Chapter 4

User-Level Access to VME and EISA

Programmed I/O (PIO) refers to loading and storing data between program variables (actually, CPU registers) and device registers. This is done by setting up a memory mapping of a VME or EISA device into the process address space, so that the program can treat device registers as if they were volatile memory locations. This chapter discusses the methods of setting up this mapping, and the performance that can be obtained. The main topics are as follows:

- "VME Programmed I/O" on page 62 discusses PIO mapping of VME devices.
- "EISA Programmed I/O" on page 67 discusses PIO mapping of EISA devices.
- "VME User-Level DMA" on page 70 discusses the use of the DMA engine in a Challenge or Onyx system.

Normally, PIO programs are designed in synchronous fashion; that is, the process issues commands to the device and then polls the device to find out when the action is complete. However, it is possible for a user process to receive interrupts from a mapped VME device—see Chapter 7, "User-Level Interrupts."

A user-level process can perform DMA transfers from a VME bus master or (in the Challenge or Onyx series) a VME bus slave, directly into the process address space. The use of these features is covered under "VME User-Level DMA" on page 70.

VME Programmed I/O

For an overview of the VME bus and its hardware implementation in Silicon Graphics systems, see Chapter 13, "VME Device Attachment."

Mapping a VME Device Into Process Address Space

As discussed in "Device Addresses" on page 4, an I/O device is represented as an address, or range of addresses, in the physical address space. A kernel-level device driver has the ability to set up a mapping between the address of a device register and an arbitrary location in the address space of a user-level process. When this has been done, the device register appears to be a variable in memory. The program can assign values to it, or refer to it in expressions.

Learning VME Device Addresses

In order to map a VME device for PIO, you must know the following points:

- the VME bus number on which the device resides

  The Crimson series supports two VME busses, numbered 0 and 1.

  Challenge and Onyx systems support as many as five VME buses. The first is number 0. Use the *hinv* command to display the numbers of others (and see "VME Bus Numbering" on page 315).
- the VME address space in which the device resides

  This will be either A16, A24, or A32—the A64 space is not supported for PIO.
- the VME address space modifier the device uses—either supervisory (s) or nonprivileged (n)
- the VME bus addresses associated with the device

  This must be a sequential range of VME bus addresses that spans all the device registers you need to map.

This information is normally supplied by the manufacturer of a third-party VME device. You can find these values for Silicon Graphics equipment by examining the */var/sysgen/system/irix.sm* file, in which each configured VME device is specified by a VECTOR line. When you examine a VECTOR line, note the following parameter values:

bustype — Specified as *VME* for VME devices. The VECTOR statement can be used for other types of buses as well.

adapter — The number of the VME bus where the device is attached.

iospace, *iospace2*, *iospace3* — Each *iospace* group specifies the VME address space and modifier, the starting bus address, and the size of a segment of VME address space used by this device.

Within each *iospace* parameter group you find keywords and numbers for the address space, modifier, and addresses for a device. The following is an example of a VECTOR line:

```
VECTOR: bustype=VME module=cdsio ipl=5 ctlr=0 adapter=0
iospace=(A24S,0xF00000,0x10000) probe_space=(A24S,0xF0FFFF,1)
```

This example specifies a VME device (*bustype=VME*) on bus 0 (*adapter=0*). The device resides in the A24 address space in supervisory mode (*iospace=(A24S...)*). Its first VME bus address is 0xF0 0000 and it covers a span of 0x01 0000 (64K) addresses—in other words, 0xF0 0000 through 0xF0 FFFF.

For third-party VME devices, look for a VECTOR line supplied by the manufacturer, usually stored in some file in */var/sysgen/system*.

Opening a Device Special File

When you know the device addresses, you can open a device special file that represents the correct range of addresses. The device special files for VME mapping are found in */dev/vme*.

The naming convention for these files is documented in the usrvme(7) reference page. Briefly, each file is named **vme***B***a***SM*, where

B — is one or two digits for the bus number, for example 0 or 53

S — is two digits for the address space, 16, 24, or 32

M — is the modifier, either *s* for supervisory or *n* for nonprivileged

The device special file for the device described by the example VECTOR line in the preceding section would be */dev/vme/vme0a24s*.

In order to map a device on a particular bus and address space, you must open the corresponding file in */dev/vme*.

Using the mmap() Function

When you have successfully opened the device special file, you use the file descriptor as the primary input parameter in a call to the **mmap()** system function.

This function is documented for all its many uses in the mmap(2) reference page. For purposes of mapping a VME device into memory, the parameters should be as follows (using the names from the reference page):

addr Should be NULL to permit the kernel to choose an address in user process space.

len The length of the span of VME addresses, as documented in the *iospace* group in the VECTOR line.

prot PROT_READ for input, PROT_WRITE for output, or the logical sum of those names when the device will be used for both input and output.

flags MAP_SHARED, with the addition of MAP_PRIVATE if this mapping is not to be visible to child processes created with the **sproc()** function (see the sproc(2) reference page).

fd The file descriptor returned from opening the device special file in */dev/vme*.

off The starting VME bus address, as documented in the *iospace* group in the VECTOR line.

The value returned by **mmap()** is the virtual address that corresponds to the starting VME bus address. When the process accesses that address, the access is implemented by data transfer to the VME bus.

Map Size Limits

There are limits to the amount and location of VME bus address space that can be mapped for PIO. The system architecture can restrict the span of mappable addresses, and kernel resource constraints can impose limits.

In all systems that support the VME bus it is possible to map all of A16 space.

In a Silicon Graphics Crimson system, the upper 8 MB of A24 space, from 0x80 0000 to 0xFF FFFF can be mapped. Only a fixed part of the A32 space can be mapped. A total of 256 MB of addresses is divided between the supervisor and nonprivileged A32 spaces on one or two VME buses. For details of the VME bus addresses, see "Crimson Mapping of A32 Space" on page 307. VME devices that respond to addresses in the ranges shown in that section can be mapped for PIO.

In the Silicon Graphics Challenge and Onyx systems, all of A24 and A32 space can be used for PIO mappings, but there is a limit on the size of each map. Each bus mapping uses a hardware register that can span as much as 8 MB of contiguous VME bus addresses—so a single **mmap()** call can map at most 8 MB. There are as many as 12 mapping registers for each bus, so by making successive **mmap()** calls for adjacent 8 MB blocks of VME space you can map up to 96 MB of VME space into user process space from a single bus.

VME PIO Access

Once a VME device has been mapped into memory, your program reads from the device by referencing the mapped address, and writes to the device by storing into the mapped address. Example 4-1 displays a sketch of a hypothetical function that maps a device and copies one register into another.

Example 4-1 Opening and Using a Hypothetical VME Device

```
#define SPECFILE "/dev/vme/vme1a16n"
typedef unsigned short int busdata; /* device data item */
typedef volatile busdata busreg;    /* device register */
#define MAPSIZE 8*sizeof(vmereg)
#define BUSADDR 0xff00
int vmefunc()
{
   busreg *mapped;
   busdata sample;
   int specfd = open(SPECFILE,O_RDWR);
   if (-1 == specfd) return error;
   mapped = mmap(NULL,      /* kernel pick address */
                 REGSIZE,   /* size of mapped area */
                 PROT_READ|PROT_WRITE, /* protection flags */
                 MAP_SHARED, /* mapping flags */
                 specfd,     /* special file */
```

```
                 BUSADDR)    /* file offset */
   if (!mapped) return error;
   sample = busreg[0];       /* read A16N at 0xff00 */
   busreg[1] = sample;       /* write A16N at 0xff02 */
}
```

A PIO read is synchronous at the hardware level. The CPU executes a register-load instruction that does not complete until data has been returned from the device, up the system bus, to the CPU (see “CPU Access to Device Registers” on page 10). This can take 1 or 2 microseconds in a Challenge system.

A PIO write is not necessarily synchronous at the hardware level. The CPU executes a register-store instruction that is complete as soon as the physical address and data have been placed on the system bus. The actual VME write operation on the VME bus can take 1 or more microseconds to complete. During that time the CPU can execute dozens or even hundreds more instructions from cache memory.

VME PIO Bandwidth

On a Challenge L or Onyx system, the maximum rate of PIO output is approximately 750K writes per second. The maximum rate of PIO input is approximately 250K reads per second. The corresponding data rate depends on the number of bytes transferred on each operation, as summarized in Table 4-1.

Table 4-1 VME Bus PIO Bandwidth

Data Unit Size	Read	Write
D8	0.25 MB/second	0.75 MB/second
D16	0.5 MB/second	1.5 MB/second
D32	1 MB/second	3 MB/second

Note: The numbers in Table 4-1 were obtained by doing continuous reads, or continuous writes, to a device in the Challenge chassis. When reads and writes alternate, add approximately 1 microsecond for each change of direction. The use of a repeater to extend to an external card cage would add 200 nanoseconds or more to each transfer.

EISA Programmed I/O

For an overview of the EISA bus and its implementation in Silicon Graphics systems, see Chapter 17, "EISA Device Drivers."

Mapping an EISA Device Into Memory

As discussed in "Device Addresses" on page 4, an I/O device is represented as an address or range of addresses in the physical address space. A kernel-level device driver has the ability to set up a mapping between the physical address of a device register and an arbitrary location in the address space of a user-level process. When this has been done, the device register appears to be a variable in memory—the program can assign values to it, or refer to it in expressions.

Learning EISA Device Addresses

In order to map an EISA device for PIO, you must know the following points:

- which EISA bus adapter the device is on

  In all present Silicon Graphics systems, there is only one EISA bus adapter, and its number is 0.

- whether you need access to the EISA bus memory or I/O address space
- the address and length of the desired registers within the address space

You can find all these values by examining files in the */var/sysgen/system* directory, especially the */var/sysgen/system/irix.sm* file, in which each configured EISA device is specified by a VECTOR line. When you examine a VECTOR line, note the following parameter values:

bustype — Specified as *EISA* for EISA devices. The VECTOR statement can be used for other types of buses as well.

adapter — The number of the bus where the device is attached (0).

iospace, *iospace2*, *iospace3* — Each *iospace* group specifies the address space, starting bus address, and the size of a segment of bus address space used by this device.

Within each *iospace* parameter group you find keywords and numbers for the address space and addresses for a device. The following is an example of a VECTOR line (which must be a single physical line in the system file):

```
VECTOR: bustype=EISA module=if_ec3 ctlr=1
iospace=(EISAIO,0x1000,0x1000)
exprobe_space=(r,EISAIO, 0x1c80,4,0x6010d425,0xffffffff)
```

This example specifies a device that resides in the I/O space at offset 0x1000 (the slot-1 I/O space) for the usual length of 0x1000 bytes. The *exprobe_space* parameter suggests that a key device register is at 0x1c80.

Opening a Device Special File

When you know the device addresses, you can open a device special file that represents the correct range of addresses. The device special files for EISA mapping are found in */dev/eisa*.

The naming convention for these files is as follows: Each file is named **eisa***B***a***M*, where

B is a digit for the bus number (0)

M is the modifier, either *io* or *mem*

The device special file for the device described by the example VECTOR line in the preceding section would be */dev/vme/eisa0aio*.

In order to map a device on a particular bus and address space, you must open the corresponding file in */dev/eisa*.

Using the mmap() Function

When you have successfully opened the device special file, you use the file descriptor as the primary input parameter in a call to the **mmap()** system function.

This function is documented for all its many uses in the mmap(2) reference page. For purposes of mapping EISA devices, the parameters should be as follows (using the names from the reference page):

addr Should be NULL to permit the kernel to choose an address in user process space.

len — The length of the span of bus addresses, as documented in the *iospace* group in the VECTOR line.

prot — PROT_READ, or PROT_WRITE, or the logical sum of those names when the device is used for both input and output.

flags — MAP_SHARED, with the addition of MAP_PRIVATE if this mapping is not to be visible to child processes created with the **sproc()** function (see the sproc(2) reference page).

fd — The file descriptor from opening the device special file in */dev/eisa*.

off — The starting bus address, as documented in the *iospace* group in the VECTOR line.

The value returned by **mmap()** is the virtual memory address that corresponds to the starting bus address. When the process accesses that address, the access is implemented by data transfer to the EISA bus.

Note: When programming EISA PIO, you must always be aware that EISA devices generally store 16-bit and 32-bit values in “small-endian” order, with the least-significant byte at the lowest address. This is opposite to the order used by the MIPS CPU under IRIX. If you simply assign to a C unsigned integer from a 32-bit EISA register, the value will appear to be byte-inverted.

EISA PIO Bandwidth

The EISA bus adapter is a device on the GIO bus. The GIO bus runs at either 25 MHz or 33 MHz, depending on the system model. Each EISA device access takes multiple GIO cycles, as follows:

- The base time to do a native GIO read (of up to 64 bits) is 1 microsecond.
- A 32-bit EISA slave read adds 15 GIO cycles to the base GIO read time; hence one EISA access takes 19 GIO cycles, best case.
- A 4-byte access to a 16-bit EISA device requires 10 more GIO cycles to transfer the second 2-byte group; hence a 4-byte read to a 16-bit EISA slave requires 25 GIO cycles.
- Each wait state inserted by the EISA device adds four GIO cycles.

Table 4-2 summarizes best-case (no EISA wait states) data rates for reading and writing a 32-bit EISA device, based on these considerations.

Table 4-2 EISA Bus PIO Bandwidth (32-Bit Slave, 33-MHz GIO Clock)

Data Unit Size	Read	Write
1 byte	0.68 MB/sec	1.75 MB/sec
2 byte	1.38 MB/sec	3.51 MB/sec
4 bytes	2.76 MB/sec	7.02 MB/sec

Table 4-3 summarizes the best-case (no wait state) data rates for reading and writing a 16-bit EISA device.

Table 4-3 EISA Bus PIO Bandwidth (16-Bit Slave, 33-MHz GIO Clock)

Data Unit Size	Read	Write
1 byte	0.68 MB/sec	1.75 MB/sec
2 byte	1.38 MB/sec	3.51 MB/sec
4 bytes	2.29 MB/sec	4.59 MB/sec

VME User-Level DMA

A DMA engine is included as part of each VME bus adapter in a Challenge or Onyx system. The DMA engine is unique to the Challenge architecture. It performs efficient, block-mode, DMA transfers between system memory and VME bus slave cards—cards that would normally be capable of only PIO transfers.

The DMA engine greatly increases the rate of data transfer compared to PIO, provided that you transfer at least 32 contiguous bytes at a time. The DMA engine can perform D8, D16, D32, D32 Block, and D64 Block data transfers in the A16, A24, and A32 bus address spaces.

Using the udmalib Functions

All DMA engine transfers are initiated by a special device driver. However, you do not access this driver through open/read/write system functions. Instead, you program it through a library of functions. The functions are documented in the udmalib(3) reference page. They are used in the following sequence:

1. Call **dma_open()** to initialize action to a particular VME card.
2. Call **dma_allocbuf()** to allocate storage to use for DMA buffers.
3. Call **dma_mkparms()** to create a descriptor for an operation, including the buffer, the length, and the direction of transfer.
4. Call **dma_start()** to execute each transfer. This function does not return until the transfer is complete.

The **dma_start()** function takes the VME bus address of the particular slave device register that will provide or accept a series of data items. Before starting a DMA transfer, and possibly between transfers, you may need to program the VME controller with other commands. You would do this using PIO (see “VME Programmed I/O” on page 62).

Tip: The **dma_start()** function operates synchronously, polling the VME adapter hardware to find out when the DMA transfer is complete. In order to get parallel execution, consider calling **dma_start()** from a separate process.

Buffer Allocation for User DMA

A buffer allocated by **dma_allocbuf()** is rounded up to a multiple of the memory page size, and is locked in memory to avoid page-faults during the DMA transfer. There is some overhead in creating a buffer, so for best performance the program should allocate the required buffers of the necessary size during initialization. However, if the total size of the buffers is a significant fraction of the available real memory, the large number of locked pages can hurt system performance.

You can only use the allocated buffer for DMA; it is not possible to provide your own buffer (for example, a buffer in a shared memory arena) for use by the DMA engine. When the data is produced by one process and written by another, this design can mean that the data has to be copied from an application buffer to the DMA buffer.

Tip: One way to avoid copying is to call **dma_allocbuf()** early, during program setup, before creating subprocesses using **sproc()**. Processes made with **sproc()** share their parent process address space, including buffer space created by **dma_allocbuf()**, so you can have a process generating or consuming data in one part of the allocated buffer space while a different process executes **dma_start()** to write or read data in a different part. It is of course essential to synchronize the use of the different buffer segments so that each area is used by only one process at a time.

Allocation of Descriptors

Each call to **dma_mkparms()** allocates a small block of memory and fills it with constants that describe a particular transfer operation. The primary input to **dma_mkparms()** is a *vme_parms_t* object (declared in *udmalib.h*) containing the following important fields:

vp_block — 1 for a block-mode transfer; 0 for a normal transfer.

vp_datumsz — The width of transfer units: VME_DS_BYTE 8-bit transfers) VME_DS_HALFWORD (16-bit transfers) VME_DS_WORD (32-bit transfers), or VME_DS_DBLWORD (64-bit transfers).

vp_dir — The direction of transfer: either VME_READ (from the VME bus) or VME_WRITE (to the VME bus).

vp_throt — For a block-mode transfer, the number of bytes to transfer in one burst without yielding the bus. Set to either VME_THROT_256 (the usual size, supported by most VME slaves that allow block transfer) or VME_THROT_2048 to allow up to 2,048 bytes per burst.

vp_release — The bus arbitration mode: VME_REL_RWD to release the bus as soon as the transfer is over, or VME_REL_ROR to release only when another bus master wants the bus. Use VME_REL_ROR for best speed when no bus masters are on the bus. Use VME_REL_RWD when other bus masters may be present and active.

vp_addrmod — The address modifier value that selects the target VME address space. Names of the form VME_*AMOD are declared for these values in *sys/vmereg.h*, for example VME_A16NPAMOD for the nonpriviliged A16 space.

During initialization you call **dma_mkparms()** to create a descriptor for each unique combination of parameters that your program will use. In each call to **dma_start()**, you pass the descriptor that contains the appropriate set of parameters. A descriptor can be used in multiple **dma_start()** calls.

Advantages of User DMA

The **dma_start()** function and its related functions operate in user space; they do not make system calls to the kernel. This has two important effects. First, overhead is reduced, since there are no mode switches between user and kernel, as there are for **read()** and **write()**. This is important, because the DMA engine is often used for frequent, small inputs and outputs.

Second, **dma_start()** does not block the calling process, in the sense of suspending it and possibly allowing another process to use the CPU. It waits in a test loop until the operation is complete. As you can infer from Table 4-4, typical transfer times range from 50 to 250 microseconds. You can calculate the approximate duration of a call to **dma_start()** based on the amount of data and the operational mode.

You can use the udmalib functions to access a VME Bus Master device, if the device can respond in slave mode. However, this would normally be less efficient than using the Master device's own DMA circuitry.

While you can initiate only one DMA engine transfer per bus, it is possible to program a DMA engine transfer from each bus in the system, concurrently.

DMA Engine Bandwidth

The maximum performance of the DMA engine for D32 transfers is summarized in Table 4-4. Performance with D64 Block transfers is somewhat less than twice the rate shown in Table 4-4. Transfers for larger sizes are faster because the setup time is amortized over a greater number of bytes.

Table 4-4 VME Bus Bandwidth, DMA Engine, D32 Transfer

Transfer Size	Read	Write	Block Read	Block Write
32	2.8 MB/sec	2.6 MB/sec	2.7 MB/sec	2.7 MB/sec
64	3.8 MB/sec	3.8 MB/sec	4.0 MB/sec	3.9 MB/sec
128	5.0 MB/sec	5.3 MB/sec	5.6 MB/sec	5.8 MB/sec
256	6.0 MB/sec	6.7 MB/sec	6.4 MB/sec	7.3 MB/sec
512	6.4 MB/sec	7.7 MB/sec	7.0 MB/sec	8.0 MB/sec

Table 4-4 (continued) VME Bus Bandwidth, DMA Engine, D32 Transfer

Transfer Size	Read	Write	Block Read	Block Write
1024	6.8 MB/sec	8.0 MB/sec	7.5 MB/sec	8.8 MB/sec
2048	7.0 MB/sec	8.4 MB/sec	7.8 MB/sec	9.2 MB/sec
4096	7.1 MB/sec	8.7 MB/sec	7.9 MB/sec	9.4 MB/sec

Note: The throughput that can be achieved in VME DMA is very sensitive to several factors:

- The other activity on the VME bus.
- The blocksize (larger is better).
- Other overhead in the loop requesting DMA operations.

  The loop used to generate the figures in Table 4-4 contained no activity except calls to **dma_start()**.
- the response time of the target VME board to a read or write request, in particular the time from when the VME adapter raises Data Strobe (DS) and the time the slave device raises Data Acknowledge (DTACK).

  For example, if the slave device takes 500 ns to raise DTACK, there will always be fewer than 2 M data transfers per second.

Example User DMA Function

The hypothetical function displayed in Example 4-2 is called to perform a series of DMA transfers from a specified device address. The code does not reflect any device initialization; all setup is presumably done by the caller. The code does not show the processing of the data, which is done in the hypothetical function **processOneBuffer()**.

Example 4-2 User-Level DMA Access to VME

```
/*
|| This function assumes that any device programming needed to
|| prepare the device for input has been done (using PIO) before
|| the function is called.  The bus number and device address
|| are function parameters.
*/
```

```
#define BLOCK_SIZE_TO_USE 4096
#include <udmalib.h>
#include <sys/vmereg.h>

extern void processOneBuffer(void *pBuffer);

int
readBlocks(int iBusNum, __uint32_t uiDevAddress)
{
    udmaid_t   *hEngine;    /* handle returned by dma_open */
    void       *pDMAbuffer; /* pointer from dma_allocbuf */
    vmeparms_t sParms;      /* operation parms for dma_mkparms */
    udmaprm_t  *hParms;     /* handle returned by dma_mkparms */
    int        iStartCode;  /* return code of dma_start */
/*
|| Open the DMA engine. Terminate if it won't open.
*/
    hEngine = dma_open(DMA_VMEBUS, iBusNum);
    if (!hEngine)
    {
        perror("dma_open");
        return(-1);
    }
/*
|| Allocate a special buffer for I/O. If that fails,
|| release the engine and terminate.
*/
    hDMAbuffer = dma_allocbuf(hEngine, BLOCK_SIZE_TO_USE);
    if (!hDMAbuffer)
    {
        perror("dma_allocbuf");
        dma_close(hEngine);
        return(-2);
    }
/*
|| Set up the VME parameters and "make" them.  A different set
|| of parameters is needed for each combination of vmeparms_t
|| values, buffer, and size.  This example uses only one set.
*/
    sParms.vp_block = 0;             /* this device does not do blocks */
    sParms.vp_datumsz = VME_DS_WORD; /* this is a 32-bit device */
    sParms.vp_dir = VME_READ;        /* input operation */
    sParms.vp_throt = VME_THROT_256; /* smaller burst size */
    sParms.vp_release = VME_REL_ROR; /* release on request */
    sParms.vp_addrmod = VME_A32NPAMOD; /* address modifier */
```

```
    hParms = dma_mkparms(hEngine, &sParms, pDMAbuffer, BLOCK_SIZE_TO_USE);
    if (!hParms)
    {
        perror("dma_mkparms");
        dma_freebuf(pDMAbuffer);
        dma_close(hEngine);
        return(-3);
    }
/*
|| Read and process blocks until error.
*/
    for(iStartCode=0;iStartCode==0;)
    {
         iStartCode = dms_start(hEngine, uiDevAddress, hParms);
         if (!iStartCode)
            processOneBuffer(pDMAbuffer);
    }
/*
|| Clean up and exit.
*/
    dma_freeparms(hEngine, hParms);
    dma_freebuf(pDMAbuffer);
    dma_close(hEngine);
    return 0;
}
```