Chapter 18

GIO Device Drivers

The GIO bus is a synchronous, multiplexed address-data bus connecting high-speed devices to main memory and CPU for Silicon Graphics workstations. This chapter gives an overview of the GIO architecture, and describes the special kernel functions used to manage a device on the GIO bus. The main topics are as follows:

- "GIO Bus Overview" on page 512 describes the hardware implementation of the GIO bus.
- "Configuring a GIO Device" on page 514 discusses the use of the VECTOR line to describe a GIO device to IRIX.
- "Writing a GIO Driver" on page 515 discusses the work done in each entry point of a GIO device driver.
- "Memory Parity Workarounds" on page 526 covers an important hardware problem.
- "Example GIO Driver" on page 528 displays major parts of a driver for a hypothetical GIO device.

GIO Bus Overview

The GIO bus is a family of buses with different electrical requirements and form factors.

GIO Bus Varieties

Two varieties of GIO bus are found in workstations supported by IRIX 6.2:

- The GIO64 bus is a 64-bit, synchronous, multiplexed address-data bus that can run at speeds up to 33 MHz. It supports both 32- and 64-bit devices. GIO64 has two slightly different varieties: non-pipelined for internal system memory, and pipelined for graphics and pipelined GIO64 slot devices. The GIO64 bus is used in the Indigo^2 and Indigo^2 Maximum Impact systems.
- The GIO32-bis variety is a 32-bit version of the non-pipelined GIO64 bus or a GIO32 bus with pipelined control signals. This bus is found on R4000-based Indigo and Indy workstations.

An additional variety, GIO32 (without the *-bis* suffix) was used in Indigo workstations with R3000 processors. These workstations are not supported by IRIX 6.2.

Card Form Factors and Compatibility

GIO devices designed for the GIO32 (Indigo R3000) bus are compatible with GIO32-bis (Indy, Indigo^2), but are not compatible with GIO64 buses.

Devices designed for GIO32-bis can be used in GIO64 bus systems.

The Indigo and Indy systems have two GIO slots. The Indigo^2 has three physical sockets, but the lower two are paired as a single logical slot—the double socket provides extra electrical and mechanical support for heavy cards. The Indigo^2 Maximum Impact has four physical sockets, with each pair ganged as one logical slot. Thus all systems have two GIO slots, electrically speaking.

The form factor depends on the specific platform in which the device is installed. GIO32-bis devices can be either single or double-wide (that is, taking one or two sockets), 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.

GIO Bus Address Spaces

Each GIO device has a range of bus addresses to which it responds. These addresses correspond to device registers or on-board memory, depending on the GIO device.

The address range for a GIO bus device is determined in part 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 and Indy systems provide two GIO address spaces known as *exp0* and *exp1*. Indigo^2 systems also have two slots, but support three GIO address spaces, referred to as *gfx*, *exp0*, and *exp1*. The *gfx* address space is used by the graphics card.

Table 18-1 shows the slot names and address spaces available on the Indigo, Indigo^2, and Indy platforms.

Table 18-1 GIO Slot Names and Addresses

Slot Name	Address	Indigo/Indy	Indigo^2
gfx	0x1f00 0000–0x1f3f ffff	N/A	Available
exp0	0x1f40 0000–0x1f5f ffff	Available	Available
exp1	0x1f60 0000–0x1f9f ffff	Available	Available

In 64-bit systems (Indigo2 Maximum Impact), two additional high-order bits are needed to select the physical address of the GIO space, so each of the above addresses is prefixed by 0x9000 0000. The 64-bit *gfx* space, for example, is at 0x9000 0000 1f00 0000.

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.

Configuring a GIO Device

A GIO device is described to the system, and related to its device driver, using a VECTOR line in a file in the */var/sysgen/system* directory (see “Configuring a Kernel” on page 236).

GIO VECTOR Line

The VECTOR line for a GIO device uses the “old style” syntax documented in */var/sysgen/system/irix.sm*. The important elements in a VECTOR line for GIO are as follows:

bustype	Specified as *GIO* for GIO devices. The VECTOR statement can be used for other types of buses as well.
module	The base name of the device driver for this device, as used in the */var/sysgen/master.d* database (see “Master Configuration Database” on page 38 and “How Names Are Used in Configuration” on page 232).
adapter	Always 0, or omitted, for GIO, since there is never more than one GIO bus adapter in current systems.
ctlr	The “controller” number is simply an integer parameter that is passed to the device driver at boot time. It can be used, for example, to specify a logical unit number.
base	Device base address, as shown in Table 18-1.
probe or *exprobe*	Specify a hardware test that can be applied at boot time to find out if the device exists.

You use the *probe* or *exprobe* parameter to program a test for the existence of the device at boot time. If the device does not respond (because it is offline or because it has been removed from the system), the *lboot* command will not invoke the device driver for this device. This facility is used in distributed */var/sysgen/system/irix.sm* files in order to choose between the graphics board in slot *gfx* or in slot *exp0*.

Writing a GIO Driver

GIO bus devices are controlled only from kernel-level drivers; there is no provision for memory-mapping GIO devices into user-level address spaces.

A GIO device driver is a kernel-level driver compiled, linked, and loaded into the kernel as described in Chapter 10, “Building and Installing a Driver.” A GIO driver can call on the kernel functions described in Chapter 8, “Structure of a Kernel-Level Driver.” However, a GIO driver has to use some special features in its *pfx***edtinit()** and *pfx***intr()** entry points.

GIO-Specific Kernel Functions

Three GIO-specific functions are used in setting up a GIO device. They are only documented here; there are no reference pages for them. The functions are declared as external in the CPU-specific include files *sys/IP20.h* and *sys/IP22.h*. (When compiling for a POWER Indigo^2, which uses an IP26 CPU, you include *sys/IP22.h* as well as *sys/IP26.h*.)

Function setgiovector()

The **setgiovector()** function registers an interrupt service function for a GIO device interrupt with the kernel’s interrupt dispatcher, or unregisters one. The function prototype is

```
void
setgiovector(int level, int slot,
            void (*func)(__psint_t, struct eframe_s *),
            __psint_t arg);
```

The arguments are as follows:

level The interrupt level; must be GIO_INTERRUPT_1 for all devices except the graphics board.

slot The slot number, 0 or 1.

func The address of the interrupt handling function (typically the *pfx***intr()** entry point of the device driver), or else NULL to unregister.

arg A “pointer-sized integer” value to be passed as the first argument of the interrupt handler when it is invoked.

Note: If either the *level* or *slot* number is out of range, **setgiovector()** issues an error message with the CE_PANIC level, causing a kernel panic.

When *func* is not NULL, the specified function is registered to receive interrupts at the given *level* from the given *slot*. When an interrupt occurs, the function is called with two arguments. The first is the value specified as *arg*, a "pointer-sized integer," typically the address of device-specific information. The second is the interrupt registers. The structure *eframe_s* is declared in *sys/reg.h*. However, this structure is of no interest.

This function can be used with a NULL for the *func* argument to unregister an interrupt routine that was previously registered. You must unregister an interrupt handler in a loadable device driver prior to unloading, when called at the *pfx***unload()** entry point (see "Entry Point unload()" on page 167).

Function setgioconfig()

The function **setgioconfig()** configures the GIO slot for a particular use. The function prototype is

```
void
setgioconfig(int slot, int flags);
```

The arguments are as follows:

slot	The slot number, 0 or 1.
flags	A set of bit-flags from the constants GIO_ARB_* declared in *sys/mc.h*.

Note: If the *slot* number is out of range, **setgioconfig()** either issues an error message with the CE_PANIC level or suffers an assertion failure, causing a kernel panic.

The flags that can be combined to make the *flags* argument are

GIO64_ARB_EXP0_SIZE_64	Configure for 64-bit transfers; otherwise transfers will be 32-bit.
GIO64_ARB_EXP0_RT	Configure as a real-time device; otherwise it will be a long burst device.
GIO64_ARB_EXP0_MST	Configure 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.

splgio0, splgio1, splgio2

Three functions can be used to set the processor interrupt mask to block GIO-bus interrupts. As of IRIX 6.2, the only systems that support the GIO bus are uniprocessor systems, in which **spl()**-type functions are effective. When writing a device driver that might be ported to a multiprocessor, you should avoid functions of this type, and use other means of getting mutual exclusion (see “Priority Level Functions” on page 213).

The prototypes of the GIO **spl()** functions are

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

Devices other than graphics drivers would typically only have a reason to use **splgio1()**, because 1 is the interrupt level of non-graphics GIO devices.

GIO Driver edtinit() Entry Point

The device driver specified by the *module* parameter is invoked at its *pfx***edtinit()** entry point, where it receives most of the other information specified in the VECTOR statement (see “Entry Point edtinit()” on page 144).

The *pfx***edtinit()** entry point is called only in response to a VECTOR line. However, a VECTOR line need not contain a *probe* or *exprobe* test of the hardware.

The driver should not assume that its hardware exists; instead it should use the **badaddr()** kernel function to test the addresses passed in the edt_t object to make sure they are usable (see “Testing Device Physical Addresses” on page 195).

Example 18-1 displays a skeleton version of the *pfx***edtinit()** entry point of a hypothetical GIO device driver. This example uses GIO-specific functions that are described in a following section, “GIO-Specific Kernel Functions” on page 515.

Example 18-1 GIO Driver edtinit() Entry Point

```
#include <sys/edt.h>
void
hypoth_edtinit(register 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)0xBf400000)
      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 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|IP26)   /* For Indy, 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 Driver Interrupt Handler

A GIO driver must contain an interrupt entry point. It does not have to be named *pfx***intr()** because it is registered using the **giosetvector()** function.

When the device generates an interrupt, the general GIO interrupt handler calls your driver's registered interrupt routine and passes it the argument that was specified to **setgiovector()** as the argument. This is typically a unit number, or the address of a device-specific information structure.

Within the interrupt routine, the driver must wake up the sleeping upper-half process, if one is waiting on the transfer to complete. In a block device driver, the interrupt routine calls **iodone()** to indicate that a block type I/O transfer for the buffer is complete (see "Waiting for Block I/O to Complete" on page 217).

Using PIO

Programmed I/O (PIO) is used to transfer small amounts of data between memory and device registers. PIO is typically used for control functions and to set up device registers prior to DMA (see "Using DMA" on page 521).

PIO can be as simple as storing a variable into a bus address (as passed to the *pfx***edtinit()** entry point). Example 18-2 displays fragmentary code of a hypothetical character device driver for a GIO device that controls a printer. This *pfx***write()** entry point copies data from the user address space to device memory using the **uiomove()** function (see "Transferring Data Through a uio_t Object" on page 194). Then it stores an explicit command in the device to start it, and sleeps until the device interrupts.

Example 18-2 Hypothetical PIO Routine for GIO

```
/* device write routine entry point (for character devices)*/
int
hypoth_write(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 */
      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;
}
```

An expression like *gdb_device[unit]*-**>***command=GBD_GO* represents storing a command value in a device register. Presumably the *gdb_device* array is set up with a device address for each slot in the *pfx***edtinit()** entry point.

The code in Example 18-2 uses **splgio1()** to block an interrupt from occurring after it has started the device in operation and before it has entered the blocked state using **sleep()**. If this was not done, there is a small window of time during which an interrupt could occur and be handled before the upper-half routine had begun sleeping. Then it would sleep forever.

An alternate way to handle this same situation in a multiprocessor system is to use a mutual-exclusion lock to get exclusive use of the device registers, and a synchronization variable to wait for the interrupt (see "Using Synchronization Variables" on page 220).

Using DMA

DMA access achieves higher throughput than PIO when the device transfers more than a few words of data at a time. DMA is typically set up by programming device registers with the target address and length, and leaving the device to generate a series of stores or loads from memory. The details of device control are hardware-dependent.

The direction of a DMA transfer is measured with respect to the device, which operates independently. A DMA operation is either a DMA *read* (of memory data out to the device) or a DMA *write* (by the device, of data into memory).

DMA buffers should be cache-aligned in memory (see "Setting Up a DMA Transfer" on page 198). Prior to a DMA read, the driver should make sure that cached data has been written to memory using **dki_cache_wb()**. Prior to a DMA write, the driver should make sure the CPU knows that cached data is invalid (or is about to become invalid) using **dki_cache_inval()** (see "Managing Memory for Cache Coherency" on page 200).

DMA To Multiple Pages

Some devices can perform DMA only in a single transfer of data to a range of contiguous addresses. Such a device must be programmed separately for each individual page of data. Other devices are capable of transferring a series of page units to different addresses; that is, they support "scatter/gather" capability. These devices can be programmed once to transfer an entire buffer of data, regardless of whether the buffer spans multiple pages.

In either case, the *pfx***strategy()** entry point of a block device driver must calculate the physical addresses of a series of one or more pages, and program them into the device. When the device does not support scatter/gather, it is set up and started on each page of data individually, with an interrupt after each page. When the device supports scatter/gather, it is programmed with a list of page addresses all at once.

DMA With Scatter/Gather Capability

Example 18-3 shows the skeleton of a *pfx***strategy()** entry point for a block device driver for a hypothetical GIO device that supports scatter/gather capability.

Example 18-3 Strategy Code for Hypothetical Scatter/Gather GIO Device

```
/* Actual device setup for DMA, etc., if your board has
 * hardware scatter/gather DMA support.
 * Called from the hypo_write() routine via physio().
 */
void
hypo_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   int npages;
   volatile unsigned *sgregisters; /* ->device regs */
   int i, v_addr;
   /* MISSING: any checking for initial state. */
   /* 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 affected by this request.
    * The numpages() macro (sysmacros.h) returns the number of pages
    * that span a given length starting at a given address, allowing
    * for partial pages.  Unrealistically, we limit this to the
    * 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_bcount);
   if(npages > GBD_NUM_DMA_PGS) {
       bp->b_resid = IO_NBPP * (npages - GBD_NUM_DMA_PGS);
       npages = GBD_NUM_DMA_PGS;
```

```
        cmn_err(CE_WARN,
            "request too large, only %d pages max", npages);
    }
    /* 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));
       v_addr += IO_NBPP;
    }
    /* Program the device for input or output */
    if ((bp->b_flags & B_READ) == 0)
       gbd_device[unit]->direction = GBD_WRITE;
    else
       gbd_device[unit]->direction = GBD_READ;
/* Start the device going and return. The caller, either a
 * file system or uiophysio(), waits for the iodone() call
 * from the interrupt routine.
 */
    gbd_device[unit]->command = GBD_GO;
}
```

DMA Without Scatter/Gather Support

When the GIO device does not provide scatter/gather capability, the driver must program the transfer of each memory page individually, ensuring that the device does not attempt to store or load across a page boundary. The usual method is as follows:

- In the *pfx***strategy()** routine, save the address of the *buf_t* for use by the *pfx***intr()** entry point.
- In the *pfx***strategy()** routine, program the device to transfer the data for the first page, and start the device going.
- In the *pfx***intr()** entry point, calculate the number of bytes remaining to transfer. If the count is zero, signal **biodone()**. If the count is nonzero, program the device to transfer the next page of data.

Under this design, there is no explicit loop over the successive pages of the transfer visible in the code. The loop is implicit in the fact that the *pfx***intr()** entry point starts a new transfer, and so will be called again, until the transfer is complete.

Example 18-4 shows the code of the *pfx***strategy()** routine for a hypothetical GIO device without scatter/gather.

Example 18-4 Strategy() Code for GIO Device Without Scatter/Gather

```
/* Actual device setup for DMA, etc., when the board
 * does NOT have hardware scatter/gather DMA support.
 * Called from the hypo_write() routine via physio().
 */
void
hypo_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   /* MISSING: any checking for initial state. */
   /* 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;
   }
   /* Save ->buf_t where interrupt handler can find it */
   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] = IO_NBPP-
      ((unsigned int)gbd_curaddr[unit] & (IO_NBPP-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) */
}
```

An alternate design might seem conceptually simpler: to put an explicit loop in the *pfx***strategy()** routine, starting each page transfer and waiting on a semaphore until the *pfx***intr()** routine is called. Such a design keeps the complexity in the *pfx***strategy()** routine, making the *pfx***intr()** routine as simple as possible. However, it has a high cost in performance because the *pfx***strategy** routine must wake up and be dispatched for every page.

Scatter/gather programming can be simplified by the use of the **sgset()** function, which calculates the physical addresses and lengths for each page in the transfer (see the sgset(D3) reference page). The **sgset()** function is limited to use with hardware that uses a fixed mapping of bus addresses to memory addresses, which is the case in the workstations supporting GIO. For example, **sgset()** cannot be used in the Challenge or Onyx line; it always returns -1 in those systems.

Memory Parity Workarounds

Beginning with IRIX 5.3, parity checking is enabled on the SysAD bus, which connects the CPU to memory in workstations that use the GIO bus (see Figure 18-1). Unfortunately, with certain GIO cards, errors can occur if memory reads complete before the Memory Controller (MC) finishes calculating parity.

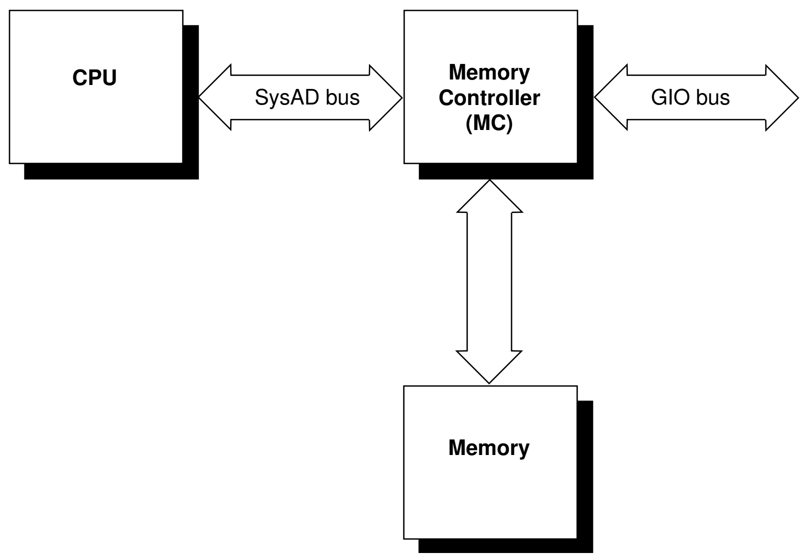

Figure 18-1 The SysAD Bus in Relation to GIO

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 or 16-bit transfers, 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 be received as a parity error on the SysAD bus.

Note: Diagnosis is complicated by the fact that this problem may not show up on every transaction. It occurs only when one of the data lines that is left floating happens to change state between the start of the MC parity calculation and the completion of the CPU read. A device and its driver can appear to function correctly for some time before the problem occurs.

When writing a driver for a GIO card that does not drive all 32 data lines, you must either disable SysAD parity checking completely, or disable it during the time your driver is performing PIO transfers. Three kernel functions are supplied for these purposes; none of them take arguments.

- **is_sysad_parity_enabled()** returns a nonzero value if SysAD parity checking is enabled.
- **disable_sysad_parity()** turns off parity checking on the SysAD bus.
- **enable_sysad_parity()** returns SysAD parity checking to normal.

To completely disable SysAD parity checking removes the system's ability to recover from a parity error in main memory. As a short-term fix, a driver could simply call **disable_sysad_parity()** in the *pfx***init()** or *pfx***edtinit()** entry point.

It is much better to disable parity checking only during the time the device is being used. The advantage here is that the software recovery procedures for memory parity errors are almost always in effect.

To selectively disable parity checking, put wrappers around your driver's PIO transactions to disable SysAD parity checking before a transfer, and to re-enable it after the PIO completes. Example 18-5 shows a skeleton of such a wrapper.

Example 18-5 Disabling SysAD Parity Checking During PIO

```
void
do_PIO_without_parity()
{
   int was_enabled = is_sysad_parity_enabled();
   if (was_enabled) disable_sysad_parity();
/* do driver PIO transfers */
if (was_enabled) enable_sysad_parity();
}
```

The reason that the function in Example 18-5 saves the current state of parity, and only re-enables parity when it was enabled on entry, is that parity checking could have been turned off in some higher-level routine. For example, an interrupt handler could be entered during execution of a device driver function that disables parity checking. If the interrupt handler turned parity checking back on regardless of its former state, errors would occur.

Example GIO Driver

The code in Example 18-6 displays a complete device driver for a hypothetical device. The driver prefix is *gbd* (for “GIO board”).

Example 18-6 Complete Driver for Hypothetical GIO Device

```
/* Source for a hypothetical 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 6.2 or later.
 * Dave Olson, 5/93.  6.2 port by Dave Cortesi 9/95.
*/

/* Compilation: Define the environment variable CPUBOARD as IP20, IP22,
 * or IP26 (the only GIO platforms). Then include the build rules from
 * /var/sysgen/Makefile.kernio to set $CFLAGS including:
#   _K32U32     kernel in 32 bit mode running only 32 bit binaries
#   _K64U64     kernel in 64 bit mode running 32/64 bit binaries (IP26)
#   -DR4000     R4000 machine (IP20, IP22)
#   -DTFP       R8000 machine (IP26)
#   -G 8        global pointer set to 8 (GIO drivers cannot be loadable)
#   -elf        produce an elf executable
 */

/* 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 8   /* 0 for no hardware scatter/gather
                             * support, else number of pages of
                             * scatter/gather per request */
#include <sys/param.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>
#include <sys/conf.h> /* for flags D_MP */

/* gbd (for Gio BoarD) is the driver prefix, specified in the
 * file /var/sysgen/master.d/gbd and in VECTOR module=gbd lines.
```

```
 * This driver is multiprocessor-safe (even though no GIO platform
 * is a multiprocessor).
 */
int gbddevflags = D_MP;

/* these defines and structures defining the (hypothetical) hardware
 * interface 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

/* device register interface to the board */
typedef struct gbd_device {
    __uint32_t  command;
    __uint32_t  count;
    __uint32_t  direction;
    __uint32_t  offset;
    __uint32_t  status; /* errors, interrupt pending, etc. */
#if (!GBD_NODMA)        /* if hardware DMA */
#if (GBD_NUM_DMA_PGS)   /* if hardware scatter/gather */
    /* board register points to array of GBD_NUM_DMA_PGS target
     * addresses in board memory.  Board can relocate the array
     * by changing the content of sgregisters.
     */
    volatile paddr_t    *sgregisters;
#else                   /* dma to contiguous segment only */
    paddr_t     startaddr;
#endif
#endif
} gbd_regs;
```

```
static struct gbd_info {
    gbd_regs    *gbd_device;    /* ->board regs */
    char        *gbd_memory;    /* ->on-board memory */
    sema_t      use_lock;       /* upper-half exclusion from board */
    lock_t      reg_lock;       /* spinlock for interrupt exclusion */
#if GBD_NODMA
    int         gbd_state;      /* transfer state of PIO driver */
    sv_t        intr_wait;      /* sync var for waiting on intr */
#else /* DMA supported somehow */
    buf_t       *curbp;         /* current buf struct */
#if (0 == GBD_NUM_DMA_PGS)  /* software scatter/gather */
    caddr_t     curaddr;        /* current address to transfer */
    int         curcount;       /* count being transferred */
    int         totcount;       /* total size this transfer */
#endif
#endif
} gbd_globals[2];

void gbdintr(int, struct eframe_s *);

/* early device table initialization routine. Validate the values
 * from a VECTOR line and save in the per-device info structure.
 */
void
gbdedtinit(register edt_t *e)
{
    int slot;           /* which slot this device is in */
    __uint32_t val = 0; /* board ID value */
    register struct gbd_info *inf;

    /* Check to see if the device is present */
    if(!badaddr(e->e_base, sizeof(__uint32_t)))
        val = *(__uint32_t *)(e->e_base);
    if ((val && GBD_MASK) != GBD_BOARD_ID) {
        if (showconfig) {
            cmn_err (CE_CONT, "gbdedtinit: board not installed.");
        }
        return;
    }
    /* figure out slot from VECTOR base= value */
    if(e->e_base == (caddr_t)0xBF400000)
        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 IP20 /* for Indigo R4000, set up board as a realtime bus master */
    setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);
#endif
#if (IP22|IP26) /* for Indigo2, set up as a pipelined, realtime bus master */
    setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);
#endif
    /* Initialize the per-device (per-slot) info, including the
     * device addresses from the edt_t.
     */
    inf = &gbd_globals[GIO_SLOT_0 ? 0 : 1];
    inf->gbd_device = (struct gbd_device *)e->e_base;
    inf->gbd_memory = (char *)e->e_base2;
    initsema(&inf->use_lock,1);
    spinlock_init(&inf->reg_lock,NULL);
    setgiovector(GIO_INTERRUPT_1,slot,gbdintr,0);
    if (showconfig) {
        cmn_err (CE_CONT, "gbdedtinit: board %x installed\n", e->e_base);
    }
}
/* OPEN: minor number used to select slot. Merely test that
 * the device was initialized.
 */
/* ARGSUSED */
gbdopen(dev_t *devp, int flag, int otyp, cred_t *crp)
{
    if(! (gbd_globals[geteminor(*devp)&1].gbd_device) )
        return ENXIO;   /* board not present */
    return 0;   /* OK */
}
/* CLOSE: Nothing to do. */
/* ARGSUSED */
gbdclose(dev_t dev, int flag, int otyp, cred_t *crp)
{
    return 0;
}
#if (GBD_NODMA) /***** Non-DMA, therefore character, device ******/
/* WRITE: for character device using PIO */
/* READ entry point same except for direction of transfer */
int
gbdwrite(dev_t dev, uio_t *uio)
{
```

```
    int unit = geteminor(dev)&1;
    struct gbd_info *inf = &gbd_globals[unit];
    int size, err=0, lk;
    /* Exclude any other top-half (read/write) user */
    psema(&inf->use_lock,PZERO)
    /* while there is data to transfer */
    while((size=uio->uio_resid) > 0) {

        /* Transfer no more than GBD_MEMSIZE bytes per operation */
        size = (size < GBD_MEMSIZE) ? size : GBD_MEMSIZE;

        /* Copy data from user-process memory to board memory.
         * uiomove() updates uio fields and copies data
         */
        if(! (err=uiomove(inf->gbd_memory, size, UIO_WRITE, uio)) )
            break;

        /* Block out the interrupt handler with a spinlock, then
         * program the device to start the transfer.
         */
        lk = mutex_spinlock(&inf->reg_lock);
        inf->gbd_device->count = size;
        inf->gbd_device->command = GBD_GO;
        inf->gbd_state = GBD_INTR_PEND; /* validate an interrupt */
        /* Give up the spinlock and sleep until gdbintr() signals */
        sv_wait(&inf->intr_wait,PZERO,&inf->reg_lock,lk);
    } /* while(size) */
    vsema(&inf->use_lock); /* let another process use board */
    return err;
}
/* INTERRUPT: for PIO only board */
/* ARGSUSED1 */
void
gbdintr(int unit, struct eframe_s *ef)
{
    register struct gbd_info *inf = &gbd_globals[unit];
    int lk;
    /* get exclusive use of device regs from upper-half */
    lk = mutex_spinlock(&inf->reg_lock);

    /* if the interrupt is not from our device, ignore it */
    if(inf->gbd_device->status & GBD_INTR_PEND) {
        /* MISSING: test device status, clean up after interrupt,
         * post errors into inf->state for upper-half to see.
         */
```

```
        /* Provided the upper-half expected this, wake it up */
        if (inf->gbd_state & GBD_INTR_PEND)
            sv_signal(&inf->intr_wait);
    }
    mutex_spinunlock(&inf->reg_lock,lk);
}

#else /******** DMA version of driver ************/

void gbd_strategy(struct buf *);

/* WRITE entry point (for character driver of DMA board).
 * Call uiophysio() to set up and call gbd_strategy routine,
 * where the transfer is actually done.
*/
int
gbdwrite(dev_t dev, uio_t *uiop)
{
    return uiophysio((int (*)())gbd_strategy, 0, dev, B_WRITE, uiop);
}
/* READ entry point same except for direction of transfer */
#if GBD_NUM_DMA_PGS > 0

/* STRATEGY for hardware scatter/gather DMA support.
 * Called from gbdwrite()/gbdread() via physio().
 * Called from file-system/paging code directly.
 */
void
gbd_strategy(register struct buf *bp)
{
    int unit = geteminor(bp->b_edev)&1;
    register struct gbd_info *inf = &gbd_globals[unit];
    register gbd_regs *regs = inf->gbd_device;
    volatile paddr_t *sgregisters;
    int npages;
    int i, lk;
    caddr_t v_addr;

    /* Get the kernel virtual address of the data. Note that
     * b_dmaaddr is NULL when 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");
        bp->b_flags |= B_ERROR;
        iodone(bp);
        return;
    }
    v_addr = bp->b_dmaaddr;

    /* Compute number of pages affected by this request.
     * The numpages() macro (sysmacros.h) returns the number of pages
     * that span a given length starting at a given address, allowing
     * for partial pages.  Unrealistically, we limit this to the
     * 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_bcount);
    if(npages > GBD_NUM_DMA_PGS) {
        bp->b_resid = IO_NBPP * (npages - GBD_NUM_DMA_PGS);
        npages = GBD_NUM_DMA_PGS;
        cmn_err(CE_WARN,
            "request too large, only %d pages max", npages);
    }

    /* Get exclusive upper-half use of device. The sema is released
     * wherever iodone() is called, here or in the int handler.
     */
    psema(&inf->use_lock,PZERO)
    inf->curbp = bp;

    /* Get exclusive use of the device regs, blocking the int handler */
    lk = mutex_spinlock(&inf->reg_lock);

    /* MISSING: set up board to transfer npages discreet segments. */
    /* Get address of the scatter-gather registers */
    sgregisters = regs->sgregisters;

    /* Provide the beginning byte offset and count to the device. */
    regs->offset = io_poff(bp->b_dmaaddr); /* in immu.h */
    regs->count = (IO_NBPP - inf->gbd_device->offset)
                    + (npages-1)*IO_NBPP;

    /* Translate the virtual address of each page to a
```

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

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

    /* release use of the device regs to the interrupt handler */
    mutex_spinunlock(inf->reg_lock,lk);

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

/* INTERRUPT: for hardware DMA support. This is over-simplified
 * because the above strategy routine never accepts a transfer
 * larger than the device can handle in a single operation.
 */
/* ARGSUSED1 */
void
gbdintr(int unit, struct eframe_s *ef)
{
    register struct gbd_info *inf = &gbd_globals[unit];
    register gbd_regs *regs = inf->gbd_device;
    int error = 0;
    int lk;

    /* get exclusive use if device regs from upper-half */
    lk = mutex_spinlock(&inf->reg_lock);

    /* If interrupt was not from this device, exit quick */
    if (! (regs->status & GBD_INTR_PEND) ) {
        mutex_spinunlock(&inf->reg_lock,lk);
        return;
    }

    /* MISSING: read board registers, clear interrupt,
```

```
     * and note any errors in the "error" variable. */
    if(error)
        inf->curbp->b_flags |= B_ERROR;

    /* release lock on exclusive use of device regs */
    mutex_spinunlock(&inf->reg_lock,lk);

    /* wake up any kernel/file-system waiting for this I/O */
    iodone(inf->curbp);

    /* unlock use of device to other upper-half driver code */
    vsema(&inf->use_lock);
}

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

/* STRATEGY: for software-controlled scatter/gather.
 * Called from the gbdwrite() routine via uiophysio().
 */
void
gbd_strategy(struct buf *bp)
{
    int unit = geteminor(bp->b_edev)&1;
    register struct gbd_info *inf = &gbd_globals[unit];
    register gbd_regs *regs = inf->gbd_device;
    int lk;

    /* 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");
        bp->b_flags |= B_ERROR;
        iodone(bp);
        return;
    }

    /* Get exclusive upper-half use of device. The sema is released
     * wherever iodone() is called, here or in the int handler.
```

```
         */
        psema(&inf->use_lock,PZERO)
        inf->curbp = 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.
         */
        inf->curaddr = bp->b_dmaaddr;
        inf->totcount = bp->b_bcount;
        inf->curcount = IO_NBPP - io_poff(inf->curaddr);
        if (bp->b_bcount < inf->curcount)
            inf->curcount = bp->b_bcount;

        /* Get exclusive use of the device regs and start the transfer
         * of the first/only segment of data. */
        lk = mutex_spinlock(&inf->reg_lock);
        regs->startaddr = kvtophys(inf->curaddr);
        regs->count = inf->curcount;
        regs->direction = (bp->b_flags & B_READ) ? GBD_READ : GBD_WRITE;
        regs->command = GBD_GO; /* start DMA */

        /* release use of the device regs to the interrupt handler */
        mutex_spinunlock(inf->reg_lock,lk);
        /* and return; upper layers of kernel wait for iodone(bp) */
}

/* INTERRUPT: for software scatter/gather. This version is more typical
 * of boards that do have DMA, and more typical of devices that support
 * block i/o, as opposed to character i/o.
 */
/* ARGSUSED1 */
void
gbdintr(int unit, struct eframe_s *ef)
{
        register struct gbd_info *inf = &gbd_globals[unit];
        register gbd_regs *regs = inf->gbd_device;
        register buf_t *bp = inf->curbp;
        int error = 0;
        int lk;


        /* get exclusive use if device regs from upper-half */
        lk = mutex_spinlock(&inf->reg_lock);
```

```
    /* If interrupt was not from this device, exit quick */
    if (! (regs->status & GBD_INTR_PEND) ) {
        mutex_spinunlock(&inf->reg_lock,lk);
        return;
    }

    /* MISSING: read board registers, clear interrupt,
     * and note any errors in the "error" variable. */
    if(error) {
        bp->b_resid = inf->totcount; /* show bytes undone */
        bp->b_flags |= B_ERROR; /* flag error in transfer */
        iodone(bp); /* we are done, tell upper layers */
        vsema(&inf->use_lock); /* make device available */
    }
    else {
        /* Note the successful transfer of one segment. */
        inf->curaddr += inf->curcount;
        inf->totcount -= inf->curcount;
        if(inf->totcount <= 0) {
            iodone(bp); /* we are done, tell upper layers */
            vsema(&inf->use_lock); /* make device available */
        }
        else {
            /* More data to transfer. Reprogram the board for
             * the next segment and start the next DMA.
             */
            inf->curcount = (inf->totcount < IO_NBPP) ? inf->totcount : IO_NBPP;
            regs->startaddr = kvtophys(inf->curaddr);
            regs->count = inf->curcount;
            regs->direction = (bp->b_flags & B_READ) ? GBD_READ : GBD_WRITE;
            regs->command = GBD_GO; /* start next DMA */
        }
    }
    /* release lock on exclusive use of device regs */
    mutex_spinunlock(&inf->reg_lock,lk);
}
#endif /*  GBD_NUM_DMA_PGS */
#endif /* GBD_NODMA */
```