System-specific Issues

This appendix lists the Silicon Graphics functions available for writing device drivers and how those function differ from the functions listed in the *IRIX Device Driver Reference Pages* .

It contains the following sections:

- "CPU Types" on page 320
- "Data Cache Write Back and Invalidation" on page 321
- "Flushing the Write Buffer" on page 324
- "Reliable Multiprocessor Spinlocks" on page 324
- "VME Slave Addressing" on page 325
- "VME Master Addressing" on page 326
- "VME-bus Space Reserved for Customer Drivers" on page 329
- "POWER Indigo2 and POWER CHALLENGE M Drivers" on page 330

Although all Silicon Graphics systems share similar architectural elements, there are several significant differences you must recognize when writing device drivers. Despite these differences, it is possible to write a device driver that runs on all Silicon Graphics systems. This appendix outlines the various CPU types used by Silicon Graphics systems and describes the CPU features that can vary. Also listed are the VME-bus addresses and interrupt vectors available for customer use.

Whenever possible, this appendix promotes the use of those IRIX kernel functions that are supported on all Silicon Graphics architectures. The hardware features that can differ across architectures are:

- Data cache write back and invalidation
- Write buffer flushing
- Hardware spinlocks (test and set variables)

CPU Types

This appendix describes the CPU types used in Silicon Graphics workstations and servers. These CPU types are summarized in Table A-1.

Table A-1 CPUs Used in Silicon Graphics Computer Systems

CPU Type	Processor	Clock Speed	System/Series
IP4	R2000	8/12.5MHz 33 MHz	4D/50, 4D/70, and 4D/80 systems (Originally known as Single-Board Computer or "SBC")
IP5	R2000	16.7 MHz	4D/100 series multiprocessor systems
IP6	R2000	12.5/20 MHz	Personal IRIS (4D/20, 4D/25) systems
IP7	R3000	25/33 MHz	4D/200 series multiprocessor systems
IP9	R3000	25MHz	4D/210 series systems
IP11	R3000	33 MHz	4D/300 series systems
IP12	R3000	30 MHz	4D/30, 4D/35 and older Indigo series systems. Indigo series systems have no VME interface, but they do support GIO.
IP15	R3000	40 MHz	4D/400 series systems
IP17	R4000 R4400	100 MHz 150MHz	Crimson series single-processor systems
IP19	R4400	150 MHz	CHALLENGE L/XL and Onyx multiprocessor systems
IP20	R4000	100 MHz	Newer Indigo series systems
IP21	R8000	75 MHz	POWER CHALLENGE and POWER/Onyx multiprocessor systems
IP22	R4000 R4000 R4400 R4400 R4400 R4600	100 MHz 100 MHz 150 MHz 150MHz 150MHz 133 MHz	Indigo^2 systems (with GIO and EISA) Indy systems Indigo^2 systems (with GIO and EISA) Indy systems (with GIO but not EISA) Indigo systems Indigo systems
IP26	R8000	75 MHz	POWER CHALLENGE M and POWER Indigo^2 systems

To determine which CPU a Silicon Graphics system uses, type:

```
% uname -m
```

at a shell prompt. This command reports the CPU type. For details on this command, see the **uname**(1) man page.

For a more detailed listing of a Silicon Graphics system's hardware configuration, type:

```
% hinv
```

at a shell prompt. See the **hinv**(1M) man page.

Data Cache Write Back and Invalidation

All CPUs use the MIPS R2000, R3000, R4000, or R8000 (formerly known as "TFP") series processor chips (R4400 and R4600 are the same as the R4000 for all practical purposes). These chips use a data cache to maximize the efficiency of fetching of heavily used memory.

The IP5, IP7, IP11, IP15, IP19, and IP21 multiprocessor CPUs have *bus-watching caches* that automatically invalidate the data cache when the system performs DMA (direct memory access) into physical memory. For these CPUs, no data cache write back or invalidation is required by software because these functions are performed by hardware.

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 from the point of view of what happens to memory.

The single-processor CPUs based on the R2000 or R3000 employ a cache architecture known as a *write through cache*. This means that all stores generated by the processor to a cached memory location go into the cache and into memory at the same time. Therefore, the data caches on these systems never contain data that is more recent than memory. However, after the system performs DMA to physical memory, the data lines in the processor's cache corresponding to this physical memory contain data that is *stale* with respect to memory. Therefore, a driver running on a single-processor CPU must explicitly invalidate the data cache before reading from the corresponding cached address, after the DMA completes.

R4000s and R8000s employ a cache architecture known as a *write back cache*. This means that stores generated by the processor go only into cache; they are written back from cache to memory only when a cache miss causes that cache line to be replaced. For this type of cache, the cache contains data that is newer than the corresponding memory locations. Memory is then *stale* with respect to the cache. Drivers that perform DMA reads from memory to device must specifically cause the cache to be written back to memory before the DMA starts. On IP19 and IP21 platforms, DMA pulls data from the processor caches, if necessary, thus providing *coherent I/O*.

Recall the code examples that read kernel data in Chapter 3, "Writing a VME Device Driver," and Chapter 5, "Writing a SCSI Device Driver." Before reading the data, the driver code used the **dki_dcache_inval**() function to invalidate the appropriate data cache lines. When the data cache lines are invalidated, accessing the kernel data causes a cache miss and, thus, forces a read from physical memory. Therefore, to ensure driver portability, your driver must always use **dki_dcache_inval**() to invalidate the data cache. This is the case even though the **dki_dcache_inval**() functions defined for the IP5 and IP7 use stub functions that do not do any actual work, although they use **dki_dcache_wb**() before starting a DMA from memory to device. See Table A-2 for a summary of cache line sizes for various MIPS processors.

If your driver uses the functions **userdma**() or **physio**() (**physio** calls **userdma**() internally), the data cache is automatically written back and invalidated for you no matter what system you are using. If your driver does not use these functions for a DMA write into cached memory, your driver must use **dki_dcache_inval**() to invalidate the data cache explicitly after the DMA completes.[1] Further, if your driver does a DMA read from memory to device, it must use **dki_dcache_wb**() to write back the data cache explicitly before the DMA is started.

---

[1] On systems with write through caches, and on IP5, IP7, IP9, IP11, and so on, the **dki_dcache_inval**() functions are stub functions that perform no actual work.

Table A-2 Cache Line Sizes by Processor Type

	Primary Cache			Secondary Cache		
Processor Type	**Size (D/I)**	**Type**	**Line Size (D/I)**	**Size (D/I)**	**Type**	**Line Size (D/I)**
R3000 (IP12)	32K/32K	D	4/64	None	None	None
R3000 (IP7)	64K/64K	D	4/64	None	None	None
R4000PC (IP20)	8K/8K	D	32/32	None	None	None
R4000SC (IP17)	8K/8K	D	32/32	1-4 MB	D	128/128
R4400MP (IP19)	16K/16K	D	32/32	1-4 MB	D	128/128
R4600PC (IP22)	16K/16K	2	32/32	None	None	None
R4600SC (IP22)	16K/16K	2	32/32	512K	2	128
R8000 (IP21)	16K/16K	D	32/32	4 MB	4	512
R8000 (IP26)	16K/16K	D	32/32	2 MB	4	128

Another consideration worth mentioning is that of buffer alignment for DMA. The R4000 processor implements a secondary cache line size of between 4 and 32 words (the secondary cache line size is dependent upon the CPU board implementation). Buffers used for DMA must be aligned on a byte boundary that is equal to the cache line size. To accomplish this, use the **kmem_alloc**() function with the *KM_CACHEALIGN* flag. This returns a buffer with the necessary alignment for the system.

Note: The R8000 has the same DMA alignment problems in general as the R4000. This is true for all systems with write back caches.

Why is this alignment necessary? Suppose you have a variable, *X*, followed by a buffer you are going to use for DMA write. If you invalidate the buffer prior to the DMA write, but then reference the variable *X*, the resulting cache miss brings part of the buffer back into the cache. When the DMA write completes, the cache is stale with respect to memory. If, however, you invalidate the cache after the DMA write completes, you destroy the value of the variable *X*.

Flushing the Write Buffer

You may, in some cases, want to call **flushbus**() to ensure that any writes in the write buffer have actually been flushed to the system bus. This is sometimes necessary when a device requires delays between PIOs, particularly between a write and a read, since they might otherwise arrive at the device back-to-back.

For example, you write to the device, delay, write to the device, delay. These writes may be buffered regardless of the delay and still be sent to the device in quick succession.

Registers and Register Optimization

Variables not declared volatile may be optimized to registers in the processor, since there are multiple processors. Multiple inconsistent copies of a variable may exist in registers if volatile is not declared.

Note: Use the *-O* compiler flag to turn optimization on. The *-g* compiler flag for debugging disables optimization.

Reliable Multiprocessor Spinlocks

The multiprocessor CPUs implement test-and-set variables in hardware. These variables are known as *spinlocks*. Your code can use functions such as **LOCK_ALLOC**(D3) and **LOCK**(D3) to take advantage of these variables to protect a critical region of code or data on one processor from interference from other processors.

All kernels for all Silicon Graphics CPUs support the spinlock functions (although these functions perform no actual work on single-processor systems). Therefore, a fully semaphored multiprocessor device driver can run unmodified on a single-processor system.

Note: Because IRIX kernels cannot, as a rule, be preempted, any driver that sits in a loop waiting for some condition to be satisfied may tie a processor up for as long as it wants. Real-time processes, such as audio, are very sensitive to such delays.

VME Slave Addressing

Recall that these base addresses are passed to the driver through the device **edtinit**() function. Therefore, while you may need a different *system* file for each Silicon Graphics platform, the driver code itself can still be portable.

Table A-3, Table A-4, and Table A-5 show the mapping of kernel virtual address, physical address, and VME-bus address for each addressing mode. For IP6 and IP12 systems, block mode and burst mode transfers are supported only during DMA, where the VME device is the master. No predefined addresses are available on CHALLENGE/Onyx series systems.

Note: When consulting Table A-3 through Table A-7, note that VME space is set up by software on various hardware platforms. Always use the macro #defines in */usr/include/sys/vmereg.h* to refer to the various VME resource address ranges by their logical names.

Table A-3 A24 Kernel/VME-bus Address Mapping

System CPU	VME Modifier	Kernel Address VME0	Kernel Address VME1	Size (MB)	VME Address Range
IP4/6/12	0x3D	0xBC000000 – 0xBCFFFFFF	NA	16	0x0 – 0xFFFFFF
IP4/6/12	0x39	0xBE000000 – 0xBEFFFFFF	NA	16	0x0 – 0xFFFFFF
IP5/7/9/17	0x39	0xB2000000 – 0xB2FFFFFF	F0000000 – F0FFFFFF	16	0x0 – 0xFFFFFF
IP5/7/9/17	0x3D	0xB3000000 – 0xB3FFFFFF	F1000000 – F1FFFFFF	16	0x0 – 0xFFFFFF

Table A-4 A32 Kernel/VME-bus Address Mapping

System CPU	VME Modifier	Kernel Address	Size (MB)	VME Address Range
R2300	0x09	0xB8000000 – 0xBBFFFFFF	192	0x18000000 – 0x1BFFFFFF
IP4/6/12	0x09	0xB0000000 – 0xBBFFFFFF	192	0x10000000 – 0x1BFFFFFF
IP5/7/9/17	0x09	0xD0000000 – 0xDFFFFFFF	256	0x10000000 – 0x1FFFFFFF
IP5/7/9/17	0x0D	0xE0000000 – 0xEFFFFFFF	256	0x10000000 – 0x1FFFFFFF

Table A-5 Dual VME-bus Address Mapping

System CPU	VME Modifier	Kernel Address VME0	Kernel Address VME1	Size (MB)	VME Address Range
IP5/7/9/17	0x9	0xD8000000– DFFFFFFF	D0000000– D7FFFFFF	128 128	18000000– 1FFFFFFF
IP5/7/9/17	0xD	0xE8000000– EFFFFFFF	E0000000– E8000000	128 128	18000000– 1FFFFFFF

VME Master Addressing

When a VME device uses DMA to access main memory, it acts as a “VME master.” Silicon Graphics systems support both A24 and A32 VME master addressing. Although there are a few minor differences in the addressing modes supported (non-privileged versus supervisor), the main difference is that the 4D/100, 4D/200, 4D/300, 4D/400, and Crimson series systems support dynamic address mapping. This allows IP5, IP7, IP9, and IP17 CPUs to access all of physical memory for A24 addressing and allows scatter-gather for both A24 and A32 addressing.

You can still write an IRIS-portable device driver if the **dma_mapalloc**() function does not return –1 and if the device driver can take advantage of the DMA mapping functions described in Chapter 5, “Writing a SCSI Device Driver.” Otherwise, you must use one of the other DMA methods.

Table A-6 describes the mapping between the address generated by a VME device and physical memory. You can perform A32 master addressing on the IP5, IP7, IP9, and IP17 in either mapped or unmapped mode.

Table A-6 A32 VME-bus/Physical Address Mapping

System CPU	VME Address	Physical Address	Size (MB)	VME Modifier	Map
IP4/6/12	0x0 – 0x0FFFFFFF	0x0 – 0x0FFFFFFF	256	0x9	No
IP5/7/9/17	0x0 – 0x0FFFFFFF	0x0 – 0x0FFFFFFF	256	0x9	No
IP5/7/9/17	0x80000000 – 0x8FFFFFFF	0x – 0x0FFFFFFF	256	0x9	Yes
IP19/21	Must be mapped by the driver	Must be mapped by the driver			

On POWER Series workstations, ranges of VME-bus address space were mapped one-to-one with K2 segment addresses. This made accessing the VME bus easy but was also limiting. Only a small amount of K2 space is available for use by VME, so very little of the VME address space was made available. Even worse, for dual VME-bus systems, the space previously available was now cut in half as it was shared between the two buses.

The CHALLENGE series supports up to five VME buses. Since K2 space is a limited resource, and dividing up what is available by five would make 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 slid 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.

VME-bus Space Reserved for Customer Drivers

Table A-7 shows the VME-bus space reserved for customers.

Table A-7 VME-bus Space Reserved for Customer Drivers

Space	VME Modifier	VME Address
A16	0x2D	0x6000 – 0x7FFF
A16	0x29	0x6000 – 0x7FFF
A24	0x3D	0x800000 – 0x9FFFFF
A24	0x39	0x000000 – 0xFFFFFF
A32	0x09	0x1A000000 – 0x1BFFFFFF
A32	0x09	0x1E000000 – 0x1FFFFFFF (IP5/7/9/17 Only)
A32	0x0D	0x1E000000 – 0x1BFFFFFF (IP5/7/9/17 Only)
A32	0x09	0x20000000 – 0x3FFFFFFF (IP19 and IP21 only)
A32	0x0D	0x20000000 – 0x3FFFFFFF (IP19 and IP21 only)

Note: For information that may not have been available when this guide went to press, refer to the comments in the */var/sysgen/system/irix.sm* file.

POWER Indigo^2 and POWER CHALLENGE M Drivers

For POWER Indigo^2 or POWER CHALLENGE M processors, uncached (K1 segment) writes to main memory must be double-word (64-bit) writes, and they must be aligned on a double-word boundary. The reason for this has to do with the POWER Indigo^2's hardware support for ECC-protected memory. Since the ECC calculation requires that data be written to main memory in 64-bit (8-byte) chunks, smaller (1-, 2-, and 4-byte) uncached writes to main memory tend to produce memory corruption. Cached accesses (both read and write), as well as uncached read operations do *not* cause problems; neither do accesses that are not to main memory, such as device register reads and writes.

Any device driver that performs uncached writes to main memory must always do writes in 8-byte quantities.[1] If the driver needs to write a smaller piece of memory, then it must do the write as a read-modify-write operation of an 8-byte piece of memory. For example, a driver with code like the following (assuming that the structure is being accessed uncached), would corrupt memory:

```
struct foo_s {
    char    b0;
    char    b1;
    char    b2;
    char    b3;
    char    b4;
    char    b5;
    char    b6;
    char    b7;
} bar1;

driver_write_bar1_b3 ()
{
    bar1.b3 = 5;
}
```

Instead, it would have to be modified to perform a read-modify-write operation on the whole double-word containing *b3*, as in the following example (on a big-endian system):

[1] Access to device registers does not fall under this restriction.

```
struct foo_s {
    union {
        __uint64    dw;     /* A 64-bit quantity */
        struct {
            char    b0;     /* READ ONLY */
            char    b1;     /* READ ONLY */
            char    b2;     /* READ ONLY */
            char    b3;     /* READ ONLY */
            char    b4;     /* READ ONLY */
            char    b5;     /* READ ONLY */
            char    b6;     /* READ ONLY */
            char    b7;     /* READ ONLY */
        } byte;
    } dw0;
} bar1;

driver_write_bar1_b3 ()
{
    bar1.dw0.dw = (bar1.dw0.dw & ~(0xff<<32)) | (5<<32);
}
```

If at all possible, use cached accesses to memory, along with cache coherency operations where necessary, instead of uncached operations. Cache coherency operations are necessary only for data shared by the CPU and a device that needs to perform DMA. See “Data Cache Write Back and Invalidation” on page 321 for more information on use of cache coherency operations.

Sharing Data Between CPU and Peripheral Devices

To avoid memory contamination, the CPU and any other device capable of DMA must not both be allowed to write data to separate parts of a double-word in memory. Since CPU accesses to smaller portions of a double-word are performed as read-modify-write operations, they are not atomic, and could be interleaved with data written by a device between the CPU read operation and the CPU write operation. Make sure that data written by a device does not fall within the same double-word as data written by the CPU.

Using mmap()

If you have a driver that allows a process to access main memory uncached through the **mmap**() call, that process must not write the memory using operations that write less than 8 bytes at a time. Again, this restriction does not apply to accesses to device registers, or to accesses to cached memory.