Chapter 6

Writing Kernel-level GIO Device Drivers

This chapter provides in-depth information about drivers that interface to the GIO (Graphics Input/Output) bus. It describes the system configuration for GIO device drivers and introduces several GIO-specific functions you must include in your device driver. There are several models for performing DMA operations. Which model you choose for your device driver depends on the capability of the device, which may either require a software implementation or have hardware support for scatter/gather. Memory-mapped, user-level drivers for GIO devices are not supported; all GIO drivers must be kernel-level drivers.

This chapter contains the following sections:

GIO-bus Architecture

The GIO bus is a family of synchronous, multiplexed address-data buses for connecting high-speed devices to main memory and CPU for entry-level Silicon Graphics systems. The GIO bus has three varieties: GIO32, GIO32-bis, and GIO64.

The members of the GIO-bus family are all similar; however, the GIO32 and GIO64 are not compatible. A GIO32 device does not work in a GIO64 slot, but a GIO32-bis device does fit in either a GIO32 or GIO32-bis option slot. It is possible to design a board that functions in systems with either a GIO32 or GIO32-bis bus.

The form factor and bus protocol depend on the specific platform in which the device is installed. GIO32 and GIO32-bis devices can be either single or double-wide (that is, taking one or both board slots), while GIO64 boards are the size of an EISA board. Slots in Indigo^2 systems can accept either an EISA board or a GIO64 board. These two types of boards share common board dimensions but have different connectors for attaching to their respective buses.

The *GIO Bus Specification* contains more detailed information about the various types of GIO buses, from both an electrical and a mechanical point of view.

Determining GIO Device Addresses

Each GIO device has a range of GIO-bus addresses to which it responds. These addresses correspond to device registers or on-board memory, depending on the GIO device.[1] GIO-bus addresses cannot be mapped into user address space. GIO devices can be classified as 32-bit or 64-bit.

The address range for GIO-bus devices is determined by the slot number of the device. The hardware must be designed to determine which slot the device is in and make the appropriate adjustments to respond to that slot's address range.

Indigo, Indigo^2, and Indy systems all have two slots available for GIO devices. However, the address spaces for Indigo^2 are slightly different than for Indigo and Indy.

For Indigo and Indy, the two slots are known as *exp0* and *exp1*. The Indigo^2's slots are known as *gfx*, *exp0*, and *exp1*.

The *gfx* slot is normally used for the graphics card, but it can be used as a regular GIO card slot if the graphics can be moved up into the *exp0* slot. This slot's address space is also available on Challenge M (Indigo^2 with no graphics) systems.

Table 6-1 shows the slot names and addresses available on the Indigo, Indigo^2, and Indy platforms.

Table 6-1 Indigo, Indigo^2, and Indy Slot Names and Addresses

Slot Name	Address	Indigo/Indy	Indigo^2
gfx	0x1f000000–0x1f3fffff	N/A	Available
exp0	0x1f400000–0x1f5fffff	Available	Available
exp1	0x1f600000–0x1f9fffff	Available	N/A

[1] Unlike VME, where the class of device determines the address range, each GIO device responds to the same address range.

GIO-bus devices use only one interrupt level — interrupt 1. Interrupts 0 and 2 are used by the graphics system and may not be used by GIO-bus devices.

Since one interrupt serves multiple GIO devices, the interrupt routine in each driver must be able to deal with the various interrupt situations:

- The interrupt is for the board.
- The interrupt is for some other GIO device.
- There is no interrupt pending.

Including GIO 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 GIO drivers. To add a new kernel-level GIO device driver, you must:

- Create a system file
- Create a master file
- Create the boot file

For GIO drivers, use the *INCLUDE* directive, which unconditionally adds the module to the kernel. Because **lboot** can probe for GIO devices, **lboot** can conditionally include a GIO device driver into the kernel.

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.

Creating a System File

This file resides in the */var/sysgen/system* directory and contains the instructions that **lboot** uses to add the software module to the kernel. This normally consists of the *VECTOR* directive; in fact, the *system* file may consist of one or more *VECTOR* directives. The filename must end in *.sm* for **lboot** to recognize and include the software module. Typically, the filename is the software module's name with the *.sm* suffix, as in *gbd.sm*.

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

vector — The interrupt vector value, as described previously. The interrupt vector for GIO devices is set using the **setgiovector**() function (see "setgiovector" on page 187). Therefore, the vector in the *VECTOR* statement must always be 0x0 for GIO devices.

unit — The device number that differentiates between more than one device of the same type. This value is related to VME-style devices. For GIO devices, this value can be anything, but for consistency, make it 0.

base — The device address(es) on the GIO bus, determined by the slot in which the board is installed. This is a K1 address (see the kvtophys(D3X) man page).

base2, *base3* — Additional addresses passed to driver **edtinit**() routine through the *edt* structure. These are K1 addresses (see the kvtophys(D3X) man page).

exprobe — The address read when **lboot** determines the existence of the device. This address is often the same as the base address. If you do not specify a probe address, the module is automatically included in the kernel. For GIO bus devices, the **exprobe**() call is used in place of the probe call. The fields used for this call are:

operation	read (*r*) or write (*w*)
address	address to probe
# *of bytes*	number of bytes to read or write
value	expected response value (the GIO ID number)
mask	mask to apply to value

Creating a Master File

The master file resides in the */var/sysgen/master.d* directory. It contains the information that **lboot** uses to create the *device switch table* as well as indicating dependencies with other kernel modules. The name of the master file must be the same as the software module. This file also contains the prefix used in building the driver entry points.

The FLAG field of the master file must include at least the character device flag *c*.

Creating a Boot File

The boot file must reside in the */var/sysgen/boot* directory. This file is the successfully compiled driver object file. The name of the boot file must end with the suffix ".o".[1]

The GBD Example

For example, to add a mythical GIO device driver to the kernel of an R4000 Indigo (IP20), you must copy the driver object file *gbd.o* to */usr/sysgen/boot*, create a master file (as shown below), and create a system file with the following *VECTOR* directive:

```
VECTOR: bustype=GIO module=gbd vector=0x0 unit=0
        base=PHYS_TO_K1(0x1f400000) base2=0xBF410000
        exprobe=(r,PHYS_TO_K1(0x1f400000),4,0x75,0xff)
```

Note that the interrupt vector (*vector*=), the base addresses, and the probe address must all be specified in hexadecimal format. The *base* address and the address in the *exprobe* must agree. In the example above, **lboot** reads four bytes at probe address PHYS_TO_K1(0x1f400000) to determine whether the device is present in slot 0. In this example, *base2* is used to point to the location of on-board memory.

1 The ".a" suffix is used for archived files.

In actual use, it is advisable to add a second *VECTOR* line to the *system* file, to perform a probe of the other GIO slot. If only the line above had been used and the GIO device were physically placed in slot 1 rather than slot 0 as specified in the *VECTOR* line, the probe would fail, and the driver would not have been included in the kernel. Using this situation as an example, the following line must be added to the *system* file:

```
VECTOR: bustype=GIO module=gbd vector=0x0 unit=0
        base=0xBF600000 base2=0xBF610000
        exprobe=(r,0xBF600000,4,0x75,0xff)
```

This ensures that a GIO device placed in either slot will be recognized.

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

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

Writing edtinit()

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

Synopsis

```
drvedtinit(e)
struct edt *e

{
   */your code here/*
}
```

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

The definition of the *edt* type structure is:

```
#define NBASE 3

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

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)(struct edt *); /* device init */
                                        /*run-time probe*/
    iospace_t e_space[NBASE];
} edt_t;

#define e_base          e_space[0].ios_vaddr
#define e_base2         e_space[1].ios_vaddr
#define e_base3         e_space[2].ios_vaddr
#define e_iobase        e_space[0].ios_iopaddr
#define e_iobase2       e_space[1].ios_iopaddr
#define e_iobase3       e_space[2].ios_iopaddr
```

The *e_bus_type* must be *ADAP_GIO* (defined in *edt.h*). The *e_ctrl*, *v_cpuintr*, and *ios_type* values must be 0. The (**e_init*)() member is not used by *drv***edtinit**(). (These fields are set by the kernel, and the driver does not interfere with them.) Your driver uses the *e_base*, *e_base2*, and *e_base3* members:

e_base, *e_base2*, *e_base3*
: These members give your driver the base addresses as specified in the *VECTOR* line. Each is assigned as an unsigned long data type.

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 **autoconfig** (via **lboot**) 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 ...
}
```

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

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

```
/* equipped device table initialization function. The edt
 * structure is defined in edt.h.
 */
void
gbdedtinit(struct edt *e)
{
   int slot, val;

   /* Check to see if the device is present */
   if(badaddr_val(e->e_base, sizeof(int), &val) ||
         (val && GBD_MASK) != GBD_BOARD_ID) {
      if (showconfig)
         cmn_err (CE_CONT,
            "gbdedtinit: board not installed.");
         return;
   }

   /* figure out slot from base on VECTOR line in
   /* system file*/
```

```
   if(e->e_base == (caddr_t)PHYS_TO_K1(0x1f400000))
      slot = GIO_SLOT_0;
   else if(e->e_base == (caddr_t)0xBF600000)
      slot = GIO_SLOT_1;
   else {
      cmn_err (CE_NOTE,
      "ERROR from edtinit: Bad base address %x\n", e-
>e_base);
      return;
   }

#ifdef IP12      /* For Indigo R3000, set up board as a
                 /* realtime bus master  */

   setgioconfig(slot,0);

#endif

#ifdef IP20     /* For Indigo R4000, set up board as a
                /* realtime bus master */

   setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);

#endif

#ifdef IP22      /* For Indigo2, set up board as a pipelined,
                 /* realtime bus master  */

   setgioconfig(slot,GIO64_ARB_EXP0_RT |
                 GIO64_ARB_EXP0_PIPED) ;

#endif

   /* Save the device addresses, because
    * they won't be available later.
    */

   gbd_device[slot == GIO_SLOT_0 ? 0 : 1] =
            (struct gbd_device *)e->e_base;
   gbd_memory[slot == GIO_SLOT_0 ? 0 : 1] =
            (char *)e->e_base2;
              /* Where "unit_#" is any parameter passed to
              /* the interrupt handler (gbdintr) */
   setgiovector(GIO_INTERRUPT_1,slot,gbdintr,unit_#);
}
```

GIO-specific Functions

The GIO-specific support functions **setgiovector**(), **setgioconfig**(), and **splgio**(n) must be included in the **init**() or **edtinit**() section of any GIO driver.

setgiovector

The **setgiovector**() function registers an interrupt service function for a GIO-bus device interrupt with the kernel's interrupt dispatcher.

Synopsis

```
setgiovector(int level, int slot, long (*func)
             (long),int arg)
```

Arguments

level Specifies which interrupt is used by the device. For GIO boards, this must always be *GIO_INTERRUPT_1*, since *GIO_INTERRUPT_0* and *GIO_INTERRUPT_2* are used by the graphics system.

slot Specifies which physical slot the GIO-bus board is plugged into; must be either *GIO_SLOT_0* or *GIO_SLOT_1*.

func Pointer to the interrupt service routine called when the associated interrupt occurs. Note that *func* may be called even when there is no pending interrupt from the particular slot specified, in which case it should simply return. The interrupt handler therefore needs to be able to determine when its device is actually interrupting and when it is not, in a timely, nondestructive manner.

arg Passed to the interrupt service routine when it is called and may contain any value. The interrupt service routine is called with the processor interrupt mask set to disable further interrupts from the device.

splgio0, splgio1, splgio2

These functions set the processor interrupt mask to block GIO-bus interrupts.

Synopsis

```
long splgio0();
long splgio1();
long splgio2();
```

setgioconfig

setgioconfig (2K) sets up the GIO-bus arbitration mode for the GIO slot specified by the *slot* parameter. The arbitration mode is specified in the *flags* parameter as a bit-wise OR of the flags documented below.

Synopsis

`setgioconfig(int` *slot*`, int` *flags*`)`

Arguments

slot Specifies which physical slot the GIO-bus board is plugged into; must be either *GIO_SLOT_0* or *GIO_SLOT_1*.

flags Flags that indicate the configuration for the GIO board. The flags are defined as follows:

For R3000-based systems using the GIO32 bus, these defines are found in */usr/include/sys/IP12.h:*

GIO_CONFIG_LONG
Configure board as a long burst device; otherwise it will be a real-time device.

GIO_CONFIG_SLAVE
Configure board as a bus slave; otherwise it will be a bus master.

For R4000-based systems using the GIO32-bis or GIO64 bus, these defines are found in */usr/include/sys/mc.h*:

GIO64_ARB_EXP0_SIZE_64
Configure slot for 64-bit transfers; otherwise transfers will be 32-bit. For Indigo, this must not be set.

GIO64_ARB_EXP0_RT
Configure slot as a real-time device; otherwise it will be a long burst device.

GIO64_ARB_EXP0_MST
Configure slot as a bus master; otherwise it will be a slave.

GIO64_ARB_EXP0_PIPED
Configure slot as a pipelined device, otherwise it will be a non-pipelined device. For Indigo^2 systems, this must be set. For Indigo, this must *not* be set.

On R4000-based Indigo and Indigo^2 systems, **setgioconfig**() uses the slot argument to determine the location of boards.

GIO Interrupt Handler

Your driver module must contain an interrupt routine. The name of this routine does not need to be *drv***intr**(), since GIO uses **setgiovector**() to register interrupt routines. When the device generates an interrupt, the general GIO interrupt handler calls your driver's interrupt routine and passes it the unit number for the device. Within your interrupt routine, you must set flags to indicate the state of the transfer and wake up sleeping processes (if any) waiting on the transfer to complete. Usually, the interrupt routine calls **iodone**() to indicate that a block type I/O transfer for the buffer is complete.

Caution: Interrupt routines must not try to sleep themselves by calling **iowait**(), **sleep**(), **psema**(), or **delay**() kernel calls, nor should they try to access the per-process global variables in the *u* type structure directly. The *u* type structure they access may not be that of the process that made the I/O request.

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.

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

In these non-DMA cases, the processor is used to copy data from user space to the device. Some device make additional operations on the data. These operations may then be triggered by writing to a register on the device, such as a printer or disk controller. Most such devices have a status register that is used to verify completion. Polling on an interrupt can be used, but it is expensive and should be used sparingly.

Listed below is part of a mythical GIO device driver for a printer controller that does not support DMA. To print data from the user, the driver copies a number of bytes (as specified by *uio_resid*) from the *uio_iov* array to an on-board memory buffer of size GBD_MEMSIZE. Following the copy of each chunk, the driver programs the device registers to indicate the size of valid data in the memory and to tell the controller to start the printing.

Note: Beginning with IRIX 5.0, direct access of the user structure *u* is not allowed. Instead, user data may be accessed only through the *uio* structure. For example, note that the field *u.u_count* is not found in this driver, and references are made to the *uio_resid* field only.

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 **splgio1**() routine.

```
/* device write routine entry point (for character devices)*/
volatile int
gbdwrite(dev_t dev, uio_t *uio)
{
   int unit = geteminor(dev)&1;
   int size, err=0, s;

   /* while there is data to transfer */
   while((size=uio->uio_resid) > 0) {

      /* Transfer no more than GBD_MEMSIZE bytes
      * to the device */
      size = size < GBD_MEMSIZE ? size : GBD_MEMSIZE;

      /* decrements size, updates uio fields, copies data */
      if(err=uiomove(gbd_memory[unit], size, UIO_WRITE, uio))
         break;

      /* prevent interrupts until we sleep */
      s = splgio1();

      /* Transfer is complete; start output */
      gbd_device[unit]->count = size;
      gbd_device[unit]->command = GBD_GO;
      gbd_state[unit] = GBD_SLEEPING;
      while (gbd_state[unit] != GBD_DONE) {
         sleep(&gbd_state[unit], PRIBIO);
      }
      /* restore the interrupt level after waking up */
      splx(s);
   }
   return err;
}
```

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

Note: If your driver uses the **sleep**() and **wakeup**() kernel routines to sleep and awaken, it is a good idea for the *drv***intr**() to verify that the actual event has occurred before actually awakening the sleeping process. (See **sleep**() for details on the sleep/wakeup process synchronization mechanism.) If your driver uses the **iowait**()/**iodone**() routines or the **psema**()/**vsema**() routines to sleep and awaken, you need not worry about it 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 the fields in the *uio* structure and uses **copyout**() (or **copyin**()) to check for invalid user addresses. Recall that *uio_resid* must be left with the number of bytes left untransferred.

DMA Operations

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 point of view is that of memory.

There are some 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 read, whereas both write back and write through caches require the cache to be invalidated before data from a DMA write 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. To this end, when a driver allocates a buffer for DMA, it must use the **kmem_alloc**() function with the *KM_CACHEALIGN* flag to obtain a buffer that is properly aligned.The interrupt service routine then calls your *drv***intr** routine. Your *drv***intr** routine can confirm that the transfer is complete (if necessary), set flags indicating the status of the transfer, and then awaken the sleeping process.

The GIO bus does not provide any address mapping registers. Any DMA operation that requires scatter/gather must be supported by GIO board hardware or a software implementation of scatter/gather.

Memory Parity Workarounds

If you are writing a GIO device driver for Indigo R4000, Indigo^2, POWER Indigo^2, or Indy systems, please make note of the following changes introduced in IRIX 5.3 (or IRIX 5.2 with Patch4, the Memory Parity Patch).

Beginning with IRIX 5.3, parity checking is enabled on the SysAD bus (see Figure 6-1). Unfortunately, with certain GIO cards, errors can occur if memory reads complete before the Memory Controller (MC) finishes calculating parity.

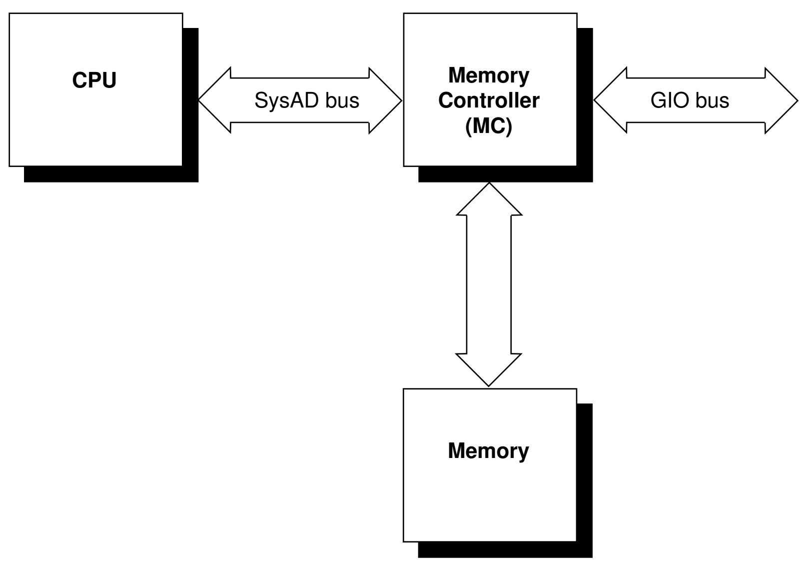

Figure 6-1 The SysAD Bus in Context

Some GIO cards do not drive all 32 GIO data lines during CPU PIO reads. These reads from the GIO card are either 8-bit (byte) or 16-bit (short word), so the lines are left floating. The problem is that to generate parity bits for the SysAD bus, the Memory Controller (MC) must calculate parity for all 32 bits. Since the calculation must occur before the CPU read completes, it is possible that one (or more) of the floating bits may change while parity is being calculated. Thus, when the CPU read completes, it may receive a parity error on the SysAD bus.

Caution: Even on GIO boards that do not drive all data lines, this problem may not show up on every transaction. It occurs only when one of the floating data lines changes state between the start of the MC parity calculation and the completion of the CPU read. Even if a driver appears to function correctly, the system may panic due to a parity error.

If you are writing a driver for a GIO card that does not drive all 32 data lines, even when fewer bits are being read, you must either:

1. Disable SysAD parity checking completely.
   This reduces the system's ability to recover from a parity error in main memory, but it is both reliable and easy to program. The way to implement this is simply to put a call to **disable_sysad_parity**() at the beginning of your driver's **init** (or **edtinit**) routine before the driver attempts any PIO reads from the GIO device.
2. Disable SysAD parity checking only when your driver is actually performing PIOs.
   The advantage here is that the software recovery procedures for memory parity errors are almost always in effect, but it requires a bit more work during driver development. Put wrappers around your driver's PIO transactions to disable SysAD parity checking before the transactions and re-enable it after the PIOs complete, as in the following code fragment:

```
{
 int was_enabled = is_sysad_parity_enabled();

 if (was_enabled)
 disable_sysad_parity();

 /* do driver PIO transactions */

 if (was_enabled)
```

```
    enable_sysad_parity();
    }
```

GIO Devices with Hardware-supported Scatter/Gather Capability

Chapter 2, “Writing a Device Driver,” tells you to use the **physio**() kernel routine to fault in and lock the physical pages corresponding to the user's buffer. **physio**() 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, **physio**() allocates a structure of type *buf* if you pass a *NULL* pointer. (**physio**() uses this structure to embody the transfer information.) **physio**() 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 a GIO device that has hardware-supporting 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:

```
/* Actual device setup for DMA, etc., if your board has
 * hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   int npages;
   volatile unsigned *sgregisters;
   int i, v_addr;

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

```
sgregisters = gbd_device[unit]->sgregisters;

/* Get the kernel virtual address of the data; note
* b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
* indicates false; in that case, the field bp->b_pages
* is a pointer to a linked list of pfdat structure
* pointers; that saves creating a virtual mapping and
* then decoding that mapping back to physical addresses.
* BP_ISMAPPED will never be false for character devices,
* only block devices.
*/
if(!BP_ISMAPPED(bp)) {
   cmn_err(CE_WARN,
      "gbd driver can't handle unmapped buffers");
   bioerror(bp, EIO);
   biodone(bp);
   return;
}

v_addr = bp->b_dmaaddr;

/* 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. Limit to
* number of scatter/gather registers on board.
* Note that this sample driver doesn't handle the
* case of requests > than # of registers!
* numpages() is a macro declared in sys/sysmacros.h
*/
npages = numpages (v_addr, bp->b_dmalen);
/*
* Provide the beginning byte offset and count to the
* device.
*/
gbd_device[unit]->offset =
      (unsigned long)bp->b_dmaaddr & (NBPC-1);
if(npages > GBD_NUM_DMA_PGS) {
   npages = GBD_NUM_DMA_PGS;
   cmn_err(CE_WARN,
   "request too large, only %d pages max", npages);
   if(gbd_device[unit]->offset)
      gbd_device[unit]->count = NBPC -
```

```
            gbd_device[unit]->offset + (npages-1)*NBPC;
         else
            gbd_device[unit]->count = npages*NBPC;
         bp->b_resid = bp->b_count - gbd_device[unit]->count;
      }
      else
         gbd_device[unit]->count = bp->b_count;

      /* Translate the virtual address of each page to a
      * physical page number and load it into the next
      * scatter/gather register. btop()
      * 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++ = btop(kvtophys(v_addr));

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

         v_addr += NBPC;
      }

      if ((bp->b_flags & B_READ) == 0)
         gbd_device[unit]->direction = GBD_WRITE;
      else
         gbd_device[unit]->direction = GBD_READ;
      gbd_device[unit]->command = GBD_GO;   /* start DMA */

      /* and return; upper layers of kernel wait for iodone(bp)
   */
   }
```

DMA on GIO Devices Without Scatter/Gather Capability

If your device does not provide scatter/gather capability, it must break up a data transfer so that DMA transfer targets in physical memory are physically contiguous to the DMA engine (this assures that no transfer crosses a page boundary). The IRIX operating system provides a utility, sgset(D3X), that simulates scatter/gather registers in software. (See the *IRIX Device Driver Reference Pages* for details on this routine.) Your driver can use this facility 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:

```
/* Actual device setup for DMA, etc., if your board
 * does NOT have hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;

   /* any checking for initial state here. */

   /* Get the kernel virtual address of the data; note
   * b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
   * indicates false; in that case, the field bp->b_pages
   * is a pointer to a linked list of pfdat structure
   * pointers; that saves creating a virtual mapping and
   * then decoding that mapping back to physical addresses.
   * BP_ISMAPPED will never be false for character devices,
   * only block devices.
   */
   if(!BP_ISMAPPED(bp)) {
      cmn_err(CE_WARN,
         "gbd driver can't handle unmapped buffers");
      bioerror(bp, EIO);
      biodone(bp);
      return;
   }

   gbd_curbp[unit] = bp;
   /*
   * Initialize the current transfer address and count.
   * The first transfer should finish the rest of the
   * page, but do no more than the total byte count.
```

```
   */
   gbd_curaddr[unit] = bp->b_dmaaddr;
   gbd_totcount[unit] = bp->b_count;
   gbd_curcount[unit] = NBPC -
      ((unsigned int)gbd_curaddr[unit] & (NBPC-1));
   if (bp->b_count < gbd_curcount[unit])
      gbd_curcount[unit] = bp->b_count;
   /* Tell the device starting physical address, count,
   * and direction */
   gbd_device[unit]->startaddr = kvtophys(gbd_curaddr[unit]);
   gbd_device[unit]->count = gbd_curcount[unit];
   if (bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */

   /* and return; upper layers of kernel wait for iodone(bp)
*/
}
```

Device Driver Example

The following pages contain the complete driver code for the mythical *gbd* GIO device. Note that it includes strategy routines for devices that have hardware support for scatter/gather as well as for those devices that have no hardware scatter/gather support.

Commonly, a single set of source files is used for multiple target machines. C preprocessor *defines* are used to define differences conditionally. Command line compile options expose the correct values. The following examples are interesting:

For an Indigo (R3000) system:

```
% cc -DIP12 -DR3000 -cckr -c gbd.c
```

For an Indigo (R4000) system:

```
% cc -DIP20 -DR4000 -cckr -c gbd.c
```

For an Indigo^2 (R4000) or Indy system:

```
% cc -DIP22 -DR4000 -cckr -c gbd.c
```

Note: For R8000 systems, omit the `-cckr` argument.

For more information on compile directives, see */var/sysgen/Makefile.kernio*.

```
/* Source for a mythical GIO board device; it can be compiled
 * for devices that support DMA (with or without scatter/
 * gather support), or for PIO mode only. This version is
 * designed for IRIX 5.1 or later.
 * Dave Olson, 5/93
 */

/* defines for compilation; would normally be passed on
compilation
 * line via Makefile */
#define _K32U32   1
#define _KERNEL   1

#define IP20   1   /* define cpu type */
```

```
#if IP20 || IP22
#define R4000   1
#elif IP12
#define R3000   1
#endif
/* end of 'normal' compilation definitions */
/* The following definitions choose between PIO vs DMA
 * supporting boards, and if DMA is supported, whether
 * hardware scatter/gather is supported. */
#define GBD_NODMA 0  /* non-zero for PIO version of driver */
#define GBD_NUM_DMA_PGS 4 /* 0 for no hardware scatter/gather
                           * support, else number of pages of
                           * scatter/gather supported per
                           * request */

#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/cpu.h>
#include <sys/buf.h>
#include <sys/cred.h>
#include <sys/uio.h>
#include <sys/ddi.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/edt.h>

   /* NOTE: This sample driver ignores the possiblity that
    * the board might be busy handling some earlier request.
    * Any real device must deal with that possiblity, of
    * course, before changing the board registers.
    */

/* these defines and structures would normally be in
 * a separate header file */

#define GBD_BOARD_ID   0x75
#define GBD_MASK   0xff  /* Use 0xff if using only first byte
                          * of ID word; use 0xffff if using
                          * whole ID word.
                          */

#define GBD_MEMSIZE 0x8000

/* command definitions */
```

```
#define GBD_GO 1

/* state definitions */
#define GBD_SLEEPING 1
#define GBD_DONE 2

/* direction of DMA definitions */
#define GBD_READ 0
#define GBD_WRITE 1

/* status defines */
#define   GBD_INTR_PEND   0x80

/* "gbd" is device prefix; also in master.d/xxx file */

/* devices interface to the board */
struct gbd_device {
   int command;
   int count;
   int direction;
   off_t offset;
   unsigned *sgregisters; /* if scatter/gather supported */
   caddr_t startaddr; /* if no scatter/gather on board */
   unsigned status;   /* errors, interrupt pending, etc. */
};

/* These are used for no scatter/gather case only, and assume
 * (since they aren't protected!) that the driver is
 * completely single threaded. */
struct buf   *gbd_curbp[2];  /* current buffer */
caddr_t      gbd_curaddr[2]; /* current address to transfer
*/
int          gbd_curcount[2];
int          gbd_totcount[2];

/* pointer to on-board registers */
volatile struct gbd_device *gbd_device[2];

char *gbd_memory[2];   /* pointer to on-board memory */

static int gbd_state[2];    /* flag for transfer state
             * (PIO driver) */

void gbdintr(int);
extern int splgio1(void);
```

```
/* equipped device table initialization routine.  The edt
 * structure is defined in edt.h.
 */
void
gbdedtinit(struct edt *e)
{
   int slot, val;

   /* Check to see if the device is present */
   if(badaddr_val(e->e_base, sizeof(int), &val) ||
         (val && GBD_MASK) != GBD_BOARD_ID) {
      if (showconfig)
         cmn_err (CE_CONT,
            "gbdedtinit: board not installed.");
         return;
   }

/* figure out slot from base on VECTOR line in
    * system file */
   if(e->e_base == (caddr_t)PHYS_TO_K1(0x1f400000))
      slot = GIO_SLOT_0;
   else if(e->e_base == (caddr_t)0xBF600000)
      slot = GIO_SLOT_1;
   else {
      cmn_err (CE_NOTE,
      "ERROR from edtinit: Bad base address %x\n", e-
>e_base);
      return;
   }

#if IP12      /* For Indigo R3000 system, set up board as a
               * realtime bus master.
               */

   setgioconfig(slot,0);

#endif

#if IP20 /* For Indigo R4000 system, set up board as a
          * realtime bus master.
          */

   setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);
#endif
```

```
#if IP22  /* for Indigo2 system, set up board as a pipelined,
           * realtime bus master */

   setgioconfig(slot,GIO64_ARB_EXP0_RT |
                 GIO64_ARB_EXP0_PIPED);

#endif

   /* Save the device addresses, because
    * they won't be available later. */

   gbd_device[slot == GIO_SLOT_0 ? 0 : 1] =
            (struct gbd_device *)e->e_base;
   gbd_memory[slot == GIO_SLOT_0 ? 0 : 1] =
            (char *)e->e_base2;
   setgiovector(GIO_INTERRUPT_1,slot,gbdintr,unit_#);
}
/* minor number used to indicate which slot; open does
 * nothing but check that board is present. */
/* ARGSUSED */
gbdopen(dev_t *devp, int flag, int otyp, cred_t *crp)
{
   if(!gbd_device[geteminor(*devp)&1])
      return ENXIO;   /* board not present */
   return 0;   /* OK */
}

/* ARGSUSED */
gbdclose(dev_t dev, int flag, int otyp, cred_t *crp)
{
   return 0;   /* nothing to do */
}

#ifdef GBD_NODMA

/* device write routine entry point (for character devices)
*/
int
gbdwrite(dev_t dev, uio_t *uio)
{
   int unit = geteminor(dev)&1;
   int size, err=0, s;

   /* while there is data to transfer */
```

```
   while((size=uio->uio_resid) > 0) {

      /* Transfer no more than GBD_MEMSIZE bytes
       * to the device */
      size = size < GBD_MEMSIZE ? size : GBD_MEMSIZE;

      /* decrements count and updates uio fields,
       * and copies data */
      if(err=uiomove(gbd_memory[unit], size, UIO_WRITE, uio))
         break;

      /* prevent interrupts until we sleep */
      s = splgio1();

      /* Transfer is complete; start output */
      gbd_device[unit]->count = size;
      gbd_device[unit]->command = GBD_GO;
      gbd_state[unit] = GBD_SLEEPING;
      while (gbd_state[unit] != GBD_DONE) {
         sleep(&gbd_state[unit], PRIBIO);
      }
      /* restore the process level after waking up */
      splx(s);
   }
   return err;
}

/* interrupt routine for PIO only board, just wake up
 * upper half of driver
 */
/* ARGSUSED1 */
void
gbdintr(int unit)
{
   /* Read your board's registers to determine if there are
    * any errors or interrupts pending. If no interrupts
    * are pending, return without doing anything.
    */
   if(!gbd_device[unit]->status & GBD_INTR_PEND)
      return;

   if (gbd_state[unit] == GBD_SLEEPING) {
      /* Output is complete; wake up top half
       * of driver, if it is waiting. */
      gbd_state[unit] = GBD_DONE;
```

```
      wakeup(&gbd_state[unit]);
   }

   /* Do anything else to board to tell it we are done
    * with transfer and interrupt here. */
return;    /* could just fall through */
}

#else   /* DMA version of driver */

void gbd_strategy(struct buf *);

/* device write routine entry point (for character devices).
 * Does nothing but call uiophysio to setup passing a pointer
 * to the gbd_strategy routine, which does most of the work.
 */
int
gbdwrite(dev_t dev, uio_t *uiop)
{
   return uiophysio((int (*)())gbd_strategy, 0, dev,
B_WRITE, uiop);
}

#if GBD_NUM_DMA_PGS > 0

/* Actual device setup for DMA, etc., if your board has
 * hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   int npages;
   volatile unsigned *sgregisters;
   int i, v_addr;

   /* Get address of the scatter/gather registers */
    sgregisters = gbd_device[unit]->sgregisters;

   /* Get the kernel virtual address of the data; note
    * b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
    * indicates false; in that case, the field bp->b_pages
    * is a pointer to a linked list of pfdat structure
    * pointers; that saves creating a virtual mapping and
```

```
     * then decoding that mapping back to physical addresses.
     * BP_ISMAPPED will never be false for character devices,
     * only block devices.
     */
     if(!BP_ISMAPPED(bp)) {
       cmn_err(CE_WARN,
          "gbd driver can't handle unmapped buffers");
       bioerror(bp, EIO);
       biodone(bp);
       return;
    }

    v_addr = bp->b_dmaaddr;

    /* 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. Limit to
     * number of scatter/gather registers on board.
     * Note that this sample driver doesn't handle the
     * case of requests > than # of registers!
     */
    npages = numpages (v_addr, bp->b_dmalen);

 /*
     * Provide the beginning byte offset and count to the
     * device.
     */
    gbd_device[unit]->offset =
          (unsigned int)bp->b_dmaaddr & (NBPC-1);
    if(npages > GBD_NUM_DMA_PGS) {
       npages = GBD_NUM_DMA_PGS;
       cmn_err(CE_WARN,
           "request too large, only %d pages max", npages);
       if(gbd_device[unit]->offset)
          gbd_device[unit]->count = NBPC -
              gbd_device[unit]->offset + (npages-1)*NBPC;
       else
          gbd_device[unit]->count = npages*NBPC;
       bp->b_resid = bp->b_count - gbd_device[unit]->count;
    }
    else
       gbd_device[unit]->count = bp->b_count;
```

```
   /* Translate the virtual address of each page to a
    * physical page number and load it into the next
    * scatter/gather register. btop()
    * 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++ = btop(kvtophys(v_addr));

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

      v_addr += NBPC;
   }

   if ((bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */
  /* and return; upper layers of kernel wait for iodone(bp)*/
}
/* not much to do in this interrupt routine, since we are
 * assuming for this driver that we can never have to do
 * multiple DMA's to handle the number of bytes requested...
 */
void
gbdintr(int unit)
{
   int error;

   /* Read your board's registers to determine if
    * there are any errors or interrupts pending.
    * If no interrupts are pending, return without
    * doing anything.
    */
   if(!gbd_device[unit]->status & GBD_INTR_PEND)
      return;

   if(error)
      bioerror(bp, EIO);
```

```
   biodone(bp);   /* we are done, tell upper layers */

   /* do anything else to board to tell it we are done
    * with transfer and interrupt here */
}

#else /*  GBD_NUM_DMA_PGS == 0; no hardware
       *  scatter/gather support */

/* Actual device setup for DMA, etc., if your board
 * does NOT have hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;

   /* any checking for initial state here. */

   /* Get the kernel virtual address of the data; note
    * b_dmaaddr may be NULL if the  BP_ISMAPPED(bp) macro
    * indicates false; in that case, the field bp->b_pages
    * is a pointer to a linked list of pfdat structure
    * pointers; that saves creating a virtual mapping and
    * then decoding that mapping back to physical addresses.
    * BP_ISMAPPED will never be false for character devices,
    * only block devices.
    */
    if(!BP_ISMAPPED(bp)) {
      cmn_err(CE_WARN,
         "gbd driver can't handle unmapped buffers");
      bioerror(bp, EIO);
      biodone(bp);
      return;
   }

   gbd_curbp[unit] = bp;
   /*
    * Initialize the current transfer address and count.
    * The first transfer should finish the rest of the
    * page, but do no more than the total byte count.
    */
   gbd_curaddr[unit] = bp->b_dmaaddr;
```

```
   gbd_totcount[unit] = bp->b_count;
   gbd_curcount[unit] = NBPC -
      ((unsigned int)gbd_curaddr[unit] & (NBPC-1));
   if (bp->b_count < gbd_curcount[unit])
      gbd_curcount[unit] = bp->b_count;
   /* Tell the device starting physical address, count,
    * and direction */
   gbd_device[unit]->startaddr = kvtophys(gbd_curaddr[unit]);
   gbd_device[unit]->count = gbd_curcount[unit];
   if (bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */

   /* and return; upper layers of kernel wait for iodone(bp)
*/
}

/* more complicated interrupt routine, not necessarily
because
 * board has DMA, but more typical of boards that do have
 * DMA, since they are typically more complicated.
 * Also more typical of devices that support block i/o, as
 * opposed to character i/o.
 */
void
gbdintr(int unit)
{
   int error;
   register struct buf *bp = gbd_curbp[unit];

   /* read your board's registers to determine if
    * there are any errors or interrupts pending.
    * If no interrupts are pending, return without
    * doing anything.
    */
   if(!gbd_device[unit]->status & GBD_INTR_PEND)
      return;

   if(error) {
      bioerror(bp, EIO);
      biodone(bp);   /* we are done, tell upper layers */
   }
   else {
```

```
        /* On successful transfer of last chunk, continue
         * if necessary */
        gbd_curaddr[unit] += gbd_curcount[unit];
        gbd_totcount[unit] -= gbd_curcount[unit];
        if(gbd_totcount[unit] <= 0)
           biodone(bp);
              /* we are done, tell upper layers */
        else {
        /* else more to do, reprogram board and
         * start next dma */
            gbd_curcount[unit] =
               (gbd_totcount[unit] < NBPC
                     ? gbd_totcount[unit] : NBPC);
            gbd_device[unit]->startaddr =
                     kvtophys(gbd_curaddr[unit]);
            gbd_device[unit]->count = gbd_curcount[unit];
            if (bp->b_flags & B_READ) == 0)
               gbd_device[unit]->direction = GBD_WRITE;
            else
               gbd_device[unit]->direction = GBD_READ;
            gbd_device[unit]->command = GBD_GO;
                 /* start next DMA */
        }
    }

    /* Do anything else to board to tell it we are done
     * with transfer and interrupt here. */
}
#endif /*  GBD_NUM_DMA_PGS */

#endif /* GBD_NODMA */
```