Chapter 5

Writing a SCSI Device Driver

This chapter gives an overview of the SCSI device driver interface and explains how to write a user-level SCSI device driver and a kernel-level SCSI device driver. It contains the following sections:

SCSI-bus Interface Overview

SCSI (Small Computer System Interface) is an industry standard I/O bus designed to provide host computers with device independence within a class of devices. All Silicon Graphics systems provide an interface to at least a single SCSI bus for peripherals that support the SCSI standard. Your device driver can place commands on the bus by using the SCSI host adapter driver. Systems with POWERchannel I/O processor boards support two SCSI interfaces per POWERchannel board; those with POWERchannel-2 boards support two native SCSI interfaces plus as many as six additional SCSI interfaces, if mezzanine boards are used, for a maximum of eight SCSI interfaces per POWERchannel-2 board. Systems equipped with VME-SCSI (Jaguar) boards provide two buses per board.

The drivers support all three SCSI standards, SCSI-1, CCS[1], and SCSI-2. Not all optional features are supported, and different systems support different features (such as synchronous, fast, and wide).

In addition, DMA address mapping registers allow noncontiguous physical memory to appear contiguous to the SCSI host adapter. This allows your driver to handle I/O across noncontiguous pages in a single transfer (scatter-gather). The IRIX operating system provides an interface that hides much of the complexity of the SCSI bus management.

Choosing a Driver Model

Choosing between a user-level and a kernel-level device driver model usually depends on the method used to transfer data to and from the device.

Note: Silicon Graphics has a generic SCSI device driver to support its SCSI hardware. This driver has entry points that enable programs to control devices unknown to the generic Silicon Graphics SCSI device driver. In other words, there are hooks to extend the SGI SCSI device driver to manage customer SCSI devices. In the strictest sense, this is not a device driver but an extension to a device driver.

---

[1] The Common Command Set (CCS) standard is superseded by SCSI-2.

User-level SCSI-bus Device Driver

If you must write a driver for a device that conforms to the SCSI standard, you can often use the */dev/scsi/sc** special files and the *dslib* library of routines and macros to write a user-level program that sends SCSI commands to a device on the SCSI bus.

Although this *dslib* interface to the SCSI device differs from the IRIX standard device interface and requires that you be familiar with the SCSI protocol, you can typically write this sort of user-level SCSI device driver in a fraction of the time that it takes to write a kernel-level SCSI device driver. As a result, even if your ultimate goal is to write a kernel-level driver for a device, you can write a user-level device driver to test the device and familiarize yourself with its features.

However, there are some limits. For example, quasi-SCSI devices that do not strictly adhere to SCSI standards for reporting errors can cause problems. In earlier IRIX releases, the user-level driver did not support SCSI commands of an unusual length (commands not of *grp0*, *grp1*, or *grp2* length). IRIX 4.0 and later supports commands of 6-12 bytes, regardless of group. Finally, the *dslib* routines and macros may be somewhat less efficient (in terms of throughput) than a kernel-level SCSI driver. This lower efficiency may not be a problem when dealing with devices such as scanners and printers. DMA is used for all I/O, just as with kernel SCSI drivers.

IRIX uses the *devscsi* driver for CDROM ISO-9660 filesystems as well as for HFS (Apple® Macintosh®) and DOS™ floppy filesystems.

Kernel-level SCSI Device Driver

You may have to write a kernel-level SCSI driver for a SCSI device when you need the best throughput possible or when you must control an unusual SCSI device. See Chapter 2, “Writing a Device Driver,” for a general description of the IRIX device driver interface.

User-level SCSI Device Drivers

This section explains how to write a user-level driver for a SCSI device. It describes the SCSI interface routines for the integral SCSI controller and gives a short example.

In IRIX 5.0 and later releases, a SCSI target may, in most cases, be used by more than one driver at a time unless it is opened in exclusive mode (currently, only the *tpsc* tape driver uses exclusive mode). See “Kernel-level SCSI Device Drivers” for more information.

Note: There have been extensive changes in the kernel SCSI driver interface in the IRIX 5.x releases. In IRIX 5.x, the *devscsi* interface is available on the **wd93**, **wd95**, and **VMESCSI** (Jaguar) drivers. In prior releases, it was available only on the **wd93**. IRIX 5.2 and 5.3 remain source compatible, however.

To control a SCSI device from a user-level program, you need the routines of the *dslib* library and a device-special file created for the device. The system comes with device-special files for SCSI adapters in the directory */dev/scsi*. These files can be especially useful because they insulate the device driver writer from changes in operating system releases, and they remain valid across all Silicon Graphics platforms, independent of the low-level SCSI driver. Their limitation is that they are created for logical unit 0 only. (If the logical unit number for your device is not zero, you need to create a device-special file for the device. See “Creating Device-special Files for User-level SCSI Drivers.”)

Which special files you use depends on the SCSI ID for the device, a number from 1-7 (wide SCSI is 1-15). The SCSI ID for a device is usually controlled by switch settings or jumpers. (See the device technical specification for details.) For a SCSI device with a SCSI ID of 3 on controller 0, use */dev/scsi/sc0d3l0*; use */dev/scsi/sc0d7l0* for a SCSI device with a SCSI ID of 7, and so on. See the man page for the **ds**(7M) command for more details on device naming.

The remainder of this chapter assumes that you are familiar with the SCSI interface. For additional information on SCSI-bus operation, see the *ANSI Standards X3.131-1986* and *X3T9.2/85-52 Rev 4B*.

Creating Device-special Files for User-level SCSI Drivers

If the logical unit for the SCSI device you want to control from a user-level program is some value other than 0, you need to create a device-special file. See the **mknod**(1) and **ds**(7) man pages for more information. To create a device-special file, use the **mknod** command:

```
% mknod filename type major# minor#
```

where:

filename — The name of the device special file to create for the device.

type — The type of special file:
c = character special file
b = block special file

major# — The major device number. For a device that will be controlled from a user-level program, this number is 195, which is the major device number of *devscsi*.

minor# — The 14-bit (decimal) minor device. The bits are defined as shown in Figure 5-1.

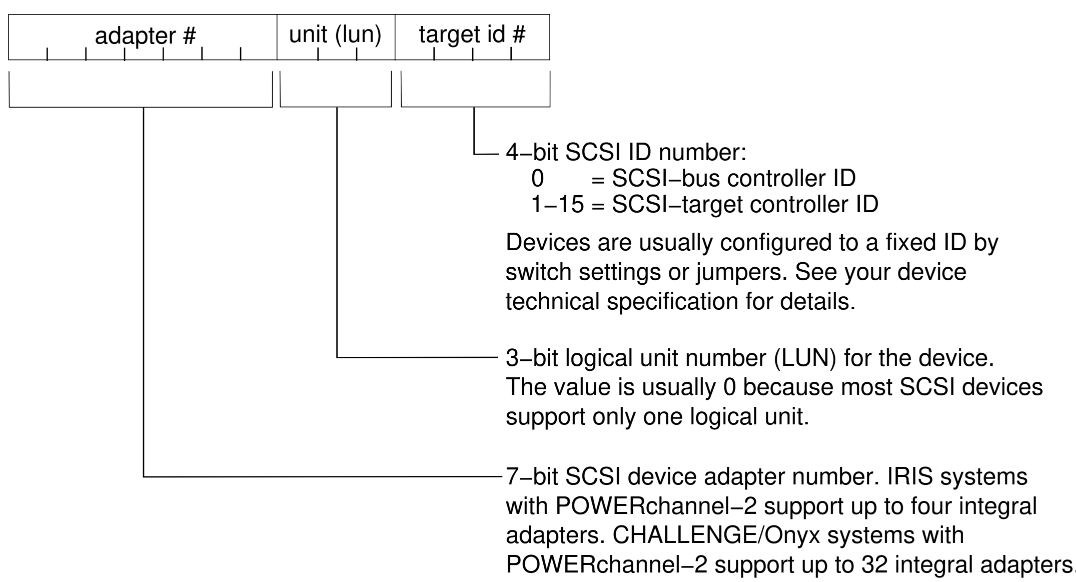

Figure 5-1 Minor Number Bit Definitions

Example

```
% mknod /dev/scsi/sc0d1l1 c 195 17
```

The minor device number is set to 17_{10}, which is 0010001_2, thus indicating a device that has logical unit number 1 and a target ID of 1.

dsreq – User-level Driver Communication Structure

Your user-level SCSI driver communicates with a SCSI device by reading and setting the values of the members of the *dsreq* type structure. Understanding this structure is essential to writing a user-level SCSI driver. Your driver can access these fields directly; however, for many common tasks that involve controlling a SCSI device, *dslib* has simple routines or macros that can access these values for you. The advantage of using these macros and routines is that, if the structure should change in a future release, you can accommodate the change by changing the internals of the macros or simply recompiling with new macro defs in *dsreq.h*. Thus, code that uses these macros and routines is likely to be portable across releases even if the structure itself changes.

The *dsreq* type structure can be found in the *sys/dsreq.h* directory. The macros associated with a *dsreq* type structure are described below. The *dslib* routines are described in “dslib Routines Description” on page 143.

dsreq Structure

```
typedef struct dsreq {
/* devscsi prefix */

ulong          ds_flags;     /* see flags defined below */
ulong          ds_time;      /* time-out in milliseconds */
ulong          ds_private;   /* for private use by caller */
/* scsi request */
caddr_t        ds_cmdbuf;    /* command buffer */
uchar_t        ds_cmdlen;    /* command buffer length */
caddr_t        ds_databuf;   /* data buffer start */
ulong          ds_datalen;   /* total data length */
caddr_t        ds_sensebuf;  /* sense buffer */
uchar_t        ds_senselen;  /* sense buffer length */
/* miscellaneous */
```

```
dsiovec_t      *ds_iovbuf;   /* scatter-gather dma control */
ushort         ds_iovlen;    /* length of scatter-gather */
struct dsreq   *ds_link;     /* for linked requests */
ushort         ds_synch;     /* synchronous xfer control*/
uchar_t        ds_revcode;   /* devscsi version code*/

/* return portion */
uchar_t         ds_ret;       /* devscsi return code*/
uchar_t         ds_status;    /* device status byte value*/
uchar_t         ds_msg;       /* device message byte value */
uchar_t         ds_cmdsent;   /* actual length command*/
ulong           ds_datasent;  /* actual length user data*/
uchar_t         ds_sensesent; /* actual length sense data*/
} dsreq_t;
```

where:

ds_flags The bits of the value for this member are used as flags that determine what the SCSI-bus driver does after you call **doscsireq**(). Use the **FLAGS** macro to access the value of this member. The interface is designed to be implemented on a number of architectures, some of which provide more low-level control of the SCSI bus than IRIX. The file, *dsreq.h* included by *dslib.h*, currently defines this macro as:

```
#define FLAGS (dp) ((dp)->ds_flags)
```

There are symbolic constants that you can use to set the bits of the *ds_flags* value. (Not all of these flags are currently honored. Your driver can test for which flags are honored by checking the returned value of an **ioctl**() call on *devscsi*. See the **ds**(7M) man page.) These constants are defined in *sys/dsreq.h*:

DSRQ_ASYNC – No/Yes sleep until request done. A SCSI device option. Not implemented.

DSRQ_SENSE – Yes/No automatically get sense on status when a check condition occurs. A SCSI device option. All requests return only on completion. Not implemented.

DSRQ_TARGET – Target/Initiator role. A SCSI device option. Not implemented.

DSRQ_SELATN – Select with/without ATN. A select option. Not implemented.

DSRQ_DISC – Identify disconnect not-allowed/allowed. A select option. Not implemented.

DSRQ_SYNXFR – Negotiate synchronous SCSI transfer. A select option.

DSRQ_SELMSG – Send supplied/generated message. A select option. Not implemented.

DSRQ_IOV – Scatter-gather not-specified/specified. A data transfer option.

DSRQ_READ – Input data from SCSI bus to the CPU. A data transfer direction.

DSRQ_WRITE – Output data to SCSI bus from the CPU. A data transfer direction.

DSRQ_BUF – Buffered/Direct data transfer. A data transfer option.

DSRQ_CALL – Notify progress upon completion. A progress/continuation callback option. Used with *DSRQ_ASYNC*. Not implemented.

DSRQ_ACKH – Hold/Don't-hold ACK asserted. A progress/continuation callback option. Not implemented.

DSRQ_ATNH – Hold/don't-hold ATN asserted. A progress/continuation callback option. Not implemented.

DSRQ_ABORT – Send an abort message. Useful only with SCSI commands that have the immediate bit set.

DSRQ_TRACE – Trace/don't-trace this request. A host option (and so not likely to be portable).

DSRQ_PRINT – Print/don't-print this request. A host option (and so not likely to be portable).

DSRQ_CTRL1 – Request with host control bit 1. A host option (and so not likely to be portable).

DSRQ_CTRL2 – Request with host control bit 2. A host option (and so not likely to be portable).

DSRQ_MIXRDWR – Request can both read and write.

ds_time This member sets the time-out value in milliseconds for the completion of a command sent to the SCSI device. You can use the **TIME** macro to access the value of this member.

The file *dsreq.h* defines this macro as:

```
#define TIME (dp) ((dp)->ds_time)
```

If you use *dslib*, you must not change this pointer or the data it references.

ds_private To access the value of the *ds_private* member, you can use the **PRIVATE** macro currently defined in *dslib.h* as:

```
#define PRIVATE(dp) ((dp)->ds_private)
```

This is intended to be used only in the library support routines.

ds_cmdbuf This member is a pointer to an array, the SCSI command descriptor you want to send to the device. You can use the **CMDBUF** macro to access this value. The file *dsreq.h* currently defines this macro as:

```
#define CMDBUF(dp) ((caddr_t) (dp)->ds_cmdbuf)
```

ds_cmdlen This member is the length (in bytes) of the SCSI command pointed to by *ds_cmdbuf*. You can use the **CMDLEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define CMDLEN(dp) ((dp)->ds_cmdlen)
```

Typically, this value is 6, 10, or 12 for SCSI commands of class 0, 1, or 2, respectively.

ds_databuf This member is a pointer to the start of a data buffer. You can use the **DATABUF** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define DATABUF(dp) ((caddr_t) (dp)->ds_databuf)
```

ds_datalen This member is the length of the data in the buffer pointed to by the *ds_databuf* member. You can use the **DATALEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define DATALEN(dp) ((dp)->ds_datalen)
```

ds_sensebuf This member is the pointer to the start of the sense buffer. The contents written to this buffer when you request sense information from a device depend on the device. See the device-specific documentation supplied by the manufacturer. You can use the **SENSEBUF** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define SENSEBUF(dp) ((caddr_t) (dp)-
>ds_sensebuf)
```

It is used only if *DSRQ_SENSE* is set in the flags.

ds_senselen This member is the length of the data in the buffer pointed to by the *ds_sensebuf* member. You can use the **SENSELEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define SENSELEN(dp) ((dp)->ds_senselen)
```

**ds_iovbuf* This member is a pointer to a *dsiovec* type structure, a SCSI device I/O vector. This structure is used for DMA scatter-gather control. The *dsiovec* type structure (defined in *dsreq.h*) has two members: *iov_base*, a pointer to a buffer containing a table of physical addresses for the entire transfer, and *iov_len*, the length of the buffer pointed to by *iov_base*.

To access the value of the **ds_iovbuf* member, you can use the macro, **IOVBUF**. The file *dsreq.h* currently defines this macro as:

```
#define IOVBUF(dp) ((caddr_t) (dp)->ds_iovbuf)
```

ds_iovlen This member is the length, in bytes, of the data for scatter-gather transfer. You can use the **IOVLEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define IOVLEN(dp) ((dp)->ds_iovlen)
```

**ds_link* Not supported.

ds_synch Not supported.

ds_revcode This member is the version code for the *devscsi* driver.

ds_ret

This member is the return code for the command executed on the SCSI device. You can use the **RET** macro to access this member. The file *dsreq.h* currently defines **RET**:

```
#define RET(dp) ((dp)->ds_ret)
```

The file *dsreq.h* defines the following symbolic constants for the value pointed to by this member:

DSRT_DEVSCSI – General failure from SCSI bus.

DSRT_HOST – General host failure, typically a SCSI-bus request.

DSRT_STAI – Protocol error during status phase.

DSRT_EBSY – Busy dropped unexpectedly; protocol error.

DSRT_UNIMPL – Protocol error. Not implemented.

DSRT_CMDO – Protocol error during command phase.

DSRT_REJECT – Message rejected; protocol error.

DSRT_PARITY – Parity error on SCSI bus; protocol error.

DSRT_PROTO – Miscellaneous protocol failure.

DSRT_MEMORY – Host memory error.

DSRT_MULT – Request rejected by SCSI bus.

DSRT_CANCEL – Lower request canceled by SCSI bus.

DSRT_REVCODE – Software obsolete, must recompile.

DSRT_AGAIN – Try again, recoverable SCSI-bus error.

DSRT_NOSEL – No unit responded to select.

DSRT_SHORT – Incomplete transfer (not an error).

DSRT_OK – Completed transfer without error status.

DSRT_SENSE – Command with status, sense data successfully retrieved from SCSI host.

DSRT_NOSENSE – Command with status, error occurred while trying to get sense data from SCSI host.

DSRT_TIMEOUT – Request idled longer than requested. Command could not complete within the limit of the time-out value.

DSRT_LONG – Target over ran data bounds.

ds_status This member is the SCSI target's status byte value for the SCSI command just executed. You can use the **STATUS** macro to access this member.

The file *dsreq.h* currently defines **STATUS**:

```
#define STATUS(dsp) ((dp)->ds_status)
```

The file *dsreq.h* defines the following symbolic constants for this byte:

STA_GOOD – the target has successfully completed the SCSI command.

STA_CHECK – indicates an error, exception, or abnormal condition. If *DSRQ_SENSE* is set, request sense is automatically done. See *ds_sensebuf*.

STA_BUSY – the target is busy, so the command was not issued.

STA_IGOOD – SCSI command with link completed.

STA_RESERV – Command aborted because it tried to access a logical unit or an extent within a logical unit that reserves that type of access to another SCSI device.

ds_msg Not implemented.

ds_cmdsent The value of this *member* is the length of the SCSI command actual sent. You can use the **CMDSENT** macro to access this member. The file *dsreq.h* currently defines **CMDSENT**:

```
#define CMDSENT(dsp) ((dp)->ds_cmdsent)
```

ds_datasent The value of this member is the length of the user data actually transferred. You can use the **DATASENT** macro to access this member. The file *dsreq.h* currently defines **DATASENT**:

```
#define DATASENT(dsp) ((dp)->ds_datasent)
```

ds_sensesent The value of this member is the length of the sense data actually received. You can use the **SENSESENT** macro to access this member. The file *dsreq.h* currently defines **SENSESENT**:

```
#define SENSESENT(dsp) ((dp)->ds_sensesent)
```

dslib Routines Description

Table 5-1 and the following section describe the *dslib* routines. You can use these routines to set the values of a *dsreq* type structure and make a request of the SCSI bus that uses the information contained in it.

Table 5-1 dslib Routines

Routine	Description
dsopen	Allocate a *dsreq* type structure and open a device.
dsclose	Free the *dsreq* structure for the SCSI device and close the device.
doscsireq	Send a command to the SCSI device or to make another request.
filldsreq	Set members of a *dsreq* type structure.
fillg0cmd	Set up the *dsreq* structure to send a **group 0 SCSI** command.
fillg1cmd	Set up the *dsreq* structure to send a **group 1 SCSI** command.
inquiry12	Issue an inquiry command and retrieve information from the device concerning such things as its type.
modeselect15	Issue a **group 0 mode select** command to a SCSI device.
modesense1a	Send a **group 0 “mode sense”** command to a SCSI device to retrieve the page information from the device.
readcapacity25	Issue a **read capacity** command to a SCSI device.
readextended28	Issue a **read extended** command to a SCSI device.
requestsense03	Issue a **request sense** command and test or probe for the device.

Table 5-1 (continued) dslib Routines

Routine	Description
senddiagnostic1d	Issue a **send diagnostic** command and test whether the device or the SCSI bus is online or offline, or run a self-test on the device.
testunitready00	Issue a **test unit ready** command to the SCSI device.
vtostr	Return a pointer to a string describing a table entry value.
write0a	Issue a **group 0 write** command to the SCSI device.
writeextended2a	Issue a **write extended** command to the SCSI device.

Note: Many of the following routines take the parameter *vu*. This parameter is not yet implemented. Consequently, its value must always be zero.

SCSI Open and Close Driver Routines

dsopen – Allocate a dsreq Type Structure and Open a Device

The **dsopen**() routine is used to allocate a *dsreq* type structure for a device on the SCSI bus and to "open" that device.

Synopsis

```
struct dsreq* dsopen(char *opath, int oflags);
```

Arguments

opath Expects the name of the special file for the device on the SCSI bus you want to open. The system comes with up to 15 device special files per adapter in the directory */dev/scsi*. These special files all assume that the logical unit for your device is 0. If the logical unit number for your device is not 0, you must create a device-special file for it. See "Creating Device-special Files for User-level SCSI Drivers."

oflags Expects the *oflag* value you would normally give to the standard **open**() system call when opening a device.

Returns

The returned value of this function is a pointer to a *dsreq* type structure.

Notes

This structure is the medium of communication between your user-level SCSI driver and the device on the SCSI bus, and almost every library routine expects the pointer this function returns as its first argument.

dsclose – Free the dsreq Structure and Close the Device

The **dsclose**() routine is used to free the *dsreq* structure for the SCSI device and close the device, when your driver is done with the device.

Synopsis

```
dsclose(struct dsreq *dsp);
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). Upon successful completion, **dsopen**() returns a pointer to a *dsreq* type structure as its function value.

SCSI Function-Building Routines – Group 1

The next five routines, **doscsireq**(), **filldsreq**(), **fillg0cmd**(), **fillg1cmd**(), **vtostr**() are utility routines used to construct your own SCSI functions. If the provided SCSI routines are sufficient, you will not need these routines.

doscsireq – Send a Command to the SCSI Device

The **doscsireq**() routine is used to send a command to the SCSI device or to make some other request of the SCSI bus. All data structures must have been set up before you can use this routine.

Synopsis

```
doscsireq(int fd, struct dsreq *dsp);
```

Arguments

fd — Expects the file descriptor for the special file opened by **dsopen**(). This file descriptor is stored in the *context* type structure pointed to by the *ds_private* member of the *dsreq* type structure allocated by the call to **dsopen**(). Use the **getfd**(*dsp*) macro to get *fd*.

dsp — A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). You control the behavior of **doscsireq**() by how you set the bits of the value stored in the *ds_flags* member of the *dsreq* type structure pointed to by this parameter. For more information, see the description given in "dsreq – User-level Driver Communication Structure."

filldsreq – set Members of a dsreq Type Structure

The **filldsreq**() routine is used to set the *ds_flags*, *ds_databuf*, and *ds_datalen* members of a *dsreq* type structure.

Synopsis

```
filldsreq(struct dsreq *dsp, uchar_t data,
          long datalen, long flags)
```

Arguments

dsp — A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). The following parameters are then used to set the values of some of the members of this structure.

data — A pointer to the start of a data buffer. The value of this parameter is written to the *ds_databuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.

datalen — The length of the data pointed to by the *data* parameter. The value of this parameter is written to the *ds_datalen* member of the *dsreq* type structure pointed to by the *dsp* parameter.

flags — The value to which you want to set the *ds_flags* member of the *dsreq* type structure pointed to by the *dsp* parameter. See the description of the *ds_flags* member given in "dsreq – User-level Driver Communication Structure."

fillg0cmd – Set Up the dsreq Structure

The **fillg0cmd**() routine is used to set up the *dsreq* structure to send a **group 0 (6-byte) SCSI** command to the SCSI device. To actually send the command, you must call the routine **doscsireq**().

Synopsis

```
fillg0cmd(struct dsreq *dsp, uchar_t *cmdbuf, b0, ..., b5)
```

Arguments

dsp — A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). The following parameters are then used to set the values of some of the members of this structure.

cmdbuf — A pointer to a SCSI command. The value of this parameter (the pointer) is written to the *ds_cmdbuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.

b0, b1, b2, b3, b4, b5 — The values of the individual bytes of the **group 0 SCSI** command to be written to the string pointed to by *cmdbuf*.

fillg1cmd – Send a Group 1 SCSI Command

The **fillg1cmd**() routine is used to set up the *dsreq* structure to send a **group 1 (10-byte) SCSI** command to the SCSI device. To actually send the command, you must call the routine **doscsireq**().

Synopsis

```
fillg1cmd(struct dsreq *dsp, uchar_t *cmdbuf, b0, ..., b9)
```

Arguments

dsp — A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). The following parameters are then used to set the values of some of the members of this structure.

cmdbuf — A pointer to a SCSI command. The value of this parameter (the pointer) is written to the *ds_cmdbuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.

b0, b1, b2, b3, b4, b5, b6, b7, b8, b9
The values of the individual bytes of the ten-byte **group 1 SCSI** command that you want to write to the string pointed to by *cmdbuf*.

vtostr – Return a Pointer to a String Describing Table Entry

The **vtostr**() routine is used to look up a value in a table and return a pointer to a string describing the table entry for that value. It is normally used to print debugging information, such as when the global variable *dsdebug* is set to a nonzero value.

Synopsis

`vtostr(long` *value*`, struct vtab *`*table*`);`

Arguments

value Expects the value you want to look up in the table named by the table parameter.

table A pointer to the name of the table in which you want to look up the value specified as the *value* parameter. You have a choice of four tables are provided with the library:

dsrqnametab – describes the *DSRQ_** flags.

dsrtnametab – describes the *DSRT_** flags.

cmdstatustab – describes the values returned in the *ds_status* member of the *dsreq* type structure.

cmdnametab – describes the values used for the SCSI commands.

Note: The tables are provided in source form in the same directory as *dslib.c* in the *dstab.c* file.

SCSI Function-Building Routines – Group 2

The next ten routines implement the most frequently used SCSI commands, other than **read**. There are so many variations on **read** that a generic version of these commands would be too clumsy, so you will have to write your own. **Read** and **write** extended are more standard, and so are provided.

inquiry12 – Issue an Inquiry Command

The **inquiry12**() routine is used to issue an **inquiry** command to a SCSI device and retrieve information from the device concerning such things as its type. Much of this is device-specific information. See vendor your documentation for more details.

Synopsis

```
inquiry12(struct dsreq *dsp, caddr_t data,
          long datalen, char vu)
```

Arguments

dsp — A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().

data — A pointer to a buffer. Upon successful completion, this command writes the inquiry information to the buffer pointed to by this parameter. Internally, the value of this parameter is written to the *ds_databuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.

datalen — The size of the buffer pointed to by the *data* parameter. Internally, the value of this parameter is written to the *ds_datalen* member of the *dsreq* type structure pointed to by the *dsp* parameter. Typically, the length must be at least 36 bytes, although 64 bytes is a more normal value for *datalen*.

vu — Not implemented.

modeselect15 – Issue a Group 0 “Mode Select” Command

The **modeselect15**() routine is used to issue a **group 0 “mode select”** command to a SCSI device. This is similar in concept to the UNIX **ioctl**() system call. It is used to control a large number of standard and vendor-specific device parameters. Typically, **modesense1A**() is used first to retrieve the current parameters; the page number(s) of interest and the length are passed in the first few bytes of the data.

Synopsis

```
modeselect15(struct dsreq *dsp, caddr_t data,
             long datalen, int save, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to the buffer containing the mode select data.
datalen	The length of the buffer in data.
save	A value that indicates whether you want the device to save the "page" information. The possible values are: 0 = do not save saveable pages. 1 = save saveable pages.
vu	Not implemented.

modesense1a – Send a Group 0 "Mode Sense" Command

The **modesense1a**() routine is used to send a **group 0 "mode sense"** command to a SCSI device to retrieve the page information from the device.

Synopsis

```
modesense1a(struct dsreq *dsp, caddr_t data, long datalen,
            char pgctrl, char pgcode, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes the "page" information to the buffer pointed to by this parameter.
datalen	The size of the buffer pointed to by the *data* parameter.
pgctrl	Expects one of four values that indicate what sort of information you want to retrieve from the page:

0 = current values
1 = changeable values
2 = default values
3 = saved values

pgcode Expects the value that indicates the "page" you want to see. There can be up to 0x3F pages. To return all pages, use 0x3F as the value of this parameter. The information on these pages varies from vendor to vendor. For more information, see the vendor-supplied documentation for the device.

vu Not implemented.

readcapacity25 – Issue a Read Capacity Command

The **readcapacity25**() routine is used to issue a **read capacity** command to a SCSI device.

Synopsis

```
readcapacity25(struct dsreq *dsp, caddr_t data,
               long datalen, long lba, char pmi, char vu)
```

Arguments

dsp A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().

data A pointer to a buffer. Upon successful completion, this command writes the capacity information to the buffer pointed to by this parameter.

datalen The size of the buffer pointed to by the *data* parameter.

lba Unused if *pmi* is 0. Otherwise, it expects the logical block value of the track for which you want capacity information.

pmi The value that tells the routine whether you want the capacity for the entire unit or for the current track (as specified by the logical block named in the *lba* parameter). When you ask for track information, the logical block returned (in the buffer pointed to by *data*) is the most distant logical block you can access without undue delay. The valid values for this parameter are:

	0 = return last logical block in unit. 1 = return last logical block in track.
vu	Not implemented.

readextended28 – Issue a Read Extended Command

The **readextended28**() routine is used to issue a **read extended** command to a SCSI device. This command has enough variations that it is quite possible you will need a custom version of it for your device. Do not preempt the function name.

Synopsis

```
readextended28(struct dsreq *dsp, caddr_t data,
                long datalen, long lba, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes information from the device to the buffer pointed to by this parameter.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
lba	Expects the logical block from which you want to read.
vu	Not implemented.

requestsense03 – Issue a Request Sense Command

The **requestsense03**() routine is used to issue a **request sense** command to a SCSI device and test or "probe" for the device. If you set *DSRQ_SENSE* in the *doscsireq* flag argument, as the included library routines do, you don't need to use this routine.

Synopsis

```
requestsense03(struct dsreq *dsp, caddr_t data,
               long datalen, char vu)
```

Arguments

dsp A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().

data A pointer to a buffer. Upon successful completion, this command writes the “sense” information to the buffer pointed to by this parameter.

datalen A value that specifies the size of the buffer pointed to by the *data* parameter.

vu Not implemented.

senddiagnostic1d – Issue a Send Diagnostic Command

The **senddiagnostic1d**() routine is used to issue a **send diagnostic** command to a SCSI device and test whether the device is functioning correctly. Upon completion of a self-test run by the device, the *ds_status* member of this *dsreq* type structure usually describes the results. (See the description given for this member in “dsreq – User-level Driver Communication Structure.”) If you request that a self-test hold the results, you must issue a **read diagnostic** command to get the results. The *dslib* does not contain a routine to issue a **read diagnostic**, but you can use **fillg0cmd**() to create one.

Synopsis

```
senddiagnostic1d(struct dsreq *dsp, caddr_t data, long dlen,
                 long self, long dofl, long uofl, chap vu)
```

Arguments

dsp A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().

data Not used.

datalen Not used.

self A value that indicates whether you want the device to run a self test that holds the results or a self test that reports the results in *ds_status*. This parameter has two valid values:

0 = run a self test, hold the results
1 = run a self test, report through *ds_status*

dofl A value that indicates whether or not you want to test if the SCSI bus is online or offline:

0 = test if bus is on-line
1 = test if bus is off-line

uofl A value that indicates whether or not you want to test if the device is on-line or off-line:

0 = test if device is on-line
1 = test if device is off-line

vu Not implemented.

testunitready00 – Issue a Test Unit Ready Command to a SCSI Device

The **testunitready00**() routine is used to issue a **test unit ready** command to a SCSI device.

Synopsis

```
testunitready00(struct dsreq *dsp);
```

Arguments

dsp A pointer to the *dsreq* type structure that you allocate for the SCSI device through a call to **dsopen**().

Upon completion of the test, the *ds_status* member of this *dsreq* type structure describes the results. See the description given for this member in "dsreq – User-level Driver Communication Structure" on page 136.

write0a – Issue a Group 0 Write Command

The **write0a**() routine is used to issue a **group 0 write** command to a SCSI device. As with **readextended**(), this routine tends to be device-specific, so it is quite possible you will need to make your own custom version.

Synopsis

```
write0a(struct dsreq *dsp, caddr_t data, long datalen,
         long lba, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to the buffer you want to write to the device.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
lba	Expects the logical block to which you want to write. (For some devices, this may actually be a byte value.)
vu	Not implemented.

writeextended2a – Issue a Write Extended Command

The **writeextended2a**() routine is used to issue a **write extended** command to a SCSI device. As with **readextended**(), this routine tends to be device-specific, so it is quite possible you will need to make your own custom version.

Synopsis

```
writeextended2a(struct dsreq *dsp, caddr_t data,
                 long datalen, long lba, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to the buffer you want to write to the device.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
lba	The logical block to which you want to write.
vu	Not implemented.

Using the dslib Routines

To use the *dslib* library routines, include the header file *dslib.h* in your code and compile the code using the compiler option *-lds*. To open and close the device, your program must use the *dslib* routines **dsopen**() and **dsclose**(), rather than the **open**() and **close**() system calls. These routines set up data structures for the other library routines.

Note: This means that only specially compiled programs can use these devices; most standard programs will not be able to use them.

The reason for this departure from IRIX is that *dslib* is actually a library of routines and macros that provide an interface to the device driver for the SCSI bus. By controlling the SCSI bus, you can control any device on that bus, providing the SCSI-bus device driver contains support for it. Such a SCSI-bus device driver is available on Silicon Graphics systems. However, using the **ioctl**() of the SCSI bus device driver directly can be complex. The routines of *dslib* provide access to **ioctl**() for the SCSI bus in a safe and relatively straightforward way.

For more on the *dslib* routines, see the online man pages **dslib**(3X) and **ds**(7M). The source for *dslib* is included in *4Dgifts.sw.giftsfull* in 5.0 in the directory */usr/people/4Dgifts/examples/devices/devscsi*.

Opening a SCSI Device

To open a device on the SCSI bus, a user-level program calls **dsopen**(), which calls the **open**() of the device driver for the SCSI bus and allocates data structures. If the open succeeds, the kernel allocates a SCSI subchannel for the device on the bus that you want to control. If this succeeds, a *dsreq* type structure is allocated, which in turn sets the values of some of its members and returns a pointer to this structure as its function value. This allocated and primed *dsreq* type structure is the medium of communication between your user program and the SCSI device.

Sending Commands to a SCSI Device

To send commands to the device on the SCSI bus, your program can modify the members of the *dsreq* type structure and call the *dslib* routine **doscsireq**(). This command uses an **ioctl**() call to the SCSI bus interface driver to set the members of the *scsisubchan* type structure for the device according to the information in the *dsreq* structure.

To simplify sending commands to a SCSI device, *dslib* provides routines and macros that set members of the *dsreq* type structure and call **doscsireq**(). For example, *dslib* provides the routine **write0a**() to give you an easy way to issue a simple group 0 "**write**" command. However, since few truly general SCSI commands exist, the *dslib* provides functions such as **fillg0cmd**() to give you a relatively painless way to send vendor-specific commands to a device on the SCSI bus. You can (and are expected) to extend these as needed, since the library source is included in the **4Dgifts** subsystem.

Closing a SCSI Device

When your user-level program is done with the SCSI device, it must call **dsclose**() to close the device. This involves freeing the *dsreq* type structure and kernel data structures, among other things.

Kernel-level SCSI Device Drivers

This section shows how to write a kernel-level driver for a SCSI device. Starting with IRIX 5.x, all supported SCSI controllers use the same interface.

Configuring a Kernel-level SCSI Driver

Chapter 2, "Writing a Device Driver," gives a detailed description of how to configure a kernel to include a new driver. This section presents only the material that is unique to SCSI drivers. Recall that you must:

1. Create the object code for the driver you want to include in the kernel.
2. Move that object code to the directory */usr/sysgen/boot.*
3. Edit the *system* file. Use a directive telling **lboot**, the configuration utility, how to include your driver and specify which memory space your device will allocate. Install it in the */var/sysgen/system* directory using the driver name appended by *.sm*.
4. Create a *master* file in the directory */usr/sysgen/master.d.*

5. Create a new kernel. (To create a debuggable kernel, see “Making a Debuggable Kernel” in Chapter 10, “Driver Installation and Testing.”)

To edit the *system* file (*/usr/sysgen/system*) to include a SCSI driver, use the *INCLUDE* directive to tell **lboot** to unconditionally add the named SCSI module into the new kernel.

To create a *master* file, create an ASCII file, enter the appropriate information (described below), and move the file to the */usr/sysgen/master.d* directory. The name of the master file must correspond to the name of the file containing the object code for the driver.

Ensure that the *FLAG* field of the master file includes at least the character device flag *c* and the software driver flag *s*. You must flag all SCSI device drivers as software drivers (drivers that do not control actual hardware) because **lboot** cannot probe for SCSI devices.[1] If **lboot** tries to probe for a SCSI device, it fails, then assumes that the device is not present, and does not include your SCSI device driver in the kernel.

For example, assume that you want your kernel to include a device driver for a SCSI device that you call *sdk*. Create the object code for the device driver, and move the *sdk.o* object file to the directory */usr/sysgen/boot*. After examining */usr/include/sys/major.h*, you determine that major device number 61 is available and can be used for the device, *sdk*. Create a file *sdk.sm* with the following line, and also add it to the *system* file:

```
INCLUDE: sdk
```

You then create a master ASCII file called */usr/sysgen/master.d/sdk* and enter:

```
*
* sdk
*
*FLAG   PREFIX   SOFT   #DEV   DEPENDENCIES
 sc     sdk_     61       -     scsi
```

[1] Although **lboot** does probe for the SCSI controller, the target devices that the SCSI controller manages cannot be probed by a memory reference/access.

Under “DEPENDENCIES,” you must list “scsi.” This indicates that the SCSI interface driver must be present. The SCSI interface is described later in this chapter.

Writing a SCSI init()

Because you use the *INCLUDE* directive to include a module into the kernel, your driver object module must contain a routine of the form *drv***init**(), where *drv* is the prefix for the device that you specified in the *master* file. If the driver module includes *drv***init**(), the system calls *drv***init**() at system boot time. Your driver can use *drv***init**() for boot-time device or driver initialization. To initialize the device *sdk* every time the system boots, you can include an **sdkinit**() routine in the driver object module *sdk.o*.

The *drv***init**() routine has no parameters:

```
sdk_init( )
{
your code
};
```

SCSI Device Interface Overview

The SCSI host adapter communicates with devices, known as *targets*, on the SCSI bus. Each SCSI target can control a number of actual devices, known as *logical units*. Most SCSI targets, however, control a single device.

There are two types of SCSI drivers: device level drivers and host adapter drivers. The host adapter drivers handle the low-level communication over the SCSI interface, such as programming the SCSI interface chip or board, negotiating synchronous or wide mode, and handling disconnect/reconnect. The device drivers handle high-level communication, primarily by issuing SCSI commands and interpreting sense data.

Some examples of host adapter drivers are *wd93*, *wd95*, and *jag*. Examples of device level drivers are *dksc*, *tpsc*, and *smfd*.

There are four basic interfaces used to communicate to a host adapter driver: **scsi_info**, **scsi_alloc**, **scsi_free**, and **scsi_command**(). Each of these interfaces is implemented as an array of pointers to functions, indexed by a host adapter driver number. The host adapter driver number is determined from the adapter number. The adapter number ranges are defined in *sys/scsi.h*, and are different on different architectures. In general, Integral SCSI controllers start at adapter 0, and Interphase VME-SCSI (Jaguar) controllers start after the last Integral adapter. Each SCSI bus is considered to be one adapter.

On a POWER Series or Crimson system, for example, Interphase VME-SCSI controller 3, bus 0, would be adapter number 10. On a CHALLENGE/Onyx system, VME-SCSI controller 3, bus 0, would be adapter number 134. See the definitions of *SCSI_SGISTART*, *SCSI_SGICOUNT*, *SCSI_JAGSTART*, and *SCSI_JAGCOUNT* in *sys/scsi.h*.

Determining Driver Number

To determine the correct number for your driver, see *scsi_driver_table* in *sys/scsi.h*. *scsi_driver_table* is an array, indexed by adapter number, that returns the adapter type, which is a constant defined by the *SCSIDRIVER_XXX* definitions. *scsi_driver_table* may be sparsely populated to accommodate possible options. This avoids the reassignment of major and minor device numbers in cases where hardware is added at a later time.

SCSI_XXXSTART definitions define the starting adapter number for the various adapter types. *SCSI_XXXCOUNT* is the number of possible adapters for a given controller type. Both *SCSIDRIVER_WD93* and *SCSIDRIVER_WD95* are part of Silicon Graphics-built SCSI controllers, and so use the *SCSI_SGISTART* and *SCSI_SGICOUNT* *#define*'s.

For example, on a CHALLENGE system, the second bus on Interphase VME-SCSI controller 4 would be considered adapter number 137. Counting *SCSI_JAGSTART* as 128, (controller 4 * 2 adapters per controller) + bus 1 (the second bus on the controller) gives 9 to be added to the value of *SCSI_JAGSTART* to give adapter (or bus) number 137 from the point of view of a SCSI device driver.

Typically, the major and minor numbers of a device are used to determine the adapter number, which can then be used to index into *scsi_driver_table.*

```
adap = sdk_adap_num(device);

driver_num = scsi_driver_table[adap];
```

scsi_info – Getting Information About a Device

Before your driver tries to access a device, it must call the host adapter **scsi_info** function. The host adapter driver then issues an **Inquiry** command to the given device and returns a pointer to a *struct scsi_target_info*. If the **Inquiry** is not successful — or if the *adapter*, *target*, or *lun* is invalid — the return value is *NULL*. The SCSI device driver must examine the inquiry data to determine whether it is the appropriate driver for the device.

Synopsis

```
struct scsi_target_info * (*scsi_info[])
                        (u_char adapter, u_char target, u_char lun);
```

Arguments

adapter	The adapter number.
target	The target number.
lun	The logical unit number.

Example

The following would be the way to use *scsi_info* to get information about target 5, LUN 0 on Interphase controller 4, bus 1 (the second bus):

```
info = (*scsi_info[SCSIDRIVER_JAG])(9, 5, 0);
```

scsi_alloc – Initializing a Connection

Before your driver can issue a command to a SCSI device through **scsi_command**, it must have the low-level driver initialize the connection. The interface to this low-level driver is through the **scsi_alloc**(), typically in the *drv***open**() routine.

Synopsis

```
int (*scsi_alloc[driver_num])
     (unsigned char adap, unsigned char target,
      unsigned char lun, int option, void (*callback)());
```

Arguments

adap The number of the SCSI adapter for the device you want to control. All IRIX systems support at least one adapter (0). Systems with POWERchannel have two or four built-in adapters and up to 16 VME-SCSI adapters (two per controller). CHALLENGE/Onyx systems can have up to 120 different built-in SCSI adapters (although one system can have only a fraction of those installed at one time) and 16 VME-SCSI adapters; Indigo^2 systems have two adapters.

target The target ID of the SCSI controller for the device your driver wants to control. This must be a value from 0 to 15, but cannot be the ID of the controller itself (typically 0 for built-in and 7 for the Jaguar). Not all host adapter implementations support SCSI device numbers 8 through 15. (Note that for this purpose, the Jaguar VME_SCSI controller has two SCSI host adapters, so only devices 0-7 are valid. Choose the bus number with *adap*.) Usually, the device is configured to a fixed ID by switch settings or jumpers. Refer to the device technical specification for details. If the system contains more than one of a particular device, the device driver normally uses the minor device number to distinguish between devices.

lun The logical unit number for the device your driver wants to control. This is often 0 because most SCSI targets support only a single logical unit.

option Two options are available to **scsi_alloc**:

SCSIALLOC_EXCLUSIVE specifies that the device driver wants to have exclusive access to the target in question. If any other device has allocated a connection, the **scsi_alloc** call fails.

SCSIALLOC_QDEPTH, the queue depth that your driver requests, is the bottom eight bits of *option*. It is considered advice only, and may or may not be followed.

callback Gets a pointer to a function that the host adapter driver calls every time it has sense data. Only one driver at a time may have a callback allocated, so, in this respect, it functions like *SCSIALLOC_EXCLUSIVE* in the option argument above. If the callback argument is not *NULL*, the host adapter driver calls it with a pointer to the sense data (an array of char) every time there is any. This allows multiple drivers to access a device while still allowing one of them to keep apprised of all sense data.

Returns

If the call to **scsi_alloc** is successful, the type of SCSI adapter being used (*SCSIDRIVER_WD93*, *SCSIDRIVER_WD95*, or *SCSIDRIVER_JAG*) is returned. Otherwise, this routine returns 0, which indicates that no connection could be established between the host adapter driver and your driver, probably because the arguments were included for the adapter.

scsi_command – Executing a SCSI Command

Once your driver has established a connection to a host adapter driver with a successful call to **scsi_alloc**[](), your driver can send SCSI commands to the device. To send SCSI commands to a device, fill out a *scsi_request* structure and then call **scsi_command**[](), passing it the address of this structure. The system uses this structure to report back on the status of the SCSI command.

Not all *scsi_request* type structure members are of interest to your device driver (some members are of interest only to the SCSI host adapter driver and may change across releases of the operating system). Following is the definition of the structure *scsi_request*:

```
struct scsi_request
{
        /* values filled in by device driver */
        u_char   sr_ctlr;
        u_char   sr_target;
        u_char   sr_lun;
        u_char   sr_tag;     /* first byte of tag message */

        u_char  *sr_command; /* scsi command */
        ushort   sr_cmdlen;  /* length of scsi command */
        ushort   sr_flags;   /* direction of data transfer */
        ulong    sr_timeout; /* in seconds */

        u_char  *sr_buffer;  /* location of data */
        uint     sr_buflen;  /* amount of data to transfer */
```

```
        u_char  *sr_sense;   /* where to put sense data in
                             /*case of CC */
        ulong    sr_senselen;/* size of buffer allocated for
                             /*sense data */
        void    (*sr_notify)(struct scsi_request *);
                             /* callback pointer */
        void    *sr_bp;      /* usually a buf_t pointer */

        /* spare pointer used by device driver */
        void    *sr_dev;

        /* spare fields used by host adapter driver */
        void    *sr_ha;      /* usually used for linked list
                             /*of req's */
        void    *sr_spare;   /* used as spare pointer, int,
                             /*etc. */

        /* results filled in by host adapter driver */
        uint     sr_status;      /* Status of command */
        u_char   sr_scsi_status; /* SCSI status byte */
        u_char   sr_ha_flags;    /* flags used by host
                                 /*adapter driver */
        short    sr_sensegotten; /* number of sense bytes
                                 /* received; -1 == err */
        uint     sr_resid;       /* amount of sr_buflen not
                                 /*transferred */
};
typedef struct scsi_request      scsi_request_t;
```

Before calling **scsi_command**[](), your driver must fill in the values indicated above. Some are optional.

sr_ctlr Number of the adapter where the command will be transferred.

sr_target ID of the target where the command will be transferred.

sr_lun Number of the logical unit where the command will be transferred.

sr_tag Type of queue tag to use (see SCSI-2 specification). Not all tag types are supported by all drivers.

sr_command Give this member a pointer to the SCSI command descriptor block that you want to send to the device.

sr_cmdlen The length (in bytes) of the command to which the *sr_command* member points. You can use three symbolic constants for this member (the constants are defined in *sys/scsi.h*), but other values are permitted for vendor unique commands, too:

SC_CLASS0_SZ is a command size of 6 bytes.

SC_CLASS1_SZ is a command size of 10 bytes.

SC_CLASS2_SZ is a command size of 12 bytes.

sr_flags This member must set zero or more options as specified below:

SRF_DIR_IN — Data associated with the command will be written into memory.

SRF_FLUSH — A cache operation is required on the data.

SRF_MAP — The *sr_buffer* address needs to be mapped -- it is a kernel virtual address.

SRF_MAPBP — *sr_bp* has a pointer to a *struct buf* that is not mapped (*BP_ISMAPPED* returns false).

SRF_AEN_ACK — This request acknowledges an error in a previous request. Once a **scsi_request** is returned with an error, all further requests are returned with *SC_ATTN*, without being acted upon, until *SRF_AEN_ACK* is set.

SRF_NEG_ASYNC — This request attempts to negotiate *async xfer* mode on this command (if currently using *sync xfers*).

SRF_NEG_SYNC — This request attempts to negotiate *sync xfer* mode on this command if currently asynchronous. It may be ignored by some adapter drivers, either always or if the driver has previously failed to negotiate *sync xfer* mode with this target. This overrides the *SCSIALLOC_NOSYNC* flag to **scsi_alloc** (if it has been specified).

sr_timeout The maximum amount of time, in Hz, that a command can take.

sr_buffer A pointer to the start of the data associated with a command. If there is no data (as with a *test_unit_ready* for instance), *sr_buffer* must be NULL.

sr_buflen The maximum amount of data that can be transferred. This is not necessarily the amount that *will* be transferred, but is an upper limit.

sr_sense A pointer to a buffer where request sense data can be copied in case a command gets a check condition status.

sr_senselen The amount of space reserved for request sense data.

sr_notify A pointer to a notification routine. The notification routine gets called when a command is complete or gets an error. This value must be set (*NULL* is not a valid value).

sr_bp This field must point to the *struct buf* corresponding that generated the *scsi_request* if the *SRF_MAPBP* flag is set in *sr_flags*. If the *SRF_MAPBP* flag is not set, then *sr_bp* is not used by the host adapter driver.

sr_dev This field is never used by a host adapter driver and is reserved for the use of your device driver.

After control returns from the host adapter driver to your driver, your driver must check the following members of the *scsi_request* type structure. These members are:

sr_status This member reports the overall status of the SCSI command (this is the host adapter driver status; SCSI bus status is in *sr_scsi_status*). *sr_status* can report one of these values:

SC_GOOD — indicates no error. The bus successfully processed the SCSI command; however, the command itself may still have failed (see *sr_scsi_status*).

SC_TIMEOUT — indicates selection timeout. The device did not respond to selection within 250 milliseconds.

SC_HARDERR — indicates hardware or SCSI device error, usually a SCSI bus reset, possibly caused by a bad phase or time-out on some other device.

SC_PARITY — indicates an unrecoverable parity error on the SCSI bus.

SC_MEMERR — indicates an unrecoverable parity or ECC error from host memory.

SC_CMDTIME — indicates the command did not complete before the time-out specified in *s_timeoutval* elapsed.

SC_ALIGN — indicates the buffer address did not meet the alignment requirements of the system. Most Silicon Graphics systems require starting buffer addresses to be on a four-byte boundary.

SC_ATTN — indicates the **scsi_request**() has been aborted by a SCSI bus reset, the device, or the driver, due to an error in another request.

SC_REQUEST — indicates the request is not a valid request. This can be because no **scsi_alloc** has been done successfully or because the **scsi_request**() has conflicting or illegal values, such as *sr_notify* being set to *NULL*.

sr_scsi_status The SCSI status byte sent by the target. *sr_scsi_status* can report any one of these values: The values of *sr_sci_status* correspond to those described in the SCSI specification:

ST_GOOD — indicates the target has successfully completed the SCSI command. On this value, *sr_sensegotten* must be checked to see if a check condition occurred on the command.

ST_CHECK — indicates that the request sense command generated by the host adapter driver in response to a check condition also got a check condition. This is a special case. If a device reports a "check condition" status, the host adapter driver automatically issues a *request sense* command and reports the status of that command in this byte. If *sr_sense* is set and *sr_senselen* is > 0, *sr_sensegotten* is set to the number of bytes of sense data received, or to -1 if an error occurs during the request sense command.

ST_COND_MET — indicates search condition satisfied.

ST_BUSY — indicates the target is busy. Normally, the driver should delay, then reissue the command.

ST_INT_GOOD — indicates this status is reported for every command in a series of linked commands. Although they are not supported, these linked commands may sometimes work.

ST_RES_CONF — indicates an attempt to access a logical unit or an extent within a logical unit that reserves that type of access to another SCSI device.

This byte corresponds to the SCSI command status byte as documented in the SCSI specifications.

sr_sensegotten The number of bytes of sense data gotten as a result of a request sense command issued in response to a check condition status from this command, if any, or -1 if a check condition occurs, but the request sense command fails.

sr_resid The difference between *sr_buflen* and the number of bytes actually transferred.

scsi_free – Freeing the Connection

When your driver is done with a SCSI device, it must call through the array **scsi_free** to indicate to the host adapter driver that your driver no longer needs to use the device. Typically, you do this in your *drv***close**() routine.

Synopsis

```
int (*scsi_free[driver_num])
    (unsigned char adap, unsigned char target,
     unsigned char lun, void (*callback)());
```

The arguments to **scsi_free** are similar to those to **scsi_alloc** except that no option argument is used.

Using the SCSI Access Routines

"SCSI Device Interface Overview" on page 159 describes the four routines that make up the device driver interface to host adapter SCSI-bus drivers. This section describes how you can use these routines in your driver.

Your *drv***open**() routine should first use **scsi_info** to determine whether a device is present on the SCSI bus and to examine the inquiry information to see what type of device it is. Remember, SCSI target 3 may be a disk on one system but a tape drive on another. So be sure to check the inquiry information to determine whether your driver is appropriate to the device in question.

After your driver has determined that it is appropriate to talk to a device, it must then use **scsi_alloc** to initialize a connection between your device driver and the host adapter driver.

Your driver can free the connection by using **scsi_free**. Remember that if your driver uses the exclusive option, no other SCSI device driver can communicate with a target until your driver calls **scsi_free**. Similarly, if your driver uses a sense callback, no other driver can use a sense callback until your driver calls **scsi_free**, although other drivers that do not need a sense callback may still communicate with the target.

Because the IRIX SCSI interface transfers data using direct memory access (DMA) only, your driver must use the kernel routine **physio**() and a *drv***strategy**() routine for SCSI I/O to transfer data to pages that a user process is using. Internally generated requests that transfer data into kernel memory (either statically allocated at compile time in the driver or allocated through **kmem_alloc**() or similar functions) do not need to use **physio**(). However, in this case your driver must do its own cache flushing.

When filling out a *scsi_request* structure in preparation for a call to **scsi_command**, be sure to fill out the *sr_ctlr*, *sr_target*, *sr_lun*, and *sr_tag* fields with appropriate values. The *sr_command* field must point to an array of characters filled out with the SCSI command byte values, and *sr_cmdlen* must be the length of the command. *sr_flags* is used to specify the direction of data transfer, if any, whether cache flushes need to be done, and what kind of mapping needs to be done. If the data transfer involves direct mapped K0SEG or K1SEG memory, then no mapping need be specified, although the host adapter may still perform some mapping. The flags must also acknowledge an error by setting the *SRF_AEN_ACK* bit on the next command or the command will fail immediately.

sr_timeout must be the command timeout in clock ticks (defined by Hz), typically 1/100 of a second.[1] *sr_buffer* must be a pointer to the start of the buffer to be transferred to, unless there is no data transfer or the

SRF_MAPBP flag in *sr_flags* is set, in which case *sr_buffer* can be *NULL*. *sr_buflen* must be an amount equal to the maximum amount of data to be transferred in the command.

If your driver wants to examine sense data from a request sense caused by a check condition that occurs as a result of the command you issued, it can set the *sr_sense* field to point to such a buffer, with *sr_senselen* set to the size of the buffer. The *sr_notify* must point to your driver "interrupt" routine. This is not a real interrupt routine—it is called indirectly as a result of an interrupt from the interrupt handler—so it can have any name you desire. This routine is called when the host adapter driver has completed processing your command.

If the *SRF_MAPBP* flag is set, *sr_bp* must point to the buf structure that generated the SCSI request. Sometimes your driver strategy function will get a buf that is not mapped into kernel virtual memory. In this case, **BP_ISMAPPED**(*bp*), where *bp* is a pointer to the buf, will return false. If *SRF_MAPBP* is not used, *sr_bp* is free to be used by your driver for other purposes. *sr_dev* is always a spare field usable by your driver, often for linking lists of active or queued requests or to keep special information.

After your **notify** function is called, your driver must check the *sr_status*, *sr_scsi_status*, and *sr_sensegotten* fields to see whether there were any errors. See */usr/include/sys/scsi.h* for additional information on return values for the *sr_status* field. The *sr_scsi_status* field corresponds to the SCSI status byte that a target sends at the end of every command. However, if a target sends a check condition (status 2), a request sense is issued instead; otherwise *sr_scsi_status* is 0. *sr_sensegotten* is nonzero: -1 if there is an error getting sense data or the amount of sense data actually received. Otherwise, *sr_resid* will be the difference between *sr_buflen* and the amount of data actually transferred.

[1] Check the header file and include the file that defines Hz. Do not hard-code the value of Hz into a driver.

SCSI Device Driver Example

The following example shows how a driver can communicate with a direct access SCSI device, such as a disk. Note the use of **scsi_info**[]() to determine that the device is actually present and of the appropriate type.

This driver is simplified and does not do as much error checking as a real driver would do. Also, this example uses a global SCSI request structure that does not work in real drivers, since multiple reads or writes would overwrite a command in progress.

```
#include "sys/param.h"
#include "sys/types.h"
#include "sys/user.h"
#include "sys/buf.h"
#include "sys/errno.h"
#include "sys/cmn_err.h"
#include "sys/cred.h"
#include "sys/ddi.h"
#include "sys/systm.h"
#include "sys/scsi.h"

int sdk_devflag = 0;

#define ADAPT    0     /* SCSI host adapter */
#define TARGET   7     /* the disk will have target ID #7 */
#define LU       0     /* and logical unit  #0 */
#define TIMEOUT (30*HZ)/* wait 30 secs for SCSI device to
                          respond */
#define DIRECTACCESS 0 /* First byte of inqry cmnd */

unchar scsi_read[]    = {0x28, 0, 0, 0, 0, 0, 0, 0, 0, 0};
unchar scsi_write[]   = {0x2a, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int    sdk_inuse = 0;
int    sdk_driver;
struct scsi_target_info *sdk_info;
struct scsi_request sdk_req;
u_char sdk_sensebuf[SCSI_SENSE_LEN];  /* SCSI_SENSE_LEN
                                         from scsi.h */

/* forward definitions*/
int sdk_strategy(struct buf *bp);
void sdk_notify(struct scsi_request *req);
```

```
/*
 * sdk_open - Open the SCSI device exclusively.
 *
 * Issue a SCSI inquiry command upon device and ensure
 * it is a direct access device.
 */
int
sdk_open(dev_t *devp, int flag, int otyp, cred_t *crp)
{
   if (sdk_inuse)
      return EBUSY;

   /* Get driver number */
   sdk_driver = scsi_driver_table[ADAPT];

   /*
    * Call through scsi_info to get inquiry data and to
    * find out if a device is at the address we want.
    */
   sdk_info = (*scsi_info[sdk_driver])(ADAPT, TARGET, LU);
   if (sdk_info == NULL)
      return ENODEV;

   /*
    * Is it a direct access device?  We could check the
    * entire inquiry buffer to ensure it is actually the
    * correct device.
    */
   if (sdk_info->si_inq[0] != DIRECTACCESS)
      return ENXIO;

   /*
    * It's a direct access device (disk drive).  Initialize
    * the connection to the host adapter driver.
    */
   if ((*scsi_alloc[sdk_driver])
      (ADAPT, TARGET, LU, 1, NULL) == 0)
      return EBUSY;

   /*
    * We have successfully allocated a connection between
    * sdk and the host adapter driver.  Initialize the
    * scsi_request structure, and mark the driver as being
    * in use.
```

```
     */
    sdk_inuse = 1;
    bzero(&sdk_req, sizeof(sdk_req));
    sdk_req.sr_ctlr = ADAPT;
    sdk_req.sr_target = TARGET;
    sdk_req.sr_lun = LU;
    sdk_req.sr_timeout = TIMEOUT;
    sdk_req.sr_sense = sdk_sensebuf;
    sdk_req.sr_senselen = sizeof(sdk_sensebuf);
    sdk_req.sr_notify = sdk_notify;

    return 0;
}

/* sdk_close - close the device and free the subchannel. */
int
sdk_close(dev_t dev, int flag, int otyp, cred_t *crp)
{
    (*scsi_free[sdk_driver])(ADAPT, TARGET, LU, NULL);
    sdk_inuse = 0;
    return 0;
}

/*
 * sdk_read - read from the SCSI device, ensuring an even
 * block count and a word-aligned address.
 */
sdk_read(dev_t dev, uio_t *uiop, cred_t *crp)

/*
 * sdk_write - write to the SCSI device, ensuring an even
 * block count and a word-aligned address.
 */
sdk_write(dev_t dev, uio_t *uiop, cred_t *crp)

/*
 * sdk_strategy - do the dirty work of the I/O.
 * Use either the SCSI read or write command as
 * appropriate.  Modify the block number and block counts
 * within the command buffer. Simply return here;
 * physio( ) will wait for an iodone( ).
 */
int
sdk_strategy(struct buf *bp)
{
```

```
    int blkno, blkcount;

    /* Prime the subchannel communication block. */

    blkno = bp->b_blkno;
    blkcount = BTOBB(bp->b_bcount);

    sdk_req.sr_command = bp->b_flags & B_READ ?
                         scsi_read : scsi_write;
    sdk_req.sr_command[2] = (char)(blkno>>24);
    sdk_req.sr_command[3] = (char)(blkno>>16);
    sdk_req.sr_command[4] = (char)(blkno>>8);
    sdk_req.sr_command[5] = (char) blkno;
    sdk_req.sr_command[7] = (char)(blkcount>>8);
    sdk_req.sr_command[8] = (char) blkcount;

    sdk_req.sr_cmdlen = SC_CLASS1_SZ;
    sdk_req.sr_flags = bp->b_flags & B_READ ? SRF_DIR_IN : 0;

    if (BP_ISMAPPED(bp)) {
        sdk_req.sr_buffer = bp->b_dmaaddr;
        sdk_req.sr_buflen = bp->b_bcount;
        sdk_req.sr_flags |= SRF_MAP;
    }
    else {
        sdk_req.sr_buffer = NULL;
        sdk_req.sr_buflen = bp->b_bcount;
        sdk_req.sr_flags = SRF_MAPBP;
    }
    sdk_req.sr_bp = bp;   /* required for SRF_MAPBP, but a
                           * convenience in all cases */

    /* Perform the SCSI operation. */
    (*scsi_command[sdk_driver])(&sdk_req);
}

/*
 * sdk_notify - SCSI command completion notification routine
 *
 * Simply check for errors and wake up physio( ) with
 * an iodone( ) on the buffer.
 * Note that a more robust driver would be more thorough
 * about error handling by retrying errors, giving more
 * information about error types, etc.
 */
```

```
void
sdk_notify(struct scsi_request *req)
{
   register struct buf *bp = req->sr_bp;

   if ((req->sr_status != SC_GOOD) ||
        (req->sr_scsi_status != ST_GOOD) ||
        (req->sr_sensegotten < 0))
   {
      cmn_err(CE_NOTE,
         "sdk: Error: driver stat 0x%x, scsi stat 0x%x"
         " sensegotten %d\n", req->sr_status,
         req->sr_scsi_status, req->sr_sensegotten);
      bioerror(bp, EIO);
         }
   else if (req->sr_sensegotten > 0) {
      cmn_err(CE_NOTE, "sdk: Error: sensekey 0x%x\n",
         sdk_sensebuf[2] & 0x0F);
      bioerror(bp, EIO);
   }
   bp->b_resid = req->sr_resid;
   biodone(bp);
}
```