Chapter 3

Writing a VME Device Driver

This chapter provides in-depth information about drivers that interface to the VME bus. It gives a brief overview of the VME-bus interface, describes system configuration for VME device drivers, and introduces several VME-specific routines you must include in your device driver. Which of the several models for performing DMA *(direct memory access)* operations you choose for your device driver depends on the capability of the device (whether the device has scatter/gather registers, for example), the address space of the device (VME A24, A32, or A64), and whether the device provides address mapping capability.

It contains the following sections:

VME-bus Interface Overview

All high-end Silicon Graphics systems—Crimson, CHALLENGE/Onyx, POWER CHALLENGE/POWER Onyx, and the POWER series— support the VME bus with a VME-bus adapter. Old mid-range systems—IRIS 4D/20, 4D/25, 4D/30, and 4D/35—also supported VME. Silicon Graphics desktop systems—Indigo, Indigo^2, and Indy—do not currently support the VME bus.

The VME bus is an industry-standard bus for interfacing devices. It supports the following features:

- Seven levels of prioritized processor interrupts
- 16-bit, 24-bit, and 32-bit data addresses and 64-bit memory addresses
- 16-bit and 32-bit accesses (and 64-bit accesses in MIPS III mode)
- 8-bit, 16-bit, 32-bit, and 64-bit data transfer
- DMA to/from main memory

The VME bus does not distinguish between I/O and memory space, and it supports multiple address spaces. This feature allows you to put 16-bit devices in the 16-bit space, 24-bit devices in the 24-bit space, 32-bit devices in the 32-bit space, and 64-bit devices in 64-bit space.[1] So you must know which of the four address spaces the board uses when you design a VME device driver.

Note: On some devices, you can use jumpers or switch settings to configure the device to use a particular address space. Some systems have DMA-mapping registers to make memory appear contiguous to the VME card.

For additional information on VME-bus operation, see the ANSI standards specification for the VME bus.

[1] 64-bit data transfers, accesses, and memory addresses do not depend on a 64-bit kernel, so they can be mapped to all MIPS R4000 series platforms.

VME-bus Adapter

The term VME-bus adapter (see Figure 3-1) refers to a hardware *conduit* that translates host CPU operations to VME-bus operations and decodes some VME-bus operations (as though the conduit were a memory board) to translate them to the host side.

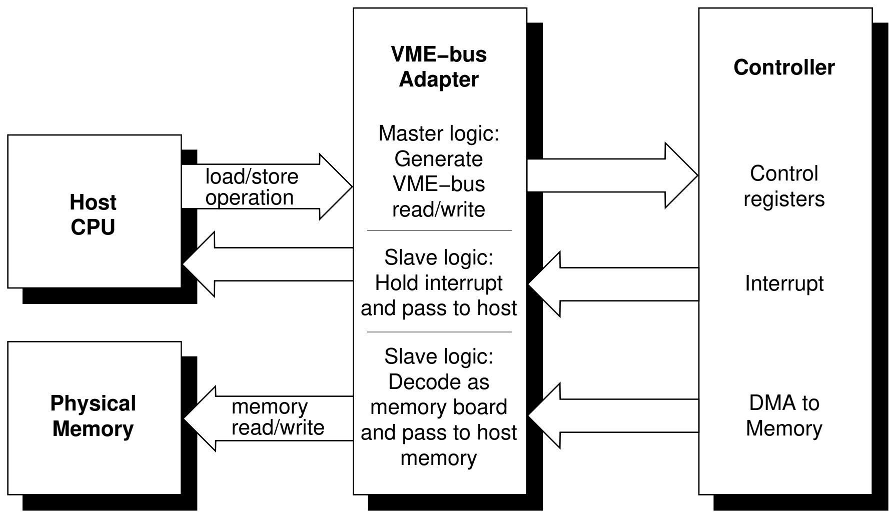

Figure 3-1 VME-bus Adapter

VME-bus Address Space

The VME bus provides 32 address bits and six address-modifier bits. It supports four address sizes: 16-bit, 24-bit, 32-bit, and 64-bits (A16, A24, A32, and, on CHALLENGE/Onyx and POWER CHALLENGE/POWER Onyx series systems, A64). The VME bus allows the master to broadcast addresses at any of these sizes. The VME bus supports data transfer sizes of 8, 16, 32, or 64 bits. To best understand the VME-bus addressing and address space, think of the device as consisting of two halves: the *master* and the *slave*. When the CPU accesses the address space of the device, the device acts as a VME slave. When the VME device accesses main memory through direct memory access (DMA) operations, the VME device acts as a VME master.

Addressing behavior for a driver depends on whether the CPU or the device is the master. For example, a VME device can be a 16-bit slave and a 32-bit master. Silicon Graphics systems support 16-, 24-, and 32-bit slaves, but only 24- and 32-bit masters.

Some Silicon Graphics systems provide additional hardware mapping registers that map a VME-bus address to an arbitrary location in physical memory. Device drivers can take advantage of this mapping hardware to provide scatter/gather capabilities (and to support DMA operations to all of memory for A24 devices). The IRIX operating system provides a procedural interface by which your device driver can allocate and use these maps. This interface also has a provision to handle multiple VME-bus systems.

For other systems, 24-bit VME masters can access only the lowest 8 MB of physical memory,[1] so device drivers may need to allocate buffers in low memory and then copy data to its final destination. See */usr/include/sys/vmereg.h* for macro #devices to facilitate VME access.

VME-bus Read-Modify-Write Cycle

The VME bus provides a read-modify-write (or RMW) cycle that allows users to read and change the contents of a device register or memory location in a single atomic operation. Although this feature is typically used to implement synchronization primitives on VME memory, you may occasionally find this feature useful for certain devices. The VME-bus adapter provides access to VME read-modify-write cycles through a set of kernel functions, such as **pio_andh_rmw**() and **pio_orw_rmw**().

Caution: The VME RMW cycle is needed *only* when a controller allows both itself and a user to write a register. If a disk controller uses a single register for the status and command information for several disk drives, for example, you could be writing a command into the register from the driver while the disk controller is updating the status. The VME RMW cycle enforces exclusive use of an address. Since this operation is expensive, in terms of resources, it should rarely be used.

[1] The highest bit is used to distinguish between user and supervisor access.

Note: Silicon Graphics products do not support VME read-modify-write operations initiated by a VME master to host memory.

VME-bus Adapter Requests

The VME-bus adapter provides four levels of bus request, 0-3, (3 has the highest priority) for DMA arbitration. Do not confuse these *bus request levels* with the interrupt priorities described below. Bus requests prioritize the use of the physical lines representing the bus and are normally set by means of jumpers on the interface board. The IRIS-4D/20, 4D/25, 4D/30, and 4D/35 support only Bus Request level 3 and Bus Grant level 3.

VME-bus Interrupts

The VME bus supports seven levels of prioritized interrupts, 1 through 7 (where 7 has the highest priority). The VME-bus adapter has a register associated with each level. On Silicon Graphics systems, all VME interrupts come in at the same CPU interrupt level. When the system responds to the VME-bus interrupt, it services all devices identified in the interrupt vector register in order of their VME-bus priority (highest number first). The operating system then determines which interrupt routine to use, based on the interrupt level and the interrupt vector value.

Note: On systems equipped with multiple VME buses, adapter 0 has the highest priority; other adapters are prioritized in ascending order (after 0).

No device can interrupt the VME bus at the same level as an interrupt currently being serviced by the CPU because the register associated with that level is busy. A device that tries to post a VME-bus interrupt at the same VME-bus priority level as the interrupt being serviced must wait until the current interrupt is processed.

Therefore, when choosing VME-bus priority levels for devices, be sure that the priority levels are well distributed. If you must double up on VME-bus priority levels, double up on those devices not likely to need the CPU at the same time.

Note: All VME interrupt levels map into one CPU interrupt level.

Distribution of VME Interrupts on Multiprocessors

On CHALLENGE/Onyx, POWER CHALLENGE/POWER Onyx, and multiprocessor POWER Series systems, VME interrupt levels can be individually locked onto any processor in the system through the IPL directive. This prevents a processor running a real-time process or a process that needs a guaranteed response from being interrupted inconveniently, and it makes system load balancing easier. To lock a particular VME interrupt level to a processor, edit the */var/sysgen/system/irix.sm* file, then run **lboot** to implement the changes. The format is:

```
IPL: level cpu#
```

where *level* is the priority level (1-7, with 7 being the highest), and *cpu#* is the number of the CPU on which you want the VME interrupts of that level to occur. For example:

```
IPL: 4 1
```

designates VME interrupt priority level 4 on CPU number 1.

CHALLENGE/Onyx and POWER CHALLENGE/POWER Onyx and multiprocessor POWER Series™ systems take advantage of multiple CPUs by distributing interrupts across all processors. These distributed interrupts are called *sprayed* interrupts. To declare a CPU that is not suitable for sprayed interrupts (usually because they will be used for real-time activities), use the *NOINTR* directive.

Example: to declare that CPU 3 will not accept sprayed interrupts, use:

```
NOINTR: 3
```

You *can* tie a VME interrupt to a processor that accepts no sprayed interrupts using the IPL directives described above. You may *not* restrict CPU 0 from receiving interrupts. You can specify multiple CPUs on the NOINTR line.

After editing the *irix.sm* file, you must run **lboot** to reconfigure the system before the changes can take effect. **autoconfig** is a script in */etc/init.d* that runs **lboot**. See the **autoconfig**(1M), **lboot**(1M), and **system**(4) man pages for details.

Choosing a Driver Model

Choosing between a user-level device driver and a kernel-level device driver model usually depends on the method used to transfer data to and from the device. However, the two driver models are not necessarily mutually exclusive. It is possible, for example, for a single VME driver to use both direct memory access (DMA) and memory mapping to transfer data.

User-level VME-bus Device Driver

The easiest way to handle a VME device is to write a user-level program that controls the device by dealing directly with the special */dev/vme* driver. You can write a user-level device driver when your users need to access a VME-bus device that is not interrupt driven and does not require DMA operations. In fact, many boards that use DMA or generate interrupts can have these features turned off for simple, user-level device drivers.

User-level VME-bus device drivers are convenient for determining whether a device responds to the correct address or simple register tests. They can also be useful for prototyping: you can quickly integrate boards whose interrupts can be turned off into a system, then later write a kernel-level driver that turns the interrupts back on for higher performance. In addition, you can use a user-level VME-bus device driver in real applications that require low-overhead access to on-board device registers or memory.

A user-level VME-bus device driver might typically handle data acquisition hardware—hardware that reads large amounts of data into device memory. Because the device memory is memory-mapped into the address space of the user program, it is available to the user program directly; the user program can avoid copying the data into host memory, processing the data in the device memory instead. However, these PIO accesses may have substantially lower performance than DMA-based kernel drivers. Refer to “Programmed I/O (PIO)” on page 88.

Kernel-level VME Device Driver

You must write a kernel-level device driver for a VME device that is interrupt-driven or that requires DMA. See Chapter 2, "Writing a Device Driver," for a description of the IRIX device driver interface.

Kernel-level General Memory-mapping Device Driver

If you want to write a driver that lets users access the VME device as memory in user space and also supports DMA and interrupts, you cannot use the general-purpose VME device driver. Instead, you must write a kernel-level device driver of the general memory-mapping model. Likewise, if you need an efficient way to share main memory between a kernel driver and a user program, you must write a device driver of the general memory-mapping model.

The general memory-mapping model is a kernel-level device driver similar to the user-level memory-mapped device driver described above. See Chapter 2, "Writing a Device Driver," for a general description of kernel-level device drivers. See Chapter 7, "Writing Kernel-level General Memory-mapping Device Drivers," for a description of the memory mapping facilities.

Writing User-level VME Device Drivers

The IRIX operating system contains special files in */dev/vme/vme** that provide access to the various address spaces on the system's VME-bus adapters. These special files allow a user-level program to map arbitrary VME devices into its address space. You can take advantage of them to write a user-level memory-mapped device driver.

Byte addresses in */dev/vme/vme** are interpreted as VME-bus addresses. Not all addresses can be read from or written to because of read-only or write-only registers and unequipped addresses. Reads or writes to invalid VME-bus addresses normally result in a SIGBUS signal being sent to the offending process.

If multiple processes have the mapping for the VME address that got an error, a SIGBUS signal is sent to each of them. On multiprocessor systems, a write to an invalid VME-bus address behaves differently from one on a single-processor system. In these cases, since writes are asynchronous, processors do not wait for the completion of the write operation. If a write operation fails, it may take up to 10 milliseconds for the user VME process to be signaled about a failed write. (VME-bus time-out is about 80 microseconds.) So, if the user VME process has to confirm the successful completion of a write, it should wait for about 10 milliseconds. If the user VME process has already terminated by the time the kernel gets the VME write error interrupt, it sends a message to SYSLOG indicating the VME adapter number and failed VME-bus address.

When your driver maps a device into the address space of a user-level program (through the **mmap**() system call), the user-level program can use simple loads and stores to and from program variables to read or modify device registers or to read or set on-board device memory. If you use memory mapping, you do not need to modify any *irix.sm* files.

Recall that mmapped device drivers are slave devices in which the hardware is memory mapped into a user's address space. No interrupt or DMA service routine is available to the user process.

The special files found in */dev/vme/** are named in the format:

`/dev/vme/`*vme<adapter-#><address-space><address-mode>*

Arguments

adapter-#	specifies which VME-bus adapter
address-space	specifies which address space, such as 16, 24, or 32 (see "VME-bus Space Reserved for Customer Drivers" on page 329.)
address-mode	identifies the addressing mode, which is *n* for non-privileged or *s* for supervisor

Use the **hinv**(1M) (*hardware inventory*) command to produce a list of valid VME-bus adapters present on the system. Adapter numbers range upwards from 0. These adapters can be used only for memory mapping VME-bus address space into the address space of a user's program. The address space can be 16, 24 or 32. The address mode is either *n* for non-privileged or *s* for supervisory. Thus, adapter 0, address space 16 in non-privileged mode is referred to as */dev/vme/vme0a16n*.

Use the technical specification for the device to determine the slave addressing mode.

The kernel driver for user-level VME is referred to as **usrvme**. If VME buses are added to an existing system, it may be necessary to run **MAKEDEV**(1M), specifying a target of **usrvme**, to have the additional */dev/vme* devices created.

Example VME Device Driver

The following code sample uses the user-level VME-bus interface to perform bus probes:

```
#include <sys/types.h>
#include <sys/mman.h>
#include <stdio.h>
#include <fcntl.h>
#include <getopt.h>
#include <errno.h>
#include <limits.h>
#include <signal.h>

int state = 0;

#define S_DIR 0x01
#define S_ADAP 0x02
#define S_SPACE 0x04
#define S_ADDR 0x08
#define S_SIZE 0x10
#define S_VAL 0x20

#define D_READ 0x1
#define D_WRITE 0x2

#define READSTATE (S_DIR|S_ADAP|S_SPACE|S_ADDR|S_SIZE)
#define WRITESTATE (READSTATE|S_VAL)

char *progname;

char *spaces[] = {
```

```
    "a16n",
    "a16s",
    "a24n",
    "a24s",
    "a32n",
    "a32s"
};

#define MAXSPACE (sizeof(spaces)/sizeof(spaces[0]))

void usage(void);
long ntol(char *);
long chkspc(char *);

char devnm[PATH_MAX];

static void probe_fail(int);

int
main(int ac, char *av[])
{
    int    c, errflg = 0;
    int    dir_f;
    long    adap_f;
    long    addr_f;
    long    size_f;
    long    val_f;
    char *space_f;
    int    fd;
    char *mapaddr;
    int    pgaddr, pgoff;
    int    pgsz;
    int    rtval;

    progname = av[0];

    while( (c = getopt(ac,av,"rws:a:b:p:v:")) != -1 )
        switch( c ) {
        case 'r':
            if( state & S_DIR ) {
                usage();
                return 1;
            }
            dir_f = D_READ;
            state |= S_DIR;
```

```
        break;
    case 'w':
        if( state & S_DIR ) {
            usage();
            return 1;
        }
        dir_f = D_WRITE;
        state |= S_DIR;
        break;
    case 's':
        if( state & S_SPACE ) {
            usage();
            return 1;
        }
        if( chkspc(optarg) ) {
            usage();
            return 1;
        }
        state |= S_SPACE;
        space_f = optarg;
        break;
    case 'a':
        if( ((adap_f = ntol(optarg)) < 0) ||
             (state & S_ADAP) ) {
            usage();
            return 1;
        }
        state |= S_ADAP;
        break;
    case 'b':
        if( ((addr_f = ntol(optarg)) < 0) ||
             (state & S_ADDR) ) {
            usage();
            return 1;
        }
        state |= S_ADDR;
        break;
    case 'p':
        if( ((size_f = ntol(optarg)) < 0) ||
             (state & S_SIZE) ) {
            usage();
            return 1;
        }
        state |= S_SIZE;
        break;
```

```
case 'v':
    if( ((val_f = ntol(optarg)) < 0) ||
        (state & S_VAL) ) {
        usage();
        return 1;
    }
    state |= S_VAL;
    break;
case '?':
    errflg++;
    break;
}

if( errflg || !(state & S_DIR) ) {
    usage();
    return 1;
}

if( (dir_f == D_READ) && (state != READSTATE) ) {
    usage();
    return 1;
}
if( (dir_f == D_WRITE) && (state != WRITESTATE) ) {
    usage();
    return 1;
}


/* check the size */
switch( size_f ) {
case 1:
case 2:
case 4:
    break;
default:
    (void)fprintf(stderr,"invalid size %d\n",size_f);
    usage();
    return 1;
}

/* create name of device */
sprintf(devnm,"/dev/vme/vme%d%s",adap_f,space_f);

/* open the usrvme device */
if( (fd = open(devnm,O_RDWR)) < 0 ) {
```

```
        perror("open");
        return 1;
    }

    /* we map in memory on page boundaries so figure out
     * the page and page offset
     */

    pgsz = getpagesize();
    pgaddr = (addr_f / pgsz) * pgsz;
    pgoff = addr_f % pgsz;

    /* map in the vme space surrounding the address */
    if( (mapaddr = mmap(
            NULL,pgsz,PROT_READ|PROT_WRITE,MAP_PRIVATE,
            fd,pgaddr)) == (void*)-1 ) {
        perror("mmap");
        return 1;
    }

    /* catch bus errors */
    signal(SIGBUS,probe_fail);

    /* do the probe */
    if( dir_f & D_READ ) {
        switch( size_f ) {
        case 1:
            rtval = *(char *)&mapaddr[pgoff];
            break;
        case 2:
            rtval = *(short *)&mapaddr[pgoff];
            break;
        case 4:
            rtval = *(int *)&mapaddr[pgoff];
            break;
        }
        printf("read probe of 0x%x\n",rtval);
    }
    else {
        switch( size_f ) {
        case 1:
            *(char *)&mapaddr[pgoff] = val_f;
            break;
        case 2:
            *(short *)&mapaddr[pgoff] = val_f;
```

```
                break;
            case 4:
                *(int *)&mapaddr[pgoff] = val_f;
                break;
            }
            printf("write probe of 0x%x\n",val_f);
            /* wait here to catch any bus errors... */
            sginap(CLK_TCK/50+1);
        }

        return 0;
}

long
ntol(str)
    char *str;
{
    char *strp;
    ulong ret;

    if( *str == '"' ) {
        str++;
        return (*str)?*str:-1;
    }

    ret = strtoul(str,&strp,0);

    if( ((ret == 0) && (strp == str)) ||
        ((errno == ERANGE) && (ret = -1)) )
        return (long)-1;

    return (long)ret;
}

long
chkspc(char *nm)
{
    int i;

    for( i = 0 ; i < MAXSPACE ; i++ )
        if( strcmp(nm,spaces[i]) == 0 )
            return 0;

    return 1;
}
```

```
void
usage()
{
 (void)fprintf(stderr,
    "usage: %s -r -a adap -s space -b busaddr -p
probesize\n",
    progname);
 (void)fprintf(stderr,
    "usage: %s -w -a adap -s space -b busaddr -p probesize -
v val\n",
    progname);
 (void)fprintf(stderr,
    "    space is one of a16n, a16s, a24n, a24s, a32n,
a32s\n");
 (void)fprintf(stderr,
    "    probesize is one of 1 2 or 4\n");
}

static void
probe_fail(int signo)
{
    fprintf(stderr,"*** probe failed\n");
    exit(1);
}
```

Using mmap

After you have reconfigured the system correctly, the user-level driver can open the special file for a generic VME device */dev/vme/vme**. To map in the device, the user program must use the **mmap**() system call. For example:

```
#include "fcntl.h"
#include "sys/mman.h"
fd = open("/dev/vme/vme0a16s", O_RDWR);
addr = mmap(
         0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, off);
```

The **mmap**() routine maps *len* bytes starting at (VME-bus) address *off* to the user virtual address *addr*. The *prot* argument is a bit mask that indicates the protection that the operating system enforces on access to the device memory. Thus, *PROT_WRITE* allows writing; *PROT_READ* allows reading. The *flags* argument can be either *MAP_PRIVATE* or *MAP_SHARED* when used with hardware devices (currently, */dev/vme/vme** makes no distinction between the two). These symbolic constants are defined in *sys/mman.h*. See the mmap(D2) man page for further information on the use of this system call.

Once the **mmap** call succeeds, reads and writes from the user virtual address *addr* for a length of *len* bytes result in the appropriate reads and writes for the VME device pointed to by the file descriptor *fd*.

Note: There is protection on a page boundary only. Even if the user-level program maps in less than a page, the entire page of device registers remains accessible to the user program. Use **getpagesize**(2) to determine the page size of the system.

The amount of VME address space that can be mapped into user address space depends on two factors:

- VME address space type (A16, A24, A32)
- hardware platform

For VME A16 address space, the entire 64 KB of A16-VME address space is mappable to user virtual address space.

For A24 address space, only 8 MB of the 16 MB of address space, starting at VME address 0x80000000, is mappable to the user virtual address space. The kernel reserves the remaining 8 MB of address space to support DMA transfers from A24 masters.

For A32 address space, there is some variation according to hardware platform.

CHALLENGE/Onyx systems support up to five VME buses. Users can map in a maximum of 96 MB of A32 address space on each VME bus.

Note: Mapping should be performed in 8 MB increments: Each **mmap**() system call can map a maximum of 8 MB of VME address space into user virtual address space, so eight such **mmap**() calls are needed to map the entire 96 MB of VME address space available.

This 96 MB of VME address space is shared between the kernel and user-level device drivers. Any installed kernel-level VME device drivers that use

VME address space reduce the amount of VME address space available for mapping by the user-level drivers.

On other IRIS platforms (IP5/7/9/17), a maximum of 256 MB can be mapped into user virtual address space. On these platforms, **mmap**() can support mapping in the entire 256 MB of VME address space in a single system call. On systems with dual VME buses, however, the amount of VME address space available for mapping is reduced by half.

See Appendix A for further details of what A32 address space is available for customer boards.

Accessing Mapped Space

VME accesses are sensitive to the access size, so extra caution is called for once VME address space is mapped into a user's virtual space. For example, A16D8 boards may support only 8-bit accesses, while A16D16 boards may support both 8-bit and 16-bit accesses. Similarly, A32D32 may support only 32-bit accesses or may support 8-bit, 16-bit, 32-bit accesses. User-level device drivers should ensure that the data structures onto which the VME address space is mapped generate the proper size of transaction on the VME bus.

VME-bus Error Handling for User-level Device Drivers

Bus errors can occur when a read or write on the VME times out. This can be triggered by user-level drivers accessing a mapped VME space for which no controller exists, or if the controller fails to respond.

Caution: If the kernel cannot determine whether the bus error is harmless to the system, the system panics.

Read Errors

If a VME-bus read error is triggered by a user-level VME driver, the driver process receives a SIGBUS signal. If the driver needs to be aware of the error, then it can catch the signal and take appropriate action. Read errors are synchronous, so the user process can get a definite idea of what PC or routine caused the error.

Write Errors

VME-bus write errors are asynchronous: when a user-level driver writes to the mapped VME address space, the CPU does not stall at that address, but continues executing further instructions. Since the write error time-out bus takes up to 80 microseconds on a VME bus and another finite amount of time for handling the error interrupt from the VME bus, handling write errors can be complicated.

A bad VME write error results in a SIGBUS signal being sent to the offending process, if that process is still running on the system. Since it takes a finite amount of time to send the signal to the user-level driver process that triggers a bad VME write, it is essential for the user-level driver to wait for up to 10 milliseconds before exiting. However, this is only necessary before exiting the system (to allow for the handling of the last few bad writes) or when the user process wants to assure that the write completed. It is not necessary to wait for 10 milliseconds after every VME write.

User-level DMA Library (*udmalib*)

A user-level interface for VME drivers provides access to DMA engines on CHALLENGE/Onyx (IP19) and POWER CHALLENGE/POWER Onyx (IP21) hardware platforms. This interface is meant to be used when performance is critical and the VME-bus board itself does not support DMA.

Users can move data between a buffer and a VME-bus board faster with a DMA engine than with normal PIO operations. However, because there is only one DMA engine per VME bus on the CHALLENGE series, the DMA engine is a scarce resource. The user-level DMA support library *udmalib* allocates the DMA engine exclusively to the first user to request it, and no other user can access it until the current user frees it up. See the **usrdma**(7M) and **udmalib**(3K) man pages for further detail and usage of user-level DMA library calls.

The following functions are supported by the user-level DMA library:

dma_open Get exclusive use of a DMA engine

dma_close Free up the DMA engine

dma_allocbuf Allocate a buffer suitable for DMA

dma_freebuf Free up a DMA buffer

dma_mkparms Define a DMA operation

dma_freeparms Free up DMA parms resources

dma_start Perform DMA operation between a buffer and the VME bus

Here is a sample code fragment using the user DMA library:

```
#include <udmalib.h>
#include <stdio.h>

do_dma(int adap, void *vmeaddr, int size)
{
   udmaparm_t    *parms;
   udmaid_t     *dp;
   vmeparms_t    vparm;
   void        *iobuf;
   int          err = 0;

   if( (dp = dma_open(DMA_VMEBUS,adap)) == NULL ){
      (void)fprintf(stderr,
                  "unable to start adapter %d\n",adap);
      return 1;
   }

   /* get a buffer and phys address */
   if( (iobuf = dma_allocbuf(dp,size)) == NULL ) {
      (void)fprintf(stderr,"iobuf alloc failed\n");
      (void)dma_close(dp);
      return 1;
   }

   vparm.vp_block = 0;
   vparm.vp_datumsz = VME_DS_HALFWORD;
   vparm.vp_dir = VME_READ;
   vparm.vp_throt = VME_THROT_256;
   vparm.vp_release = VME_REL_RWD;
   vparm.vp_addrmod = 0xd;

   /* create DMA parms */
   if( (parms = dma_mkparms(dp,&vparm,iobuf,size)) == NULL
)  {
      (void)fprintf(stderr,"dma failed\n");
      (void)dma_freebuf(dp,iobuf);
```

```
        (void)dma_close(dp);
        return 1;
    }

    if( err = dma_start(dp,vmeaddr,parms) )
        (void)fprintf(stderr,"dma failed\n");

    if( dma_freebuf(dp,iobuf) ||
         dma_freeparms(dp,parms) ||
         dma_close(dp) )
        (void)fprintf(stderr,"dma release failed\n");

    return err;
}
```

Writing Kernel-level VME Device Drivers

Determining VME Device Addresses

Each VME device has a set of VME-bus addresses to which it responds. These addresses correspond to device registers or on-board memory, depending on the VME device. Your driver can map these VME addresses into the host processor address space: your users can access the device with simple reads and writes. VME devices can be classified as A16, A24, A32, or A64 VME slaves. Each class specifies a range of addresses to which the device responds. You determine the slave addressing mode from the technical specification for the device.

Once you determine the addressing mode, choose VME addresses that do not conflict with existing VME device drivers. For each slave addressing class, Silicon Graphics has reserved a range of addresses for use by user-written drivers. These ranges are listed in */var/sysgen/system/irix.sm* and tabulated in Appendix A, "System-specific Issues".

You must choose a VME-bus interrupt priority level for the device. The VME-bus interrupt priority level must be a value from one to seven. Later, all VME interrupts are channeled into one CPU interrupt level. The priority of this CPU interrupt is below that of the clock and any on-CPU devices.

In the past, it was necessary to reserve a VME interrupt vector. Since most VME devices can program the interrupt vector through software, a dynamic allocation scheme for vectors now hands VME vectors out to drivers at initialization time. However, if your VME device has a hard-wired or jumpered VME vector, it is still possible to reserve the VME vector that the device requires.

When CPU interrupts are assigned to the VME bus, the CPU services the VME-bus interrupts in order of their VME-bus priority. For each CPU interrupt, the system services only one device per VME-bus priority level. If more than one VME-bus interrupt occurs at the same VME-bus interrupt priority level, all but one device must wait until the next time the CPU services the VME-bus CPU interrupt.

After picking an appropriate set of addresses and an interrupt priority level, you must program the VME device to respond accordingly. Usually, you do this with jumpers or switches. Some VME devices allow you to program the VME vector and interrupt priority level at boot time (from your driver's *drv***edtinit**() routine).

If the device performs DMA, you need to know the addressing mode by which the device accesses main memory. This addressing mode is called its *master* addressing mode, as opposed to the *slave* addressing mode described above. Silicon Graphics supports A24, A32, and A64[1] VME master addressing. (POWERchannel-2 has master DMA capability.) The master addressing mode determines the driver structure to some degree.

Including VME Device Drivers in the Kernel

Chapter 2, "Writing a Device Driver," provides general information on adding a driver to the kernel. This section describes specifics concerning VME drivers.

To add a new kernel-level device driver, you must create your own *irix.sm* file that contains the appropriate directive telling **lboot** how to include your driver. The filename, which must end with "*.sm*", belongs in the directory */var/sysgen/system*. Because **lboot** can probe for VME devices, **lboot** can conditionally include a VME device driver into the kernel.

If the current system contains the VME device, **lboot** includes the driver; otherwise, it saves memory by leaving it out. Use the *VECTOR* directive to include a VME device conditionally. In addition to the module name, the *VECTOR* directive requires that you fill out these fields:

adapter	The adapter number identifying which VME bus out of possibly several.
bustype	This must be set to VME.
ctlr	The device number that differentiates between more than one device of the same type.

[1] R4000 – R4400 use 64-bit MIPS III mode in the CHALLENGE/Onyx chassis.

exprobe_space This is an extended probe that can do reads and writes with compares against expected values to search for a VME device at an address. The first *arg* defines a sequence of reads and writes. The second *arg* specifies the address space. The third *arg* specifies the probe address. The fourth *arg* is the size of the probe, 1-4 bytes. The fifth *arg* is the expected read or write value. The last argument is a mask that the fifth *arg* is ANDed against.

iospace, *iospace2*, *iospace3*
This is a triple identifying a VME-bus space, address, and size. The bus space is one of *A16NP*, *A16S*, *A24NP*, *A24S*, *A32NP*, *A32S*.

ipl The VME interrupt priority level. This must be a value from 1 to 7, as described above.

probe_space This is a triple identifying a VME-bus space, address, and read byte count. The address specified is read by **lboot** to determine the existence of device. If you do not specify a probe address, the module is automatically included in the kernel.

vector The VME interrupt vector value. This must not be used unless the VME device has hard-wired or jumpered vector values.

You must also create a master file under */var/sysgen/master.d.* A master file has four sections:

- a tabulated ordering of flags
- phrases and values interpreted by the configuration program and used to build device tables
- a list of stub routines
- a section of C code.

The first, non-blank, non-comment line is interpreted for flags, phrases, and values. Other non-comment lines that follow, up to a line that begins with a dollar sign, specify stubs. Anything that follows the line beginning with a dollar sign is processed to interpret special characters, then compiled into the kernel.

The name of the master file is the same as the name of the object file for the driver, but the master file must *not* have the *.o* suffix. The FLAG field of the master file must include at least the character device flag *c*. (You do not need the *s* flag for VME device drivers because **lboot** can probe for VME devices.) See */var/sysgen/master.d/README* and the **master**(4) man page.

Note: For network drivers, the FLAG field would blank with a "-" (hyphen).

As an example, suppose you want to add a mythical VME device driver to the kernel. You must copy the driver object file *vdk.o* to */var/sysgen/boot*, and you must add a line similar to the following to a file called *vdk.sm* in */var/sysgen/system*:

```
VECTOR: bustype=VME module=vdk ipl=1 ctlr=0 adapter=0
iospace=(A16S,0x400,0x200) iospace2=(A16S,0x800,0x100)
probe_space=(A16S,0x404,2)
```

Note: The above lines must all be on one line in the *irix.sm* file.

Note that the bus addresses and sizes are specified in hexadecimal format. The *ctlr*= value helps identify a device when more than one uses the same driver. If there is more than one device, give each a unique number, starting from zero. In the above example, **lboot** reads two bytes at probe address 0x404 to determine whether the device is present.

After examining */usr/include/sys/major.h*, you determine that major device number 51 is available and can be used for this device. You then create a master file *vdk* in */var/sysgen/master.d*, and enter:

```
*FLAG     PREFIX    SOFT      #DEV        DEPENDENCIES
 c        vdk       51        -
```

Writing edtinit()

If you use the *VECTOR* directive to configure a driver into the kernel, your driver can use a routine of the form *drv***edtinit**(), where *drv* is the driver prefix. If your device driver object module includes a *drv***edtinit**() routine, the system executes the *drv***edtinit**() routine when the system boots. In general, you can use your *drv***edtinit**() routine to perform any device driver initialization you want.

Synopsis

```
drvedtinit(e)
struct edt *e
{
   your code here
}
```

edt Type Structure

When the system calls your *drv***edtinit**() routine, it hands the routine a pointer to a structure of type *edt*. (This structure type is defined in the *sys/edt.h* header file.)

Structure Definition

```
typedef struct iospace {
    unchar    ios_type;      /* io space type in adapter */
    iopaddr_t ios_iopaddr;   /* io base address */
    ulong     ios_size;
    caddr_t   ios_vaddr;     /* kernel virtual address */
} iospace_t;

#define NBASE 3

typedef struct edt {
   uint_t    e_bus_type;     /* vme, scsi, eisa, ... */
   unchar    v_cpuintr;      /* cpu to send intr to */
   unchar    v_setcpuintr;   /* cpu field is valid */
   uint_t    e_adap;         /* adapter */
   uint_t    e_ctlr;         /* controller identifier */
   void*     e_bus_info;     /* bus-dependent info */
   int       (*e_init)( );   /* device init/run-time probe */
   iospace_t e_iospace[NBASE];
};
```

Based on *e_bus_type*, **lboot** will set up *e_bus_info* to point to the corresponding bus-dependent data structure (that is, *vme_intrs*). With this two-layer structure, it is easier to extend *edt* to support EISA, GIO, or other types of buses.

CHALLENGE/Onyx and POWER CHALLENGE/POWER Onyx systems can support more I/O address space than the kernel can, so it is necessary to provide a way of allocating only the part of the I/O space that is needed into the kernel address space. Programming the I/O board/adapter registers dynamically assigns the mapping. The *edt* structure must describe the device by adapter ID and adapter bus address. The kernel uses this information to initialize the kernel virtual address.

On POWER Series workstations, ranges of VME-bus address space are mapped one-to-one with K2 segment addresses. This makes accessing the VME bus easy, but is also limiting. Only a small amount of K2 space is available for use by VME, so very little of the VME address space is made available. Even worse, for dual VME-bus systems, the space previously available is now halved because the two buses must share it.

The CHALLENGE series supports up to five VME buses. Since K2 space is a limited resource, and dividing up what is available by five makes the extra VME buses next to useless, a new approach was tried. The CHALLENGE series does not have a direct K2 address map into VME-bus space. Each VME-bus adapter has the ability to map fifteen 8 MB windows of VME-bus space into K2 space. These windows can be moved around at will to give the illusion of a much larger address space.

To access a VME space, a user must allocate a PIO map, which provides a translation between a kernel address and VME space. These mappings can be “FIXED” or “UNFIXED.” As on POWER Series platforms, a FIXED mapping is a one-to-one mapping of a range of VME-bus space into the driver’s address space. An UNFIXED window takes advantage of the sliding window ability on the CHALLENGE series, which supports both FIXED and UNFIXED mappings.

In an UNFIXED map, VME-bus space cannot be accessed directly; instead, access is provided through special **bcopy**() routines used to move data between VME space and kernel buffers. While it is not always possible to get a FIXED mapping, an UNFIXED mapping is always available. The special **bcopy**() routines work for both FIXED and UNFIXED mappings. On

POWER Series and earlier workstations, UNFIXED mappings are treated as FIXED mappings.

The PIO mapping routines also have a general interface that allows them to be used for mapping in bus spaces other than VME.

The support routines for PIO mapping are:

pio_mapalloc	Allocate a PIO map.
pio_mapaddr	Map bus space to a driver accessible address (FIXED maps only).
pio_mapfree	Free a previously allocated PIO map.
pio_badaddr	Check to see whether a bus address is equipped.
pio_wbadaddr	Check to see whether a bus address is equipped.
pio_bcopyin	Copy data from bus space to kernel buffer.
pio_bcopyout	Copy data from kernel buffer to bus space.

These PIO maps are normally set up in the driver's *drv***edtinit**() routine.

e_iospace is enhanced to be a structure of I/O base address, size and type of the I/O mapping, and kernel virtual address space. *ios_type*, *ios_iopaddr*, and *ios_size* are initialized by **lboot** from the system, and *ios* is assigned when a driver is initialized.

e_adap is added to specify the adapter number, while *e_ctlr* is for physical controller ID.

To pass the desired interrupt CPU to the driver via the *irix.sm* file, use the *VECTOR* directive. The line

```
VECTOR: module=XXX intrcpu=3
```

directs **lboot** (via **autoconfig**) to set the *v_intrcpu* field for the module's **edt** struct to 3 and the *v_setintrcpu* field to 1, indicating that *v_intrcpu* is valid. If no *intrcpu*= statement appears in the VECTOR line, *v_setintrcpu* is set to 0. The module's **edtinit** function may then use these fields to route interrupts as desired.

```
void
XXXedtinit (struct edt *ep)
{
     if (ep->setcpuintr)
               dest_cpu = ep->cpuintr;
     else
          dest_cpu = <some default>;

     ...machine-specific intr routing ...
}
```

vme_intrs Structure

In the case of a VME driver, the field *e_bus_info* will point to the *vme_intrs* structure.

Structure Definition

```
typedef struct vme_intrs {
    int      (*v_vintr)();    /* interrupt routine */
    unsigned char  v_vec;     /* vme vector */
    unsigned char  v_brl;     /* interrupt priority level */
    unsigned char  v_unit;    /* software identifier */
} vme_intrs_t;
```

The only field that must be accessed is *v_brl*, which contains the *ipl=value* from the *VECTOR* line. The *v_vec* field must be used only if the *VECTOR* line uses the *vector= directive* and your device requires a jumpered or hard-wired VME interrupt vector.

Note: Although **lboot** knows not to include a VME device driver in the kernel for a device not present, it is a good idea for your *drv***edtinit**() routine to probe for its device with **badaddr**(). This lets you write a driver that is prepared if the device is removed from the system after the kernel has been built or when the kernel runs on another system.

Continuing with this mythical VME device driver example, its *drv***edtinit**() routine could look like:

```
struct drvctlrinfo ctlrinfo[MAXCTLR];
drvedtinit(edt_t *e)
{
   int i, vec;
   struct vme_intrs   *info;
   volatile struct drvdevice *dp;
   struct drviopb      iopb;
   piomap_t *pmap;

   pmap = pio_mapalloc(e->e_bus_type,e->e_adap,&e->e_space[0],
      PIOMAP_FIXED,"DRV");

   /* make sure adapter exists and addresses are valid */
   if( pmap == 0 )
      return;
   dp = pio_mapaddr(pmap,e->e_iobase);
   /* probe for the device */
   if( badaddr(&dp->csr,sizeof(dp->csr)) ) {
      cmn_err(CE_WARN,"drv: ctlr %d not installed\n",e->e_ctlr);
      pio_mapfree(pmap);
      return;
   }

   /* save the controller's device registers pointer */
   ctlrinfo[e->e_ctlr]->devregs = dp;

   /* dynamically allocate an interrupt vector */
   vec = vme_ivec_alloc(e->e_adap);
   if( vec == -1 ) {
      cmn_err(CE_WARN,"drv: ctlr %d, no irq vector\n", e->e_ctlr);
      pio_mapfree(pmap);
      return;
   }

   /* register our interrupt routine with the kernel */
   vme_ivec_set(e->e_adap,vec,drvintr,e->e_ctlr);
   iopb.ipl = info->v_brl;
   iopb.vec = vec;
   .
   .
   .
}
```

Two new routines **vme_ivec_alloc**(*uint_t, adapter*) and **vme_ivec_set**(*adapter, vec, intr_func, arg*) are implemented to dynamically allocate an interrupt vector and register this vector into **vme_ivec**(). This scheme supports multiple vectors and loadable drivers. **vme_ivec_alloc**() and **vme_ivec_set**() are used in an **edtinit** routine; **vme_ivec_free**() can be called to free up a vector that has been allocated.

You can specify the vector in the *irix.sm* file for old VME boards with a hard-wired interrupt vector. 0x30-0x3f and 0x70-0x7f are reserved for customer boards.

VME Interrupt Handler

Your driver module must contain a routine of the form *drv***intr**(), where *drv* is the driver prefix. When the device generates an interrupt, the general VME interrupt handler calls your driver's *drv***intr**() routine.

When the VME interrupt handler calls your *drv***intr**(), it passes it the value registered with the **vme_ivec_set**() routine for the device. Within your *drv***intr**() routine, you must set flags to indicate the state of the transfer and to wake any sleeping processes waiting for the transfer to complete. Usually, the interrupt routine calls **biodone**() to indicate that an I/O transfer for the buffer is complete.

Caution: Interrupt routines (*drv***intr**()) must not try to sleep themselves by using **biowait**(), **sleep**(), **psema**(), or **delay**() kernel calls.

With the new dynamic interrupt vector allocation scheme, the argument passed to the individual *drv***intr**() is *arg*, which is set by **vme_ivec_set**() in **edtinit**(). *arg* can be an index, a controller number, the address, or any argument that the driver wants to pass to interrupt the service routine.

The IRIX 5.x and 6.0 *vme_ivec* structure is shown below:

```
struct vme_ivec {
    int   (*vm_intr)(int);
    int    vm_arg;
}vme_ivec[ADAPTER][MAX_VME_VECTS];
```

Note: Although the prototype for the VME interrupt handler routine in the **vme_intrs** (*field v_vintr*) and **vme_ivec** (*field vintr*) structures indicates that it returns an integer value, the return value is not used. The prototype should indicate that the function is of type **void***. It was left unchanged to avoid breaking existing VME device drivers.

To support the loadable device drivers, multiple adapters, and multiple interrupt vectors, the *vem_ivec* table is changed to be dynamically allocated at system boot time. The size depends on the number of VME adapters currently supported by the running system. After the table is allocated, the kernel fills the entries for the devices specified in the *irix.sm* file.

Programmed I/O (PIO)

When transferring large amounts of data, your device driver should use direct memory access (DMA). Using DMA, your driver can program a few registers, return, and put itself to sleep while it awaits an interrupt that indicates the transfer is complete. This frees up the processor for use by other processes. See the ioctl(D2) man page.

Sometimes you must write a driver for a device that does not support DMA. Even if the device does support DMA, you may not want to use DMA to transfer amounts of data so small as not to warrant the DMA overhead.

In these cases, the host processor usually copies the data from the user space to on-board memory. Your driver can then program the device registers to notify the device that the memory is ready. The device controller can then copy the data from its on-board memory to the peripheral device.

Listed below is part of a mythical VME device driver for a printer controller that does not support DMA.

To print data from the user, the driver copies data from the user's buffer to an on-board memory buffer of size *VDK_MEMSIZE*. Following the copy of each chunk, the driver programs the device register to indicate the size of valid data in memory and to tell the controller to start printing.

The driver then sleeps, waiting for an interrupt to indicate that the printing is complete and that the on-board memory buffer is available again. To prevent a race condition, in which the interrupt responds before the calling process can sleep, the driver uses the **splvme**() and **splx**() routines. See the spl(D3) man page.

```
int vdk_state;      /* flag for transfer state */

int
vdkwrite(dev_t dev, uio_t *uiop, cred_t *crp)
{
   register int size;
   register int i;
   int s;
   int err;

   /* while there is data to transfer */
   while( uiop->uio_resid > 0 ) {

      /* can only move VDK_MEMSIZE bytes at a time */
      size = MIN(uiop->uio_resid,VDK_MEMSIZE);

      if( (err = uiomove(vdk_memory,size,UIO_WRITE,uiop)) != 0 )
         return err;

      /* block interrupts until we sleep */
      s = splvme(); /* may not be sufficient on MP */

      /* start printing */
      vdk_device->count = size;
      vdk_device->command = VDK_GO;
      vdk_state = VDK_SLEEPING;

      while ( vdk_state != VDK_DONE )
         sleep(&vdk_state,PRIBIO);

      /* restore the process level after waking up */
      splx(s); /* clears any MP locks as well */
   }

   return 0;
}

void
vdkintr(int unit)
{
   ...
   /* printing is complete */
```

```
    if( vdk_state == VDK_SLEEPING ) {
        vdk_state = VDK_DONE;
        wakeup(&vdk_state);
    }
    ...
}
```

The driver's use of the *volatile* declaration informs the optimizer that this variable points to a hardware value that may change. Otherwise, the optimizer may determine that one write to *vdk_device*->*command* or storage of the value in a register is sufficient.

Note: If your driver uses the **sleep**() and **wakeup**() kernel routines to sleep and awaken, it is a good idea for the top half to verify that the event has occurred before awakening the sleeping process. (See sleep(D3) for details on the sleep/wakeup process synchronization mechanism.) If your driver uses the **biowait**()/**biodone**() routines or the **psema**()/**vsema**() routines to sleep and awaken, you need not worry about its awakening by accident. However, the routines **psema**() and **vsema**() are specific to IRIX and are probably not supported on other operating systems.

The **uiomove**() kernel routine is a useful procedure to call in these situations because it automatically updates *uio* and *iovec* structures while checking for valid user addresses. Remember that *uiop*->*uio_resid* must be left with the number of bytes remaining untransferred.

Note: **uiomove**() uses **bcopy**() to transfer data. **bcopy**() transfers data as fast as possible between locations in system memory. **bcopy**() takes advantage of CPU-specific commands to optimize performance. On the R4000 processor, **bcopy**() tries to move eight bytes at a time. Most VME-bus boards cannot move data eight bytes at a time, so using this routine directly may not work. A work-around would be to use **uiomove**() to copy the data from a user buffer into a kernel buffer, then to use **pio_bcopy**() to copy the data from the kernel buffer to the VME-bus board. **pio_bcopy**() allows the user to specify the element size being transferred.

DMA Operations

As indicated in “Programmed I/O (PIO),” use DMA (direct memory access) when the device supports it. In its simplest form, DMA is easy to use: your driver gives the device the physical memory address, and the transaction begins. Your driver can then put itself to sleep while it waits for the transfer to complete, thus freeing the processor for other tasks. When the transfer is complete, the device interrupts the processor. On most systems, when large amounts of data are involved, DMA devices obtain higher overall throughput than devices that do only PIO.

DMA operations are categorized as *DMA reads* or *DMA writes*. DMA operations that transfer from memory to a device, and hence read memory, are DMA reads. DMA operations that transfer from a device to memory are DMA writes. Thus, you may want to think of DMA operations as being named the from the point of view of what happens to memory.

There are important cache considerations for drivers using DMA. The cache architecture of the system dictates the appropriate cache operations. Write back caches require that data be written back from cache to memory before a DMA write, whereas both write back and write through caches require the cache to be invalidated before data from a DMA read is used. See “Data Cache Write Back and Invalidation” in Appendix A and the dki_dcache_wbinval(D3X) man page for a discussion of these issues.

Another concern for driver writers is that DMA buffers may require cache-line alignment. If a driver allocates a buffer for DMA, it must use the **kmem_alloc**() function, using the *KM_CACHEALIGN* flag.

The interrupt service routine then calls your *drv***intr**() routine. Your *drv***intr**() routine can check that the transfer is complete (if necessary), set flags indicating the status of the transfer, and then awaken the sleeping process.

Unfortunately, the details of how you implement the simple scheme described above is complicated by the use of virtual memory, different VME addressing modes, and a variety of device and system implementations. To sort through these potentially confusing choices, ask the following questions in order. If the answer to any question is “yes,” go on to the section indicated. Otherwise, proceed to the next question.

A32 Addressing Scatter/Gather Support

Modern computer architectures support *virtual memory*—memory in which the user's view of memory is logically contiguous, but the underlying physical pages are not. Because VME devices understand only physical page addresses, your driver would ordinarily be forced to do transfers one page at a time. At the start of each one-page transfer, your driver would have to awaken the sleeping process and compute the physical address for the next virtual page.

Because this is not efficient, many devices now provide a method to store the address mapping for the entire transfer up front. Your driver can usually do this merely by programming the device with a table of physical addresses for all of the upcoming transfer. This method of regrouping of noncontiguous physical memory is called *scatter/gather*.

If your VME device supports scatter/gather, uses A32 addressing, has less than 4 GB of physical memory, and you are not on a CHALLENGE/Onyx series system, proceed to "VME Devices with Scatter/Gather Capability."

DMA Mapping for High-end Systems and Older Systems

Older Silicon Graphics systems and current high-end systems provide for address mapping of physical addresses so that even devices that do not support scatter/gather in the controller can transfer to and from noncontiguous physical pages with ease. This facility, called *DMA mapping*, is available on 4D/100 through 4D/400, Crimson, CHALLENGE/Onyx and POWER CHALLENGE/POWER Onyx series systems. Indigo, Indigo^2, and Indy workstations have no VME-bus support. DMA mapping works equally well for both VME A24 and A32 master addressing. See "Using DMA Maps" for a description of how to use DMA mapping.

Does the VME Device Perform A24 Master Addressing?

If the VME device uses A24 addressing, and your system does not support DMA mapping, the controller can access only the first 8 MB of physical memory. Because user programs may use physical pages beyond 8 MB, your device driver must do DMA into a kernel buffer and copy from that buffer to the user's pages. See "DMA on A24 Devices with No DMA Mapping."

A32 Addressing with No Scatter/Gather

If you are writing a driver for an A32 VME device that does not support scatter/gather on a workstation that does not support DMA mapping, see "DMA on A32 Devices with No Scatter/Gather Capability" for advice on how to implement this driver type.

VME Devices with Scatter/Gather Capability

Chapter 2, "Writing a Device Driver," tells you to use the **physiock**() kernel routine to fault in and lock the physical pages corresponding to the user's buffer. **physiock**() also remaps these physical pages to a kernel virtual address that remains constant even when the user's virtual addresses are no longer mapped.

Internally, **physiock**() allocates a structure of type *buf* if you pass a *NULL* pointer (**physiock**() uses this structure to embody the transfer information). **physiock**() then calls your *drv***strategy**() routine and passes it a pointer to the *buf* type structure that it has allocated and primed. Your *drv***strategy**() routine must then loop through each page, starting at the kernel virtual address, and load each device scatter/gather register in turn with the corresponding physical address. Use the **kvtophys**() routine to convert a kernel virtual address to a physical address.

For example, suppose the mythical device is now an A32 VME device that supports scatter/gather. The scatter/gather registers for the device are simply a table of integers that store the physical pages corresponding to the current transfer. To start the transfer, the driver gives the device the beginning byte offset, byte count, and transfer direction. The code is:

```
#include <sys/sysmacros.h>
/* pointer to device registers */
volatile struct vdk_device *vdk_device;
vdkstrategy(bp)
struct buf *bp;
{
   int npages;
   register volatile int *sgregisters;
   register int i, v_addr;

   /* Get address of the scatter/gather registers */
```

```
     *sgregisters = vdk_device->sgregisters;

    /* Get the kernel virtual address of the data */
     v_addr = bp->b_un.b_addr;

    /* Compute number of pages received.
     * The dma_len field provides the number of pages to
     * map. Note that this may be larger than the actual
     * number of bytes involved in the transfer. This is
     * because the transfer may cross page boundaries,
     * requiring an extra page to be mapped.*/
     npages = numpages (v_addr, bp->b_bcount);

    /* Translate the virtual address of each page to a
     * physical page number and load it into the next
     * scatter/gather register.  The btoct macro
     * converts the byte value to a page value after
     * rounding down the byte value to a full page.
     */
     for (i = 0; i < npages; i++) {
       *sgregisters++ = btoct(kvtophys(v_addr));

    /*
    /* Get the next virtual address to translate.
     * (NBPC is a symbolic constant for the page
     * size in bytes.)
     */

    v_addr += NBPC;
    }

    /*
     * Provide the beginning byte offset and count to the
     * device.
     */

   vdk_device->offset = (unsigned int)bp->b_dmaaddr & (NBPC-1);
    vdk_device->count = bp->b_bcount;
    if ((bp->b_flags & B_READ) == 0)
       vdk_device->direction = VDK_WRITE;
    else
       vdk_device->direction = VDK_READ;
}
```

Using DMA Maps

On IRIS 4D/100, 4D/200, 4D/300, 4D/400, Crimson, CHALLENGE/Onyx, and POWER CHALLENGE/POWER Onyx series systems, a number of registers perform mapping from physical pages to other physical pages. Because the addresses that are mapped are really no longer "physical" addresses, they are now referred to as "bus virtual" addresses. Your driver should allocate these system mapping registers when it opens the device, remap these registers for every transfer, and then free them when it closes the device.

Internally, all the mapping routines deal with a DMA map structure. The values stored in members of the structure are subject to change from release to release. Therefore, when your driver manipulates the DMA maps, it must use the following routines only. Your driver must not try to access the structure members directly.

dma_mapalloc	Allocate a DMA Map
dma_map	Map a Virtual Address Space
dma_mapaddr	Return the Bus Virtual Address

Note: When using DMA maps, be sure that the source or destination address begins on a 32-bit word boundary.

Caution: Once you free a DMA map, it is gone and you can no longer use it. Free it only after the DMA operation has been successfully aborted.

Example Using DMA Maps

Suppose the mythical VME device is an A24 device for use with an IRIS-4D/100 series workstation. The driver begins the transfer by giving the device the starting address, byte count, and transfer direction. Some driver excerpts could look like the following:

```
#define MAXTRANSFER  4      /* maximum transfer size in pages */
                            /* pointer to device registers */
volatile struct vdk_device*vdk_device;
dmamap_t    vdk_map;         /* pntr to DMA map */
struct buf *vdk_curbp;      /* current buffer */
caddr_t     vdk_curaddr;     /* current address to transfer */
int         vdk_curcount;
```

```
static      int vdk_vmeadap   /* computed during edtinit */

vdkopen(dev, flag, otyp, crp)
dev_t   *dev;
int     flag, otyp;
credit  *crp;
{
...
    vdk_map =
        dma_mapalloc(DMA_A24VME, vdk_vmeadap,
            MAXTRANSFER, 0);
...
}

vdkclose(dev, flag, otyp, crp)
dev_t   dev;
int     flag, otyp;
{
...
        dma_mapfree(vdk_map);
...
}

vdkstrategy(bp)
struct buf *bp
{
...

/* Save structure pointer for the interrupt  routine, vdkintr */
    vdk_curbp = bp;

    /* Set up the mapping registers */
    bp->b_resid = bp->b_bcount;
    vdk_curaddr = bp->b_dmaaddr;
    vdk_curcount = dma_map
        (vdk_map, vdk_curaddr, bp->b_resid);

/* Tell the device starting bus virtual address and count */
    vdk_device->startaddr =
            dma_mapaddr(vdk_map, vdk_curaddr);
    vdk_device->count = count;
    if (bp->b_flags & B_READ) == 0)
        vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
```

```
        vdk_device->command = VDK_GO;
        /* Set up for next transfer */
        vdk_curaddr += count;
        ...
}

vdkintr(unit)
int unit;
{
    int count;
    register struct buf *bp = vdk_curbp;
    ...

    if(error) {
        bp->b_flags |= B_ERROR;
        iodone(bp);
        return( );
    }

    /*On successful transfer of last chunk, continue if necessary.*/
    bp->resid -= vdk_curcount;
    if (bp->b_resid > 0) {
        count = dma_map(vdk_map, vdk_curaddr, bp->b_resid);
        vdk_device->startaddr = dma_mapaddr(vdk_map,vdk_curaddr);
        vdk_device->count = count;
        if (bp->b_flags & B_READ) == 0)
            vdk_device->direction = VDK_WRITE;
        else
            vdk_device->direction = VDK_READ;
        vdk_device->command = VDK_GO;
        vdk_curaddr += count;
    } else {
        biodone(bp);
    }
    ...
}
```

Note: As with other examples, error checking is omitted, but should not be omitted in real code.

DMA on A24 Devices with No DMA Mapping

VME A24 addressing specifies an address space that may be smaller than all of physical memory. Silicon Graphics workstations that support the VME bus provide a DMA mapping capability so that your driver can access all of physical memory.

Some Silicon Graphics systems allow A24 masters to access only the first 8 MB of physical memory. Your driver must declare a static buffer assigned to contiguous physical pages in low memory, and it must do DMA transfers to and from this buffer only. After a transfer is complete, the driver can copy the data from this buffer to the user's memory. Because kernel static data uses contiguous physical memory pages, scatter/gather hardware is not needed. Keep this buffer no more than a few pages long; otherwise, the kernel BSS segment may be too large for the system to boot it. The limits on kernel size vary from system to system and sometimes across releases and other included kernel drivers.

Using a DMA read operation, your driver can transfer data from a device directly to physical memory. Any words in the processor data cache corresponding to this memory are now stale. To invalidate the data cache lines associated with the physical memory addresses, your driver must call the **dki_dcache_inval**() routine. If your driver calls the kernel routine, **physiock**(), it need not call **dki_dcache_inval**() because **physiock**() calls the **userdma**() routine and thus invalidates the data cache.

Using a DMA write operation, your driver can transfer data from memory to the device. Prior to the transfer, any words in the processor data cache corresponding to this memory may be more up-to-date than the corresponding memory. In this case, memory is said to be stale with respect to cache, and any words in the cache corresponding to this memory must be written back to memory before the DMA starts. Use the **dki_dcache_wbinval**() routine to write the contents of the cache back to memory. If your driver calls the kernel routine **physiock**(), it need not call **dki_dcache_wbinval**() because **physiock**() calls the **userdma**() routine and thus writes back and invalidates the data cache.

The driver below does a DMA transfer into memory that has not been prepared by **physiock**(). The driver can do this because the data is in a kernel buffer, so there is no need to lock it in memory and remap it. See

Appendix A, "System-specific Issues," for more information on data cache management.

For example, suppose the mythical VME device is now an A24 master. The driver begins the transfer by programming the starting address, byte count, and transfer direction.

Note: On systems with multiple word cache lines, this buffer must be aligned on a cache line boundary for correct operation. Normally, some extra bytes (currently 128) must be allocated, and a pointer into the buffer, whose address is adjusted to be cache-line aligned, must be used (see *SCACHE_ALIGNED* in *sys/immu.h*).

Alternatively, allocate a buffer during the *drv***edtinit** routine with **kmem_alloc** and the *SCACHE_ALIGN* flag, and verify that the address is in the low 8 MB of physical memory with **kvtophys**().

A driver excerpt follows:

```
#define V DKBUFSIZE 4096 /* kernel buffer size */
/* pointer to device rgstrs */
volatile struct vdk_device *vdk_device;
char vdk_buffer[VDKBUFSIZE] ; /* kernel bufr */
struct buf *vdk_curbp; /* current bufr */
caddr_t vdk_curaddr;   /* current address to transfer */
caddr_t vdk_curcount   /* current count being trnsfrd */

vdkstrategy(buf *bp)
{
    ...
    bp->b_resid = bp->b_bcount;
    vdk_curaddr = bp->b_un.b_addr;
    vdk_curbp = bp;
    vdk_curcount = MIN(bp->b_resid,VDKBUFSIZE);

    /* for a write operation, copy from the user's memory
     * to the kernel buffer
     */
    if( (bp->b_flags & B_READ) == 0 )
        bcopy(vdk_curaddr,vdk_buffer,vdk_curcount);

    /* tell the device the phys address of kernel
     * buffer and count
     */
```

```
    vdk_device->startaddr = kvtophys(vdk_buffer);
    vdk_device->count = vdk_curcount;
    if( (bp->b_flags & B_READ) == 0 )
        vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
    vdk_device->com = VDK _GO;

    ...
}

vdkintr(int unit)
{
    int count; error
    register struct bug *bp = vdk_curbp;

    ...
    /* check for an error */
    if( error ) {
        bioerror(bp,EIO);
        biodone(bp);
        return;
    }
    /* For a read operation, copy the data from the kernel
     * buffer to the user's pages. The bcopy routine
     * must be used with the kernel virtual address of
     * the user's buffer since the user's virtual
     * addresses aren't mapped anymore.
     *
     * Note that the data cache is flushed before
     * copying from a cached address. Ordinarily physiock()
     * does this for you. See Appendix A for when and how
     * to flush the data cache.
     */
    if( (bp->b_flags & B_READ) != 0 ) {
        dki_dcache_flush(vdk_buffer, vdk_curcount);
        bcopy(vdk_buffer, vdk_curaddr, vdk_curcount);
    }

    /* Decrement the residual count and bump up the current
     * transfer address. If there are any bytes left to
     * transfer, do it again.
     */
    bp->b_resid -= vdk_curcount;
    vdk_curaddr += vdk_curcount;
```

```
    if( bp->b_resid == 0 ) {
        biodone(bp);
        return;
    }

    vdk_curcount = MIN(bp->b_resid,VDKBUFSIZE);

    if( (bp->b_flags & B_READ) == 0 )
        bcopy(vdk_curaddr,vdk_buffer,vdk_curcount);

    vdk_device->startaddr = kvtophys(vdk_buffer);
    vdk_device->count = vdk_curcount;
    if( (bp->b_flags & B_READ) == 0 )
        vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
    vdk_device->com = VKD_GO;
}
```

DMA on A32 Devices with No Scatter/Gather Capability

If neither your device nor your workstation provides scatter/gather capability, your driver must break up a data transfer so that no transfer crosses a page boundary. The IRIX operating system provides a utility called sgset(D3X), which simulates scatter/gather registers in software. It should not be used on systems that support DMA address mapping. (See the *IRIX Device Driver Reference Pages* for details on this routine.) Your driver can use this utility to perform the virtual-to-physical mapping up front. Or, as the example below shows, your driver can do this mapping following the transfer of each page:

```
/* pointer to device registers */
volatile struct vdk_device    *vdk_device;
struct buf   *vdk_curbp      /* current buffer */
caddr_t      vdk_curaddr;    /* current address to transfer
*/
int          vdk_curcount;
vdkstrategy(bp)
struct buf    *bp;
{
    ...
    vdk_curbp = bp;
    bp->b_resid = bp->b_bcount;
    /*
     * Initialize the current transfer address and count.
     * The first transfer must finish the rest of the
     * page, but do no more than the total byte count.
     */
    vdk_curaddr = bp->b_un.b_addr;
    vdk_curcount = NBPC -
        ((unsigned int)vdk_curaddr & (NBPC-1));
    if (bp->b_resid < vdk_curcount)
        vdk_curcount = bp->b_resid;
    /* Tell the device starting physical address, count,
     * and direction */
    vdk_device->startaddr = kvtophys(vdk_curaddr);
    vdk_device->count = vdk_curcount;
    if (bp->b_flags & B_READ) == 0)
    vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
    vdk_device->command = VDK_GO;
    vdk_curaddr += vdk_curcount;
```

```
    biowait(bp);
    ...
}
vdkintr(unit)
int unit;
{
    int count, error;
    register struct buf *bp = vdk_curbp;
    ...
    if(error) {
        bioerror (bp,EIO);
        biodone(bp);
        return;
    }
    /* On successful transfer of last chunk, continue
     * if necessary. */
    vdk_curaddr -= vdk_curcount;
    if (bp->b_resid > 0) {
            count =
                (bp->b_resid < NBPC ? bp->b_resid : NBPC);
            vdk_device->startaddr = kvtophys(vdk_curaddr);
            vdk_device->count = count;
            if (bp->b_flags & B_READ) == 0)
                vdk_device->direction = VDK_WRITE;
            else
                vdk_device->direction = VDK_READ;
            vdk_device->command = VDK_GO;
            vdk_curaddr += count;
    } else {
            biodone(bp);
    }
    ...
}
```