Chapter 15

SCSI Device Drivers

All Silicon Graphics systems support the Small Computer Systems Interface (SCSI) bus for the attachment of disks, tapes, and other devices. This chapter details the kernel-level support for SCSI device drivers.

If your aim is to control a SCSI device from a user-level process, this chapter contains some useful background information to supplement Chapter 5, “User-Level Access to SCSI Devices.” If you are designing a kernel-level SCSI driver, this chapter contains essential information on kernel support. The major topics in this chapter are as follows:

- “SCSI Support in Silicon Graphics Systems” on page 352 gives an overview of the hardware and software support for SCSI.
- “Host Adapter Facilities” on page 354 documents the use of the host adapter driver to access a SCSI device.
- “Designing a SCSI Driver” on page 369 notes design differences from other driver types, and includes an example driver skeleton.
- “Example SCSI Device Driver” on page 370 lists a skeleton driver to illustrate the use of the interface.
- “Designing a Host Adapter Driver” on page 375 documents the facility for creating and installing customized host adapter drivers.
- “SCSI Reference Data” on page 377 tabulates SCSI codes and messages for reference.

In addition, you may want to review the following additional sources:

intro(7) reference page	Documents the naming conventions for disk and tape device special files.
dksc(7) reference page	Documents the Silicon Graphics disk volume partition layout and the ioctl support in the base-level SCSI drivers.
ANSI *X3.131-1986* and *X3T9.2/85-52 Rev 4B*.	SCSI standards documents.
http://www.abekrd.co.uk/SCSI2/	Web page containing the complete SCSI-2 standard in HTML form.

SCSI Support in Silicon Graphics Systems

All current Silicon Graphics systems rely on the SCSI bus as the primary attachment for disks and tapes. The IRIX kernel provides extensive support for OEM drivers for SCSI devices.

Note: As used here, the term "adapter" means a SCSI controller such as the Western Digital W93 chip, which attaches a unique chain of SCSI devices. In this sense, a SCSI adapter and a SCSI bus are the same. "Adapter number" is used instead of "bus number."

SCSI Hardware Support

The Silicon Graphics computer systems supported by IRIX 6.2 can attach multiple SCSI adapters, as follows:

- The Indy workstation has at least one SCSI adapter on its motherboard, and can have up to two additional adapters on a GIO option board.
- The Indigo^2 series supports two SCSI adapters on the motherboard.
- The Challenge S system has two SCSI adapters on the motherboard, and can have one or two additional on each of one or two additional GIO option boards, for a maximum of sixadapters.

- The Challenge M system supports one SCSI adapter on the CPU board and can have up to two additional adapters on a GIO option board.
- The POWERchannel (IO3) boards used in the Crimson line support two SCSI adapters per board.
- The Power Channel-2™ (IO4) boards used in the Challenge and Onyx series support two SCSI adapters, plus many as six additional SCSI adapters on mezzanine cards, for a maximum of eight adapters per IO4. In addition, VME-SCSI adapters (*Jag* units) can be installed on the VME bus in these systems.

In all systems, DMA mapping hardware allows a SCSI adapter to treat discontiguous memory locations as if they were a contiguous buffer, providing scatter/gather support.

IRIX Kernel SCSI Support

The IRIX kernel contains two levels of SCSI support. An inner SCSI driver, the *host adapter driver*, manages all communication with SCSI hardware adapters. The kernel-level SCSI device drivers for particular devices prepare SCSI commands and call on the host adapter driver to execute them. This design centralizes the management of SCSI adapters. Centralization is necessary because the use of the SCSI bus is multiplexed across many devices, while recovery and error-handling need central handling. In addition, use of the host adapter driver makes it simpler to write a SCSI device driver.

Host Adapter Drivers

Different host adapter drivers are loaded, depending on the hardware in the system. Some examples of host adapter drivers are *wd93*, *wd95*, and *jag*.

The host adapter drivers support all levels of the SCSI standard: SCSI-1, the Common Command Set (CCS, superceded by SCSI-2), and SCSI-2. Not all optional features of the standard are supported. Different systems support different feature combinations (such as synchronous, fast, and wide SCSI), depending on the available hardware.

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.

A host adapter driver is not, strictly speaking, a proper device driver because it does not support all the entry points documented in Chapter 8, “Structure of a Kernel-Level Driver.” You can think of it as a specialized library module for SCSI-bus management or as a device driver, whichever you prefer. The software interface to the host adapter driver is documented under “Host Adapter Facilities” on page 354.

Caution: Connect/disconnect strategy is enabled on any SCSI bus by default (the option is controlled by a constant defined in the host adapter driver descriptive file in */var/sysgen/master.d*). When disconnect is enabled on a bus, and a device on that bus refuses to disconnect, it can cause timeouts on other devices.

SCSI Device Drivers

SCSI device drivers handle high-level device management, primarily by setting up SCSI commands for the host adapter driver to execute, and by interpreting returned sense data. Examples of device drivers are *dksc*, *tpsc*, and *smfd*.

Host Adapter Facilities

The principal difference between a SCSI driver and other kernel-level drivers is that, while other kinds of drivers are expected to control devices directly using PIO and DMA, a SCSI driver operates its devices indirectly, by making function calls to the host adapter driver. This section documents the functional interface to the host adapter driver.

Purpose of the Host Adapter Driver

The reason that IRIX uses host adapter drivers is that the SCSI bus is shared among multiple devices of different types, each type controlled by a different driver. A disk, a tape, a CDROM, and a scanner could all be cabled from the same SCSI adapter. Each device has a different driver, but each driver needs to use the adapter, a single chip-set, to communicate with its device.

If IRIX allowed multiple drivers to operate the host adapter, there would be confusion and errors from the conflicting uses. IRIX puts the management of each host adapter under the control of a host adapter driver, whose job is to issue commands on its bus and report the results. The host adapter is tailored to the hardware of the particular host adapter and to the architecture of the host system.

The interface to the host adapter driver is the same no matter what type of hardware the adapter uses. This insulates the individual device drivers from details of the adapter hardware.

The driver for each type of device is responsible for preparing the SCSI command bytes for its device, for passing the command requests to the correct host adapter driver, and for interpreting sense and status data as it comes back.

Host Adapter Concepts

IRIX 6.2 permits a total of 10 unique host adapter drivers—five supplied by Silicon Graphics and up to five from other vendors. Each host adapter is customized to manage one type of adapter hardware. Each adapter driver has an adapter type number that is declared in *sys/scsi.h*. The constant names, are listed in Table 15-1.

Table 15-1 Host Adapter Driver Classes

Driver Constant	Driver Description
SCSIDRIVER_NULL	No driver exists; invalid adapter number or nonexistent adapter.
SCSIDRIVER_WD93	The *wd93* driver, for adapters based on the Western Digital WD93 chip set.
SCSIDRIVER_JAG	The *jag* driver, for adapters based on the VME-SCSI bridge used in the Challenge and Onyx systems.
SCSIDRIVER_WD95	The *wd95* driver, for adapters based on the Western Digital WD95 chip set.
SCSIDRIVER_SCIP	The *scip* driver, for adapters based on the augmented WD95 chip set used in Challenge and Onyx systems.
SCSIDRIVER_QL	The *ql* driver, for adapters based on the QLogic chip set.
SCSIDRIVER_3RD_PARTY_START	First number available for OEM host adapter drivers.
SCSIDRIVER_3RD_PARTY_END	Last number available for OEM host adapter drivers.

Caution: The constant names listed in Table 15-1 compile to different values in different hardware systems. For this reason, you should avoid using these names in your driver; if you use one, your driver object file has to be recompiled for each CPU type.

The *lboot* command loads a host adapter driver for each unique type of adapter in the system. *lboot* is directed by VECTOR statements in the */var/sysgen/system/irix.sm* file (see "Configuring a Kernel" on page 236).

You can examine VECTOR lines in */var/sysgen/system/irix.sm* to see how many adapters your system has, and which of the host adapter drivers listed in Table 15-1 is loaded for each one.

The adapter number, the target number, and the logical unit number are important parameters to all the functions of the host adapter driver.

Target Numbers

The purpose of a host adapter driver is to carry communications between a device driver and a *target*. A target is a device on the SCSI chain that responds to SCSI commands. A target can be a single device, or it can be a controller that in turn manages other devices.

A target is identified by a number between 0 and 15. Normally this number is configured into the device with switches or jumpers. The SCSI controller, usually target number 0 but 7 for the jag controller, cannot be used as a target.

The target number must be conveyed to the device driver somehow. The target numbers of Silicon Graphics disk and tape devices are passed in the device minor number.

Not all adapters support the range of 0-15 targets. The Jaguar VME-SCSI unit contains two independent adapters, each supporting target numbers 0-7.

Logical Unit Numbers (LUNs)

When the target is a controller, it manages one or more sub-devices, each one a *logical unit* of that target. The logical unit being addressed is identified by a logical unit number (LUN). When the target is a single device, its LUN is 0.

Overview of Host Adapter Functions

IRIX 6.2 permits a total of 10 unique host adapter drivers, but each of the ten must provide the same functional interface, which is based on simple concepts. The interface to host adapter drivers is declared in *sys/scsi.h*. Each adapter driver must provide the functions listed in Table 15-2.

Table 15-2 Host Adapter Function Summary

Function	Header Files	Can Sleep?	Purpose
scsi_info(D3)	scsi.h	Y	Issue the SCSI Inquiry command and return the results.
scsi_alloc(D3)	scsi.h	Y	Open a connection between a driver and a target device.
scsi_free(D3)	scsi.h	Y	Release connection to target device.
scsi_command(D3)	scsi.h	Y?	Transmit a SCSI command on the bus and return results.
scsi_abort()	scsi.h	Y?	Transmit a SCSI ABORT command (see caution).
scsi_reset()	scsi.h	Y?	Reset the SCSI adapter or bus (see caution).

The normal sequence of operations is as follows:

1. In the *pfx***open()** entry point (or, rarely, in an initialization entry point), the device driver calls **scsi_info()** to test the device characteristics. The results verify that the target device exists and is of the expected type.
2. In the *pfx***open()** entry point, the device driver calls **scsi_alloc()** to set up communications with the target device. This allocates resources in the host adapter driver.
3. In the *pfx***strategy()** or *pfx***ioctl()** entry points, the device driver constructs SCSI command strings and calls **scsi_command()** to have them executed.
4. In the *pfx***close()** entry point, the device driver calls **scsi_free()** to release any held resources related to this device.

Caution: The program interface to the **scsi_abort()** and **scsi_reset()** functions is likely to change in a near-future release. If you need these powerful functions, it is a good idea to code the function calls as macros that can easily be changed.

How the Host Adapter Functions Are Found

A SCSI device driver can be asked to manage devices on different adapters. But different adapters can use different hardware, and be managed by different host adapter drivers. When opening one device, the device driver might need to call **scsi_alloc()** as provided by the *wd93* driver. When opening a different device, the driver might need the **scsi_alloc()** function from the *jag* driver. How can a driver locate the correct host adapter function for a given device?

The answer is provided by a set of function vector tables that are indexed by adapter number, and that yield the address of the appropriate function for that adapter.

Using the Function Vector Tables

The function vector tables are maintained by the *scsi* driver module and filled in by each host adapter driver as it is initialized. The four vector tables are declared in *sys/scsi.h*. The declaration of table *scsi_command* is as follows:

```
extern void (*scsi_command[])(struct scsi_request *req);
```

This declaration states that *scsi_command* is an array of pointers to functions. Each function has the prototype

```
void function(struct scsi_request *req);
```

The four tables *scsi_command*, *scsi_alloc*, *scsi_free*, and *scsi_info* are similar. Each is an array of pointers to functions. Each array is indexed by the adapter type number. If *iAdapT* is an integer variable containing the adapter type number for a device, the following statements are valid calls to the four host adapter functions (the function arguments are examined in detail in the following topics):

```
#include <sys/scsi.h>
pTargInfo = (*scsi_info[iAdapT])(iAdap,iTarg,iLun);
iAllocRet = (*scsi_alloc[iAdapT])(iAdap,iTarg,iLun,0,NULL);
(void) (*scsi_command[iAdapT])(&request);
(void) (*scsi_free[iAdapT])(iAdap,iTarg,iLun,NULL);
```

Each statement is a function call, but in each case, the name of the function is replaced by an expression that indexes the appropriate table.

Learning the Adapter Type Number

Clearly, a SCSI driver needs to know the adapter type number for each device that it manages. Otherwise it cannot call the host adapter functions to manage that device.

The adapter type number for each adapter in the system is stored in an array maintained by the *scsi* driver. The array is declared as follows in *sys/scsi.h*:

```
extern u_char scsi_driver_table[];
```

When indexed by the number of the adapter in use, this table returns the adapter type number of the host adapter driver for that adapter.

Learning the Adapter Number

Now all that remains is for the device driver to learn the adapter number with each device that it manages. There are two simple ways to do this.

One method is to get the number in the *edt_t* structure. When a device is configured using a VECTOR line, the VECTOR should contain an *adapter=n* parameter. This number is stored in the *e_adap* field of the *edt_t* structure that is passed to the *pfx***edtinit()** entry point. Code to retrieve it in a hypothetical driver is shown in Example 15-1.

Example 15-1 Storing the Adapter Type Number in pfxedtinit()

```
#include <sys/scsi.h>
typedef struct devVital_s {
   uchar devAdapNum;
   uchar devAdapType;
...} devVital_t;
void hypo_edtinit(edt_t *edt)
{
   devVital_t *pVitals;
   ...
   pVitals->devAdapNum = edt->e_adap;
   pVitals->devAdapType = scsi_driver_table[edt->e_adap];
...
}
```

A second method is to get it from the device minor number. For all Silicon Graphics disk and tape devices, the adapter number is encoded into both the visible name and the minor number of the device special file. You can use the bits of the minor number of any device in a similar way (see “Minor Device Number” on page 35).

Under the second plan, **geteminor()** is used to extract the minor number is extracted from the *dev_t* value passed to each entry point (see “Device Number Functions” on page 181). The adapter number is calculated by shifting and masking the minor number. Hypothetical example code is shown in Example 15-2. The code of Example 15-2 can be extended to macros for the logical unit and control unit in obvious ways.

Example 15-2 Extracting an Adapter Number From a Minor Device Number

```
/* Hypothetical minor bits: 00 aaaaaaaa ccccuuuu */
#define MINOR_ADAP_SHIFT 8
#define MINOR_ADAP_MASK 0x00ff
#define MINOR_ADAP(devt) (MINOR_ADAP_MASK & \
                  (geteminor(devt) >> MINOR_ADAP_SHIFT))
```

When the adapter number is known, the expression to call a host adapter function can be converted to a macro as well, possibly making the code more readable. The macro in Example 15-3 encapsulates a call to **scsi_alloc()**. This code takes advantage of the fact that the adapter number is an argument to the function in any case.

Example 15-3 Macro to Encapsulate a Call to scsi_alloc()

```
#define SCSI_ALLOC(adap,targ,lun,opt,func) \
   (*scsi_alloc[scsi_driver_table[adap]]) \
   (adap,targ,lun,opt,func)
```

It could be argued that the double indexing in Example 15-3 imposes needless overhead. An approach with minimum overhead is to reserve space in the device-information structure for four function addresses, and to store the addresses of the host adapter functions with the other unique device information when the device is initialized.

Using scsi_info()

Before a SCSI driver tries to access a device, it must call the host adapter **scsi_info()** function. This function issues an Inquiry command to the specified adapter, target, and logical unit. If the Inquiry is not successful—or if the adapter, target, or LUN is invalid—the return value is NULL. Otherwise, the return value is a pointer to a *scsi_target_info* structure.

The SCSI driver can learn the following things from a call to **scsi_info()**:

- If the return is NULL, there is a serious problem with the device or the information about it. Write a descriptive log message with **cmn_err()** and return ENODEV.
- The *si_inq* field points to the Inquiry bytes returned by the device. Examine them for device-dependent information.
- Note the value in *si_maxq* for the largest number of pending SCSI commands that can be queued to this host adapter driver.
- Test the bits in *si_ha_status* for information about the capabilities and error status of the host adapter itself. The possible bits are declared in *sys/scsi.h*. For example, SRH_NOADAPSYNC indicates that the specified target, or possibly the host adapter itself, does not support synchronous transfer. Not all bits are supported by all host adapter drivers.

You can also call **scsi_info()** at other times; some of the returned information can be useful in error recovery. However, be aware that **scsi_info()** for some host adapters is slow, and can use serialized access to hardware.

Using scsi_alloc()

Depending on its particular design, the host adapter driver may need to allocate memory for data structures, DMA maps, or buffers for sense and inquiry data, before it is ready to execute commands to a particular target. The call to **scsi_alloc()** gives the host adapter driver the opportunity to prepare in these ways.

Because the host adapter driver may allocate virtual memory, it may sleep. Some host adapter drivers allocate all the resources they need on the first call to **scsi_alloc()** and do little or nothing on subsequent calls. However, you cannot predict whether your driver will make the first call or not.

A SCSI device driver will typically call the **scsi_alloc()** function from the *pfx***open()** entry point. However, if the driver needs to issue commands to the device at initialization time, it would call **scsi_alloc()**, use **scsi_command()**, and call **scsi_free()**, all within the *pfx***init()** or *pfx***edtinit()** entry point.

A call to **scsi_alloc()** specifies these parameters:

adap, *targ*, *lun*	Numbers that identify the device on the bus.
option	An integer comprising two parameters, a flag and a count.
callback	Address of a function to be called whenever sense data is gotten from the device.

The option parameter may include the SCSIALLOC_EXCLUSIVE flag to request exclusive use of the target. For example, a tape device driver would typically use exclusive access, while a disk device driver would not. If another driver has allocated a path to the same device, **scsi_alloc()** return EBUSY.

The option parameter may include SCSIALLOC_NOSYNC to specify that this device should not, or cannot, use synchronous transfer mode. That setting can be overridden for single commands by a flag to **scsi_command()** (see Table 15-4 on page 364).

The option parameter can also include a small integer value indicating the number of SCSI commands the driver would like to start before any have completed. The call to **scsi_info()** returns the maximum number of commands that can be queued in this way; a larger number is ignored.

The callback function address can be specified as NULL. The specified callback function is called only when sense data is gotten from the allocated device. Only one driver that allocates a path to a device can specify a callback function. If the path is not held exclusively, any other drivers must specify a null address for their callback functions.

Using scsi_free()

A SCSI driver typically calls **scsi_free()** from the *pfx***close()** entry point. That is the time when the driver knows that no processes have the device open, so the host adapter should be allowed to release any resources it is holding just for this device.

In addition, **scsi_free()** releases the device for use by other drivers, if the driver had allocated it for exclusive use.

Using scsi_command()

A SCSI device driver sends SCSI commands to its device by storing information in a *scsi_request* structure and passing the structure to the **scsi_command()** function for the adapter. The host adapter driver schedules the command on the SCSI bus that it manages, and returns to the caller. When the command completes, a callback function is invoked.

Tip: When debugging a driver using a debugging kernel (see "Preparing the System for Debugging" on page 243), you can display the contents of a *scsi_request* structure using *symmon* or *idbg* (see "Commands to Display I/O Status" on page 267).

Input to scsi_command()

The device driver prepares the *scsi_request* fields shown in Table 15-3.

Table 15-3 Input Fields of the scsi_request Structure

Field Name	Contents
sr_ctlr	The adapter number.
sr_target	The target number.
sr_lun	The logical unit number.
sr_tag	If this target supports the SCSI-2 tagged-queue feature, and this command is directed to a queue, this field contains the queue tag message. Constant names for queue messages are in *sys/scsi.h*: SC_TAG_SIMPLE and two others.
sr_command	Address of the bytes of the SCSI command to issue.
sr_cmdlen	The length of the string at **sr_command*. Constants for the common lengths are in *sys/scsi.h*: SC_CLASS0_SZ (6), SC_CLASS1_SZ (10), and SC_CLASS2_SZ (12).
sr_flags	Flags for data direction and DMA mapping, see Table 15-4.
sr_timeout	Number of ticks (HZ units) to wait for a response before timing out. The host adapter driver supplies a minimum value if this field is zero or too small.
sr_buffer	Address of first byte of data. Must be zero when *sr_bp* is supplied and SRF_MAPBP is specified in *sr_flags*.
sr_buflen	Length of data or buffer space.
sr_sense	Address of space for sense data, in case the command ends in a check condition.

Table 15-3 (continued) Input Fields of the scsi_request Structure

Field Name	Contents
sr_senselen	Length of the sense area.
sr_notify	Address of the callback function, called when the command is complete. A callback address is required on all commands.
sr_bp	Address of a *buf_t* object, when the command is called from a block driver's *pfx***strategy()** entry point and buffer mapping is requested in *sr_flags*.
sr_dev	Address of additional information that could be useful in the callback routine **sr_notify*.

The callback function address in *sr_notify* must be specified. (Device drivers for versions of IRIX previous to 5.x may set a NULL in this field; that is no longer permitted.)

The possible flag bits that can be set in *sr_flags* are listed in Table 15-4.

Table 15-4 Values for the sr_flags Field of a scsi_request

Flag Constant	Purpose
SRF_DIR_IN	Data will be received in memory. If this flag is absent, the command sends data from memory to the device.
SRF_FLUSH	The data cache for the buffer area should be flushed (for output) or marked invalid (for input) prior to the command. This flag should be used whenever the buffer is local to the driver, not mapped by a *buf_t* object. It causes no extra overhead in systems that do not require cache flushing.
SRF_MAPUSER	Set this flag when doing I/O based on a *buf_t* and the B_MAPUSER bit is set in *b_flags*.
SRF_MAP	Set this flag when doing I/O based on a *buf_t* and the BP_ISMAPPED macro returns nonzero.
SRF_MAPBP	The *sr_bp* field points to a *buf_t* in which BP_ISMAPPED returns false. The host adapter driver maps in the buffer.
SRF_AEN_ACK	This request is an acknowledgment of an AEN (Asynchronous Event Notification) message from the target. Following an AEN, any command without this flag is rejected with status SC_ATTN.

Table 15-4 (continued) Values for the sr_flags Field of a scsi_request

Flag Constant	Purpose
SRF_NEG_SYNC	Attempt to negotiate synchronous transfer mode for this command. Ignored by some host adapter drivers. Overrides SCSIALLOC_NOSYNC (see "Using scsi_alloc()" on page 361).
SRF_NEG_ASYNC	Attempt to negotiate asynchronous mode for this command. Ignored unless the device is currently using synchronous mode.

When none of the three flag values beginning SR_MAP are supplied, the sr_buffer address must be a physical memory address. The SR_MAPUSER and SR_MAPBP flags are normally used when the command is issued from a *pfx***strategy()** entry point in order to read or write a buffer controlled from a *buf_t* object.

Command Execution

The host adapter driver validates the contents of the *scsi_request* structure. If the contents are valid, it queues the command for transmission on the adapter and returns. If they are invalid, it sets a status flag (see Table 15-6), calls the *sr_notify* function, and returns.

In any event, the *sr_notify* function is called when the command is complete. This function can be called from the host adapter interrupt handler, so it must assume that it is called in interrupt state.

The device driver should wait for the notify function to be called. The usual way is to share a semaphore (see "Semaphores" on page 222), as follows:

- Prior to calling **scsi_command()**, initialize the semaphore to 0 (the semaphore is being used to wait for an event).
- Immediately after the call to **scsi_command()**, call **psema()** for the semaphore.
- In the notify function, call **vsema()** for the function.

If the request is valid, the device driver will sleep in the **psema()** call until the command completes. If the request is invalid, the semaphore will already have been posted when **psema()** is called.

When the device driver holds an exclusive lock prior to issuing the command, a synchronization variable provides an appropriate way to wait for command completion (see "Using Synchronization Variables" on page 220).

Values Returned in a scsi_request Structure

The host adapter driver sets the results of the request in the *scsi_request* structure. The sr_notify function is the first to inspect the values summarized in Table 15-5.

Table 15-5 Values Returned From a SCSI Command

Field Name	Purpose
sr_status	Software status flags, see Table 15-6.
sr_scsi_status	SCSI status byte, see Table 15-7.
sr_ha_flags	Host adapter status flags, see Table 15-8.
sr_sensegotten	When no sense command was issued, 0. When a sense command was issued following an error, the number of bytes of sense data received. When an error occurred during a sense command, -1
sr_resid	The difference between *sr_buflen* and the number of bytes actually transferred.

The *sr_status* field should be tested first. It contains an integer value; the possible values are summarized in Table 15-6.

Table 15-6 Software Status Values From a SCSI Request

Constant Name	Meaning
SC_GOOD	The request was valid and the command was executed. The command might still have failed; see *sr_scsi_status*.
SC_TIMEOUT	The device did not respond to selection within 250 milliseconds.
SC_HARDERR	A hardware error occurred; inspect *sr_senselen* to see how much sense data was received, if any.
SC_PARITY	SCSI bus parity error detected.
SC_MEMERR	System memory parity or ECC error detected.
SC_CMDTIME	The device responded to selection but the command did not complete before *sr_timeout* expired.
SC_ALIGN	The buffer address was not aligned as required by the adapter hardware. Most Silicon Graphics adapters require word (4-byte) alignment.

Table 15-6 (continued) Software Status Values From a SCSI Request

Constant Name	Meaning
SC_ATTN	Either a unit attention was received, or this command follows an AEN and did not contain the SR_AEN_ACK flag (see Table 15-4).
SC_REQUEST	An error was detected in the input values; the command was not attempted. The error could be that **scsi_alloc()** has not been called; or it could be due to missing or incorrect values.

One or more bits are set in the *sc_scsi_status* field. This field represents the status following the requested command, when the requested command executes correctly. When the requested command ends with Check Condition status, a sense command is issued and the SCSI status following the sense is placed in *sc_scsi_status*. In other words, the true indication of successful execution of the requested command is a zero in *sr_sensegotten*, because this indicates that no sense command was attempted.

Possible values of *sc_scsi_status* are summarized in Table 15-7.

Table 15-7 SCSI Status Bytes

Constant Name	Meaning
ST_GOOD	The target has successfully completed the SCSI command. If a check condition was returned, a sense command was issued. The *sr_sensegotten* field is nonzero when this was the case.
ST_CHECK	This bit is only set for the special case when a check condition occurred on a sense command following a check condition on the requested command. The *sr_sensegotten* field contains -1.
ST_COND_MET	Search condition was met.
ST_BUSY	The target is busy. The driver will normally delay and then request the command again.
ST_INT_GOOD	This status is reported for every command in a series of linked commands. Linked commands are not supported by Silicon Graphics host adapters.
ST_RES_CONF	A conflict with a reserved logical unit or reserved extent.

One or more bits can be set in *sr_ha_flags* to document a host adapter state or problem. These flags are summarized in Table 15-8.

Table 15-8 Host Adapter Status After a SCSI Request

Constant Name	Meaning
SRH_CANTSYNC	Unable to negotiate synchronous mode.
SRH_SYNCXFR	Synchronous mode was used. If not set, asynchronous mode was used.
SRH_TRIEDSYNC	Synchronous mode negotiation was attempted; see the SHR_CANTSYNC bit for the result.
SRH_BADSYNC	When SRH_CANTSYNC is set, this bit indicates that the negotiation failed because the device cannot negotiate.
SRH_NOADAPSYNC	When SRH_CANTSYNC is set, this bit indicates that the host adapter does not support synchronous negotiation, or that the system has been configured to not use synchronous mode for this device.
SRH_WIDE	This adapter supports Wide mode.
SRH_DISC	This adapter supports Disconnect mode and is configured to use it.
SRH_TAGQ	This adapter supports tagged queueing and is configured to use it.
SRH_MAPUSER	This host adapter driver can map user addresses.

Using scsi_abort()

The purpose of the **scsi_abort()** function is to issue a SCSI ABORT command to a specified target and logical unit. The prototype of the function is:

```
int (*scsi_abort[adapter-type])(struct scsi_request *req);
```

The only fields of the scsi_request that are input to this function are those that identify the device: *sr_ctlr*, *sr_target*, and *sr_lun*. The ABORT command is issued on the bus as soon as possible but there could be a delay if the bus is busy. Status is returned in *sr_status*. The function returns a nonzero value when the ABORT command is issued successfully, and a zero when the ABORT command fails (which probably indicates a serious bus problem).

Using scsi_reset()

The purpose of **scsi_reset()** is to reset the adapter hardware and possibly the attached bus, for example by asserting the reset line on the bus for at least 25 microseconds. The prototype of the function is

```
int (*scsi_reset[adapter-type])(uchar adap);
```

The adapter number is reset and a nonzero value is returned. If the host adapter driver does not support this function, or if it is unable to reset the hardware, it returns 0.

Designing a SCSI Driver

A kernel-level SCSI device driver has the driver architecture described in Chapter 8, "Structure of a Kernel-Level Driver," and it uses the basic system services documented in Chapter 9, "Device Driver/Kernel Interface." You prepare a SCSI driver and configure it into the kernel as described in Chapter 10, "Building and Installing a Driver."

However, a SCSI driver uses additional services, including those of the host adapter driver, and its configuration is slightly different from other drivers.

SCSI Driver Initialization

A SCSI driver can be included in the kernel through a VECTOR, INCLUDE, or USE line in the system file in */var/sysgen/system* (see "Configuring a Kernel" on page 236). When included through a VECTOR line, the *pfx***edtinit()** entry point is called for each VECTOR line given. The VECTOR can describe a logical unit or a control unit, according to your design choice. However, a VECTOR line for a high-level SCSI driver can not include a *probe* or *exprobe* operand, because the hardware is owned by the host adapter driver, not by the SCSI device driver.

When included through a USE line, a SCSI driver is initialized at its *pfx***init()** entry point. In this case, the driver must obtain the adapter number by some other means. (See "Initialization Entry Points" on page 143.)

Opening a SCSI Device

When the *pfx***open()** entry point is called, the SCSI driver uses the appropriate **scsi_info()** function to verify the device and get hardware dependent Inquiry data from it. If the device is operational, the driver calls **scsi_alloc()** to open a communications path to it.

Accessing a SCSI Device

In general, it is simplest to put all access to a device within a *pfx***strategy()** entry point, even in a character device driver. When the *pfx***read()**, *pfx***write()**, or *pfx***ioctl()** entry point needs to read or write data, it can prepare a *uio_t* to describe the data, and call **uiophysio()** to direct the operation through the single *pfx***strategy()** entry point.

The notify routine passed in the *sr_notify* field plays the same role as the *pfx***intr()** entry point in other device drivers. It is called asynchronously, when the SCSI command completes. It may not call a kernel function that can sleep. However, it does not have to be named *pfx***intr()**, and a SCSI driver does not have to provide a *pfx***intr()** entry point.

Configuring a SCSI Driver

A SCSI driver can be either a block or a character driver, or it can support both interfaces. When preparing the descriptive file for */var/sysgen/master.d*, you must use the *s* flag, specifying a software-only driver, and list *scsi* as a dependency, in the descriptive line in */var/sysgen/master.d*. See "Describing the Driver in /var/sysgen/master.d" on page 233.

Example SCSI Device Driver

The following example shows how a driver can communicate with a direct access SCSI device, such as a disk. This driver is simplified and does not do as much error checking as a real driver would do. Also, this example uses a single, global SCSI request structure that does not work in real drivers, since multiple reads or writes would overwrite a command in progress.

Tip: A more complete sample SCSI driver is available on the Developer's Toolbox CD. See information about the Developer Program under "Developer Program" on page xxxvii.

```
#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; /* not old, not _MP either */

#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);
}
```

Designing a Host Adapter Driver

IRIX 6.2 provides the ability to load and install third-party host adapter drivers. This section documents the special features of this type of driver.

Overview of Host Adapter Driver Architecture

A host adapter driver is a low-level driver for a SCSI bus adapter. A host adapter driver is similar to other device drivers described in this book in many ways:

- Like other device drivers, it uses the kernel facilities described in Chapter 9, "Device Driver/Kernel Interface."
- It is compiled and linked like other drivers (see Chapter 10, "Building and Installing a Driver").
- It is configured to the system using files in */var/sysgen/master.d*, and loaded by *lboot*. A host adapter driver should not be loadable; if it is loadable, it should not unload.
- Like other drivers, it can have entry points *pfx***start()** or *pfx***edtinit()** for initialization, *pfx***intr()** for interrupt handling, and *pfx***halt()** for shutdown.

Unlike other drivers, a host adapter driver does not provide any entry points for serving the needs of system functions, such as *pfx***read()**, *pfx***poll()**, or *pfx***strategy()**. Instead, it supplies the entry points used by SCSI device drivers.

Host Adapter Initialization

In its initialization, the host adapter driver does three things:

- initializes the adapter hardware it supports
- acquires an adapter type number
- stores pointers to its functions in the function pointer arrays

Initializing the Hardware

If it is called at its *pfx***edtinit()** entry point, the host adapter driver receives adapter hardware information in an *edt_t* structure. Integration of the driver in this way, using a VECTOR line, is preferred. It removes the need to hard-code device addresses; and it allows the use of an *exprobe* operand to load the driver only when its adapter hardware is present.

If it is not loaded by a VECTOR line, the driver must be loaded with a USE line in the system database (see "Configuring a Kernel" on page 236) and it must find out the address of its adapter hardware by other means.

The driver may also include a *pfx***start()** entry point for general initialization, including the two steps of acquiring a type number and setting up its entry point addresses.

Acquiring a Type Number

Every host adapter driver must have a unique adapter type number. The type numbers for Silicon Graphics drivers are declared in *sys/scsi.h*. An OEM driver acquires a number dynamically, by calling the kernel function **get_driver_number()**. The prototype of this function is

```
uchar get_driver_number(void);
```

If successful, the function returns a number between SCSIDRIVER_3RD_PARTY_START and SCSIDRIVER_3RD_PARTY_END, inclusive (see Table 15-1 on page 355). If unsuccessful, it returns -1, and the driver cannot initialize.

Storing Entry Point Addresses

After it has its type number, the driver can store the address of each of its functional entry points in the arrays used by its callers. For example it stores the address of its command execution function in the *scsi_command* array, indexed by its type number.

The driver must support the functions summarized in Table 15-2 on page page 357. For each function there is a corresponding array of function pointers, in which the driver stores the address of its function, indexed by its driver type number.

SCSI Reference Data

This section contains reference material in the following categories:

- “SCSI Error Messages” on page 377 describes the general form of messages written by host adapter drivers into the system log.
- “Adapter Error Codes (Table scsi_adaperrs_tab)” on page 378 lists the possible adapter error codes and their message strings.
- “SCSI Sense Codes (Table scsi_key_msgtab)” on page 379 lists the primary sense codes and the corresponding message strings.
- “Additional Sense Codes (Table scsi_addit_msgtab)” on page 380 lists the possible additional sense codes (ASCs) and their message strings.

SCSI Error Messages

The host adapter drivers such as *wd93*, *wd95*, and *jag* send error messages to the system log using the **cmn_err()** function (see “Producing Diagnostic Displays” on page 249).

These messages almost always contain the adapter number (sometimes called the bus number or controller number). They sometimes contain the number of the target device, and sometimes add the number of the logical unit that was addressed.

Messages from the *wd93* driver specify the adapter number as `Bus=`*n*. The target device is shown as `ID=`*n* and the logical unit as `LUN=`*n*.

Messages from the *wd95* and *jag* drivers contain one, two, or three or more decimal numbers. In all cases, the first number is the adapter number, the second is the target ID, and the third (when present) is the logical unit number.

When error messages list a sense code, refer to “SCSI Sense Codes (Table scsi_key_msgtab)” on page 379 and to “Additional Sense Codes (Table scsi_addit_msgtab)” on page 380.

When the error message reports an error from the adapter itself, refer to “Adapter Error Codes (Table scsi_adaperrs_tab)” on page 378.

SCSI Error Message Tables

The *scsi* module contains a set of error message tables that you can use to generate error messages based on SCSI sense codes and other data. The contents of these tables is documented here for reference, and to assist in decoding messages from SCSI drivers.

Each table is an array of pointers to strings; for example the *scsi_key_msgtab* table is defined beginning as follows:

```
char *scsi_key_msgtab[SC_NUMSENSE] = {
   "No sense",          /* 0x0 */
   "Recovered Error",   /* 0x1 */
...};
```

Each of the tables is declared as extern in *sys/scsi.h*.

Adapter Error Codes (Table scsi_adaperrs_tab)

The table with the external name *scsi_adaperrs_tab* contains message strings to document the adapter error codes that can be returned in the *scsirequest.sr_status* field (see Table 15-6). The *scsi_adaperrs_tab* table contains NUM_ADAP_ERRS entries (9, defined in *sys/scsi.h*). The first entry (index 0x0) contains a pointer to a null string. The other entries are documented in Table 15-9.

Table 15-9 Adapter Error Codes

Adapter Error Code	Constant Name	Message Text
0x1	SC_TIMEOUT	Device does not respond to selection .
0x2	SC_HARDERR	Controller protocol error or SCSI bus reset.
0x3	SC_PARITY	SCSI bus parity error.
0x4	SC_MEMERR	Parity/ECC error in system memory during DMA.
0x5	SC_CMDTIME	Command timed out.
0x6	SC_ALIGN	Buffer not correctly aligned in memory.
0x7	SC_ATTN	Unit attention received on another command causes retry.
0x8	SC_REQUEST	Driver protocol error.

SCSI Sense Codes (Table scsi_key_msgtab)

The table with the external name *scsi_key_msgtab* is indexed by the primary sense code. Its contents are listed in Table 15-10. The table contains SC_NUMADDSENSE entries (16, defined in *sys/scsi.h*), of which the last two should not occur.

Table 15-10 Primary Sense Key Error Table

Sense Key	Message	Most Common Cause
0x0	`No sense`	No error information available.
0x1	`Recovered error`	The device recovered by itself.
0x2	`Device not ready`	No media, or drive not spun up.
0x3	`Media error`	An actual media problem.
0x4	`Device hardware error`	Usually a device hardware error.
0x5	`Illegal request`	Invalid command or data issued.
0x6	`Unit attention`	Device was reset or power-cycled.
0x7	`Data protect error`	Usually device is write-protected.
0x8	`Unexpected blank media`	Tried to read at end of a tape.
0x9	`Vendor unique error`	Varies.
0xA	`Copy aborted`	Copy command aborted by host (not used).
0xB	`Aborted command`	Target device aborted command.
0xC	`Search data successful`	Search data command OK (not used).
0xD	`Volume overflow`	Tried to write past EOT on tape.
0xE	`Reserved (0xE)`	0xE should not be seen.
0xF	`Reserved (0xF)`	0xF should not be seen.

Additional Sense Codes (Table scsi_addit_msgtab)

The table with the external name *scsi_addit_msgtab* is indexed by the Additional Sense Code (ASC) value, when one is present. The table contains SC_NUMADDSENSE entries (0x71, defined in *sys/scsi.h*). Some values have no standard definition; for these, the table contains a NULL value. Therefore you should always test the table value for a valid pointer before using it to format a message.

Table 15-11 lists the contents of this message table. Undefined (NULL) table entries are omitted.

Table 15-11 Additional Sense Code Table

ASC Value	Corresponding Message String
0x01	`No index/sector signal`
0x02	`No seek complete`
0x03	`Write fault`
0x04	`Not ready to perform command`
0x05	`Unit does not respond to selection`
0x06	`No reference position`
0x07	`Multiple drives selected`
0x08	`LUN communication error`
0x09	`Track error`
0x0a	`Error log overflow`
0x0c	`Write error`
0x10	`ID CRC or ECC error`
0x11	`Unrecovered data block read error`
0x12	`No address mark found in ID field`
0x13	`No address mark found in Data field`
0x14	`No record found`
0x15	`Seek position error`

Table 15-11 (continued) Additional Sense Code Table

ASC Value	Corresponding Message String
0x16	`Data sync mark error`
0x17	`Read data recovered with retries`
0x18	`Read data recovered with ECC`
0x19	`Defect list error`
0x1a	`Parameter overrun`
0x1b	`Synchronous transfer error`
0x1c	`Defect list not found`
0x1d	`Compare error`
0x1e	`Recovered ID with ECC`
0x20	`Invalid command code`
0x21	`Illegal logical block address`
0x22	`Illegal function`
0x24	`Illegal field in CDB`
0x25	`Invalid LUN`
0x26	`Invalid field in parameter list`
0x27	`Media write protected`
0x28	`Media change`
0x29	`Device reset`
0x2a	`Log parameters changed`
0x2b	`Copy requires disconnect`
0x2c	`Command sequence error`
0x2d	`Update in place error`
0x2f	`Tagged commands cleared`
0x30	`Incompatible media`

Table 15-11 (continued) Additional Sense Code Table

ASC Value	Corresponding Message String
0x31	`Media format corrupted`
0x32	`No defect spare location available`
0x33[a]	`Media length error`
0x36	`Toner/ink error`
0x37	`Parameter rounded`
0x39	`Saved parameters not supported`
0x3a	`Medium not present`
0x3b	`Forms error`
0x3d	`Invalid ID msg`
0x3e	`Self config in progress`
0x3f	`Device config has changed`
0x40	`RAM failure`
0x41	`Data path diagnostic failure`
0x42	`Power on diagnostic failure`
0x43	`Message reject error`
0x44	`Internal controller error`
0x45	`Select/reselect failed`
0x46	`Soft reset failure`
0x47	`SCSI interface parity error`
0x48	`Initiator detected error`
0x49	`Inappropriate/illegal message`
0x4a	`Command phase error`
0x4b	`Data phase error`
0x4c	`Failed self configuration`

Table 15-11 (continued) Additional Sense Code Table

ASC Value	Corresponding Message String
0x4e	`Overlapped commands attempted`
0x53	`Media load/unload failure`
0x57	`Unable to read table of contents`
0x58	`Generation (optical device) bad`
0x59	`Updated block read (optical device)`
0x5a	`Operator request or state change`
0x5b	`Logging exception`
0x5c	`RPL status change`
0x5d	`Self diagnostics predict unit will fail soon`
0x60	`Lamp failure`
0x61	`Video acquisition error/focus problem`
0x62	`Scan head positioning error`
0x63	`End of user area on track`
0x64	`Illegal mode for this track`
0x70[b]	`Decompression error`

a. Specified as tape only.

b. DAT only; may be in SCSI3.

WD93 States and Phases

Some of the SCSI states and phases that can be detected by the *wd93* host adapter driver are listed in Table 15-12 for reference, in case they appear in a debugging log message. These states and phases are declared in the */usr/include/sys/wd93.h* header file. The comments in the table have been extracted from that file and supplemented with additional information.

"Out" is from the CPU to the SCSI device in these descriptions, and "receive" and "send" are also from the SCSI device point of view, since the target controls all the bus phases except for initial selection.

Table 15-12 SCSI State Error Messages

State Message	Sense Key	Comments
`ST_RESET`	0x00	SCSI chip reset by reset command or power-up.
`ST_SELECT`	0x11	Selection of target complete (after C93SELATN).
`ST_SATOK`	0x16	Select-And-Transfer completed successfully, that is, all phases have completed in a normal manner.
`ST_TR_DATAOUT`	0x18	Transfer command done, target requesting data.
`ST_TR_DATAIN`	0x19	Transfer command done, target sending data.
`ST_TR_STATIN`	0x1b	Target is sending status in.
`ST_TR_MSGIN`	0x1f	Transfer command done, target sending message.
`ST_TRANPAUSE`	0x20	Transfer command has paused with ACK.
`ST_SAVEDP`	0x21	Save Data Pointers message during SAT normal state when device is disconnecting from the bus.
`ST_A_RESELECT`	0x27	Reselected after disconnect (93A).
`ST_UNEXPDISC`	0x41	Device disconnected without sending a disconnect message. This sometimes happens when devices with removable media have had the media removed during a transfer.
`ST_PARITY`	0x43	Command terminated due to parity error on the SCSI bus.

Table 15-12 (continued) SCSI State Error Messages

State Message	Sense Key	Comments
ST_PARITY_ATN	0x44	Command terminated due to parity error (ATN is asserted so that host can send a message to device; the transfer is just aborted).
ST_TIMEOUT	0x42	Time-out during Select or Reselect, that is, the device never responded to an attempt to select it; normally seen only during hardware inventory probing, but sometimes happens after a SCSI bus reset if device takes a long time to recover from the reset or is powered off.
ST_INCORR_DATA	0x47	Incorrect message or status byte.
ST_UNEX_RDATA	0x48	Unexpected receive data phase device tried to send more data than the SCSI chip is programmed to expect. This can be OK, as when a high-level request is made to transfer more data than the DMA hardware can map on a single request. In this case, simply reprogram the DMA hardware for the next chunk of data and restart the transfer (but don't send a new SCSI command to the device). When printed as part of an error message, it can sometimes be caused by a SCSI cabling problem, or (particularly with devscsi user drivers) by a mismatch in the byte count given to the driver and the byte count implied by the SCSI command sent to the device.
ST_UNEX_SDATA	0x49	Unexpected send-data phase (same as above, but device is asking for more data).
ST_UNEX_CMDPH	0x4a	Unexpected command phase
ST_UNEX_SSTATUS	0x4b	Unexpected send status phases occur at the end of SCSI command (that is, byte count remaining is 0); if they happen at other times, the chip interrupts. This can happen when you ask a device for more data than it can give you, and in this case, you just return a short I/O count to the caller. When printed as part of an error message, it usually implies a cabling or termination problem.
ST_UNEX_RMESGOUT	0x4e	Unexpected request-message-out phase; usually indicates a SCSI cabling problem.

Table 15-12 (continued) SCSI State Error Messages

State Message	Sense Key	Comments
`ST_UNEX_SMESGIN`	0x4f	Unexpected send-message-in phase. Usually indicates a SCSI cabling problem; also happens when device sends an unsolicited disconnect message when preparing to disconnect from the bus.
`ST_RESELECT`	0x80	WD33C93 has been reselected.
`ST_93A_RESEL`	0x81	Reselected while idle (93A).
`ST_DISCONNECT`	0x85	Disconnect has occurred.
`ST_NEEDCMD`	0x8a	Target is ready for a command.
`ST_REQ_SMESGOUT`	0x8e	REQ signal for send message out.
`ST_REQ_SMESGIN`	0x8f	REQ signal for send message in above 3 usually seen only during sync negotiations.