Writing Kernel-level GIO Device Drivers For IRIX 4.0.x

This chapter contains the following subsections:

- Section 1, "Determining GIO Device Addresses"
- Section 2, "Including GIO Device Drivers in the Kernel"
- Section 3, "Writing edtinit()"
- Section 4, "GIO-Specific System Calls"
- Section 5, "GIO Interrupt Handler"
- Section 6, "Programmed I/O (PIO)"
- Section 7, "DMA Operations"
- Section 8, "GIO Devices with Hardware Supported Scatter-Gather Capability"
- Section 9, "DMA on GIO Devices with No Scatter-Gather Capability"
- Section 10, "Device Driver Example"

This chapter provides in-depth information about drivers that interface to the GIO bus. System configuration for GIO device drivers is described, and several GIO-specific functions you should include in your device driver are introduced. This chapter also describes several models for performing DMA operations. Which model you choose for your device driver depends on the capability of the device. The device may have hardware support for scatter-gather or may require a software implementation of scatter-gather. Memory mapped, user level drivers for GIO devices are not supported. All GIO drivers must be kernel level, user level drivers are not supported.

This chapter is meant as a supplement to the IRIX Device Driver Programming Guide, Document number 007-0911-020. Consult this document for additional information regarding system architecture and device drivers for IRIX.

1 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. GIO bus addresses cannot be mapped into user address space. GIO devices can be classified as 32 bit or 64 bit. Unlike VME, where the class of device determines the address range, GIO devices each respond to the same address range.

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

2 Including GIO Device Drivers in the Kernel

Chapter 6, "Kernel–Driver Interface Overview," 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 add a directive to the *system* file (*/usr/sysgen/system*). For SCSI drivers, you 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.

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 Section 4.1, "Setgiovector"). Therefore, the vector in the VECTOR statement should 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. This is determined by the slot in which the board is installed. This is a K1 address (see *kvtophys(3K)* man page).
base2, base3	additional addresses passed to driver edtinit routine via edt structure. These are K1 addresses (see *kvtophys(3K)* 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 as follows:

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

Also recall that you should create a master file under */usr/sysgen/master.d. (*The name of the master file is the same as the name of the object file for the driver, but the master file should not have the ".o" suffix.) The FLAG field of the master file should at least include the character device flag c.

As an example, suppose you want to add a mythical GIO device driver to the kernel. You must copy the driver object file *gbd.o* to */usr/sysgen/boot*, and you must add a line similar to the following to the *system* file:

```
VECTOR: module=gbd vector=0x0 unit=0 base=0xBF400000
   base2=0xBF410000 exprobe=(r,0xBF400000,4,0x75,0xff)
```

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 above example, *lboot* reads four bytes at probe address, 0xBF400000, 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.

In actual use it is advisable to add a second VECTOR line to the *system* file. This should perform a probe of the other GIO slot. If only the line above had been used and the GIO device was 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 should be added to the *system* file:

```
VECTOR: 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 */usr/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        -
```

3 Writing edtinit()

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

```
drvedtinit(e)
struct edt *e

{
   your code here
}
```

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

```
struct edt {
    paddr_t   e_base, e_base2, e_base3;
    struct vme_intrs    *e_intr_info;
    int    (*e_init)( );
/* device initialization and run-time probe */
};
```

The **e_intr_info* and (**e_init*)() members are of no interest to your *drvedtinit* routine. Your driver only 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.

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 *drvedtinit* routine 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 *drvedtinit* routine could look like:

```
/* early device table initialization routine. The edt
 * structure is defined in edt.h.
 */
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 == 0xBF400000)
        slot = GIO_SLOT_0;
    else if(e->e_base == 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, set up board as a
                  * realtime bus master
                  */

    setgioconfig(slot,0);

#endif

#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         /* for Indigo2, set up board as a pipelined,
                  * realtime bus master
                  */

    setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST |
                 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,0);
}
```

4 GIO-Specific Functions

There are three GIO Bus specific support routines that must be included in the init or edtinit section of any GIO driver. These are the setgiovector, setgioconfig, and splgio(n) routines.

4.1 Setgiovector

The *setgiovector(2K)* routine registers an interrupt service function for a GIO bus device interrupt with the kernel's interrupt dispatcher.

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

The *level* parameter specifies which interrupt is used by the device. For GIO bus boards this should always be **GIO_INTERRUPT_1** since **GIO_INTERRUPT_0** and **GIO_INTERRUPT_2** are used by the graphics system.

The *slot* parameter specifies which physical slot the GIO bus board is plugged into and should be either **GIO_SLOT_0**, or **GIO_SLOT_1**.

The *func* parameter is a pointer to the interrupt service routine that will get 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, non-destructive manner.

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

4.2 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.

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

For R3000 based machines 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 realtime device
GIO_CONFIG_SLAVE	Configure board as a bus slave, otherwise it will be a bus master

For R4000 based machines 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, this must not be set. For Indigo^2, this must be set.

On R4000 based Indigos and Indigo^{2}s, setgioconfig uses the slot argument to determine the location of boards.

4.3 Splgio0, Splgio1, Splgio2

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

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

5 GIO Interrupt Handler

Your driver module should contain an interrupt routine. The name of this routine must agree with that used in the VECTOR line in */usr/sysgen/system*. Normally this routine is called *drvintr* (where *drv* is the driver prefix). When the device generates an interrupt, the general GIO interrupt handler calls your driver's *drvintr* routine. When the GIO interrupt handler calls your *drvintr*, it passes it the unit number for the device. Within your *drvintr* routine, you should 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*(K) to indicate that a block type I/O transfer for the buffer is complete.

Caution: Interrupt routines (*drvintr*) must not try to sleep themselves by calling *iowait*(K), *sleep*(K), *psema*(K), or *delay*(K) kernel calls. Neither should they try to access the per-process global variables in the *u* type structure directly. The *u* type structure they can access may not be that of the process that made the I/O request.

6 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. In addition, even if the device does support DMA, you may not want to use DMA to transfer amounts of data so small that the overhead of DMA is not warranted.

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

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 data from *u.u_base* 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.

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 *supplement* and *splx*(K) routines.

```
/* device write routine entry point (for character devices) */
gbdwrite(dev_t dev)
{
    int unit = minor(dev)&1;
    int size;
    int s;

    while (u.u_count > 0) {
        /* while there is data to transfer */

        /* Transfer no more than GBD_MEMSIZE bytes
         * to the device */

        size = (u.u_count <
            GBD_MEMSIZE ? u.u_count : GBD_MEMSIZE);

        /* decrements u.u_count while copying data */
        iomove(gbd_memory[unit], size, B_WRITE);
        if (u.u_error)
            break;

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

```
        /* Transfer is complete; start output */
        gbd_device[unit]->count = u.u_count;
        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);
    }
}
```

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 *drvintr* to verify that the actual event has occurred before actually awakening the sleeping process. (See *sleep*(K) 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 *iomove*(K) kernel routine is a useful procedure to call in these situations because it automatically updates *u.u_count*, *u.u_offset*, and *u.u_base* and uses *copyout*(K) (or *copyin*(K)) to check for invalid user addresses. Recall that *u.u_count* must be left with the number of bytes left untransferred.

7 DMA Operations

As indicated in Section 6, "Programmed I/O (PIO)", you should use DMA 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 a DMA read or a DMA write. DMA operations that transfer from memory to device, and hence read memory, are DMA reads. DMA operations that transfer from device to memory are DMA writes. Thus, the point of view is that of memory. A disk read results in a DMA write, and a disk write results in a DMA read.

There are some cache considerations for drivers using DMA. The cache architecture of the machine 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 Section A.2, "Data Cache Write Back and Invalidation," and *dki_dcache_wbinval*(K) for a discussion of these issues.

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

The interrupt service routine then calls your *drvintr* routine. Your *drvintr* routine can check 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.

8 GIO Devices with Hardware Supported Scatter-Gather Capability

Chapter 5, "Creating User-level Device Drivers," 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 *drvstrategy* routine and passes it a pointer to the *buf* type structure that it has allocated and primed. Your *drvstrategy* routine should 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*(K) 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:

```
/* block device read/write entry point, if your board has
 * hardware scatter/gather DMA support.
 */
gbdstrategy(struct buf *bp)
{
    int unit = minor(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");
        bp->b_flags |= B_ERROR;
        iodone(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. 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));

   /*
   /* 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) */
}
```

9 DMA on GIO Devices with No Scatter-Gather Capability

If your device does not provide any scatter-gather capability, your driver must break up a data transfer so that no transfer crosses a page boundary. The IRIX operating system provides a utility, *sgset*(K), that simulates scatter-gather registers in software. (See Appendix C, "Man Pages for the Kernel Functions," 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:

```
gbdstrategy(struct buf *bp)
{
    int unit = minor(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");
        bp->b_flags |= B_ERROR;
        iodone(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) */
}
```

10 Device Driver Example

On the following pages is 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.

Normally defines for a driver are kept in a separate header file. In the case of this mythical device these defines would be found in *gbd.h*. For this example these defines are contained in the driver source file itself.

To compile this example, the following command line would be used assuming the source code is found in a file named *gbd.c*:

For an Indigo (R3000): `cc -DIP12 -DR3000 -cckr -c gbd.c`

For an Indigo (R4000): `cc -DIP20 -DR4000 -cckr -c gbd.c`

For an Indigo[2] (R4000): `cc -DIP20 -DR4000 -cckr -c gbd.c`

```
#define _KERNEL /* more typically set on compile line */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/cpu.h>
#include <sys/buf.h>
#include <sys/user.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 seperate
 * 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_NUM_DMA_PGS  4/* 0 for no hardware scatter/gather
   * support, else number of pages of scatter/gather
   * supported per request */

#define GBD_NODMA 0/* non-zero for PIO version of driver */

#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);
```

```
/* early device table initialization routine.  The edt
 * structure is defined in edt.h.
 */
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 == 0xBF400000)
        slot = GIO_SLOT_0;
    else if(e->e_base == 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, set up board as a
                 * realtime bus master
                 */

    setgioconfig(slot,0);

#endif

#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         /* for Indigo2, set up board as a pipelined,
                  * realtime bus master
                  */

    setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST |
                 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,0);
}


#ifdef GBD_NODMA

/* device write routine entry point (for character devices) */
gbdwrite(dev_t dev)
{
    int unit = minor(dev)&1;
    int size;
    int s;

    while (u.u_count > 0) {
        /* while there is data to transfer */

        /* Transfer no more than GBD_MEMSIZE bytes
         * to the device */

        size = (u.u_count <
            GBD_MEMSIZE ? u.u_count : GBD_MEMSIZE);

        /* decrements u.u_count while copying data */
        iomove(gbd_memory[unit], size, B_WRITE);
        if (u.u_error)
            break;

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

```
        /* Transfer is complete; start output */
        gbd_device[unit]->count = u.u_count;
        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);
    }
}


/* interrupt routine for PIO only board, just wake up
 * upper half of driver
 */
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 */
}

#else/* DMA version of driver */

#if GBD_NUM_DMA_PGS > 0
```

```
/* block device read/write entry point, if your board has
 * hardware scatter/gather DMA support.
 */
gbdstrategy(struct buf *bp)
{
    int unit = minor(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 handled unmapped buffers");
       bp->b_flags |= B_ERROR;
       iodone(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.  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));

       /*
       /* 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)
        bp->b_flags |= B_ERROR;

    iodone(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 */

gbdstrategy(struct buf *bp)
{
    int unit = minor(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 handled unmapped buffers");
        bp->b_flags |= B_ERROR;
        iodone(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) {
        bp->b_flags |= B_ERROR;
        iodone(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)
            iodone(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 */
```