User-Level Access to SCSI Devices

IRIX contains a programming library, called *dslib*, that allows you to control SCSI devices from a user-level process. This chapter documents the functions in dslib, including the following topics:

- "Overview of the dsreq Driver" on page 78 gives a summary of the features and use of the generic SCSI device driver.
- "Generic SCSI Device Special Files" on page 78 documents the format of the names and major and minor numbers of generic SCSI files.
- "The dsreq Structure" on page 82 gives details of the request structure that is the primary input to the generic SCSI driver.
- "Testing the Driver Configuration" on page 88 documents the use of the DS_CONF **ioctl()** operation.
- "Using dslib Functions" on page 90 describes the functions that make it simpler to use the generic SCSI driver.
- "Using the Special DS_RESET and DS_ABORT Calls" on page 89 describes two special functions of the generic SCSI driver.

You must understand the SCSI interface in order to command a SCSI device. For several SCSI information resources, see "Other Sources of Information" on page xxxvii.

If you are specifically interested in using audio data from a CDROM or DAT drive, you should use the special-purpose libraries for CDROM and DAT that are included in the IRIS Digital Media Development Environment. These libraries are built upon the generic SCSI driver, but provide convenient, audio-oriented functions. For more information on these libraries, see the *IRIS Digital Media Programming Guide*, document number 008-1799-040.

If your interest is in controlling SCSI devices at the kernel level, see Part V, "SCSI Device Drivers."

Overview of the dsreq Driver

IRIX includes a generic SCSI device driver, the *dsreq* driver, through which a user-level program can issue SCSI commands to SCSI devices. This is a character device driver that supports only **open()**, **close()** and **ioctl()** operations (see “Kinds of Kernel-Level Drivers” on page 45, and also the open(2), close(2) and ioctl(2) reference pages).

The formal documentation of the *dsreq* driver is found in the ds(7) reference page. In order to invoke its services, you prepare a *dsreq* data structure describing the operation and pass it to the device driver using an **ioctl()** call. The device driver issues the SCSI command you specify, and sleeps until it has completed. Then it returns the status in the *dsreq* structure.

You can request operations for input and output as well as issuing control and diagnostic commands. The *dsreq* structure for input and output operations specifies a buffer in memory for data transfer. The *dsreq* driver handles the task of locking the buffer into memory (if necessary) and managing a DMA transfer of data.

The programming interface supported by the generic SCSI driver is quite primitive. A library of higher-level functions makes it easier to use. This library is formally documented in the dslib(3) reference page, and is described under “Using dslib Functions” on page 90.

Generic SCSI Device Special Files

The creation and use of device special files is discussed under “Device Special Files” on page 32. A device special file represents a device, and is the mechanism for associating a device with a kernel-level device driver.

The device special files in the */dev/scsi* directory are all associated with the *dsreq* driver. A basic set of these names is created automatically by the */dev/MAKEDEV* script (see “The Script MAKEDEV” on page 36). You have to create additional device special files if you need to control logical units other than logical unit 0.

Form of Names in /dev/scsi

Each device special filename in the */dev/scsi* directory reflects the values of the device's adapter (bus) number, SCSI ID, and logical unit number (LUN).

Tip: The character between the SCSI ID and the LUN in these names is the letter "l." When reading or copying these device names, take care not to write a digit 1 instead. This is a frequent error.

Names of SCSI Devices on a SCSI Bus

Devices attached directly to a SCSI bus have names in this form:

sc	Prefix "sc" for SCSI attachment.
0 to 137	Number of the SCSI adapter, typically 0 or 1.
d	Constant letter "d" for device.
0 to 7 (to 15 for wide SCSI)	SCSI ID of the target device or control unit, as set by switches on the device itself.
l (letter ell)	Constant letter "l" for logical unit.
0 to 7	Logical unit number (LUN) of this device, typically 0.

A typical device name would be */dev/scsi/sc1d3l0* meaning a SCSI device configured as ID 3 on SCSI bus 1. Either this device has no logical units, or this is the first logical unit on device 3.

Names of SCSI Devices on the Jag (VME Bus) Controller

Machines in the Challenge and Onyx systems can optionally have SCSI devices attached to the VME bus through a bridge using the *jag* device driver. These devices are also represented in */dev/scsi* with names of the following form:

jag	Prefix "jag" for VME/SCSI attachment.
0 to 4	Number of the VME adapter, typically 0 or 1.
d	Constant letter "d" for device.
0 to 7 (to 15 for wide SCSI)	SCSI ID of the target device or control unit, as set by switches on the device itself.
l (letter ell)	Constant letter "l" for logical unit.
0 to 7	Logical unit number (LUN) of this device, typically 0.

A typical device name would be */dev/scsi/jag1d3l0* meaning a SCSI device configured as ID 3 on VME bus 1. Either the device has no logical units, or this is the first logical unit on device 3.

Major and Minor Device Numbers in /dev/scsi

Device special files in */dev/scsi* have one of the following major device numbers:

- 195 for devices on a SCSI bus (files */dev/scsi/sc**).
- 196 for devices on a *jag* (VME) SCSI bridge (files */dev/scsi/jag**).

The minor number of these files encodes the adapter number, the SCSI ID, and the LUN, using the bit assignments shown in Figure 5-1.

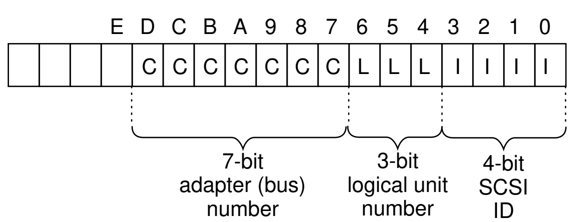

Figure 5-1 Bit Assignments in SCSI Device Minor Numbers

Creating Additional Names in /dev/scsi

The script */dev/MAKEDEV*, which runs each time the system boots, creates 16 files for each existing SCSI or jag bus. These files represent the possible SCSI ID numbers 0-15 on each bus, with a logical number of 0. If you want to control a device with LUN 0, the device special file exists.

In order to control a device with a LUN of 1-7, you must create an additional device special file, using the *mknode* or *install* command (see the install(1) reference page). For example, before you can operate logical unit 2 of device 5 on SCSI bus 1, you must create */dev/scsi/sc1d5l2* using a command such as

```
install -F /dev/scsi -m 600 -u root -g sys \
-chr 195,165 sc1d5l2
```

Relationship to Other Device Special Files

The files in */dev/scsi* describe many of the same devices that are described by files in */dev/dsk*, */dev/tape*, and other directories. There is a security exposure in that a user-level program could use a */dev/scsi* file to do almost anything to a disk or tape, including total erasure.

The *dsreq* device driver forces exclusivity with itself; that is, a given */dev/scsi* file can be opened only by one process at a time. However, a device could be open through the *dsreq* driver at the same time it is open by another process, or by a filesystem, through a different device special file and device driver. For example, a disk volume could be simultaneously open through the name */dev/scsi/sc0d0l0* and through */dev/rdsk/dks0d1s0*.

The process that opens a generic SCSI device can request exclusivity using the O_EXCL option to **open()**. In that case, the open is rejected when the device is already open through another driver; and no other driver can open the device until the generic device file is closed.

The dsreq Structure

The primary input to most *dsreq* **ioctl()** calls, as well as the primary input to most dslib functions, is the *dsreq* structure. This structure is declared in */usr/include/sys/dsreq.h*, a header file that rewards careful study.

The important fields of the *dsreq* structure are shown in Table 5-1. Some of the field values are expanded in the following topics. The *sys/dsreq.h* file declares macros for access to many of the fields. Use these macros (listed in Table 5-1) in both expressions and assignments in order to insulate your code against future changes.

Table 5-1 Fields of the dsreq Structure

Field Name	Macro	Purpose
ds_flags	FLAGS(dp)	Bits used to determine device driver actions. See "Values for ds_flags" on page 83.
ds_time	TIME(dp)	Timeout value in milliseconds. If the command does not complete, it is ended with an error code. The driver sets a default of 5000 (5 seconds) when this is set to zero. **dsopen()** initializes it to 10000.
ds_private	PRIVATE(dp)	Field for use by the calling program. **dsopen()** uses this field to point to its "context" data (see "Using dsopen() and dsclose()" on page 92).
ds_cmdbuf	CMDBUF(dp)	Address of SCSI command string to be sent.
ds_cmdlen	CMDLEN(dp)	Length of the SCSI command string.
ds_databuf	DATABUF(dp)	Address of a single data buffer. See "Data Transfer Options" on page 85.
ds_datalen	DATALEN(dp)	Length of data buffer.
ds_sensebuf	SENSEBUF(dp)	Address to receive sense data after an error.
ds_senselen	SENSELEN(dp)	Length of sense buffer in bytes.
ds_iovbuf	IOVBUF(dp)	Address of an *iov_t* structure. See "Data Transfer Options" on page 85.
ds_iovlen	IOVLEN(dp)	Length of data described by *ds_iovbuf*.
ds_link		This field is not supported, and should be zero-filled.

Table 5-1 (continued) Fields of the dsreq Structure

Field Name	Macro	Purpose
ds_synch		This field is not supported, and should be zero-filled.
ds_revcode		Intended for the version code of the *dsreq* driver, not currently set to a useful value.
ds_ret	RET(dp)	Return code for the requested operation. See Table 5-3 on page 85.
ds_status	STATUS(dp)	SCSI status byte from the operation. See Table 5-4 on page 87.
ds_msg	MSG(dp)	The first byte of a message returned by the target. See Table 5-5 on page 87.
ds_cmdsent	CMDSENT(dp)	Length of command string actually sent (same as *ds_cmdlen*, unless an error occurs).
ds_datasent	DATASENT(dp)	Length of data transferred.
ds_sensesent	SENSESENT(dp)	Length of sense data received.

The dslib library contains functions to simplify the preparation and execution of a *dsreq* request; see "Using dslib Functions" on page 90.

Values for ds_flags

The possible flag values in the *ds_flags* field are listed in Table 5-2. The flag values are designed for the most flexible, capable type of bus, device, and device driver. Not all values are supported, and different host adapters can support different combinations.

Table 5-2 Flag Values for ds_flags

Constant Name	Supported by Any Driver?	Meaning When Set to 1
DSRQ_ASYNC	Yes	Return at once, do not sleep until the operation is complete.
DSRQ_SENSE	Yes	Get sense data following an error on the requested command.
DSRQ_TARGET	No	Act as the SCSI target, not the SCSI initiator.

Table 5-2 (continued) Flag Values for ds_flags

Constant Name	Supported by Any Driver?	Meaning When Set to 1
DSRQ_SELATN	Yes	Select with ATN.
DSRQ_DISC	Yes	Allow identify disconnect.
DSRQ_SYNXFR	Yes	Negotiate a synchronous transfer if possible. Needed only to switch into synchronous mode. Repeated negotiation is wasteful.
DSRQ_ASYNXFR	Yes	Negotiate an asynchronous transfer. Needed only to return to asynch after a synchronous transfer. Repeated negotiation is wasteful.
DSRQ_SELMSG	No	A specific select is coded in the message. This feature is not supported.
DSRQ_IOV	Yes	Use the *iov_t* from *ds_iovbuf*, not the single buffer from *ds_databuf* (see "Data Transfer Options" on page 85).
DSRQ_READ	Yes	This is a data input command (as opposed to an immediate command or an output).
DSRQ_WRITE	Yes	This is a data output command (as opposed to an immediate command or an input).
DSRQ_MIXRDWR	No	This command can both read and write.
DSRQ_BUF	No	Buffer the input and copy to the supplied buffer, instead of direct input to the buffer.
DSRQ_CALL	No	Notify completion (with DSRQ_ASYNC).
DSRQ_ACKH	No	Hold ACK asserted.
DSRQ_ATNH	No	Hold ATN asserted.
DSRQ_ABORT	No	Send ABORT messages until the bus is clear.Useful only with SCSI commands that have the immediate bit set.
DSRQ_TRACE	Yes	Trace this request (accepted but has no effect).
DSRQ_PRINT	Yes	Print this request (accepted but has no effect).
DSRQ_CTRL1	Yes	Request with host control bit 1.
DSRQ_CTRL2	Yes	Request with host control bit 2.

In order to find out which flags are supported by a particular driver, use the DS_CONF operation (see “Testing the Driver Configuration” on page 88).

Data Transfer Options

When reading or writing data, you have two design options:

- You can transfer a single segment of data directly between the device and a buffer you supply (set neither DSRQ_BUF nor DSRQ_IOV).
- You can transfer segments of data between the device and a series of one or more memory locations based on an *iov_t* object (set DSRQ_IOV).

All read/write requests are done using DMA. The “scatter/gather” support of DSRQ_IOV is presently restricted to only one memory segment, so it is not greatly different from single-buffer I/O. If you elect to use it, the *iov_t* structure is declared in *sys/iov.h* (see also the part of the read(2) reference page that deals with the **readv()** function).

During a direct transfer using either a single buffer or scatter/gather, the data buffer spaces are locked in memory.

The maximum amount of data you can transfer in one operation is set by the host adapter driver for the bus, and can be retrieved with an **ioctl()** (see “Testing the Driver Configuration” on page 88). The maximum length for a buffered transfer is returned by the same **ioctl()**. It can be less than the direct-transfer size because there may be a limit on the size of kernel memory that can be allocated.

Return Codes and Status Values

A zero return code in the *ds_ret* field signifies success. The possible nonzero return codes are summarized in Table 5-3 and are declared in *sys/dsreq.h*. Not all return codes are possible with every driver.

Table 5-3 Return Codes From SCSI Operations

Constant Name	Meaning
DSRT_DEVSCSI	General failure from SCSI driver.
DSRT_MULT	General software failure, typically a SCSI-bus request.

Table 5-3 (continued) Return Codes From SCSI Operations

Constant Name	Meaning
DSRT_CANCEL	Operation cancelled in host adapter driver.
DSRT_REVCODE	Software level mismatch, recompile application.
DSRT_AGAIN	Try again, recoverable SCSI-bus error.
DSRT_HOST	Failure reported by host adapter driver for the bus in use.
DSRT_NOSEL	No unit responded to select.
DSRT_SHORT	Incomplete transfer (not an error). See *ds_datasent*.
DSRT_OK	Not returned at this time.
DSRT_SENSE	Command returned with status; sense data successfully retrieved from SCSI host (see *ds_sensesent*).
DSRT_NOSENSE	Command with status, error occurred while trying to get sense data from SCSI host.
DSRT_TIMEOUT	Command did not complete in the time allowed by *ds_timeout*.
DSRT_LONG	Data transfer overran bounds (*ds_datalen*).
DSRT_PROTO	Miscellaneous protocol failure.
DSRT_EBSY	Busy dropped unexpectedly; protocol error.
DSRT_REJECT	Message rejected; protocol error.
DSRT_PARITY	Parity error on SCSI bus; protocol error.
DSRT_MEMORY	Memory error in system memory.
DSRT_CMDO	Protocol error during command phase.
DSRT_STAI	Protocol error during status phase.
DSRT_UNIMPL	Command not implemented; protocol error.

The possible SCSI status value in the *ds_status* field are summarized in Table 5-4.

Table 5-4 SCSI Status Codes

Constant Name	Meaning
STA_GOOD	The target has successfully completed the SCSI command.
STA_CHECK	An error or exception was detected. Sense was attempted if DSRQ_SENSE was specified.
STA_BUSY	Command not attempted; addressed unit is busy.
STA_IGOOD	Linked SCSI command 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.

The possible SCSI message byte values in the *ds_msg* field are summarized in Table 5-5.

Table 5-5 SCSI Message Byte Values

Constant Name	Meaning
MSG_COMPL	Command complete.
MSG_XMSG	Extended message (only byte returned).
MSG_SAVEP	Initiator should save data pointers.
MSG_RESTP	Initiator restore data pointers.
MSG_DISC	Disconnect.
MSG_IERR	Initiator detected error.
MSG_ABORT	Abort.
MSG_REJECT	Optional message rejected, not supported.
MSG_NOOP	Empty message.
MSG_MPARITY	Parity error during Message In phase.
MSG_LINK	Linked command complete.
MSG_LINKF	Linked command complete with flag.

Table 5-5 (continued) SCSI Message Byte Values

Constant Name	Meaning
MSG_BRESET	Bus device reset.
MSG_IDENT	Value 0x80, first of the 0x80-0xFF identifier messages.

Testing the Driver Configuration

Different buses have different host adapter drivers that can have different features. The *dsreq* device driver supports an **ioctl()** call that retrieves the configuration of the driver for the bus where the device resides. This call fills in the fields of a structure of type *dsconf* (declared in *sys/dsreq.h*) listed in Table 5-6.

Table 5-6 Fields of the dsconf Structure

Field Name	Contents
dsc_flags	DSRQ flags honored by this driver (see Table 5-2 on page 83)
dsc_preset	DSRQ preset values (defaults) that are merged with the input *ds_flags* using logical OR in any request.
dsc_bus	Number of this SCSI bus, as encoded in the device minor number.
dsc_imax	Maximum target ID for this bus (7 for SCSI, 15 for wide SCSI).
dsc_lmax	Maximum number LUN values per ID on this bus.
dsc_iomax	Maximum length of a single I/O transfer.
dsc_biomax	Maximum length of a buffered I/O transfer.

The code in Example 5-1 shows a function that tests if a particular flag is supported by a particular bus. The input arguments are a file descriptor for an open device special file, and a flag value (or values) from *sys/dsreq.h*.

Example 5-1 Testing the Generic SCSI Configuration

```
uint
test_dsreq_flags(int dev_fd, uint flag)
{
   dsconf_t config;
   int ret;
   ret = ioctl(dev_fd, DS_CONF, &config);
   if (!ret) { /* no problem in ioctl */
      return (flag & config.dsc_flags);
   } else { /* ioctl failure */
      return 0; /* not supported, it seems */
   }
}
```

A program could use the function in Example 5-1 to find out if a particular feature is supported. For example, a test of support for the DSRQ_SYNXFER feature could be coded as follows:

```
if (test_dsreq_flags(the_dev, DSRQ_SYNXFER)) {
   /* synchronous negotiation is supported */...
```

Using the Special DS_RESET and DS_ABORT Calls

Two special functions of the generic SCSI driver are available only as **ioctl()** calls, not through dslib functions.

Using DS_ABORT

The DS_ABORT **ioctl()** sends a SCSI ABORT message to the bus, target, and LUN defined by the file descriptor. The resulting status is returned in the *dsreq* that is also specified. The host adapter driver waits until no commands are pending on that bus, so there is no point in using this function to cancel anything but an immediate command such as a rewind. And example of this call is as follows:

```
ioctl(dev_fd, DS_ABORT, &some_dsreq);
```

Using DS_RESET

The DS_RESET **ioctl()** function causes a reset of the SCSI bus specified by the file descriptor. The resulting status is returned in the *dsreq* that is also specified. This powerful operation should be used with great care, because it terminates all pending activity on the bus.

Using dslib Functions

The functions in the dslib library are built upon calls to the dsreq device driver, and simplify the process of allocating a dsreq structure, setting values in it, and executing commands. The formal documentation of the library is found in dslib(3). The source code is distributed with the system in the */usr/share/src/irix/examples/scsi* directory so that you can read and extend it. (This directory installs as part of the irix_dev software component, and the examples directory does not install by default.)

dslib Functions

In order to use the functions in the library, you include */usr/include/dslib.h* in your code, and link with the *-lds* option so as to link */usr/lib/libds.so*. Then the functions summarized in Table 5-7 are available.

Table 5-7 dslib Function Summary

Function Name	Purpose	Page
ds_ctostr	Look up a string in a table using an integer key.	page 95
ds_vtostr	Look up a string in a table using an integer key.	page 95
dsopen	Open a device special file and allocate a *dsreq* for use with it.	page 92
dsclose	Free the *dsreq* structure and close the device.	page 92
doscsireq	Perform an operation on a device as specified in a *dsreq*.	page 93
filldsreq	Set values in fields of a *dsreq* structure.	page 93
fillg0cmd	Set up the *dsreq* structure for a group 0 SCSI command.	page 94
fillg1cmd	Set up the *dsreq* structure for a group 1 SCSI command.	page 94

Table 5-7 (continued) dslib Function Summary

Function Name	Purpose	Page
inquiry12	Issue an Inquiry command and retrieve information from the device concerning such things as its type.	page 96
modeselect15	Issue a group 0 Mode Select command to a SCSI device.	page 96
modesense1a	Send a group 0 Mode Sense command to a device to retrieve a parameter page from the device.	page 97
read08	Issue a group 0 Read command in disk-drive form.	page 98
readextended28	Issue a group 1 Read command in disk-drive form.	page 98
readcapacity25	Issue a Read Capacity command.	page 99
requestsense03	Issue a Request Sense command and test or probe for the device.	page 100
reserveunit16	Issue a Reserve Unit command.	page 100
releaseunit17	Issue a Release Unit command.	page 100
senddiagnostic1d	Issue a Send Diagnostic command to test if the device or the SCSI bus is online, or run a self-test on the device.	page 101
testunitready00	Issue a Test Unit Ready command to the SCSI device.	page 102
write0a	Issue a group 0 Write command to the SCSI device.	page 102
writeextended2a	Issue an extended Write command to the SCSI device.	page 102

Using dsopen() and dsclose()

The **dsopen()** function opens a device special file for a generic SCSI device, and allocates a *dsreq* structure initialized for use with that device. The function prototype is

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

The arguments are

opath	The name of the device special file as a character string, for example "/dev/scsi/jag0d7l0" (see "Form of Names in /dev/scsi" on page 79).
oflags	The *oflag* value expected by **open()** when opening this device special file. O_EXCL has special meaning; see "Relationship to Other Device Special Files" on page 81.

If the **open()** call fails or memory cannot be allocated, the function returns NULL. Otherwise it allocates a *dsreq* structure as well as generous buffers for command and sense strings. The following fields of the *dsreq* are initialized:

ds_time	Set to 10000 (10 second timeout).
ds_private	Set to the address of the context that contains the *dsreq* as well as the command and sense buffers.
ds_cmdbuf	Set to the address of the command buffer.
ds_cmdlen	Set to the length of the allocated command buffer.
ds_sensebuf	Set to the address of the allocated sense buffer.
ds_senselen	Set to the length of the sense buffer.

Other fields of the *dsreq* are cleared to zero.

Note: Other functions in dslib assume that a *dsreq* has been initialized by **dsopen()**. In particular they assume the *ds_private* value points to a context block. You should not attempt to use any *dsreq* structure with a dslib function except one returned by **dsopen()**; and you should not use a *dsreq* opened for one file with another file.

The **dsclose()** function releases the *dsreq* structure and close the device. Its prototype is

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

The only argument is the *dsreq* created by **dsopen()**.

Issuing a Request With doscsireq()

The **doscsireq()** function issues a SCSI request by passing a *dsreq* to the SCSI device driver using an **ioctl()** call. The *dsreq* must have been prepared completely beforehand. The function prototype is

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

The arguments are as follows:

fd The file descriptor for the open device file.

dsp The address of the *dsreq* prepared by **dsopen()**.

Normally the returned value is the SCSI status byte. When the requested operation ends with Busy or Reserve Conflict status, the function sleeps 2 seconds and tries the operation up to four times. The returned value is -1 when the device driver rejects the ioctl() or the third retry ends in failure.

SCSI Utility Functions

The functions **filldsreq()**, **fillg0cmd()**, **fillg1cmd()**, **ds_vtostr()**, and **ds_ctostr()** are not oriented toward particular SCSI operations, but are used to construct your own task-oriented SCSI functions.

Using filldsreq()

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

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

The arguments are as follows:

dsp The address of a *dsreq* prepared by dsopen().

data The address of a buffer area.

datalen The length of the buffer area.

flags Flag values for *ds_flags* (see “Values for ds_flags” on page 83).

The bits in *flags* are added to *ds_flags* with an OR; they do not replace the contents of the field.

Note: Besides the specified values, the function also sets 10000 in *ds_timeout* and clears *ds_link*, *ds_synch*, and *ds_ret* to zero.

Using fillg0cmd() and fillg1cmd()

The **fillg0cmd()** function stores a group 0 (6-byte) SCSI command in a command buffer. The **fillg1cmd()** stores a group 1 (10-byte) SCSI command in the buffer. Both functions set the *ds_cmdbuf* and *ds_cmdlen* fields of a *dsreq*. The function prototypes are:

```
void fillg0cmd(struct dsreq *dsp, uchar_t *cmdbuf, b0, ..., b5)
void fillg1cmd(struct dsreq *dsp, uchar_t *cmdbuf, b0, ..., b9)
```

The arguments are as follows:

dsp — The address of any *dsreq*.

cmdbuf — The address of a buffer to receive the command string.

b0, *b1*,... — Expressions for the successive bytes of a SCSI command.

In typical use, the arguments are as follows:

dsp — The address of a *dsreq* initialized by **dsopen()**.

cmdbuf — The command buffer allocated by **dsopen()**, whose address is stored in the *ds_cmdbuf* field of the *dsreq*.

b0 — A SCSI command verb expressed as one of the constants declared in dslib.h, for example G0_INQU.

A typical call resembles the following:

```
fillg0cmd(dsp, (uchar_t *)CMDBUF(dsp), G0_INQU, 1, inq_page, 0,
B1(datalen),0);
```

The macros B1(), B2(), and B4() defined in *sys/dsreq.h* are useful for expressing halfword and word values as byte sequences.

Using ds_vtostr() and ds_ctostr()

The dslib library module contains six static tables that can be used to convert between numeric values and character strings for message display. The tables are summarized in Table 5-8. The table definitions are in the source file *dstab.c*.

Table 5-8 Lookup Tables in dslib

External Name	Type	Table Contents
cmdnametab	vtab	Names for SCSI command bytes, for example “Test Unit.”
cmdstatustab	vtab	Names for SCSI status byte codes, for example “BUSY.”
dsrqnametab	vtab	Descriptions of flag values from *ds_flags*, for example “select with (without) atn” for DSRQ_SELATN.
dsrtnametab	vtab	Descriptions of return values in ds_ret, for example “parity error on SCSI bus” for DSRT_PARITY.
msgnametab	vtab	Descriptions of SCSI message bytes, for example “Save Pointers.”
sensekeytab	ctab	Descriptions of SCSI sense byte values, for example “Illegal Request.”

The **ds_vtostr()** function searches any of the five vtab tables for the string matching an integer key. The **ds_ctostr()** function searches a ctab (currently, only sensekeytab is a ctab) for the string matching a key. The function prototypes are

```
char * ds_vtostr(unsigned long v, struct vtab *table);
char * ds_ctostr(unsigned long v, struct ctab *table);
```

Each function searches the specified table for a row containing the numeric value *v*, and returns address of the corresponding string. If there is no such row, the functions return the address of a zero-length string.

Using Command-Building Functions

The remaining functions in dslib each construct and execute a specific type of common SCSI command. Each function follows this general pattern:

1. Use **fillg0cmd()** or **fillg1cmd()** to set up the command string, based on the function's arguments.
2. Use **filldsreq()** to set up the remaining fields of the *dsreq* structure.
3. Execute the command using **doscsireq()**.
4. Return the value returned by **doscsireq()**.

You can construct similar, additional functions using the utility functions in this same way. In particular you are likely to need to construct your own function to issue Read commands.

inquiry12()—Issue an Inquiry Command

The **inquiry12()** function prepares and issues an Inquiry command to retrieve device-specific information. The function prototype is

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

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the inquiry response.
datalen	The length of the buffer, at least 36 and typically 64.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

modeselect15()—Issue a Group 0 Mode Select Command

The **modeselect15()** function prepares and issues a group 0 Mode Select command. This command is used to control a variety of standard and vendor-specific device parameters. Typically, **modesense1A()** is first used to retrieve the current parameters. The function prototype is

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

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a mode data page to send.
datalen	The length of the data.
save	The least significant bit sets the SP bit in the command.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

modesense1a()—Send a Group 0 Mode Sense Command

The **modesense1a()** function prepares and issues a group 0 Mode Sense command to a SCSI device to retrieve a page of device-dependent information. The function prototype:

```
int modesense1a(struct dsreq *dsp, caddr_t data, long datalen,
               int pagectrl, int pagecode, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the page of data.
datalen	The length of the buffer.
pagectrl	The least significant 2 bits are set as the PCF bits in the command.
pagecode	The least significant 6 bits are set as the page number.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

For reference, the PCF codes are as follows:

0	Current values.
1	Changeable values.
2	Default values.
3	Saved values.

For reference, some page numbers are as follows:

0	Vendor unique.
1	Read/write error recovery.
2	Disconnect/reconnect.
3	Direct access device format; parallel interface; measurement units.
4	Rigid disk geometry; serial interface.
5	Flexible disk; printer options.
6	Optical memory.
7	Verification error.
8	Caching.
9	Peripheral device.
63 (0x3f)	Return all pages supported.

read08() and readextended28()—Issue a Read Command

The **read08()** and **readextended28()** functions prepare and issue particular forms of SCSI Read commands. The Read and extended Read commands have so many variations that it is unlikely that either of these functions will work with your device. However, you can use them as models to build additional variations on Read. Do not preempt the function names.

The function prototypes are

```
int
read08(struct dsreq *dsp, caddr_t data, long datalen,
               long lba, int vu);
int
readextended28(struct dsreq *dsp, caddr_t data, long datalen,
               long lba, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the data.
datalen	The length of the buffer (not exceeding 255 for **read08**)
lba	The logical block address for the start of the read (not exceeding 16 bits for **read08**)
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

The functions set the transfer length in the command to the number of bytes given by datalen. This is often incorrect; many devices want a number of blocks of some size. Function **read08()** sets only 16 bits from lba as the logical block number, although the SCSI command format permits another 5 bits to be encoded in the command. For these and other reasons you are likely to need to create customized Read functions of your own.

readcapacity25()—Issue a Read Capacity Command

The **readcapacity25()** function prepares and issues a Read Capacity command to a SCSI device. The function prototype is

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

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the capacity data.
datalen	The length of the buffer, typically 8.
lba	Last block address, 0 unless *pmi* is nonzero.
pmi	The least-significant bit is used to set the partial medium indicator (PMI) bit of the command.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

When *pmi* is 0, *lba* should be given as 0 and the command returns the device capacity. When *pmi* is 1, the command returns the last block following block *lba* before which a delay (seek) will occur.

requestsense03()—Issue a Request Sense Command

The **requestsense03()** function prepares and issues a Request Sense command. If you include DSRQ_SENSE in the *flag* argument to **doscsireq()**, a Request Sense is sent automatically after an error in a command. The function prototype is

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

The arguments are:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the sense data.
datalen	The length of the buffer.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

reserveunit16() and releaseunit17()—Control Logical Units

The **reserveunit16()** function prepares and issues a Reserve Unit command to reserve a logical unit, causing it to return Reservation Conflict status to requests from other initiators. The **releaseunit17()** function prepares and issues a Release Unit command to release a reserved unit. The function prototypes are

```
int
reservunit16(struct dsreq *dsp, caddr_t data, long datalen,
            int tpr, int tpdid, int extent, int res_id, int vu);
int
releaseunit17(struct dsreq *dsp,
            int tpr, int tpdid, int extent, int res_id, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of data to send with the Reserve Unit. (This may be NULL for **reservunit16()** which does not normally transfer data.)
datalen	The length of the data (typically 0).
tpr	The least-significant bit is used to set the Third-Party Reservation bit in the command: 1 means the reservation is on behalf of another initiator.
tpdid	The device ID for the device to hold the reservation: 0 unless *tpr* is 1.
extent	The least-significant bit sets the least-significant bit of byte 1 of the command string.
res_id	Passed as byte 2 of the command string.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

senddiagnostic1d()—Issue a Send Diagnostic Command

The **senddiagnostic1d()** function prepares and issues a Send Diagnostic command. The function prototype is

```
int
senddiagnostic1d(struct dsreq *dsp, caddr_t data, long datalen,
                 int self, int dofl, int uofl, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a page or pages of diagnostic parameter data to be sent.
datalen	The length of the data (0 if none).
self	The least-significant bit sets the Self Test (ST) bit in the command: 1 means return status from the self-test; 0 means hold the results.
dofl	The least-significant bit sets the Device Offline bit of the command: 1 authorizes tests that can change the status of other logical units.

uofl — The least-significant bit sets the Unit Offline bit of the command: 1 authorizes tests that can change the status of the logical unit.

vu — The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

When *self* is 1, the status reflects the success of the self-test. You should either set the DSRQ_SENSE flag in the *dsreq* so that if the self-test fails, a Sense command will be issued, or be prepared to call **requestsense03()**. When *self* is 0, you can use a Read Diagnostic command to return detailed results of the test (however, dslib does not contain a predefined function for Read Diagnostic).

testunitready00—Issue a Test Unit Ready Command

The **testunitready00()** function prepares and issues a Test Unit Ready command to a SCSI device. The function prototype is

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

This function is reproduced here in Example 5-2 as an example of how other command-oriented functions can be created.

Example 5-2 Code of the testunitread00() Function

```
int
testunitready00(struct dsreq *dsp)
{
   fillg0cmd(dsp, CMDBUF(dsp), G0_TEST, 0, 0, 0, 0, 0);
   filldsreq(dsp, 0, 0, DSRQ_READ|DSRQ_SENSE);
   return(doscsireq(getfd(dsp), dsp));
}
```

write0a() and writeextended2a()—Issue a Write Command

The **write0a()** function prepares and issues a group 0 Write command. The **writeextended2a()** function prepares and issues an extended (10-byte) Write command. As with Read commands (see "read08() and readextended28()—Issue a Read Command" on page 98), Write commands have many device-specific features, and you will very likely have to create your own customized version of these functions.

The function prototypes are

```
int
write0a(struct dsreq *dsp, caddr_t data, long datalen,
        long lba, int vu);
int
writeextended2a(struct dsreq *dsp, caddr_t data, long datalen,
        long lba, int vu);
```

The arguments are as follows:

dsp The address of a *dsreq* structure prepared by **dsopen()**.

data The address of the data to be sent.

datalen The length of the data (at most 255 for **write0a**)

lba The logical block address (at most 16 bits for **write0a**)

vu The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

Example dslib Program

The program in Example 5-3 illustrates the use of the dslib functions. This is an edited version of a program that can be obtained in full from Dave Olson's home page, *http://reality.sgi.com/employees/olson/Olson/index.html*.

Example 5-3 Program That Uses dslib Functions

```
#ident "scsicontrol.c: $Revision $"

#include <sys/types.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <dslib.h>
```

```
typedef struct
{
    unchar  pqt:3;  /* peripheral qual type */
    unchar  pdt:5;  /* peripheral device type */
    unchar  rmb:1,  /* removable media bit */
        dtq:7;  /* device type qualifier */
    unchar  iso:2,  /* ISO version */
        ecma:3, /* ECMA version */
        ansi:3; /* ANSI version */
    unchar  aenc:1, /* async event notification supported */
        trmiop:1,   /* device supports 'terminate io process msg */
        res0:2, /* reserved */
        respfmt:3;  /* SCSI 1, CCS, SCSI 2 inq data format */
    unchar  ailen;  /* additional inquiry length */
    unchar  res1;   /* reserved */
    unchar  res2;   /* reserved */
    unchar  reladr:1,   /* supports relative addressing (linked cmds) */
        wide32:1,   /* supports 32 bit wide SCSI bus */
        wide16:1,   /* supports 16 bit wide SCSI bus */
        synch:1,    /* supports synch mode */
        link:1, /* supports linked commands */
        res3:1, /* reserved */
        cmdq:1, /* supports cmd queuing */
        softre:1;   /* supports soft reset */
    unchar  vid[8]; /* vendor ID */
    unchar  pid[16];    /* product ID */
    unchar  prl[4]; /* product revision level*/
    unchar  vendsp[20]; /* vendor specific; typically firmware info */
    unchar  res4[40];   /* reserved for scsi 3, etc. */
    /* more vendor specific information may follow */
} inqdata;

struct msel {
    unsigned char rsv, mtype, vendspec, blkdesclen; /* header */
    unsigned char dens, nblks[3], rsv1, bsize[3];   /* block desc */
    unsigned char pgnum, pglen; /* modesel page num and length */
    unsigned char data[240]; /* some drives get upset if no data requested
        on sense*/
};

#define hex(x) "0123456789ABCDEF" [ (x) & 0xF ]

/* only looks OK if nperline a multiple of 4, but that's OK.
 * value of space must be 0 <= space <= 3;
 */
```

```
void
hprint(unsigned char *s, int n, int nperline, int space)
{
    int   i, x, startl;

    for(startl=i=0;i<n;i++)  {
        x = s[i];
        printf("%c%c", hex(x>>4), hex(x));
        if(space)
            printf("%.*s", ((i%4)==3)+space, "    ");
        if ( i%nperline == (nperline - 1) ) {
            putchar('\t');
            while(startl < i) {
                if(isprint(s[startl]))
                    putchar(s[startl]);
                else
                    putchar('.');
                startl++;
            }
            putchar('\n');
        }
    }
    if(space && (i%nperline))
        putchar('\n');
}

/* aenc, trmiop, reladr, wbus*, synch, linkq, softre are only valid if
 * if respfmt has the value 2 (or possibly larger values for future
 * versions of the SCSI standard). */

static char pdt_types[][16] = {
    "Disk", "Tape", "Printer", "Processor", "WORM", "CD-ROM",
    "Scanner", "Optical", "Jukebox", "Comm", "Unknown"
};
#define NPDT (sizeof pdt_types / sizeof pdt_types[0])

void
printinq(struct dsreq *dsp, inqdata *inq, int allinq)
{

    if(DATASENT(dsp) < 1) {
        printf("No inquiry data returned\n");
        return;
    }
    printf("%-10s", pdt_types[(inq->pdt<NPDT) ? inq->pdt : NPDT-1]);
```

```
if (DATASENT(dsp) > 8)
    printf("%12.8s", inq->vid);
if (DATASENT(dsp) > 16)
    printf("%.16s", inq->pid);
if (DATASENT(dsp) > 32)
    printf("%.4s", inq->prl);
printf("\n");
if(DATASENT(dsp) > 1)
    printf("ANSI vers %d, ISO ver: %d, ECMA ver: %d; ",
        inq->ansi, inq->iso, inq->ecma);
if(DATASENT(dsp) > 2) {
    unchar special = *(inq->vid-1);
    if(inq->respfmt >= 2 || special) {
        if(inq->respfmt < 2)
            printf("\nResponse format type %d, but has "
              "SCSI-2 capability bits set\n", inq->respfmt);

        printf("supports: ");
        if(inq->aenc)
            printf(" AENC");
        if(inq->trmiop)
            printf(" termiop");
        if(inq->reladr)
            printf(" reladdr");
        if(inq->wide32)
            printf(" 32bit");
        if(inq->wide16)
            printf(" 16bit");
        if(inq->synch)
            printf(" synch");
        if(inq->synch)
            printf(" linkedcmds");
        if(inq->cmdq)
            printf(" cmdqueing");
        if(inq->softre)
            printf(" softreset");
    }
    if(inq->respfmt < 2) {
        if(special)
            printf(".  ");
        printf("inquiry format is %s",
            inq->respfmt ? "SCSI 1" : "CCS");
    }
}
putchar('\n');
```

```
    printf("Device is  ");
    /*  do test unit ready only if inquiry successful, since many
        devices, such as tapes, return inquiry info, even if
        not ready (i.e., no tape in a tape drive). */
    if(testunitready00(dsp) != 0)
        printf("%s\n",
            (RET(dsp)==DSRT_NOSEL) ? "not responding" : "not ready");
    else
        printf("ready");
    printf("\n");
}

/* inquiry cmd that does vital product data as spec'ed in SCSI2 */
int
vpinquiry12( struct dsreq *dsp, caddr_t data, long datalen, char vu,
  int page)
{
  fillg0cmd(dsp, (uchar_t *)CMDBUF(dsp), G0_INQU, 1, page, 0, B1(datalen),
    B1(vu<<6));
  filldsreq(dsp, (uchar_t *)data, datalen, DSRQ_READ|DSRQ_SENSE);
  return(doscsireq(getfd(dsp), dsp));
}

int
startunit1b(struct dsreq *dsp, int startstop, int vu)
{
  fillg0cmd(dsp,(uchar_t *)CMDBUF(dsp),0x1b,0,0,0,(uchar_t)startstop,B1(vu<<6));
  filldsreq(dsp, NULL, 0, DSRQ_READ|DSRQ_SENSE);
  dsp->ds_time = 1000 * 90; /* 90 seconds */
  return(doscsireq(getfd(dsp), dsp));
}


int
myinquiry12(struct dsreq *dsp, uchar_t *data, long datalen, int vu, int neg)
{
  fillg0cmd(dsp, (uchar_t *)CMDBUF(dsp), G0_INQU, 0,0,0, B1(datalen), B1(vu<<6));
  filldsreq(dsp, data, datalen, DSRQ_READ|DSRQ_SENSE|neg);
  dsp->ds_time = 1000 * 30; /* 90 seconds */
  return(doscsireq(getfd(dsp), dsp));
}

int
dsreset(struct dsreq *dsp)
{
```

```
  return ioctl(getfd(dsp), DS_RESET, dsp);
}

void
usage(char *prog)
{
    fprintf(stderr,
    "Usage: %s [-i (inquiry)] [-e (exclusive)] [-s (sync) | -a (async)]\n"
    "\t[-l (long inq)] [-v (vital proddata)] [-r (reset)] [-D (diagselftest)]\n"
    "\t[-H (halt/stop)] [-b blksize]\n"
    "\t[-g (get host flags)] [-d (debug)] [-q (quiet)] scsidevice [...]\n",
        prog);
    exit(1);
}

main(int argc, char **argv)
{
    struct dsreq *dsp;
    char *fn;
    /* int because they must be word aligned. */
    int errs = 0, c;
    int vital=0, doreset=0, exclusive=0, dosync=0;
    int dostart = 0, dostop = 0, dosenddiag = 0;
    int doinq = 0, printname = 1;
    unsigned bsize = 0;
    extern char *optarg;
    extern int optind, opterr;

    opterr = 0; /* handle errors ourselves. */
    while ((c = getopt(argc, argv, "b:HDSaserdvlgCiq")) != -1)
    switch(c) {
    case 'i':
        doinq = 1;  /* do inquiry */
        break;
    case 'D':
        dosenddiag = 1;
        break;
    case 'r':
        doreset = 1;    /* do a scsi bus reset */
        break;
    case 'e':
        exclusive = O_EXCL;
        break;
    case 'd':
        dsdebug++;  /* enable debug info */
```

```
        break;
    case 'q':
        printname = 0;  /* print devicename only if error */
        break;
    case 'v':
        vital = 1;  /* set evpd bit for scsi 2 vital product data */
        break;
    case 'H':
        dostop = 1; /* send a stop (Halt) command */
        break;
    case 'S':
        dostart = 1;    /* send a startunit/spinup command */
        break;
    case 's':
        dosync = DSRQ_SYNXFR;   /* attempt to negotiate sync scsi */
        break;
    case 'a':
        dosync = DSRQ_ASYNXFR;  /* attempt to negotiate async scsi */
        break;
    default:
        usage(argv[0]);
    }

    if(optind >= argc || optind == 1)   /* need at 1 arg and one option */
        usage(argv[0]);

    while (optind < argc) { /* loop over each filename */
        fn = argv[optind++];
        if(printname) printf("%s:  ", fn);
        if((dsp = dsopen(fn, O_RDONLY|exclusive)) == NULL) {
            /* if open fails, try pre-pending /dev/scsi */
            char buf[256];
            strcpy(buf, "/dev/scsi/");
            if((strlen(buf) + strlen(fn)) < sizeof(buf)) {
                strcat(buf, fn);
                dsp = dsopen(buf, O_RDONLY|exclusive);
            }
            if(!dsp) {
                if(!printname) printf("%s:  ", fn);
                fflush(stdout);
                perror("cannot open");
                errs++;
                continue;
            }
        }
```

```
/* try to order for reasonableness; reset first in case
 * hung, then inquiry, etc. */

if(doreset) {
    if(dsreset(dsp) != 0) {
        if(!printname) printf("%s:  ", fn);
        printf("reset failed: %s\n", strerror(errno));
        errs++;
    }
}

if(doinq) {
    int inqbuf[sizeof(inqdata)/sizeof(int)];
    if(myinquiry12(dsp, (uchar_t *)inqbuf, sizeof inqbuf, 0, dosync)) {
        if(!printname) printf("%s:  ", fn);
        printf("inquiry failure\n");
        errs++;
    }
    else
        printinq(dsp, (inqdata *)inqbuf, 0);
}

if(vital) {
    unsigned char *vpinq;
    int vpinqbuf[sizeof(inqdata)/sizeof(int)];
    int vpinqbuf0[sizeof(inqdata)/sizeof(int)];
    int i, serial = 0, asciidef = 0;
    if(vpinquiry12(dsp, (char *)vpinqbuf0,
        sizeof(vpinqbuf)-1, 0, 0)) {
        if(!printname) printf("%s:  ", fn);
        printf("inquiry (vital data) failure\n");
        errs++;
        continue;
    }
    if(DATASENT(dsp) <4) {
        printf("vital data inquiry OK, but says no"
            "pages supported (page 0)\n");
        continue;
    }
    vpinq = (unsigned char *)vpinqbuf0;
    printf("Supported vital product pages: ");
    for(i = vpinq[3]+3; i>3; i--) {
        if(vpinq[i] == 0x80)
            serial = 1;
        if(vpinq[i] == 0x82)
```

```
                    asciidef = 1;
                printf("%2x  ", vpinq[i]);
            }
            printf("\n");
            vpinq = (unsigned char *)vpinqbuf;
            if(serial) {
                if(vpinquiry12(dsp, (char *)vpinqbuf,
                    sizeof(vpinqbuf)-1, 0, 0x80) != 0) {
                    if(!printname) printf("%s:  ", fn);
                    printf("inquiry (serial #) failure\n");
                    errs++;
                }
                else if(DATASENT(dsp)>3) {
                    printf("Serial #: ");
                    fflush(stdout);
                    /* use write, because there may well be
                     *nulls; don't bother to strip them out */
                    write(1, vpinq+4, vpinq[3]);
                    printf("\n");
                }
            }
            if(asciidef) {
                if(vpinquiry12(dsp, (char *)vpinqbuf,
                sizeof(vpinqbuf)-1, 0, 0x82) != 0) {
                if(!printname) printf("%s:  ", fn);
                printf("inquiry (ascii definition) failure\n");
                errs++;
                }
                else if(DATASENT(dsp)>3) {
                printf("Ascii definition: ");
                fflush(stdout);
                /* use write, because there may well be
                 *nulls; don't bother to strip them out */
                write(1, vpinq+4, vpinq[3]);
                printf("\n");
                }
            }
        }

        if(dostop && startunit1b(dsp, 0, 0)) {
            if(!printname) printf("%s:  ", fn);
            printf("stopunit fails\n");
            errs++;
        }
```

```
        if(dostart && startunit1b(dsp, 1, 0)) {
            if(!printname) printf("%s:  ", fn);
            printf("startunit fails\n");
            errs++;
        }

        if(dosenddiag && senddiagnostic1d(dsp, NULL, 0, 1, 0, 0, 0)) {
            if(!printname) printf("%s:  ", fn);
            printf("self test fails\n");
            errs++;
        }
dsclose(dsp);
    }
    return(errs);
}
```