buffer is taken to be installed at *OFFSET* (whatever the device physical installation address is) in the `vme24d16` memory space.

NOTE *Since all Sun VMEbus machines have a built-in, on-board frame buffer, this example is only meaningful for color frame buffers.*

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

void syserr();
long lseek();

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

main()
{
    int fd;

    /* Open the system memory device containing the frame buffer */
    if ((fd = open("/dev/vme24",O_RDWR)) < 0)
         syserr("open");

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

    writeFB(fd);
    exit(0);
}

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");
        }
    }
}
```

5.4. 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 trip hardware traps and crash the system.
- 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. There is a kernel debugger, `kadb`, and this is 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 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 run in user space.

For all 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 by 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.

NOTE *On all Sun systems, the loadable drivers feature makes driver development much easier because the code-compile-reboot-test cycle is reduced to code-compile-load-test.*

Debugging with `printf()`

With the availability of `kadb`, the kernel debugger, the importance of `printf()` in the debugging of device drivers has been significantly reduced. Still, even with `kadb` 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 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 will 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).

The window system should not be up when you use `printf()` to debug a driver because its output will go to the console window. On the Sun386i system, it is best to set the global variable `newlog` to 0.

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. Likewise adding* `printf()` *statements may make your driver begin to work properly when bugs are actually still existent, due to alterations of the system timing. 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 finished — 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, reset or change the print-control variable `xxdebug`. Or you can use `adb` if you wish, running it at user level in a separate window:

```
example adb -w /vmunix /dev/mem
```

(`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 so BE CAREFUL.*

Incidentally, `/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 can also be useful to 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 of output 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 `/var/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+10/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, it's 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`. `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/mem. In other ways, however, kadb is much different. To give just one example: on Sun-3 and Sun-3x 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. (This problem does not exist on the Sun386i. On the Sun386i, when kadb is told to read a byte, it does. Within kadb, the B command is used to read a single byte and the v command to write one).
- Sometimes a kernel which will boot find by itself will not boot under kadb because it is too large to be loaded along with kadb.

5.5. 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 will 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.*

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 summarized:

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, and most important, is the error-reporting mechanism based upon the buffer-header. *This is the only mechanism that can be used when returning errors from* *xx*`strategy()`, *xx*`start()`, and *xx*`intr()`. (See the discussion of *xx*`intr()` error reporting in the *Summary of Device Driver Routines* chapter. Finally, it is possible to directly set the global error register, `u.u_error`, from routines in the top half of the driver.

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 an 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. 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 `/var/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 using the ASSERT macro which calls an `assert()` routine which prints error context information and then calls `panic()`.

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;
```

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

5.6. System Upgrades

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

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 *SunOS 4.1 Release Notes* for this release.

5.7. Loadable Drivers

All Sun machines support loadable drivers in SunOS 4.1. This feature allows you to add a device driver to a running system without rebooting the system or rebuilding the kernel. The loadable drivers feature reduces time spent on driver development, and makes it easier for users to install drivers from other vendors.

This section explains how to convert a non-loadable driver to be a loadable driver.

Conversion of a non-loadable driver to a loadable driver requires an initialization or "wrapper" module to be written. The module `zzinit.c` below is an example of a wrapper module that contains the same kind of information ordinarily provided by a config file and by the linker. Almost all wrappers are identical to the example below. Usually, only the actual structure initialization values are different.

The following module is a typical example of an initialization routine for a driver named `zz` that has one controller and one device on that controller.

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/buf.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sundev/mbvar.h>
#include <sun/autoconf.h>
#include <sun/vddrv.h>

extern zzopen(), nulldev(), zzstrategy(), zzdump();
extern zzsize(), zzread(), zzwrite(), zzioctl();
extern zzint(), nodev(), seltrue();

extern struct mb_driver zzcdriver;    /* defined in driver */

/*
 * Driver block device entry points (normally in <sun/conf.c>)
 */
struct bdevsw zzbdev = {
    zzopen, nulldev, zzstrategy, zzdump, zzsize, 0
};

/*
 * Driver character device entry points (normally in <sun/conf.c>)
 */
struct cdevsw zzcdev = {
    zzopen, nulldev, zzread, zzwrite, zzioctl, nodev,
    nulldev, seltrue, 0
};

/*
 * Controller structure (normally in ioconf.c) (see <sundev/mbvar.h>)
 */
struct mb_ctlr zzcctlr[] = {
    &zzcdriver, 0, 0, (caddr_t) 0x00001000, 2, 6,
    SP_ATMEM, 0
};

/*
 * Device structure (normally in ioconf.c) (see <sundev/mbvar.h>)
 */
struct mb_device zzcdevice[] = {
    &zzcdriver, 0, 0, 0, (caddr_t) 0x00000000, 0, 0, 0x0,
    0, 0x0
};

/*
 * The following structure is defined in <sun/vddrv.h>
 *
 * If the number of controllers is 0, then the address of the
 * controller structure array must be NULL. Similarly, if the number
 * of devices is 0, then the address of the device structure array
 * must be NULL. The bdevsw or cdevsw entries can be NULL if there
 * is no block or character device for the driver.
 */
struct vdldrv vd = {
    VDMAGIC_DRV,        /* Type of module. This one is a driver. */
    "zzdrv",            /* Name of the module. */
    zzcctlr,            /* Address of the mb_ctlr structure array */
    &zzcdriver,         /* Address of the mb_driver structure */
    zzcdevice,          /* Address of the mb_device structure array */
    1,                  /* Number of controllers */
    1,                  /* Number of devices */
    &zzbdev,            /* Address of the bdevsw entry */
    &zzcdev,            /* Address of the cdevsw entry */
    0,                  /* Block device number. 0 means let system choose. */
    0,                  /* Char. device number. 0 means let system choose.*/
};

/*
 * This is the driver entry point routine. The name of the default entry point
 * is xxxinit. It can be changed by using the "-entry" command to modload.
 *
 * inputs: function code - VDLOAD, VDUNLOAD, or VDSTAT.
 *      pointer to kernel vddrv structure for this module.
 *      pointer to appropriate vdioctl structure for this function.
 *      pointer to vdstat structure (for VDSTAT only)
 *
 * return: 0 for success. VDLOAD function must set vdp->vdd_vdtab.
 *      non-zero error code (from errno.h) if error.
 *
 */

xxxinit(function_code, vdp, vdi, vds)
    unsigned int function_code;
    struct vddrv *vdp;
    addr_t vdi;
    struct vdstat *vds;
```

```
{
    switch (function_code) {
        case VDLOAD:
            vdp->vdd_vdtab = (struct vdlinkage *)&vd;
            return (0);
        case VDUNLOAD:
            return (unload(vdp, vdi));
        case VDSTAT:
            return (0);
        default:
            return (EIO);
    }
}

static unload(vdp, vdi)
        struct vddrv *vdp;
        struct vdioctl_unload *vdi;
{
    extern struct buf zztab;

    struct buf *dp;

    dp = &zztab;
    if (dp->b_actf) {
        return(-1); /* The driver still has an active request. */
    }

    /* The driver can do any device shutdown stuff that it needs to do */

    return(0);
}
```

Your driver routines can be placed in the wrapper module if you like. If your driver is big, it is more appropriate to break it into several modules.

If you decide to place your driver in the wrapper module, then the driver can be compiled with the following command line:

```
example# cc -c -O -DKERNEL -D[arch] [options] zzinit.c
```

where `[arch]` is the specific architecture that you are compiling for. Values that will normally be here are `-Dsun3`, `-Dsun3x`, `-Dsun4`, `-Dsun4c`, and `-Dsun386`. The `[options]` field includes other options that normally occur in the kernel makefile.

If the driver consists of more than one module, then you must use the link editor, `ld`(1), with the `-r` option to preserve relocation information. For example you might type:

```
example# cc -c -O -Dsun386 -DKERNEL zzinit.c

example# cc -c -O -Dsun386 -DKERNEL zz1.c

example# cc -c -O -Dsun386 -DKERNEL zz2.c

example# ld -r -o zz.o zzinit.o zz1.o zz2.o
```

Thus the object module can be created either by the `cc(1)` command, when the driver resides in one module, or by the `ld(1)` command, when the driver resides in several modules.

In either case the resulting loadable module is an ordinary relocatable object file (`zzinit.o` or `zz.o`). It can be installed in the kernel using the `modload(8)` command. Only the Sun386i stores the object file as a `COFF` file.[4]

The kernel-level support for loadable modules is contained in the driver for the `/dev/vd` pseudo-device. Loading a module involves a four-step process. First, `modload` runs `ld(1)` to determine the size of the linked module. Then, using an `ioctl` call to `/dev/vd`, `modload` reserves a section of memory to hold the module. The memory is dynamically allocated by the vd driver and its starting address is returned. `Modload` runs `ld` a second time to relocate the module and to resolve references to external kernel symbols. Finally, using another `ioctl` call, it copies the module into the kernel and passes control to the module's wrapper function.

[4] "COFF" = Common Object File Format, a UNIX object-file standard to which Sun386i assembler and link-editor output files (normally `a.out`) comply. See `coff(5)`.

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	Inter-rupt				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 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 <sys/param.h>
#include <sys/buf.h>
#include <sys/file.h>
#include <sys/dir.h>
#include <sys/user.h>
#include <sys/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 an *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 it's 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 on 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()`, `peekl()`, `poke()`, `pokec()`, and `pokel()`. 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" on Sun-3's and Sun-3X's, "half-words" on Sun-4's). `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, *xx*`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. If *xx*`probe()` finds the device, the `md_alive` field in the device structure is set to 1, otherwise it's set to 0. `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*`dinfo()` 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 will 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 == 0 || 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 will do so first in memory and then in the actual hardware register. We will 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;
{
    int unit = SKUNIT(dev);

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

See notes on the uio structure below. 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. For example, 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 finite 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 `<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 has 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 will 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->sk_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

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->sk_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 a 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 will 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->sk_count > 0) {   /* still more characters */
        sk_reg->sk_data = *sk->sk_cp++;
        sk->sk_count--;

        /* stop giving characters if device not ready */
        /* Note: the softcopy isn't needed for reads */
        /* contact the device */

        /* DELAY(10) might go here */

        if (!(sk_reg->sk_csr & SK_DEVREADY))
            break;
    }

    /* error-retry logic would go here */

    /* still more characters */
    if (sk->sk_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 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);
```

at the top of the `while` loop. Clearly this is an area where experimentation with the real device is called for.

6.7. `intr()` and `poll()` 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, `mb_driver`) data structure for that device driver. It is installed there during the kernel configuration process based upon information in the config file.

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()`, incidentally, 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;
	}

	/* I/O transfer completed */
	if ((sk->sk_bp->b_flag & B_ERROR) != 0 ||
		sk->sk_count == 0) {

		/* 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 header associated with the failed transfer so that it can 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". The Skeleton device, as we've defined it here, is modeless and has no such special functions so we don't have an *xx*`ioctl()` routine. (Though we will add one below in a variation of the Skeleton driver that supports a form of asynchronous I/O). For details about driver *xx*`ioctl()` routines, and the other driver routines, see the *Summary of Device Driver Routines* appendix.

6.9. Skeleton Driver Variations

The Skeleton I/O board isn't particularly realistic, but is does serve to illustrate the construction of a basic character driver. In this section, we will propose some variations on the basic device, each designed to illustrate a useful technique.

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.

NOTE *DMA is different on the Sun386i system. For information about it, see the* `dma_setup()` *and* `dma_done()` *routines in the Kernel Support Routines appendix.*

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 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`.

On the Sun-4 (or Sun-3x), the DVMA space is defined by the same address range used on the Sun-3, `0x0` to `0xFFFFF` for 24-bit or 32-bit addressing. Its system virtual address, however, is `0xFFF00000`.

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 will 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 will extend the `sk_reg` structure which defines the device registers. We will 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  */

    (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. (This wouldn't be necessary if the address was sure to be in the DVMA transfer area, which always

ends at 0xFFFFF or below. However, the transfer address can also be elsewhere in the VMEbus address space).

- All the I/O now is started by skstrategy() and continues until skpoll() is called — thus we can delete the sk_cp and sk_count fields from the sk_device structure.
- 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. Note that, with skstart() eliminated, the call to wakeup(), as well as the clearing of sk_busy, have been moved to skintr().
- Finally, skintr() needs to free up the Main Bus resources, so it will call mbrelse().

Here are the new skintr() and skpoll() routines:

```
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 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);

    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 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);
}
```

Variation with "Asynchronous I/O" Support

In this next section, we will assume that we want to further modify the Skeleton driver to support "asynchronous I/O". This may, at first sight, seem an odd thing to do, for asynchronous I/O is most commonly used for network and serial-line devices that have little in common with the Skeleton device. In actual fact, however, asynchronous I/O is *not* limited in application to such devices — its purpose is to support user processes which need to avoid blocking during I/O operations, and such functionality is of interest for serial lines, sockets, STREAMS and various character devices.

First, note that the term "asynchronous I/O" is used, in the UNIX world, to indicate two separate mechanisms. In practice, these mechanisms are closely related, and both of them will be covered in this section:

- The first is "non-blocking I/O". This is a type of I/O which, when incapable of immediately proceeding to completion, notifies its user process of this fact rather than simply going to `sleep()`. It thus gives the user process a choice of responses.

  In the UNIX system, non-blocking I/O is traditionally provided by the `select()` system call, which allows a user process to query a device to see if it's ready before making a `read()` or `write()` request to it, and thus to avoid being blocked. (It should be noted that `select()` isn't really non-blocking I/O proper. It's better thought of as an alternative to device polling, which can waste considerable CPU time).

- The second UNIX asynchronous I/O mechanism is best called "asynchronous notification". With this mechanism available, the user process no longer needs to keep trying an I/O operation until it succeeds, because the driver will `signal()` it (with a `SIGIO`) when one of its I/O channels clears. The code necessary to support such asynchronous notification is closely related to that necessary to support `select()`, and it should routinely be provided at the same time as `select()` support.

Select Routines

The Skeleton driver hasn't really been defined as a device that we would expect to have a `select()` routine. Such routines are most useful for devices which aren't always ready, and since we've defined the Skeleton device as being write only and arbitrarily fast, we wouldn't expect it to clog. Still, for the purposes of this example, we will assume that the Skeleton board is sufficiently slow that it's reasonable to have its driver support `select()`.

`select()` is more typically used in serial-line drivers which are multiplexed between multiple lines. Before reading, for example, a terminal's keyboard, such drivers need to ensure that there are characters waiting. If they didn't, they would block so often that their overall performance would be unacceptable.

`select()` works by providing user processes with a means of determining if I/O is possible on a given file descriptor. Alternatively, it has a multiplexing feature that makes it possible to determine which of a set of specified descriptors is ready to go. It can be told to return immediately, or to block the calling process until at least one descriptor is ready. A timeout argument can be specified to keep the process from blocking forever, or to allow the process to periodically do something else. See `select(2)` for details.

The driver's `select()` routine may or may not support the full functionality of the `select()` system call. The minimum that it can reasonably do is allow the user program to poll the specified device to determine if it's ready:

```
skselect(dev, rw)
    dev_t dev;
    int rw;
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    int s = spl5();

    md = skdinfo[SKUNIT(bp->b_dev)];
    sk_reg = md->md_addr;

    /* Check if the device is ready */
    if (sk_reg->sk_csr & SK_DEVREADY) {
        (void) splx(s);
        return (1);
    }
    (void) splx(s);
    return (0);
}
```

Note that, in this example, the `rw` flag has been ignored because the Skeleton device is write only. If, however, it were a read/write device, `skselect()` would switch on `rw`, and do a separate readiness test for each of the `READ` and `WRITE` cases. Throughout this example we will show only write cases: read cases would be handled identically.

To extend `skselect()` to allow user processes to block for specified periods of time (or, for that matter, indefinitely) while waiting for an OK to proceed with an I/O operation, more must be done. To begin with, we must add two fields to the `sk_device()` structure. Both of them must be initialized to 0.

```
struct sk_device {
    ...
    struct proc *sk_wsel;      /* user proc structure */
    int sk_state;              /* select state flag */
};
```

We also need the flag

```
#define SK_WCOLL    0x01
```

which will be used to indicate that a write-select collision has occurred, that is to say, that more than one process has attempted to select the device.

Then, `skselect()` must be changed, as follows:

```
skselect(dev, rw)
    dev_t dev;
    int rw;
{
    register struct mb_device *md;
    register struct sk_reg *sk_reg;
    register struct sk_device *sk;

    int s = spl5();

    md = skdinfo[SKUNIT(dev)];
    sk_reg = md->md_addr;
    sk = &skdevice[SKUNIT(dev)];

    /* Check if the device is ready */
    if (sk_reg->sk_csr & SK_DEVREADY) {
        (void) splx(s);
        return (1);
    }

    /* Here's the new code */
    if (sk->sk_wsel &&
       (sk->sk_wsel->p_wchan == (caddr_t) &selwait))
        sk->sk_state |= SK_WCOLL;
    else
        sk->sk_wsel = u.u_procp;

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

`selwait`, an external integer imported via `<sys/systm.h>`, is the "channel" which the `select()` system call, and only the `select()` system call , uses when it calls `sleep()`.

If the device is ready to go, `skselect()` behaves just as it did above: it returns immediately with a 1. If, however, the device isn't ready, a check is made to see if it has already been selected. If it hasn't been, the field `sk_wsel` is set to point to the `proc` structure of the process doing the select. In effect, we're remembering the first process to select the device. If no other processes select the same device, this structure will later be used as a "fast path" to the selecting process.

If, however, `skselect()` finds that `sk_wsel` has already been set, the test:

```
(sk->sk_wsel->p_wchan == (caddr_t) &selwait)
```

is made to see if the process indicated by `sk->sk_wsel` is sleeping as a result of a call to `select()`. If it is, the code

```
sk->sk_state |= SK_WCOLL;
```

is executed to indicate that a select "collision" has occurred, that is, that a second (or third, etc.) process attempted to select the device while the first process was still waiting for it to become available.

The rest of the select-related code is executed at interrupt time, so it goes into `skintr()`. One clean way of inserting it is to create a new routine, `skwakeup()`, and to call it from `skintr()` instead of calling `wakeup()`. (See the non-DMA version of `skintr()`, above):

```
skwakeup(sk)
    register struct sk_device *sk;
{
    if (sk->sk_wsel) {   /* select is pending */

        /* wake up the process */
        selwakeup(sk->sk_wsel, sk->sk_state & SK_WCOLL);

        /* reset the select flags */
        sk->sk_state &= ~SK_WCOLL;
        sk->sk_wsel = 0;
    }
    wakeup((caddr_t) sk);
}
```

`selwakeup()` thus receives a `NULL` second parameter unless a select collision occurred. If such a collision did occur, *all* processes which are sleeping as a result of a `select()` (any select) are awakened by a call to `wakeup()` on the `selwait` channel. Most of them will just go back to sleep, and the ones that don't will race for the device. This isn't very efficient, but it doesn't happen very often. Usually, the device will be selected by a single process, and the `proc` structure will be used to wake only that process.

Note that `selwakeup()` does nothing if `sk->sk_wsel` is 0, or if there are no processes sleeping on `selwait`. Thus, if a process has called `select()`, but not gone to sleep (because the device was immediately ready) the subsequent interrupt will simply reset the flags.

Adding Asynchronous Notification

If the driver is to support asynchronous notification as well as `select()`, a bit more is necessary. First, a new flag is necessary to indicate that the user has requested asynchronous notification:

```
#define SK_ASYNC    0x02
```

And a new field is necessary in the `sk_device` structure, which now becomes:

```
struct sk_device {
    . . .
    struct proc *sk_wsel;       /* user proc structure */
    int sk_state;               /* select state flag */
    short p_pgrp;               /* user process group leader */
};
```

The new field, `p_pgrp` must, like the others, be initialized to 0. And `p_pgrp` must be initialized in `skopen()` to indicate the process group leader of the user process opening the device:

```
    . . .
    if (sk->p_pgrp == 0)
        sk->p_pgrp = (u.u_procp)->p_pid;
```

Next, we must provide a way for the user process to request that the driver enable asynchronous notification. Of course it would be possible for it to always operate in asynchronous mode, but then user processes would constantly get sent `SIGIO` signals by the driver, whether they expected them or not. Besides, if the Skeleton driver has multiple *modes*, we must introduce an `skioctl()` routine to toggle them, and that gives us an opportunity to discuss *ioctl* routines. Actually, there are potentially three system calls that can be used to put a driver into asynchronous mode, or, for that matter, into any mode. The most common of these is `ioctl(2)`, and that is shown here. Note, though, that the other two possibilities are `fcntl(2)` and `open(2)`.

Adding an **`ioctl()`** routine

The first step in introducing an `ioctl()` routine is to define the macros which user processes will use to issue commands to the device and its driver. (For details, see the discussion of `ioctl()` routines in the *Summary of Device Driver Routines* appendix to this manual).

In the case of `skioctl()`, these macros are few and simple, for `skioctl()` will only toggle the driver mode between synchronous and asynchronous. There's no need for the `ioctl()` macros to either ship data from, or return it to, the user program.

`ioctl`– related command codes are exported to user processes by means of macros kept, by convention, in `/usr/include/sys`. In the case of the

Skeleton driver, only two macros are necessary, and we will put them into `<sys/skcmds.h>`:

```
#define SKSETSYNC    _IO(k,0)
#define SKSETASYNC   _IO(k,1)
```

The `_IO` macro is the simplest of the `ioctl()` macros, being intended for purposes like this, where no argument data need be transferred. Here, all that's necessary is to define a convention by which 0 indicates synchronous mode (the default) and 1 indicates asynchronous mode. Note the first parameter, 'k'. It's used, quite arbitrarily, to identify the `ioctl()` to be vectored to the Skeleton driver.

The additions to the driver are very simple. First, it must include the file containing its control macros:

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

Then, in `skioctl()` it simply takes the information encoded by the `_IO` macro to toggle the driver's state:

```
skioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
{
    register struct sk_device *sk;
    sk = &skdevice[SKUNIT(dev)];

    switch (cmd) {

    case SKSETSYNC:
        sk->sk_state &= ~SK_ASYNC;
        break;
    case SKSETASYNC:
        sk->sk_state |= SK_ASYNC;
        break;
    }
}
```

That's it. And now that `skioctl()` can set the `SK_ASYNC` flag, `skwakeup()` can reasonably test for it and, if it's set, call `gsignal()` to send the `SIGIO` signal to the user process group. Note that the `SK_ASYNC` signal must be cleared after calling `gsignal()`.

```
skwakeup(sk)
    register struct sk_device *sk;

    if (sk->sk_wsel) {  /* select is pending */

        /* wake up the process */
        selwakeup(sk->sk_wsel, sk->sk_state & SK_WCOLL);

        /* reset the select flags */
        sk->sk_state &= ~SK_WCOLL;
        sk->sk_wsel = 0;
    }
    if (sk->sk_state & SK_ASYNC) {
        gsignal(sk->p_pgrp, SIGIO);
        sk->sk_state &= ~SK_ASYNC;
        }
    wakeup((caddr_t) sk);
}
```

The final step in adding a `select()` routine to a driver is to edit the kernel `conf.c` file, and to plug the name of the new `select()` routine into the `cdevsw` structure in the place of the "nodev" or "seltrue" that is already there.

7 Configuring the Kernel

7.1. Background Information

In this chapter, we will assume that you've written your driver. The next step, obviously, is to build a kernel that includes your new driver. This process isn't difficult; Sun systems support easy kernel configuration, even without access to system source code. If the driver is a loadable driver then the kernel is not re-built and therefore the discussion of rebuilding the kernel does not apply. In this case, see the *Loadable Drivers* section of the *Driver Development Topics* chapter.

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 *Network Programming* and the `config(8)` man page. (Incidentally, `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 will 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 will be adding a new driver to the kernel, you will 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 default initializations of the `cdevsw` and `bdevsw` switches.
- The second is either `/usr/sys/'arch -k'/conf/files`, `/usr/sys/sun3/conf/files`, `/usr/sys/sun3x/conf/files`, `/usr/sys/sun4/conf/files`, or `/usr/sys/sun386/conf/files`, (depending upon the type of your machine). This file tells `config` where to find the source code for the kernel and its drivers.

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 SunOS 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 chapter 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 of entries in the config file, whereas new drivers are indicated by editing one or all of the files `conf.c`, `/usr/sys/conf.common/files.cmn` and `/usr/sys/sun[3,3x,4,4c]/conf/files` (or `/usr/sys/sun386/conf/files`). 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 will be managing. The drivers will use these macros *at compile time* to control conditional compilation and to size device tables.
- *mbglue.s* — contains assembly-level code that translates from the hardware interrupt mechanisms to the device-interrupt routines for the installed devices. *It does not exist on Sun-4 or Sun386i machines.*
- *Makefile* — a makefile that, when executed, will actually make the new kernel, compiling and linking files as necessary. Note that the entries in `/usr/sys/sun[3,3x,4,4c]/conf/files` (or `/usr/sys/sun386/conf/files`) 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 will need a configuration directory and a config file for your new kernel. `config` will create a configuration directory in `/usr/sys/'arch -k'` with the same name as the new config file in `/usr/sys/'arch -k'/conf`, so all you have to do is create that file:

```
example#  cd /usr/sys/'arch -k'/conf
example#  cp GENERIC 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 will 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.

We will install the Skeleton device within `vme16d16` by way of a 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).

On a Sun-3, Sun-3X or Sun-4, it would likewise be reasonable to choose the smallest of the available address spaces:

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.

One more point about the config file. The number of installed devices will be determined, for each driver, by `config`, and it will generate the appropriate `sk.h` header file for you in the configuration directory.

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 goals:

```
#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,
skiotcl, nodev,
skselect, skmmap, 0, 0
    },
    .
    .
}
```

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 will 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*`reset()`, *xx*`select()` and *xx*`mmap()`. The Skeleton driver uses the *xx*`ioctl()` routine from the previous chapter. *xx*`reset()` is *never* used so all devices set its entry to `nodev`, a special routine which always returns an error condition. *xx*`select()` is called when a user process does a `select(2)` system call; it returns 1 if the device can be immediately selected. In this example we are using the routine from the previous chapter. An alternative would be to recognize that since the Skeleton device is write only and arbitrarily fast, it's always selectable. In this case we could 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 `/usr/sys/conf.common/files_cmn` but into `/usr/sys/'arch -k'/conf/files`. Assuming that the driver source is in `/usr/sys/sundev`, here's the line you must add to `/usr/sys/'arch -k'/conf/files`:

```
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`, `/usr/sys/conf.common/files_cmn` and `/usr/sys/'arch -k'/conf/files` 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` finishes generating the `makefile`, it automatically goes on to generate the dependencies (unless you tell it not to with the **-n** command-line flag). The generation of the dependencies takes a long time, and before it starts, `config` will notify you with the message:

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

Now you can change directory to the new configuration directory, `../SKELETON` in this case, and make the new system:

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

Now you must add a new device entry to the `/dev` directory. The connections between the 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 /usr/sys/'arch -k'/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
```

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
```

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 a 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 `/usr/sys/'arch -k'/conf/files`:

```
    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. The driver can then use `rmalloc()` to allocate (from `kernelmap`) virtual space for the second-space registers/memory, and then call `mapin()` to map them into kernel space. (See the *Kernel Support Routines* appendix for details about `mapin()`).

7.4. Adding and Removing Loadable Drivers

All Sun architectures support loadable drivers. A loadable driver doesn't need to be linked with the kernel `.o` files. Nor does the system have to be rebooted or rebuilt for loadable drivers to be used. You can simply add a loadable driver to a running system. Once you have a driver in the loadable form, you can load it into the running system with the `modload`(8) command. You must be the superuser to do this.

Take care when loading an undebugged driver for the first time. Although there are many consistency checks made when a driver is loaded, it is still possible for drivers to crash the system. One of the more common crashes occurs when the running kernel is not `/vmunix`. `modload` assumes by default (unless the `-A` switch is provided) that the running kernel is `/vmunix`. It resolves driver references to kernel addresses by reading the symbol table from `/vmunix`. If `/vmunix` is not the running system, then the system is likely to crash when the driver is used.

A typical example of the `modload` command is:

```
example# modload zz.o -conf <config_file> -exec <exec_file>
```

This tells the kernel that the driver object module is in `zz.o`. (See the *Loadable Drivers* section of the *Driver Development Topics* chapter for information about how to build a loadable driver.)

Configuration information for the driver and optionally the block and character major numbers are specified in the file *config_file*. If `modload` is successful, the file *exec_file* is executed. This file is typically a script used to make the `/dev` entries for the driver. `modload`(8) has many options; see its man page for details.

Error messages from `modload` can appear in two places. The `modload` utility itself prints error messages to standard output on the terminal from which `modload` is run. In addition, modload-related kernel code can print information to the console. For this reason, we recommend that the console output be visible when you issue the `modload` command.

When it is loading a driver, `modload` may fail for a variety of reasons. For example, the driver initialization routine may not do all that is required (as described in the *Loadable Drivers* section of the *Driver Development Topics* chapter). Or the linkage structure in the driver wrapper module may have invalid addresses. Since it is not possible to return a unique error code for every possible condition, a single error code is returned and additional information is often printed on the console.

To inquire about device drivers after they are loaded, use the `modstat`(8) command. It displays the module id of the driver, the name of the device, and the major numbers of the block and character devices, as well as some additional information about the module.

The module id is required to unload a driver. A driver can be unloaded by using the `modunload`(8) command, as in this example:

```
example# modunload -id 2 -exec <exec_file>
```

This example assumes that the `modstat` command displayed the driver's module id as 2. The file *exec_file* is executed and if the execution is successful the driver is unloaded. Typically this file is a script that removes the `/dev` entries for the driver.

An example of a script that could be used with `modload` is as follows:

```
#!/bin/csh -f
if $3 != "0" then
    if ( ! -r /dev/zz0) then
        echo /etc/mknod /dev/zz0 b $3 0
        /etc/mknod /dev/zz0 b $3 0
    endif
endif

if $4 != "0" then
    if ( ! -r /dev/xrfd0a) then
        echo /etc/mknod /dev/xrfd0a c $4 0
        /etc/mknod /dev/rzz0 c $4 0
    endif
endif
```

The script is invoked with the following arguments:

```
<module_id> <module_type> <block_major_number> <character_major_number>
```

`modunload` could be invoked with the following script to remove the `/dev` entries for the driver:

```
#!/bin/csh -f
rm -f /dev/zz0
rm -f /dev/rzz0
```

8

Pseudo-Device Drivers — A Ramdisk

SunOS supports "software devices", sometimes called "pseudo devices", which have no associated physical devices. Such devices can be quite useful. The system memory devices, for example, are pseudo devices, and they can be used to access installed peripheral devices, as is shown in the discussion of frame-buffer installation in *Direct Opening of Memory Devices* section of this manual. The memory devices allow such direct physical-device access by providing a means by which processes can read and write physical memory outside their own address space. For example, the `ps` command uses the kmem pseudo-device driver to access the kernel's process tables by way of the physical memory to which the kernel is mapped.

This section will introduce pseudo-devices by way of a real, working pseudo-device ramdisk. As you will see, such a ramdisk requires none of the subtlety that makes physical disk drivers so difficult.[5] Yet it does buy speed, since ramdisks avoid two distinct kinds of file-system overhead:

- In normal use, IO buffers get paged out, despite the use of the kernel buffer cache to minimize unnecessary I/O operations. A ramdisk is an especially big win on reads, since reading processes must normally block while requested data is brought into the buffer cache.
- During normal file-system operation, file control information (like inodes) must be written synchronously with data. This overhead doesn't exist with ramdisks.

Ramdisks can be used for `/tmp`.

NOTE *In SunOS 4.1 there is a facility already in place to do this for you, called tmpfs. It addresses the benefits mentioned herein.*

This way, if a system crash results in the loss of ramdisk files, it's not a serious problem. Note that for some applications, particularly those that involve temporary files larger than ramdisk memory, using `/tmp` isn't a very good idea. An alternative is to mount the ramdisk as `/aux`, and to use it explicitly each time you think it's safe. Ramdisks have only a minimal impact on applications

[5] The ramdisk given here is *very* crude. A production version should have its memory allocated at boot time and should be pageable. And with the memory-management system introduced in SunOS 4.0, a ramdisk probably won't improve performance anyway. In general, you'll be better off letting UNIX manage memory as a page cache, rather than devoting some of that cache to a ramdisk, or use tmpfs.

software — once they're set up they are entirely transparent. (Note that ramdisks — like devices in general — can be shared by multiple processes. This driver can thus be used as an indirect means of sharing memory.)

8.1. A Ramdisk Driver

The following ramdisk driver consumes a half-megabyte of kernel memory, which is allocated to the ramdisk pseudo-device.

Ramdisk Source Code

Put the source code for the ramdisk driver into /sys/sundev/ram.c.

```
/*
 * Ramdisk pseudo-device to support I/O to real memory
 * (a statically allocated kernel array).
 */

#include "ram.h"
#if NRAM > 0
#include <sys/param.h>        /* Includes <sys/types.h> */
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/buf.h>

#define RAMSIZE (1024*512)     /* Half a megabyte */
char ram[NRAM][RAMSIZE];

ramopen(dev,wrtflag)
    dev_t dev;
    int wrtflag;
{
    return(minor(dev) >= NRAM ? ENXIO : 0);
}

ramsize(dev)
    dev_t dev;
{
    return(minor(dev) >= NRAM ? -1 : btodb(RAMSIZE));
}

ramread(dev,uio)
    dev_t dev;
    register struct uio *uio;
{
    if ((unsigned)uio->uio_offset > RAMSIZE)
        return(EINVAL);
    return(uiomove(ram[minor(dev)]+uio->uio_offset,
        MIN(uio->uio_resid, RAMSIZE - uio->uio_offset),
        UIO_READ, uio));
}

ramwrite(dev,uio)
    dev_t dev;
    register struct uio *uio;
{
    if ((unsigned)uio->uio_offset > RAMSIZE)
        return(EINVAL);
    return(uiomove(ram[minor(dev)]+uio->uio_offset,
        MIN(uio->uio_resid, RAMSIZE - uio->uio_offset),
        UIO_WRITE, uio));
}

ramstrategy(bp)
    register struct buf *bp;
{
    register long offset = dbtob(bp->b_blkno);

    if ((u_long)offset > RAMSIZE) {
        bp->b_error = EINVAL;
        bp->b_flags |= B_ERROR;
    } else {
        caddr_t raddr = ram[minor(bp->b_dev)]+offset;
        unsignednbytes=MIN(bp->b_bcount, RAMSIZE-offset);
        if (bp->b_flags & B_PAGEIO)
            bp_mapin(bp);

        if (bp->b_flags & B_READ)
            bcopy(raddr, bp->b_un.b_addr, nbytes);
        else
            bcopy(bp->b_un.b_addr, raddr, nbytes);
        bp->b_resid = bp->b_bcount - nbytes;
    }
    iodone(bp);
}
#endif
```

Pseudo-device drivers, by definition, have no corresponding physical devices. Thus, they have no probe routines.

Note the routine `ramsize`. All block drivers provide such a routine, which is charged with returning the sector size of the device in the peculiar units which the kernel expects. (This information is then used to maximize the speed of `fsck`). `ramsize()` calls the `btodb()` conversion routine, passing it an argument in bytes, and receiving from it an appropriately scaled result.

Ramdisk Installation

The more detailed discussion of these and related configuration procedures can be found in the *Configuring the Kernel* chapter of this manual. Edit `/usr/sys/`arch -k`/conf/files`, adding the following line to the end of it:

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

Then, edit both the `bdevsw` and `cdevsw` arrays in `/sys/sun/conf.c`, adding entries for the ramdisk to each of them. (In this discussion, we will only use the ramdisk as a block device, but the driver provides all the entry points necessary for use as either a block or a character driver).

```
#include "ram.h"
if NRAM > 0
int ramopen(), ramread(), ramwrite();
int ramstrategy(), ramsize();
#else
#define ramopen      nodev
#define ramread      nodev
#define ramwrite     nodev
#define ramstrategy  nodev
#define ramsize      nodev
#endif
    .
    .
    .
    {
ramopen, nulldev, ramstrategy, nulldev, /* 22 */
ramsize, 0
    }
    .
    .
    .
    {
ramopen, nulldev, ramread, ramwrite,    /* 63 */
nodev, nodev, seltrue, nodev, 0, 0
    }
```

Next, move into /dev and create device entries to correspond to the entries in conf.c.

```
example# cd /dev
example# /etc/mknod ram0 b 22 0
example# /etc/mknod rram0 c 63 0
```

The next step is to make a new configuration directory for the variant of you kernel that will include the ramdisk. Copy your kernel configuration file and add the line:

```
pseudo-device       ram
```

to the pseudo-device section of the copy. If your config file was named GENERIC, you might name the ramdisk variation GENERIC_RAM.

Then, make a version of the system kernel that includes the ramdisk:

```
example# /etc/config GENERIC_RAM
example# cd ../GENERIC_RAM
example# make
example# cp /vmunix /vmunix.old
example# cp vmunix /vmunix
example# /etc/reboot
```

During the reboot, note that the size of the kernel has gotten very large. After the reboot, make and associate a "filesystem" with the block ramdisk device:

```
/etc/mkfs /dev/ram0 1024 8 8 8192 1024 16 5 100
/etc/mount /dev/ram0 /tmp
```

That's 1024 blocks total (512 Kb), broken out as 8 sectors of 8 tracks of 8192 bytes per block with 1024 byte fragment size with 16 cylinders per group with 5% minimum free (as in `df(1)`) and 100 revolutions per second. (This two line sequence should probably be put in the `/etc/rc.local` script). The logical block size of the file system (8192) must be the same size as the pagesize (8K).

Once the ramdisk filesystem is mounted onto `/tmp`, then any program which creates and uses files on `/tmp` will use the ramdisk. Reads and writes to these files will be very fast. Measured performance indicates that I/O on files of about 10K bytes is about 5 times as fast as with a physical disk, and that this factor increases to about 10 for very large files.

Following is a test program that will exercise the driver. Another way to test it simply is to copy some data to the pathname via the `cp` command.

Ramdisk Test Program

Here is a program to test that the ramdisk works:

```
#define BUFSIZ 1024
#define CYCLES 100
#define RAMDISK

/*
 * Ramdisk test program
 */

main()
{
   int fd;                        /* file descriptor */
   int nb;                        /* number of bytes transferred */
   int i;                         /* generic loop counter variable */
   int count=BUFSIZ;
   char buffer[BUFSIZ];
   int iterations=0, error=0, done=0;

#ifndef RAMDISK
   /* Open a file on the regular filesystem */
   if ((fd = open("testfile", 2)) == -1 ) {
           perror("ramdisk test (normal opening)");
           exit(1);
   }
#else
   /* Open a file in the ram disk filesystem */
   if ((fd = open("/tmp/testfile", 2)) == -1 ) {
           perror("ramdisk test (ram opening)");
           exit(1);
   }
#endif

   do {
           lseek(fd, 0, 0);
           if (write(fd, buffer, count) != count) {
                   perror("ramdisk test (writing)");
                   exit(1);
           }
           lseek(fd, 0, 0);
           if (read(fd, buffer, count) != count) {
                   printf("count= %d0, count);
                   perror("ramdisk test (reading)");
                   exit(1);
           }
           if (iterations++ == CYCLES ) done++;
   }
   while ( !error && !done );
   close(fd);
   exit(0);
}
```

PART TWO: Appendices

A

Summary of Device Driver Routines

A.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>`.

A.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 primarily in block-device drivers. See the *The "Skeleton" Character Device Driver* chapter for additional information about many of these routines.

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.

xxattach() — 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. 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_intr->v_vptr`) is initially set to the unit number of the interrupting device, but it's often convenient to reset it to contain a pointer to a local structure.

NOTE *This does not work on the Sun 386i. It is hardcoded to be your* `irq`

xxclose() — 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()`.

xxintr() — 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 absolutely prohibited from calling `sleep()` or referencing the kernel `user` structure.

xx`intr()` receives one 32-bit parameter, which is, by default, the unit number of the device that interrupted. However, you can arrange for it to receive something else by changing the value in `md_intr->v_vptr`. (See *xx*`attach()`, above).

This does not apply to the Sun386i. The Sun386i receives two arguments. The first is the current priority level (`cpl`) and the second is the interrupt request (`irq`). The `irq` is hardwired so it cannot be changed. The interrupt routine can never receive the unit number. The unit number can be obtained by saving the interrupt request channel (board level + 8) at attach time and then figuring out which device received the interrupt at interrupt time.

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 instrumental in certain tasks which, by their nature, must be shared with top-half routines. Examples of such tasks are the maintenance of character I/O buffers and `select()`-related bookkeeping structures. (In the `select()` case, *xx*`intr()` also has the job of calling `selwakeup()` to wakeup sleeping processes).

Note that whenever *xx*`intr()` maintains a data structure or resource in cooperation with top-level routines, the top-level code *must* be protected by a mutual-exclusion lock. Interrupts are automatically disabled when an interrupt routine is called, so it is generally unnecessary for *xx*`intr()` to disable interrupts before it does its part of the job.

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 buffer header `b_flags` field (and, if an error code other than `EIO` is desired, by assigning that error code into the buffer `b_error` field). The error code will then be propagated up to 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`. *xx*`intr()` doesn't itself return anything.

Note that the driver *xx*`intr()` routine cannot itself set the `errno` register, since that register is actually a field in the `user` structure (`u.u_error`), and the `user` structure *must not* be accessed at interrupt time. Instead, *xx*`intr()` passes the error to the kernel via the buffer, and the kernel sets `u.u_error`.

- 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. (Note that the `b_resid` field in the buffer header will typically indicate the number of bytes of data that were still untransferred at the error return).

- The error return should abort the I/O request that produced the error and then place the device in its normal idle state.

*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 *xx*`ioctl()`. It is the escape mechanism by which miscellaneous operations are implemented.

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 255 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 255 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 written back 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 (in `<sys/vcmd.h>`):

```
#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. This character is not checked and is rather arbitrarily chosen. In this case, the "v" stands for "Versatec". The second parameter is the command code itself. The third is the size of the `ioctl()` argument, which cannot exceed 127 bytes for SunOS 3.x and 255 bytes for SunOS 4.x. 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 an *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 a 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 the subset of the page table entry (PTE) containing the page frame number and the page type to its caller in the kernel. *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.

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 size for transfers, but
              less than maxphys unless the new maxphys kernel
              label is increased */

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 will be needed later.
- Finally, *xx*open() will typically wait for the device to come on-line, and return an error if it doesn't.

NOTE *If xx*open() supports "clone open", that is to say, if it will allow a user to open a driver without specifying a minor device, then it is important that it does not do anything that may lead to its being blocked before it actually chooses the minor device that it is going to clone. Otherwise, there's a possibility of someone else grabbing the device while *xx*open() is blocked.

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 functions 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(), peekl(), poke(), pokec() and pokel().

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 than 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 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 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 use it will cause the kernel to `panic()` with a bus error.

Note that the amount of memory mapped in by the autoconfiguration code is determined by the size given in the `mb_driver->mdr_size` field, and not by the value returned from *xx*`probe()`, which is used only for the go/nogo test.

*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 the read operation fails, *xx*`read()` passes the error code which *xx*`intr()` set in the buffer header up to the kernel. The kernel then passes it on to the user program, which usually plugs it into the users global error variable `errno`.

*xx***select()** — Select Support

```
xxselect(dev, rw)
    dev_t dev;
    int rw;
```

The *xx*`select()` routine is necessary if the driver is to support the `select()` system call. *rw* is either `FREAD`, `FWRITE` or 0. (Simple character devices won't have occasion to use the 0 value, which is intended for exceptional conditions. It is used by network devices). These constants are defined in <sys/file.h>.

If *xx*`select()` only supports polling, then it simply determines if the device specified by the major/minor pair encoded within `dev` is ready to go, returning a 1 if it is and a 0 if it's not. Interrupts must be disabled while this check is performed, so *xx*`select()` should always do a

```
s = spl6();
```

immediately, and a

```
splx(s)
```

before returning.

If, however, *xx*`select()` allows user processes to wait for a device to become ready, it must do somewhat more work. In this case, the driver will have to maintain a local per-device structure which can associate a process with each device. It can do so with the current process `proc` structure, a pointer to which can be found in `u.u_procp`. (If the device can read and write independently, separate processes must be tracked for the two cases). The local structures must also contain some state information, which will be used by *xx*`select()` (as well as *xx*`intr()`) for bookkeeping purposes. The details are somewhat complicated, and are illustrated in the *Variation with "Asynchronous I/O" Support* section of the *The "Skeleton" Character Device Driver* chapter of this manual.

xxstrategy() — High-Level I/O

```
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()`.

xxwrite() — 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 the write operation fails, *xx*`write()` passes the error code which *xx*`intr()` set in the buffer header up to the kernel. The kernel then passes it on to the user program, which usually plugs it into the global error variable `errno`.

B

Kernel Support Routines

These routines are in alphabetical order, on the assumption that this will make them easier to find.

bcopy() — Byte Copy for Nonoverlapping Regions

```
void
bcopy(from, to, count)
    caddr_t from, to;
    u_int count;
```

Copies *count* bytes from the address designated by *from* to the address designated by *to*. The operands may not overlap; i.e. the interval [*from*, *from+count*] must be disjoint from the interval [*to*, *to+count*]. Use `ovbcopy()` to copy overlapping regions.

bp_mapin() — Map a user buffer into kernel space

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

`bp_mapin()` allocates kernel virtual address space from the kernelmap, maps the data referred to by the buffer *bp* into the space allocated, and converts *b_un.b_addr* to the new address, which is now valid at any time in the kernel. The driver must call `bp_mapin()` after calling `physio()` but before starting the data transfer - usually somewhere early in the driver's `xxstrategy()` routine, before using the *b_un.b_addr* field from the buffer header.

Device drivers that use the kernel routine `physio()` may be affected by a change in the implementation of `physio()` from SunOS 4.0 on. The change will only affect drivers that touch the actual data in the I/O buffer themselves, from the bottom half of the driver (the interrupt routine).

The `physio()` routine no longer maps the user's I/O buffer into kernel virtual address space using the kernelmap. The result is that the data address field in the buffer header (*bp->b_un.b_addr*) is now the same as the user context virtual address. This has no impact on a driver that touches the data in the I/O buffer only in its top half.

However, when the interrupt routine is running, this data address will not necessarily be valid. If the driver tries to touch the data in the buffer during its interrupt processing, a variety of errors will occur, ranging from silently touching the wrong data to kernel bus error traps. This is a change from earlier SunOS releases, where the buffer address was valid at any time in the kernel, after `physio()` had been called.

For drivers that need to reference the data in the I/O buffer from interrupt level, the correct approach in SunOS 4.0 and later is to use `bp_mapin()` and `bp_mapout()`. Using `bp_mapin()` to access the user buffer inside the device driver will ensure the user buffer is aligned on the same cache line as `mbsetup()`, for machines with a write-back cache.

bp_mapout() — Map out a user buffer in kernel space

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

`bp_mapout()` undoes the kernel mapping and releases the space in the kernelmap.

btodb() — Convert Bytes to Disk Sectors

```
btodb(bytes)
    int bytes;
```

Converts *bytes* into standard kernel block-size units. `btodb()` is called (for block drivers) from *xx*`size()`. It is listed here because it is called from the example ramdisk pseudo-device driver.

bzero() — Initialize Byte Memory Region to Zero

```
void
bzero(base, count)
    caddr_t base;
    u_int count;
```

Initializes to zero *count* bytes starting from the address designated by *base*.

copyin() — Move Data From User to Kernel Space

```
copyin(udaddr, kaddr, n)
    caddr_t udaddr, kaddr;
    u_int n;
```

`copyin()` moves data from the user address space to the kernel address space. It is commonly used when writing *xx*`ioctl()` routines. See `copyout()`.

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, `EFAULT` on a memory error, and other `Exxx` errors on pagefaults which cannot be resolved. The value of n can be up to 1 megabyte, any more is dependent on the specific machine architecture.

copyout() — Move Data From Kernel to User Space

```
copyout(kaddr, udaddr, n)
    caddr_t kaddr, udaddr;
    u_int n;
```

`copyout()` moves data from the kernel address space to the user address space. It is commonly used when writing *xx*`ioctl()` routines. See `copyin()`. *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, `EFAULT` on a memory error, and other `Exxx` errors on pagefaults which cannot be resolved. The value of n can be up to 128 kilobytes, any more is dependent on the specific machine architecture.

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()`.

dma_done() — Free the DMA Channel

```
dma_done(chan)
    int chan;
```

On Sun386i only. After a DMA transfer completes, dma_done() must be called to mark the channel as not busy so that another transfer can proceed.

dma_setup() — Set Up for a DMA Transfer

```
dma_setup(dma)
    struct dma_request *dma;
```

On Sun386i only. dma_setup() is called after the driver has gotten a contiguous set of virtual addresses from mbsetup() and before the device is programmed to start sending or receiving data. The dma_request structure (defined in <sun386/dma.h>) contains all the information required to set up the 82380 DMA chip on the Sun386i.

Unlike the Sun-3, Sun-3X, Sun-4 line of machines, the Sun386i has a memory management unit as an integral part of the CPU (the 80386). Therefore, to use the DMA facility of the Sun386i for a device driver, you must interface to the 82380 chip, which contains the DMA controller.

The primary interface to the DMA chip is the dma_request structure. You must fill in the fields in this structure and then call dma_setup() with a pointer to the structure. dma_setup() takes the contiguous virtual addresses, which were obtained from a call to mbsetup(), and sets up a linked list of physical addresses to be loaded into the DMA chip as needed.

dma_setup() returns a value of zero if the setup was successful, and non-zero if there is a problem. Reasons for failure are: the channel was busy, the transfer was zero pages long, or memory could not be allocated for the linked list of buffers.

The fields in dma_request structure are defined as follows:

```
/*
 * DMA request structure passed to dma_setup().
 * See the Intel 82380 Tech Ref for more info.
 */
struct dma_request {
    u_char  dma_channel;            /* Channel number: 0 - 7 */
    u_char  dma_xfer_mode;          /* Transfer mode */
#define   DMA_DEMAND_MODE   0
#define   DMA_SINGLE_MODE   1
#define   DMA_BLOCK_MODE    2
#define   DMA_CASCADE_MODE  3
    char    dma_rdwr;               /* Transfer direction */
#define   DMA_READ          2         /* (Relative to requester) */
#define   DMA_WRITE         1
    u_long  dma_count;              /* Transfer count */
    u_long  dma_req_space;          /* Requester address space */
#define   DMA_MEMORY        0       /* Memory or memory-mapped */
#define   DMA_IO            1         /* I/O mapped */
    u_int   dma_req_size;           /* Size of xfers to/from requester */
#define   DMA_BUS_32        1         /* 32-bit transfers */
#define   DMA_BUS_16        2         /* 16-bit transfers */
#define   DMA_BUS_8         3         /* 8-bit transfers */
    char    dma_req_hold;           /* 1 = hold address, 0 = increment */
    caddr_t dma_req_addr;           /* Requester (virtual) address */
    u_long  dma_target_space;       /* Target address space */
    u_int   dma_target_size;        /* Size of xfers to/from target */
    char    dma_target_hold;        /* Hold/increment target address */
    caddr_t dma_target_addr;        /* Target (virtual) address */
};
```

In this context, the "requester" is the device that requests service from the 82380 (normally a peripheral such as a disk controller). The "target" is the "device" with which the requester wants to communicate (normally system memory).

The fields of the `dma_request` structure are used as follows:

`dma_channel`
: Specifies the channel that the requester will use for the transfer.

`dma_xfer_mode`
: Refers to the type of transfer that the requester is capable of supporting. The SCSI controller, for instance, uses the `DMA_SINGLE_MODE` of transfer, as does the floppy controller. Refer to the peripheral manufacturer's specification sheet and the 82380 data sheet for more details.

`dma_rdwr`
: is the direction of data transfer *relative to the requester.* `DMA_WRITE` means transfer from the requester to the target and `DMA_READ` means transfer from the target to the requester.

`dma_count`
: is the byte count for the transfer.

`dma_req_space`
: is the address space in which the requester resides, i.e., whether the device is memory mapped (`DMA_MEMORY`) or I/O mapped (`DMA_IO`).

`dma_req_size`
: is the size of the requester's data path (`DMA_BUS_8` = 8 bits, `DMA_BUS_16` = 16 bits, `DMA_BUS_32` = 32 bits) and therefore the amount of data transferred with each DMA bus cycle.

`dma_req_hold`
: indicates whether the 82380 should hold the requester address constant throughout the DMA transfer, or increment it with each bus cycle. Typically the requester address is the address of the device's I/O register, which is fixed, so dma_req_hold is set to "1".

`dma_req_addr`
: is the requester's virtual address.

`dma_target_space`
: is the address space in which the target resides (usually `DMA_MEMORY`).

`dma_target_size`
: is the size of the target's data path (`DMA_BUS_32` for system memory).

`dma_target_hold`
: indicates whether the 82380 should hold or increment the target address during the DMA transfer. For memory devices, the 82380 should increment the target address with each bus cycle, so "dma_target_hold" is set to 0.

`dma_target_addr`
: is the target's virtual address.

Once all these fields are set up by the driver, the driver calls the `dma_setup()` routine. The following pseudo-code routines demonstrate how to use the DMA routines:

```
#include <machine/dma.h>
#include <sundev/mbvar.h>

struct   mb_device *xxinfo;       /* Device info */
caddr_t   xx_ioaddr = XX_ADDR;    /*Address of device's I/O port */

xx_example(bp)
   struct   buf *bp;
{
   struct   mb_device *md = xxinfo[0];
   unsigned int target_addr;
   unsigned int transfer_count;
   int   channel;
   int   readflag;

  /*
   * Set up DMA transfer.
   */
   target_addr = MBI_ADDR(mbsetup(md->md_hd, bp, 0));
   transfer_count = bp->b_bcount
   channel = md->md_dmachan;
   readflag = ((bp->b_flags & B_READ) ? 1 : 0);

   if (xx_dma_setup(target_addr, transfer_count,
     channel, readflag)  !=  0)
      return(-1);

 /*
  * Code to talk to the device, initiate the transfer,
  * and wait for transfer completion.
  */
   .
   .
   .

 /*
  * Free DMA resources.
  */
   xx_dma_done(channel);
   mbrelse(md->md_hd, &target_addr);

   return(0);
}

xx_dma_setup(addr, count, chan, rdflag)
  unsigned int addr;
  unsigned int count;
  int   chan;
  int   rdflag;
{
  struct   dma_request dreq;

  dreq.dma_channel = chan;              /* Dma channel */
  dreq.dma_xfer_mode =
    DMA_SINGLE_MODE;                    /* Single mode transfer */
  dreq.dma_rdwr =
    (rdflag ? DMA_WRITE : DMA_READ);   /* Direction */
  dreq.dma_count = count;               /* Transfer count */

  dreq.dma_req_space = DMA_MEMORY;    /*Memory-mapped requester*/
  dreq.dma_req_size  = DMA_BUS_8;     /* 8-bit data path */
  dreq.dma_req_hold  = 1;               /* Hold address constant */
  dreq.dma_req_addr  = xx_ioaddr;     /* I/O port virt. address */

  dreq.dma_target_space = DMA_MEMORY; /*Target is system memory*/
  dreq.dma_target_size = DMA_BUS_32;  /* 32-bit data path */
  dreq.dma_target_hold = 0;            /*Increment addr each cycle*/
  dreq.dma_target_addr = addr;         /* Buffer virtual address */
  return(dma_setup(&dreq));
}

xx_dma_done(chan)
   int   chan;
{
   dma_done(chan);
}
```

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()`.

hat_getkpfnum() — Address to Page Frame Number

```
u_int
hat_getkpfnum(addr)
    addr_t addr;
```

`hat_getkpfnum` takes a virtual address and returns its associated Page Frame Number. This number has already been masked down to one that can appropriately be returned by the driver *xx*`mmap()` routine.

inb() — Read a Byte from an I/O Port

```
inb(port)
    short port;
```

Sun386i only. `inb()` returns the byte value from the specified port address in the I/O space. (See `outb()`).

iodone() — Indicate I/O Complete

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

In the skeleton driver example, `iodone` is called to indicate that I/O associated with the buffer header *bp* is complete, and that it can be reused. `iodone` sets the `DONE` flag in the buffer header, then does a `wakeup` call with the buffer pointer as argument. `iodone()` is called from the bottom half right after the call to `wakeup()`. See `iowait()`.

iowait() — Wait for I/O to Complete

```
int 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()`. `iowait()` also returns the error value. See `iodone()`.

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. If called from an interrupt routine, `kmem_alloc()` can return a `NULL`. (Though `kmem_alloc()` generally should not be called from the interrupt level.) It returns a NULL if its request can't be satisfied. Note that `kmem_alloc()` takes a while, and shouldn't be used frivolously. 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. `kmem_free()`, like `kmem_alloc()`, should not be called from the interrupt level.

log() — Log Kernel Errors

```
log(pri_code, . . .)
    int pri_code;
    . . .
```

The kernel provides a `log()` function analogous to the `syslog(3)` function supplied with the C library for user programs. The first argument to `log()` is a priority code, as defined in `<sys/syslog.h>`, and is identical to the priority codes used by `syslog(3)`. The subsequent arguments are a `printf()` format string and the values to be printed under its control. Unlike `syslog()`, the format string must be terminated with a newline (\n) if a newline is to be printed at the end of the message.

Messages logged with `log()` will not pass though the normal kernel `printf()` mechanism if the `syslogd` daemon is running. They will get written to the system message buffer just as `printf()` messages are. The `syslogd` daemon will read them using a special device driver, and will log them as messages from the "kern" facility with the given priority.

If such a message is to be printed on the console, `syslogd` will do so, using its standard format which includes a time stamp. Messages printed with `printf()` will get logged as messages from the "kern" facility with a priority of `LOG_CRIT`, except that `syslogd` will not print them on the console as they have already been printed there by the kernel. The kernel does not time stamp messages that it prints; thus, messages logged with `log()` will be time stamped if they are printed on the console, while messages printed with `printf()` will not. Furthermore, `syslogd` does not lock out interrupts while printing messages, so messages logged with `log()` will not tie up the machine while they are being printed, unless `syslogd` is not printing and the kernel must print the message itself.

`machineid()` — Get Host Id From Eprom

```
machineid()
```

`machineid()` takes no arguments and returns an unsigned int which contains the same value returned by `gethostid()`. This is useful when the driver, running in kernel space, needs to know the hostid of the machine it is running on.

`MBI_ADDR()` — Get Address in DVMA Space

```
MBI_ADDR(mb_cookie)
    int mb_cookie;
```

`MBI_ADDR()` is a macro that takes the "cookie" (abstract number) returned by `mbsetup()` and converts it into a 32-bit transfer address, which may be either in the DVMA space or a VMEbus address space. This is the address that is then given to the bus-master device, though it may first need to be checked (especially for older devices) to ensure that it is not larger than the device capacity. See `mbsetup()` and `mbrelse()`.

`mb_mapalloc()` — Get Address in DVMA Space

```
caddr_t mb_mapalloc(map, bp, flags, waitfp, arg)
    struct map *map;
    register struct buf *bp;
    int flags;
    int (*waitfp)();
    caddr_t arg;
```

This is one of two new routines that device drivers can use to allocate DVMA space for I/O transfers. These routines are a move toward separating the allocation and maintenance of DVMA resources from the complex framework of the mainbus ("mb") structures. They also simplify matters in the case when no DVMA space can be allocated. The old `mbsetup()` and `mballoc()` interfaces are retained for compatibility with current drivers, so use of the new routines is entirely optional.

There are two main differences between the old and new routines. The first is that the new routines use a generic *map* structure instead of a pointer to a struct *mb_hd*. This provides for systems which do not have a "mainbus" but which do have DVMA capability.

The second difference is the way in which the allocation routines behave if there is no DVMA space available; the old scheme would return NULL and force drivers to call the allocation routines at some later point, either by way of a periodic timer in the driver or by being interrupted by I/O completion. The new interfaces use a "callback" scheme to inform drivers when DVMA has become available again. The driver passes in a pointer to the routine it wishes to be called back with and an argument to the callback routine. The argument is data private to the device driver (i.e. the allocation routines don't examine or modify it) and can optionally be used as a hint by the driver to itself. After the driver's callback request is queued, the allocation routines return NULL.

At this time, the driver puts the request which failed on a wait queue of its own, since the allocation routines only queue the callback routine, not the request itself. This method allows drivers to manage their own queues and to perform any optimizations on the request ordering they deem useful. For simplicity and economy of kernel resources, callback routines are only placed on the wait queue if they are not already there. Subsequent requests using an already queued callback routine will be ignored. The remaining responsibility of the driver's callback routine is to return DVMA_RUNOUT (defined in `<sys/mbvar.h>`) when DVMA has run out, as the allocation routines must know when to stop pulling callback routines off the wait queue.

map is a pointer to the DVMA allocation map structure, *buf* is the buffer header associated with this request, *flags* is set by the device driver to indicate special processing for this request, *waitfp* is a pointer to a function to be queued by the allocation routines if DVMA space is not available and the driver has set the *flags* parameter to *MB_CANTWAIT*, and *arg* is the argument to the callback function.

The following example shows a simple device driver `start()` routine that uses the `mb_mapalloc()` function to obtain DVMA space.

```
xxstart(arg)
    caddr_t arg;
{
    struct buf *bp;
    struct xxunit *un;
    int bufaddr, unit;

    for (bp = bufq; bp; bp = bp->av_forw) {
        unit = dkunit(bp);
        un = &xxunits[unit];
        .
        .
        if (bufaddr = mb_mapalloc(un->un_mc->mc_mh->mh_map, bp
            MB_CANTWAIT, xxstart, arg)) {
            .
            .
            xxgo();
        } else {
            bufq = bp;
            return (DVMA_RUNOUT);
        }
    }
    bufq = bp;
    return (0);
}
```

Points of note: The variable *bufq* is a queue of buffer pointers maintained by the driver; incoming requests are put on this queue, as well as requests that could not get DVMA space. The callback routine is the `xxstart()` routine itself, ignoring any arguments. This could have been a separate function within the driver, but we are showing simplicity here.

If DVMA is not available (i.e., if the return value of `mb_mapalloc()` is NULL), then `mb_mapalloc()` will queue up a pointer to the `xxstart()` function. When DVMA space frees up, `xxstart()` will be called back and will attempt to run its queue again. Note that even though space is now available, there is no guarantee that it will be sufficient to map this particular request. In such an event, `mb_mapalloc()` will simply requeue the request.

Since the `xxstart()` routine can be invoked by other driver routines or by the DVMA allocation routines, care should be taken in how such arguments are used.

`mb_nbmapalloc()` — Get Address in DVMA Space

```
caddr_t mb_nbmapalloc(map, addr, bcnt, flags, waitfp, arg)
    map *map;
    caddr_t addr;
    int bcnt, flags;
    int (*waitfp)();
    caddr_t arg;
```

This is the second of two new routines that device drivers can use to allocate DVMA space for I/O transfers. This routine is for devices which do not use the *buf* structure, but still need to request DVMA space.

The only difference between this and `mb_mapalloc()` is that the *buf* structure has been replaced by *addr* and *bcnt*, which represent the buffer address and byte count, respectively.

`mapin()` — Map Physical to Virtual Addresses

```
mapin(ppte, vpagenum, physpagenum, sizeinpages, access)
    struct pte *ppte;
    u_int vpagenum, physpagenum;
    int sizeinpages, access;
```

`mapin()` maps physical addresses to virtual addresses. Device drivers use it to set up kernel virtual memory so that device registers and memory can be directly accessed. This is useful for devices which:

- interface to the kernel by way of two different memory spaces. Since the autoconfiguration process only sets up one space, such cases are best handled by having the *xx*`attach()` routine use `mapin()` to set up the other.
- can consume variable amounts of virtual memory space, and for which, therefore, an optimum mapping cannot be made at autoconfiguration time. This is the case, for example, with certain kinds of variable-resolution frame buffers.

Drivers that call `mapin()` in their *xx*`attach()` routines must first call `rmalloc(kernelmap, ...)` to get the kernel virtual addresses which `mapin()` requires. (Actually, `rmalloc()` will return indexes to kernel virtual addresses—see below). Note that, when a driver calls `mapin()`, it should also call `mapout()` to return the mapped virtual memory when its no longer needed.

ppte is a pointer to the PTE which performs the mapping. This is the PTE in `Sysmap` (defined in `<sun[33X4]/pte.h>`) which corresponds to the map index returned from `rmalloc(kernelmap, ...)`. That is, *ppte* can be given as `&Sysmap[kmx]`, where `kmx` is the map index returned by `rmalloc()`.

vpagenum is the number of the virtual page where the physical memory is to be mapped. `kmx`, the map index returned by `rmalloc()`, can be used to calculate a virtual address, which can then be converted to a virtual page number like so:

```
vpagenum = btoc(Sysbase) + kmx;
```

Here `Sysmap` is the external array of page table entries used to map virtual addresses, starting at the (kernel virtual) base address `Sysbase`. `btoc()` is a macro (see `<machine/param.h>`) which converts addresses to page numbers, and, if necessary, performs the appropriate rounding.

Note that there are a number of general-purpose macros designed to convert between kernel map indexes and virtual addresses. These macros are in `<sys/vmmac.h>`. One of them, `kmxtob` expects an (int) kernel map index and returns the virtual address by page number. Another, `btokmx` expects a (caddr_t) virtual address and returns the integer kernel map index.

physpagenum is the physical page number of the memory being mapped into kernel virtual memory. Actually, it is the physical page number with the appropriate type bits for the given physical memory space—these types bits (PGT_*) are given in `<sys/pte.h>`.

sizeinpages is the size in pages of the memory being mapped. It can be easily computed by using the `btoc()` macro to convert the size (in bytes) of the memory being mapped into pages (since `btoc()` will round up as needed).

access is the PTE-level access flags. The flags (PG_*) are defined in `<sys/pte.h>`. The value passed by the auto-configuration process when it calls `mapin()` (the standard device driver case) is "PG_V|PG_KW", which indicates valid system pages with their write-enable flags set.

See `fbmapin()` and `fbmapout()` in `fbutils.c` (in the *Sample Driver Listings* appendix) for examples of real `mapin()` and `mapout()` calls. It is advisable to map in small portions of a device's memory (less than or equal to 6M bytes) instead of fewer mappings of large memory chunks.

`mapout()` — Remove Physical to Virtual Mappings

```
mapout (ppte, sizeinpages)
    struct pte *ppte;
    int sizeinpages;
```

`mapout()` is used to unmap a chunk of physical memory from the virtual memory that `mapin()` associated it with. Its parameters are as given in `mapin()`, above. Drivers typically need to call `mapout()` only when they have made their own calls to `rmalloc()` and `rmfree()`. It should be called just before `rmfree()`.

`mballoc()` — Allocate a Main Bus Buffer

```
mballoc(mh, addr, bcnt, flags)
    struct mb_hd *mh;
    caddr_t addr;
    int bcnt, flags;
```

`mballoc()` is a wrapper for `mbsetup()`. It allocates a `buf` struct, zeroes it out, stuffs the `b_un.b_addr` field with *addr*, sets the `b_flags` word to `B_BUSY`, sets the `b_bcount` word to *bcnt* and calls `mbsetup()`. The arguments passed to `mbsetup()` are *mh*, the address of the buf struct, and *flags*.

`mbrelse()` — Free Main Bus Resources

```
mbrelse(mb_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(mb_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. (These fields are b_un.b_addr, b_flags and b_bcount). *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. If, on the other hand, *flag* is 0, the requesting process will be put to sleep until the necessary map resources become available.

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 program being serviced.

NOTE *Don't set* B_PHYS *in bp's* b_flags *field if DVMA is between kernel address space and the device.*

Upon success, mbsetup returns a number which must be saved for the call to mbrelse. This number can also be passed to MBI_ADDR(), which will transform it into a transfer address.

outb() — Send a Byte to an I/O Port

```
outb(port, data)
    short port;
    u_char data;
```

Sun386i only. On the Sun386i, many devices, such as the floppy, are accessed by way of the I/O space. outb() sends a byte value to the I/O address specified. I/O device addresses are in the range of 0 to 0xFFFF. (See inb()).

ovbcopy() — Copy Overlapping Byte Memory Regions

```
void
ovbcopy(from, to, count)
    caddr_t from, to;
    u_int count;
```

Copies *count* bytes from the address designated by *from* to the address designated by *to*. The operands may overlap. If they do not, it is more efficient to use `bcopy()` instead.

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 has already been called — thus preventing recursive calls to `panic()`). A kernel core image is dumped.

peek(), peekc(), peekl() — Check and Read

```
short
peek(address)
    short *address;

short
peekc(address)
    char *address;

peekl(address, value)
    long *address;
    long *value;
```

`peek` and its variants are called with an address from which they read. They return –1 if the addressed location doesn't exist, otherwise they return the value that was fetched from that location. They are for use only in *xx*`probe()`. See `poke` and its variants, below.

physio() — Block I/O Service Routine

```
physio(strategy, buf, dev, rw_flag, minphys, uio)
    void  (*strategy) ();
    struct  buf  *buf;
    dev_t  dev;
    int  rw_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. *This is the only way to lock down pages of user memory.*
- 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 this:

```
loop:
        /* error and termination checking (based on values in uio) /*
        s = spl6();
        while (buf->b_flags & B_BUSY) {
                buf->b_flags |= B_WANTED;
                sleep(buf, PRIBIO+1);
        }
        (void) splx(s);
        /* set up buffer for I/O */
        while (more data) {
                buf->b_flags = B_BUSY | B_PHYS | rw_flag;
                /* more buffer I/O set up */
                (*minphys) (buf);
                /* lock down pages of user memory */
                (*strategy) (buf);
                iowait(buf);
                s = spl6();
                /* unlock buffer */
                if (buf->b_flags & B_WANTED)
                        wakeup(buf);
                (void) splx(s);
                /* bookkeeping */
        }
        buf->b_flags &= ~(B_BUSY|B_WANTED|B_PHYS);
        /* error checking and bookkeeping (based on values in uio) */
        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. *rw_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 one of the error codes defined in `errno.h` if an I/O error occurs, and a 0 upon success. Error codes are not returned on the stack, but by way of the `b_error` field in the buffer header.

`poke()`, `pokec()`, `pokel()` — Check and Write

```
poke(address, value)
    short *address;
    short value;

pokec(address, value)
    char *address;
    char value;

pokel(address, value)
    long *address;
    long value;
```

`poke` and its variants are called with an *address* to store into, and a *value* to be stored. They return 1 if the addressed location doesn't exist, and 0 if it does. They are for use only in *xx*`probe()`. See `peek` and its variants, above.

`printf()` — Kernel Printf Function

The kernel provides a `printf()` function analogous to the `printf()` function supplied with the C library for user programs. The kernel `printf()`, however, is different than 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 `mb_device.md_intpri` or `mb_ctlr.mc_intpri`, where it is put by the config 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()`. The structure element is of type `p_pid`.

`rmalloc()` — General-Purpose Resource Allocator

```
u_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`. `size` used here is in the unit of the map `mp`. For the map `kernelmap`, `size` is in pages, as you are just allocating virtual space. For the map `iopbmap`, this size is in bytes, as you are allocating virtual space tied to real physical memory.

`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 8K, and should be used only for temporary or very small buffers. The `iopbmap` is a byte-aligned table. The address it returns is not aligned on a long word boundary. If a non-aligned address is accessed, a panic may result. Callers of `rmalloc()` should ask for a few bytes of memory more than they need, and round up the address to a full word boundary if necessary. This applies to both Sun-3's and Sun-4's, but it is more critical to Sun-4's, since they can only address using full word alignment.

`rmfree()` — Recycle Map Resource

```
rmfree(mp, size, addr)
    struct map *mp;
    long size;
    u_long addr;
```

`rmfree` recycles the map resource allocated with `rmalloc`.

`selwakeup()` — Wakeup a Select-blocked Process

```
selwakeup(p, coll)
    register struct proc *p;
    int coll;
```

`selwakeup()` is called from driver interrupt routines to `wakeup()` processes which are asleep as a result of calls to `select()`. If both of its parameters are 0, it does nothing. If *coll* is 0, thus indicating that no `select()` collision occurred — that only one process is waiting for the device — `selwakeup()` just wakes up the waiting process indicated by *p*. If, however, a collision did occur, it issues a `wakeup((caddr_t)&selwait)`, thus waking all select-sleeping processes. (The `selwait` channel is used exclusively to indicate select-related sleeping). These waking processes then race for access to the device, with the first selector getting no special treatment.

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). In other areas, *address* is also referred to as *chan* for the channel that a device uses, or *event* signifying an action or state associated with a specific device. *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 of 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 SunOS `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 SunOS you always do:

```
s = spln();
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);
```

However, semaphores are not easily implemented to lockout around hardware interrupts so SunOS 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 (higher numbers indicate higher priorities). 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 will 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 SunOS 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.

splr() — Raise Priority Level

```
splr(s)
    int s;
```

`splr` called with an argument *s* that raises the priority level by *s* units.

suser() — Verify Super User

```
suser()
```

Returns a 1 if the current user is root, 0 if not. `suser()` is commonly called by `ioctl()` routines that are restricted to the superuser, and that thus need to check who's calling them.

swab() — Swap Bytes

```
swab(from, to, nbytes)
    caddr_t  from;
    caddr_t  to;
    int  nbytes;
```

`swab` swaps bytes within 16-bit 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 for Sun-3, Sun-3X, and Sun386i machines, a hundredth of a second for Sun-4s. 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 is 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, Exxx upon failure. Since this routine uses copyin() and copyout(), the amount of memory that can be moved is dependent on these routines.

For more information about the uio structure, see *Some Notes About the UIO Structure* in the *The "Skeleton" Character Device Driver* chapter 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(), uwritec() — `uio` Structure Read/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.

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().

C

User Support Routines

These routines are often useful in user-level programs that manipulate devices.

free() — Free Allocated Memory

```
free(ptr)
    char *ptr;
```

free(3) can be used to recycle the virtual memory allocated by a variety of memory allocators, including valloc(3) and malloc(3) (the most general purpose of the allocators).

getpagesize() — Return Pagesize

```
int getpagesize()
```

getpagesize(2) 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.

mmap() — Map Memory from One Space to Another

```
caddr_t
mmap(addr, len, protection, flags, fd, off)
    caddr_t addr;
    int len, protection, flags, 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 (or into the kernel address space). 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* (the device's physical installation address within *fd*'s memory), into the caller's address space beginning at *addr* and continuing for *len* bytes. (By default, mmap() will pick a good value for *addr*). The mapping established by mmap() replaces any previous mappings for the process's pages in the range [*addr*, *addr* + *len*).

fd is a file descriptor obtained by opening the character special device to be `mmap()` 'ed. *protection* specifies the read/write accessibility of the mapped pages. The values desired are expressed by or'ing the flags values `PROT_READ`, `PROT_EXECUTE`, and `PROT_WRITE`. A `write()` must fail if `PROT_WRITE` has not been set, though its behavior can be influenced by setting `MAP_PRIVATE` in the *flags* parameter.

flags provides additional information about the handling of mapped pages. Its possible values are:

`MAP_SHARED`	Share Changes
`MAP_PRIVATE`	Changes are Private
`MAP_TYPE`	Mask for Type of Mapping
`MAP_FIXED`	Interpret addr Exactly
`MAP_RENAME`	Assign Page to File

addr and *off* must be multiples of the page size (which can be found by using `getpagesize()`). Pages are automatically unmapped when *fd* is closed — they should be explicitly unmapped with `munmap()`. `mmap()` returns a -1 on error, and returns a pointer on success.

For a detailed overview of SunOS memory mapping, see the *Memory Management* chapter of the *Sun System Services Overview*. For specific details about `mmap()` and its related facilities, see `munmap()` below and the `mmap(2)`, `munmap(2)`, `mincore(2)`, `mprotect(2)`, and `msync(2)` manual pages.

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, marked invalid. If an address within an unmapped page is subsequently referenced, and if that page is in the "data segment" of a UNIX† process, then a page of zeros will be created under the address. However, if the address is outside a data segment, such a reference will cause a segmentation violation. `munmap()` returns a -1 on error, 0 on success. See `mmap()` above and the `mmap(2)` manual page for more details.

† UNIX is a registered trademark of AT&T.

D

Sample Driver Listings

The following source 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-3 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.

NOTE *This is no longer supported by Sun, but is included here for reference purposes only.*

VP is a driver for the Versatec Printer Interface. It's a fairly good example of a DMA device driver.

PP is the listing of the Sun386i Parallel Port Driver.

D.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 <sys/param.h>
#include <sys/buf.h>
#include <sys/file.h>
#include <sys/dir.h>
#include <sys/user.h>
#include <sys/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 sk_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->sk_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->sk_count > 0) {   /* still more characters */
          sk_reg->sk_data = *sk->sk_cp++;
          sk->sk_count--;
          /* stop giving characters if device not ready */
          /* Note: the softcopy isn't needed for reads */

          /* DELAY(10) might go here */

          if (!(sk_reg->sk_csr & SK_DEVREADY))   /* contact the device */
               break;
     }

     /* error-retry logic would go here */

     if (sk->sk_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->sk_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 */

    (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);
     sk->sk_busy = 0;
     wakeup((caddr_t) sk); /* free device to sleeping strategy routine */
     iodone(sk->sk_bp);    /* free buffer to waiting physio */
}

/* alternate routines which show examples of uwritec(), ureadc() usage
 * The skwrite() routine below could be used in place of the skwrite,
 * skstrategy, skstart routines
 */

skwrite (dev, uio)
dev_t dev;
struct uio *uio;
{
     struct mb_device *md;
          struct sk_reg *sk_reg;
        int c;

     md = skdinfo[SKUNIT(dev)];
     sk_reg = (struct sk_reg *)md->md_addr;
        while (uio->uio_iovcnt > 0 && uio->uio_iov->iov_len > 0) {
                if ((c = uwritec(uio)) == -1)
                        return(EFAULT);
                    sk_reg->sk_data = (char)c;
        }
        return(0);
}


skread (dev, uio)
dev_t dev;
struct uio *uio;
{
     struct mb_device *md;
          struct sk_reg *sk_reg;

     md = skdinfo[SKUNIT(dev)];
     sk_reg = (struct sk_reg *)md->md_addr;
        while (uio->uio_iovcnt > 0 && uio->uio_iov->iov_len > 0) {
                if (ureadc(sk_reg->sk_data, uio))
                        return(EFAULT);
        }
        return(0);
}
```

D.2. Sun-3 Color Graphics Driver

```
/*
 *
 * (cg2reg.h) Description of Sun-3 hardware color frame buffer.
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Structure defining the way in which the address bits to the
 * SUN-3 color frame buffer are decoded.
 */

#define CG2_WIDTH   1152
#define CG2_HEIGHT  900
#define CG2_SQUARE  1024
#define CG2_DEPTH   8

struct cg2memfb {
     union bitplane {                /* Word mode memory */
          short word[CG2_HEIGHT][CG2_WIDTH/(8*sizeof(short))];
          short sword[CG2_SQUARE][CG2_SQUARE/(8*sizeof(short))];
     } memplane[8];
     union byteplane {               /* Pixel mode memory */
          u_char pixel[CG2_HEIGHT][CG2_WIDTH];
          u_char spixel[CG2_SQUARE][CG2_SQUARE];
     } pixplane;
};

struct cg2statusreg {
     unsigned unused  : 4;           /* Reserved for future use */
     unsigned resolution : 4;        /* Screen resolution */
                                          /* 0 = 900 x 1152 */
                                          /* 1 = 1024 x 1024 */
     unsigned retrace : 1;           /* rdonly: monitor in retrace */
     unsigned inpend  : 1;           /* rdonly: interrupt pending */
     unsigned ropmode : 3;           /* Rasterop mode */
     unsigned inten   : 1;           /* Enable interrupt at end of retrace */
     unsigned update_cmap : 1;
                                     /* Copy TTL cmap to ECL cmap next vert retrace*/
                                     /* Silently disables writing to TTL cmap */
     unsigned video_enab : 1;        /* Enable video DACs */
};

struct cg2fb {
     union {                         /* ROP mode memory */
          union bitplane ropplane[8];    /* Word mode memory with ROP */
          union byteplane roppixel;      /* Pixel mode memory with ROP */
     } ropio;
     union {                         /* Rasterop unit control */
          struct memropc ropregs;    /* Normal register access */
```

```
		struct {
			char pad[2048];		/* For pixmode src reg prime */
			struct memropc ropregs;	/* Byte xfer loads alternate */
		} prime;			/* Source register bits */
		char pad[4096];
	} ropcontrol[9];
	union {				/* Status register */
		struct cg2statusreg reg;
		short word;
		char pad[4096];
	} status;
	union {				/* Per plane mask register */
		unsigned short reg;	/* 8 bits 1bit -> wr to plane*/
		char pad[4096];
	} ppmask;
	union {				/* Word pan register */
		unsigned short reg;	/* High 16 bits of 20-bit pixel address*/
					/* Pixel addr = CG2_WIDTH*y+x */
		char pad[4096];
	} wordpan;
	union {				/* Zoom and line offset register */
		struct {
		    unsigned unused  : 8;
		    unsigned lineoff : 4;/* y offset into zoomed pixel */
		    unsigned pixzoom : 4;/* Zoomed pixel size - 1 */
		} reg;
		short word;
		char pad[4096];
	} zoom;
	union {				/* Pixel pan register */
		struct {
		    unsigned unused   : 8;
		    unsigned lorigin  : 4;	/* Low 4 bits of pix addr*/
		    unsigned pixeloff : 4;	/* Zoomed pixel x offset/4 */
		} reg;
		short word;
		char pad[4096];
	} pixpan;
	union {				/* Variable zoom register */
					/* Reset zoom after line no */
		unsigned short reg;	/* Line number 0..1024/4  */
		char pad[4096];
	} varzoom;
	union {				/* Interrupt vector register */
		unsigned short reg;	/* Line number 0..1024/4  */
		char pad[4096];
	} intrptvec;
	u_short redmap[256];		/* Shadow color maps */
	u_short greenmap[256];
	u_short bluemap[256];
};

/*
```

```
 * ROPMODES -- Parallel, LD_SDT, LS_SRC, Read/Write,
 *                  on read or write?, on wrdmode or pixmode?
 */

#define   PRWWRD    0    /* parallel  8 plane,  read    write,   wrdmode */
#define   SRWPIX    1    /* single      pixel,  read    write,   pixmode */
#define   PWWWRD    2    /* parallel  8 plane,  write   write,   wrdmode */
#define   SWWPIX    3    /* single      pixel,  write   write,   pixmode */
#define   PRRWRD    4    /* parallel  8 plane,  read    read,    wrdmode */
#define   PRWPIX    5    /* parallel 16 pixel,  read    write,   pixmode */
#define   PWRWRD    6    /* parallel  8 plane,  write   read,    wrdmode */
#define   PWWPIX    7    /* parallel 16 pixel,  write   write,   pixmode */

/*
 * ROP control unit numbers
 */

#define CG2_ROP0     0    /* Rasterop unit for bit plane 0  */
#define CG2_ROP1     1    /* Rasterop unit for bit plane 1  */
#define CG2_ROP2     2
#define CG2_ROP3     3
#define CG2_ROP4     4
#define CG2_ROP5     5
#define CG2_ROP6     6
#define CG2_ROP7     7
#define CG2_ALLROP   8    /* Writes to all units enabled by PPMASK,  */
                          /* reads from plane zero  */

#define   CG_SRC          0xCC
#define   CG_DEST         0xAA
#define   CG_MASK         0xf0
#define   CG_NOTMASK      0x0f
#define   CGOP_NEEDS_MASK(op)       ( (((op)>>4)^(op)) & CG_NOTMASK)

/*
 * Defines for accessing the rasterop units
 */

#define   cg2_setrsource(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_source1 = (val))
#define   cg2_setlsource(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_source2 = (val))
#define   cg2_setfunction(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_op = (val))
#define   cg2_setpattern(fb, ropunit, val)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_pattern = (val))
#define   cg2_setshift(fb, ropunit, shft, dir)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_shift =\
           (shft)|((dir)<<8)  )
#define   cg2_setwidth(fb, ropunit, w, count)\
          ((fb)->ropcontrol[(ropunit)].ropregs.mrc_width = (w));\
           ((fb)->ropcontrol[(ropunit)].ropregs.mrc_opcount = (count))
```

```
/*
 * Defines for accessing the zoom and pan registers
 */

#define    cg2_setzoom(fb, pixsize)\
           ((fb)->zoom.reg.pixzoom = (pixsize)-1)
#define    cg2_setpanoffset(fb, xoff, yoff)\
           ((fb)->pixpan.reg.pixeloff = (xoff)>>2;\
            (fb)->zoom.reg.lineoff = (yoff)
#define    cg2_setpanorigin(fb, x, y)\
           ((y) = ((fb)->status.reg.resolution == 1) ?\
            (y)*CG2_SQUARE+(x) : (y)*CG2_WIDTH+(x);\
            (fb)->pixpan.reg.lorigin = (y)&0xf;\
            (fb)->wordpan.reg = (y)>>4)
#define    cg2_setzoomstop(fb, y) ((fb)->varzoom.reg = (y)>>2)

/*
 *   Defines that facilitate addressing the frame buffer
 */

#define    cg2_pixaddr(fb, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->pixplane.spixel[(y)][(x)] :\
            &(fb)->pixplane.pixel[(y)][(x)] )
#define    cg2_wordaddr(fb, plane, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->memplane[(plane)].sword[(y)][(x)>>4] :\
            &(fb)->memplane[(plane)].word[(y)][(x)>>4])
#define    cg2_roppixaddr(fb, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->ropio.roppixel.spixel[(y)][(x)] :\
            &(fb)->ropio.roppixel.pixel[(y)][(x)])
#define    cg2_ropwordaddr(fb, plane, x, y)\
           (((fb)->status.reg.resolution) ?\
            &(fb)->ropio.ropplane[(plane)].sword[(y)][(x)>>4]:\
            &(fb)->ropio.ropplane[(plane)].word[(y)][(x)>>4])
#define    cg2_width(fb )                                 \
           ( ((fb)->status.reg.resolution) ? CG2_SQUARE : CG2_WIDTH )
#define    cg2_height(fb )\
           ( ((fb)->status.reg.resolution) ? CG2_SQUARE : CG2_HEIGHT )
#define    cg2_linebytes(fb, mode)\
           ( ((fb)->status.reg.resolution)\
            ? ( ((mode)&1)?CG2_SQUARE:CG2_SQUARE/8 )\
            : ( ((mode)&1)?CG2_WIDTH:CG2_WIDTH/8    ))
#define    cg2_prskew(x) ((x) & 15)
#define    cg2_touch(a) ((a)=0)
```

```
/* (cg2var.h) More Sun-3 color frame buffer definitions
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Information pertaining to the Sun-3 color buffer but not to pixrects in
 * general is stored in the struct pointed to by the pr_data attribute of the
 * pixrect. One property of the color buffer not shared with all pixrects is
 * that it has a color map. The color map type and colormap contents are
 * specified by the putcolormap operation.
 */

struct cg2pr {
     struct    cg2fb *cgpr_va;
     int       cgpr_fd;
     int       cgpr_planes;          /* Color bit plane mask register */
     struct    pr_pos cgpr_offset;
};

#define cg2_d(pr) ((struct cg2pr *)(pr)->pr_data)
#define cg2_fbfrompr(pr) (((struct cg2pr *)(pr)->pr_data)->cgpr_va)
#define cg2_ropword(cgd, plane, ax, ay)\
          (cg2_ropwordaddr((cgd)->cgpr_va, (plane),\
           (cgd)->cgpr_offset.x+(ax),(cgd)->cgpr_offset.y+(ay)) )
#define cg2_pixel(cgd, ax, ay)\
          (cg2_pixaddr((cgd)->cgpr_va,\
           (cgd)->cgpr_offset.x+(ax),(cgd)->cgpr_offset.y+(ay)) )
#define cg2_roppixel(cgd, ax, ay)\
          (cg2_roppixaddr((cgd)->cgpr_va,\
           (cgd)->cgpr_offset.x+(ax),(cgd)->cgpr_offset.y+(ay)) )
#define cg2_prd_skew(cgd, ax)\
           (((cgd)->cgpr_offset.x+(ax)) & 15)

extern    struct pixrectops cg2_ops;

int       cg2_rop();
int       cg2_putcolormap();
int       cg2_putattributes();

#ifndef KERNEL
int       cg2_stencil();
int       cg2_batchrop();
struct    pixrect *cg2_make();
int       cg2_destroy();
int       cg2_get();
int       cg2_put();
int       cg2_vector();
struct    pixrect *cg2_region();
int       cg2_getcolormap();
int       cg2_getattributes();
#endif !KERNEL
```

```
/*
 * (cgtwo.c) Sun-3 (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, some 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
 * tacks, 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 <sys/param.h>          /* general kernel parameters */
#include <sys/buf.h>            /* I/O buffers */
#include <sys/errno.h>          /* system error reporting */
#include <sys/ioctl.h>          /* ioctl definitions */
#include <sys/map.h>            /* resource allocation maps */
#include <sys/vmmac.h>          /* virtual memory related conversion macros */

/* <machine> is a symbolic link to sun[234] */
#include <machine/pte.h>        /* page table entries */
#include <machine/mmu.h>        /* memory-management unit */
#include <machine/psl.h>        /* process status register */

#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_impl_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 */
```

```
/* probe size in bytes -- enough for the useful part of the board */
#define CG2_PROBESIZE CG2_MAPPED_SIZE

/* Mainbus device data */
int cgtwoprobe(), cgtwoattach();

struct mb_device *cgtwoinfo[NCGTWO];
struct mb_driver cgtwodriver = {
     cgtwoprobe, 0, cgtwoattach, 0, 0, 0,
     CG2_PROBESIZE, "cgtwo", cgtwoinfo, 0, 0, 0, 0
};

/* Driver per-unit data */
struct cg2_softc {
     int flags;          /* misc. flags; bits defined in cg2var.h */
                         /* (struct cg2pr, flags member) */
     struct cg2fb *fb;     /* virtual address */
     int w, h;           /* resolution */
#if NWIN > 0
     Pixrect pr;            /* kernel pixrect and private data */
     struct cg2pr prd;
#endif NWIN > 0
} cg2_softc[NCGTWO];

/* default structure for FBIOGATTR/FBIOGTYPE ioctls */
static struct fbgattr fbgattr_default =  {
/* real_type owner */
     FBTYPE_SUN2COLOR, 0,
/* fbtype: type h w depth cms size */
     { FBTYPE_SUN2COLOR, 0, 0, 8,    256,  CG2_MAPPED_SIZE },
/* fbsattr: flags emu_type */
     { FB_ATTR_DEVSPECIFIC, -1,
/* dev_specific: FLAGS, BUFFERS, PRFLAGS */
     { FB_ATTR_CG2_FLAGS_PRFLAGS, 1, 0 } },
/* emu_types */
     { -1, -1, -1, -1}
};

/* Double buffering enable flag */
int cg2_dblbuf_enable = 1;

#if NWIN > 0

/* SunWindows specific stuff */

/* kernel pixrect ops vector */
static struct pixrectops pr_ops = {
     cg2_rop,
     cg2_putcolormap,
     cg2_putattributes
};
#endif NWIN > 0
```

```
cgtwoprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     register struct cg2fb *fb = (struct cg2fb *) reg;
     register struct cg2_softc *softc;

     /*
      * Check if board is present and strapped for 2M decoding.
      * If this fails, remap for 4M decoding and try again.
      */
     if (probeit(fb)) {
          fbmapin((caddr_t) fb, fbgetpage((caddr_t) fb) +
               (int) btop(CG2_MAPPED_OFFSET), CG2_MAPPED_SIZE);

          if (probeit(fb))
               return 0;
     }

     softc = &cg2_softc[unit];
     softc->fb = fb;
     softc->flags = 0;

     /*  check for supported resolution  */
     switch (fb->status.reg.resolution) {
     case CG2_SCR_1152X900:
          softc->w = 1152;
          softc->h =  900;
          softc->flags = CG2D_STDRES;
          break;
     case CG2_SCR_1024X1024:
          softc->w = 1024;
          softc->h = 1024;
          break;
     default:
          printf("%s%d: unsupported resolution (%d)0,
               cgtwodriver.mdr_cname, unit,
               fb->status.reg.resolution);
          return 0;
     }

     return CG2_PROBESIZE;
}

static
probeit(fb)
     register struct cg2fb *fb;
{
     union {
          struct cg2statusreg reg;
          short word;
     } status;
```

```
#define    allrop(fb, reg)     ((short *) &(fb)->ropcontrol[CG2_ALLROP].ropregs.reg)
#define    pixel0(fb)     ((char *) &fb->ropio.roppixel.pixel[0][0])

     /*
      * Probe sequence:
      *
      * set board for pixel mode access
      * enable all planes
      * set rasterop function to CG_SRC
      * disable end masks
      * set fifo shift/direction to zero/left-to-right
      * write 0xa5 to pixel at (0,0)
      * check pixel value
      * enable subset of planes (0xcc)
      * set rasterop function to ~CG_DEST
      * write to pixel at (0,0) again
      * enable all planes again
      * read pixel value; should be 0xa5 ^ 0xcc = 0x69
      */
     status.word = peek(&fb->status.word);
     status.reg.ropmode = SWWPIX;
     if (poke(&fb->status.word, status.word) ||
          poke((short *) &fb->ppmask.reg, 255) ||
          poke(allrop(fb, mrc_op), CG_SRC) ||
          poke(allrop(fb, mrc_mask1), 0) ||
          poke(allrop(fb, mrc_mask2), 0) ||
          poke(allrop(fb, mrc_shift), 1 << 8) ||
          pokec(pixel0(fb), 0xa5) ||
          pokec(pixel0(fb), 0) ||
          peekc(pixel0(fb)) != 0xa5 ||
          poke((short *) &fb->ppmask.reg, 0xcc) ||
          poke(allrop(fb, mrc_op), ~CG_DEST) ||
          pokec(pixel0(fb), 0) ||
          poke((short *) &fb->ppmask.reg, 255) ||
          peekc(pixel0(fb)) != (0xa5 ^ 0xcc))
          return 1;

     return 0;

#undef     allrop
#undef     pixel0
}


cgtwoattach(md)
     struct mb_device *md;
{
     register struct cg2_softc *softc = &cg2_softc[md->md_unit];
     register struct cg2fb *fb = softc->fb;
     register int flags = softc->flags;

#define    dummy     flags
```

```
/* set interrupt vector */
if (md->md_intr)
     fb->intrptvec.reg = md->md_intr->v_vec;
else
     printf("WARNING: no interrupt vector specified in config file0);

/*
 * Determine whether this is a Sun-2 or Sun-3 color board
 * by setting the wait bit in the double buffering register
 * and seeing if it clears itself during retrace.
 *
 * On the Sun-2 color board this just writes a bit in the
 * "wordpan" register.
 */
fb->misc.nozoom.dblbuf.word = 0;
fb->misc.nozoom.dblbuf.reg.wait = 1;

/* wait for leading edge, then trailing edge of retrace */
while (fb->status.reg.retrace)
      /* nothing */ ;
while (!fb->status.reg.retrace)
      /* nothing */ ;
while (fb->status.reg.retrace)
      /* nothing */ ;

if (fb->misc.nozoom.dblbuf.reg.wait) {
      /* Sun-2 color board */
      fb->misc.nozoom.dblbuf.reg.wait = 0;
      flags |= CG2D_ZOOM;
}
else {
      /* Sun-3 color board (or better) */
      flags |= CG2D_32BIT | CG2D_NOZOOM;

      if (fb->status.reg.fastread)
            flags |= CG2D_FASTREAD;

      if (fb->status.reg.id)
            flags |= CG2D_ID | CG2D_ROPMODE;

      /*
       * Probe for double buffering feature.
       * Write distinctive values to one pixel in both buffers,
       * then two pixels in buffer B only.
       * Read from buffer B and see what we get.
       *
       * Warning: assumes we were called right after cgtwoprobe
       */
      cg2_setfunction(fb, CG2_ALLROP, CG_SRC);
      fb->ropio.roppixel.pixel[0][0] = 0x5a;
      fb->ropio.roppixel.pixel[0][0] = 0xa5;
      fb->misc.nozoom.dblbuf.reg.nowrite_a = 1;
      fb->ropio.roppixel.pixel[0][0] = 0xc3;
```

```
          fb->ropio.roppixel.pixel[0][4] = dummy;
          if (fb->ropio.roppixel.pixel[0][0] == 0x5a) {
               fb->misc.nozoom.dblbuf.reg.read_b = 1;

               if (fb->ropio.roppixel.pixel[0][0] == 0xa5 &&
                    fb->ropio.roppixel.pixel[0][4] == 0xc3 &&
                    cg2_dblbuf_enable)
                    flags |= CG2D_DBLBUF;
          }
          fb->misc.nozoom.dblbuf.word = 0;
     }

     softc->flags = flags;

#ifndef sun2
     /* re-map into correct VME space if necessary */
     {
          int page = fbgetpage((caddr_t) fb);

          if (((flags & CG2D_32BIT) != 0) !=
               ((page & PGT_MASK) == PGT_VME_D32))
               fbmapin((caddr_t) fb,
                    page ^ (PGT_VME_D16 ^ PGT_VME_D32),
                    CG2_MAPPED_SIZE);
     }
#endif !sun2

     /* print informative message */
     printf("%s%d: Sun-%c color board%s%s0,
          md->md_driver->mdr_dname, md->md_unit,
          flags & CG2D_ZOOM ? '2' : '3',
          flags & CG2D_DBLBUF ? ", double buffered" : "",
          flags & CG2D_FASTREAD ? ", fast read" : "");
}

cgtwoopen(dev, flag)
     dev_t dev;
     int flag;
{
     return fbopen(dev, flag, NCGTWO, cgtwoinfo);
}

/*ARGSUSED*/
cgtwoclose(dev, flag)
     dev_t dev;
{
     register struct cg2_softc *softc = &cg2_softc[minor(dev)];
     register struct cg2fb *fb = softc->fb;

     /* fix up zoom and/or double buffering on close */

     if (softc->flags & CG2D_ZOOM) {
          fb->misc.zoom.wordpan.reg = 0;      /* hi pixel adr = 0 */
          fb->misc.zoom.zoom.word = 0;  /* zoom=0, yoff=0 */
          fb->misc.zoom.pixpan.word = 0;     /* pix adr=0, xoff=0 */
          fb->misc.zoom.varzoom.reg = 255; /* unzoom at line 4*255 */
     }

     if (softc->flags & CG2D_NOZOOM)
          fb->misc.nozoom.dblbuf.word = 0;

     return 0;
}

cgtwommap(dev, off, prot)
     dev_t dev;
     off_t off;
     int prot;
{
     return fbmmap(dev, off - CG2_MAPPED_OFFSET,
          prot, NCGTWO, cgtwoinfo, CG2_MAPPED_SIZE);
}

/*ARGSUSED*/
cgtwoioctl(dev, cmd, data, flag)
     dev_t dev;
     int cmd;
     caddr_t data;
     int flag;
{
     register struct cg2_softc *softc = &cg2_softc[minor(dev)];

     switch (cmd) {

     case FBIOGTYPE: {
          register struct fbtype *fbtype = (struct fbtype *) data;

          *fbtype = fbgattr_default.fbtype;
          fbtype->fb_height = softc->h;
          fbtype->fb_width  = softc->w;
     }
     break;

     case FBIOGATTR: {
          register struct fbgattr *gattr = (struct fbgattr *) data;

          *gattr = fbgattr_default;
          gattr->fbtype.fb_height = softc->h;
          gattr->fbtype.fb_width = softc->w;

          if (softc->flags & CG2D_NOZOOM)
               gattr->sattr.dev_specific[FB_ATTR_CG2_FLAGS] |=
                    FB_ATTR_CG2_FLAGS_SUN3;

          if (softc->flags & CG2D_DBLBUF)
               gattr->sattr.dev_specific[FB_ATTR_CG2_BUFFERS] = 2;
          gattr->sattr.dev_specific[FB_ATTR_CG2_PRFLAGS] = softc->flags;
     }
     break;

     case FBIOSATTR:
          break;

#if NWIN > 0

     case FBIOGPIXRECT:
          ((struct fbpixrect *) data)->fbpr_pixrect = &softc->pr;

          /* initialize pixrect */
          softc->pr.pr_ops = &pr_ops;
          softc->pr.pr_size.x = softc->w;
          softc->pr.pr_size.y = softc->h;
          softc->pr.pr_depth = CG2_DEPTH;
          softc->pr.pr_data = (caddr_t) &softc->prd;

          /* initialize private data */
          bzero((char *) &softc->prd, sizeof softc->prd);
          softc->prd.cgpr_va = softc->fb;
          softc->prd.cgpr_fd = 0;
          softc->prd.cgpr_planes = 255;
          softc->prd.ioctl_fd = minor(dev);
          softc->prd.flags = softc->flags;
          softc->prd.linebytes = softc->w;

          /* enable video */
          softc->fb->status.reg.video_enab = 1;

          break;

#endif NWIN > 0

     /* get info for GP */
     case FBIOGINFO: {
          register struct fbinfo *fbinfo = (struct fbinfo *) data;

          fbinfo->fb_physaddr =
               (fbgetpage((caddr_t) softc->fb) << PGSHIFT) -
               CG2_MAPPED_OFFSET & 0xffffff;
          fbinfo->fb_hwwidth = softc->w;
          fbinfo->fb_hwheight = softc->h;
          fbinfo->fb_ropaddr = (u_char *) softc->fb;
     }
     break;

     /* set video flags */
     case FBIOSVIDEO:
          softc->fb->status.reg.video_enab =
               (* (int *) data) & FBVIDEO_ON ? 1 : 0;
          break;
```

```
    /* get video flags */
    case FBIOGVIDEO:
        * (int *) data = softc->fb->status.reg.video_enab
            ? FBVIDEO_ON : FBVIDEO_OFF;
        break;

    case FBIOVERTICAL:
        cgtwo_wait(minor(dev));
        break;

    default:
        return ENOTTY;
    }

    return 0;
}

/* wait for vertical retrace interrupt */
cgtwo_wait(unit)
    int unit;
{
    register struct mb_device *md = cgtwoinfo[unit & 255];
    register struct cg2_softc *softc = &cg2_softc[unit & 255];
    int s;

    if (md->md_intr == 0)
        return;

    s = splx(pritospl(md->md_intpri));
    softc->fb->status.reg.inten = 1;
    (void) sleep((caddr_t) softc, PZERO - 1);
    (void) splx(s);
}

/* vertical retrace interrupt service routine */
cgtwointr(unit)
    int unit;
{
    register struct cg2_softc *softc = &cg2_softc[unit];

    softc->fb->status.reg.inten = 0;
    wakeup((caddr_t) softc);

#ifdef lint
    cgtwointr(unit);
#endif
}
```

```
/*
 * (fbutil.c) Frame Buffer Driver Support Utilities
 * Copyright (c) 1985, 1987 by Sun Microsystems, Inc.
 */

/*
 * The routines in this file, called from many the Sun frame buffer drivers,
 * perform the essential operations necessary for all memory-mapped drivers.
 */

#include <sys/param.h>          /* machine dependent kernel parameters */
#include <sys/buf.h>            /* I/O buffers */
#include <sys/errno.h           /* System error reporting */
#include <sys/mman.h>           /* Memory-mapping definitions */
#include <sys/vmmac.h>          /* Virtual memory related conversion macros */

/* <machine> is a symbolic link set to sun[234] */
#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 then returns. Everything is OK if the
 * device is 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 struct mb_device *md;

     if (minor(dev) >= numdevs ||
          (md = mb_devs[minor(dev)]) == 0 ||
         md->md_alive == 0)
          return ENXIO;

     return 0;
}

/*
 * Work from the device address and an offset within its address
 * space to get the page frame number for the page to be mapped.
 */
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;
```

```
{
     if ((u_int) off >= size)
          return -1;

     return fbgetpage(mb_devs[minor(dev)]->md_addr + off);

}

/* Get page frame number and page type */
fbgetpage(addr)
     caddr_t addr;
{
     return (int) hat_getkpfnum((addr_t) addr);
}

/*
 * Simplified mapin and mapout. Note that, since these
 * routines are implemented in terms of Usrptmap (which has been
 * preserved for compatibility reasons) they will work with either SunOS
 * release 4.0 or with earlier releases.
fbmapin(virt, phys, size)
     caddr_t virt;
     int phys;
     int size;
{
     mapin(&Usrptmap[btokmx((struct pte *) virt)], btop(virt),
          (u_int) phys, btoc(size), PG_V | PG_KW);
}

fbmapout(virt, size)
     caddr_t virt;
     int size;
{
     mapout(&Usrptmap[btokmx((struct pte *) virt)], btoc(size));
}

#ifdef sun2
/*
 * Some Sun-2 frame-buffer devices allowed the user to enable/disable interrupts, and
 * even to change the interrupt level. Thus, fbintr is necessary so that the
 * kernel will always be able to find the interrupting device. If fbintr finds
 * an interrupting device, it returns with a 1 after calling intclear to turn
 * off its interrupt.
 */
fbintr(numdevs, mb_devs, intclear)
     int numdevs;
     register struct mb_device **mb_devs;
     int (*intclear)();
{
     register struct mb_device *md;

     while (--numdevs >= 0)
```

```
          if ((md = *mb_devs++) &&
               md->md_alive &&
               (*intclear)(md->md_addr))
               return 1;

     return 0;
}
#endif sun2
```

D.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 <sys/param.h>          /* general kernel parameters */
#include <sys/buf.h>            /* I/O buffers */
#include <sys/file.h>           /* open file information */
#include <sys/dir.h>            /* file system directories */
#include <sys/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, 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)		/* use user page */
		off -= SKYPGSIZE;

	return (hat_getkpfnum((addr_t) 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_dlreg;
        else
             s->sky_dlreg = 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_dlreg;
             else
                  s->sky_dlreg = 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;
}
```

D.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      /* form feed 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 <sys/param.h>       /* general kernel parameters */
#include <sys/dir.h>         /* file system directories */
#include <sys/user.h>        /* kernel per-process status */
#include <sys/buf.h>         /* I/O buffers */
#include <sys/systm.h>       /* miscellaneous kernel variables */
#include <sys/kernel.h>      /* kernel global variables */
#include <sys/map.h>         /* resource allocation maps */
#include <sys/ioctl.h>       /* ioctl definitions */
#include <sys/vcmd.h>        /* for all Versatec interface drivers */
#include <sys/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);
}
```

D.5. Sun386i Parallel Port Driver

```
/*
 * (ppreg.h) Sun-386i Parallel Port Registers
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

/* Register addresses.
 */

ushort ppregs[][NPPREGS] = {
     { 0x378, 0x37a, 0x379 }, /* port 1 regs */
};

/* Printer Control Reg bits */
#define    PC_INTENABLE    0x10  /* +IRQ ENABLE: enable ACK interrupts */
#define    PC_SELECT       0x08  /* +SLCT IN: select printer */
#define    PC_INIT         0x04  /* -INIT: init printer */
#define    PC_LINEFEED     0x02  /* +AUTO FD XT: set auto linefeed */
#define    PC_STROBE       0x01  /* +STROBE: strobe data */

#define    PC_NORM         (PC_INTENABLE|PC_SELECT|PC_INIT)
#define    PC_OFF          (PC_SELECT|PC_INIT)
#define    PC_RESET        0

/* Printer Status Reg bits */
#define    PS_READY        0x80  /* -BUSY: printer not busy */
#define    PS_NOTACK       0x40  /* -ACK: ACK state */
#define    PS_NOPAPER      0x20  /* +PE: printer out of paper */
#define    PS_SELECT       0x10  /* +SLCT: printer is selected */
#define    PS_NOERROR      0x08  /* -ERROR: printer error condition */

#define    PSREADY(s)      ((s)&PS_READY)
#define    PSSELECT(s)     ((s)&PS_SELECT)
#define    PSNOPAPER(s)    ((s)&PS_NOPAPER)
#define    PSERROR(s)      (((s)&PS_NOERROR) == 0)
```

```
/*
 * Parallel Port (printer) driver.
 * Copyright (c) 1987 by Sun Microsystems, Inc.
 */

#include "pp.h"
#if NPP > 0

#include <sys/param.h>
#include <sys/buf.h>
#include <sys/uio.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sundev/mbvar.h>

/*
 * Buffers for use by physio().
 */
struct buf ppbuf[NPP];
#define    PPBUFSIZ  64     /* Size of buffer written to printer */

/*
 * Software state structure, one for each printer
 */
struct ppstate {
  int      pp_flags;                /* Printer state: */
#define    PP_OPEN    0x01          /*     Currently open */
#define    PP_WANT    0x02          /*     Someone waiting for printer */
#define    PP_TIMER   0x08          /*     Watchdog timer is running */
#define    PP_BUSY    0x10          /*     I/O in progress */
  u_char   pp_timer;                /* For detecting timeout situations*/
  u_char   pp_lostintr;             /* For tracking lost interrupts*/
  u_char   pp_notready;             /* Printer not ready (no paper, etc.) */
  int      pp_unit;                 /* Unit number*/
  struct   mb_device *pp_md;        /* Pointer to mb info */
  struct   buf   *pp_bp;            /* Pointer to current 'buf' */
  char     pp_buf[PPBUFSIZ];        /* Buffer */
  char     *pp_cp;                  /* Current byte in current buffer */
  int      pp_count;                /* Number of bytes left to print */
  u_short pp_regbase;               /* Device register base in i/o space */
} ppstate[NPP];

#define    PPREG_DATA     (pp->pp_regbase)
#define    PPREG_CTRL     (pp->pp_regbase + 2)
#define    PPREG_STAT     (pp->pp_regbase + 1)


#define    PPUNIT(dev)    (minor(dev))
#define    PPPRI          (PZERO + 1)     /* Sleeps are interruptable */

extern int hz;
#define    PPWATCHDOG      3       /* Watchdog interval: see 'pptimeout()' */
#define    PPTICKS         (30/PPWATCHDOG + 1)
```

```
#define   PPMSGTICKS     (180/PPWATCHDOG)

#ifdef DEBUG
/*
 * Debugging stuff.
 */
#define DBINIT       0x0001
#define DBIO         0x0002
#define DBOPEN       0x0004
#define DBCLOSE      0x0008
#define DBSTRAT      0x0010
#define DBSTART      0x0020
#define DBTMOUT      0x0040
#define DBINTR       0x0080

int ppdebug = 0xffff;
#define ppprint(flg,x)   (((flg)&ppdebug) ? printf x : 0)

#else
#define ppprint(flg,x)
#endif DEBUG

int ppprobe(), ppattach(), ppintr(), pptimeout();

struct mb_driver ppdriver = {
     ppprobe, 0, ppattach, 0, 0, ppintr, 0, "pp", 0, 0, 0, 0,
};

/*ARGSUSED*/
ppprobe(reg, unit)
     caddr_t reg;
     int unit;
{
     ppprint(DBINIT, ("ppprobe\n"));

     if (unit >= NPP)
          panic("pp: too many units");

     ppstate[unit].pp_regbase = (u_short)reg;
     return(1);
}

ppattach(md)
     register struct mb_device *md;
{
     register struct ppstate *pp;

     ppprint(DBINIT, ("ppattach\n"));

     pp = &ppstate[md->md_unit];
     pp->pp_md = md;

     /* Initialize printer.
     * Holding PC_INIT low for 50 usecs does the trick.
     */
     outb(PPREG_CTRL, PC_RESET);
     DELAY(50);
     outb(PPREG_CTRL, PC_OFF);
     DELAY(10);
}

ppopen(dev, flags)
     dev_t dev;
     int flags;
{
     register struct ppstate *pp = &ppstate[PPUNIT(dev)];
     int status;

     ppprint(DBOPEN, ("ppopen: unit %d\n", PPUNIT(dev)));

     if (PPUNIT(dev) >= NPP  ||  pp->pp_md->md_alive == 0)
          return(ENXIO);
     if (flags & FREAD)         /* Can't read a write-only device */
          return(ENODEV);

     pp->pp_unit = PPUNIT(dev);

     while (pp->pp_flags & PP_OPEN) {    /* Enforce exclusive access */
          ppprint(DBOPEN, ("ppopen: in use - waiting...\n"));
          if (flags & FNDELAY)
               return(EBUSY);
          pp->pp_flags |= PP_WANT;
          if (sleep((caddr_t)&pp->pp_flags, PPPRI|PCATCH)) {
               return(EINTR);
          }
     }

     status = inb(PPREG_STAT);
     if (PSNOPAPER(status)  ||  ! PSSELECT(status)  ||  PSERROR(status)) {
          if (PSNOPAPER(status))
               uprintf("pp%d: printer out of paper\n", pp->pp_unit);
          else
               uprintf("pp%d: printer not ready\n", pp->pp_unit);
          (void)wakeup((caddr_t)&pp->pp_flags);
          pp->pp_flags = 0;
          return(EIO);
     }

     outb(PPREG_CTRL, PC_NORM);     /* Enable interrupts */

     if ((pp->pp_flags & PP_TIMER)  ==  0) {
          /*
           * Kick off watchdog timer.
           */
          timeout(pptimeout, (caddr_t)pp, PPWATCHDOG*hz);
          pp->pp_timer = 0;
```

```
            pp->pp_flags |= PP_TIMER;
        }

        pp->pp_flags |= PP_OPEN;
        return(0);
}

/*
 * ppclose:
 *      Close the printer device.
 *      Turn off interrupts.
 *      Wake up anyone waiting to open the printer.
 */
ppclose(dev)
        dev_t dev;
{
        register struct ppstate *pp = &ppstate[PPUNIT(dev)];

        ppprint(DBCLOSE, ("ppclose: unit %d\n", PPUNIT(dev)));

        outb(PPREG_CTRL, PC_OFF);       /* Disable interrupts */

        if (pp->pp_flags & PP_WANT)
            wakeup((caddr_t)&pp->pp_flags);
        pp->pp_flags = 0;
}


ppwrite(dev, uio)
        dev_t dev;
        struct uio *uio;
{
        int ppminphys(), ppstrategy();

        ppprint(DBIO, ("ppwrite\n"));

        return(physio(ppstrategy, &ppbuf[PPUNIT(dev)], dev, B_WRITE,
            ppminphys, uio));
}

/*
 * ppstrategy:
 */
ppstrategy(bp)
        register struct buf *bp;
{
        register struct ppstate *pp = &ppstate[PPUNIT(bp->b_dev)];

        ppprint(DBSTRAT|DBIO, ("ppstrategy\n"));

        pp->pp_bp = bp;
        pp->pp_count = bp->b_bcount;
        pp->pp_cp = pp->pp_buf;
```

```
     if (copyin(bp->b_un.b_addr, pp->pp_buf, bp->b_bcount)) {
          bp->b_flags |= B_ERROR;
          bp->b_error = EFAULT;
          ppiodone(pp);
          return;
     }

     pp->pp_flags |= PP_BUSY;
     pp->pp_timer = PPTICKS;          /* Set timer */
     pp->pp_lostintr = 0;             /* Reset "lost interrupt" counter */
     pp->pp_notready = 0;             /* Reset "notready" counter */
     ppintr();
     ppiowait(pp, bp);
     pp->pp_timer = 0;           /* Turn off timer */

     ppprint(DBSTRAT, ("ppstrategy: ***done\n"));
}

ppminphys(bp)
     register struct buf *bp;
{
     if (bp->b_bcount > PPBUFSIZ)
          bp->b_bcount = PPBUFSIZ;
}

/*
 * ppintr:
 *     Handle 'ack' interrupts from printer.
 */
ppintr()
{
     register struct ppstate *pp;
     int status;     /* printer status */
     int s;

     ppprint(DBINTR, ("ppintr\n"));

     pp = &ppstate[0];           /* XXX - only works for unit #0 */

     s = splx(pritospl(pp->pp_md->md_intpri));

     status = inb(PPREG_STAT);
     ppprint(DBINTR, ("ppintr: status = 0x%x\n", status));

     /* Were we expecting an interrupt? */
     if ( ! (pp->pp_flags & PP_BUSY)) {
          ppprint(DBINTR, ("ppintr: unsolicited interrupt\n"));
          splx(s);
          return;
     }


     if (pp->pp_count > 0) {
          /* AT Tech Ref says data must be in data reg at least
           * 0.5 usec before and after we strobe, and strobe must
           * last at least 0.5 usec.
           */
          outb(PPREG_DATA, *pp->pp_cp);
          pp->pp_cp++;
          pp->pp_count--;
          DELAY(1);
          outb(PPREG_CTRL, PC_NORM|PC_STROBE);
          DELAY(1);
          outb(PPREG_CTRL, PC_NORM);
     }

     else
          ppiodone(pp);

     splx(s);
}

/*
 * pptimeout:
 *      Check occasionally for lost interrupts or
 *      printer errors (no paper, printer off line, etc.).
 */
pptimeout(arg)
     caddr_t arg;
{
     register struct ppstate *pp = (struct ppstate *)arg;
     int  status;    /* Printer status */
     int  error = 0;
     int  s;

     ppprint(DBTMOUT, ("pptimeout\n"));

     s = splx(pritospl(pp->pp_md->md_intpri));


     /* If we're not currently doing anything, we can go away. */
     if ((pp->pp_flags & PP_OPEN)  ==  0) {   /* Not open */
          splx(s);
          return;
     } else if (pp->pp_timer <= 0) {          /* Not currently active */
          timeout(pptimeout, (caddr_t)pp, PPWATCHDOG*hz);
          splx(s);
          return;
     }

     status = inb(PPREG_STAT);

     /* Check for printer errors. */
     if (PSNOPAPER(status)) {
          if ((pp->pp_notready++ % PPMSGTICKS)  ==  0)
               uprintf("pp%d: printer out of paper\n", pp->pp_unit);
```

```
    } else if ( ! PSSELECT(status)  ||  PSERROR(status)) {
        if ((pp->pp_notready++ % PPMSGTICKS)  ==  0)
            uprintf("pp%d: printer not ready\n", pp->pp_unit);
    } else if (--pp->pp_timer == 0) {
        /* Timer has expired - see what's wrong. */
        ppprint(DBTMOUT, ("pptimeout: status = 0x%x\n", status));

        if (PSREADY(status)) {
            /*
             * We must have dropped an interrupt.
             * If this is the first one we've dropped, assume
             * it's a fluke and carry on.  Otherwise, give up.
             */
            if (pp->pp_lostintr++ == 0) {
                ppprint(DBTMOUT, ("pptimeout: dropped intr\n"));
                pp->pp_timer = PPTICKS;   /* Reset timer */
                ppintr();
            } else {
                printf("pp%d: not getting interrupts\n",
                    pp->pp_unit);
                error = 1;
            }
        } else {
            /* Printer is hung */
            error = 1;
        }
    }

    if ( ! error) {
        timeout(pptimeout, (caddr_t)pp, PPWATCHDOG*hz);
    } else {
        pp->pp_bp->b_flags |= B_ERROR;
        ppiodone(pp);
        pp->pp_flags &= ~PP_TIMER;
    }

    splx(s);
}


/*ARGSUSED*/
ppioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
{
    return(ENOTTY);
}

/*
 * ppiowait:
 *    Private version of 'biowait()'.
 */
ppiowait(pp, bp)
     struct ppstate *pp;
     register struct buf *bp;
{
     int s;

     s = splx(pritospl(pp->pp_md->md_intpri));
     while ( ! (bp->b_flags&B_DONE)) {
          if (sleep((caddr_t)bp, PPPRI|PCATCH)) {
               bp->b_flags |= (B_ERROR|B_DONE);
               bp->b_error = EINTR;
          }
     }

     splx(s);
}

/*
 * ppiodone:
 *     Private version of 'biodone()'.
 */
ppiodone(pp)
     register struct ppstate *pp;
{
     register struct buf *bp = pp->pp_bp;

     bp->b_flags |= B_DONE;
     wakeup((caddr_t)bp);

     pp->pp_flags &= ~PP_BUSY;
}

#endif NPP
```

Index