The Linux SCSI Generic (sg) HOWTO

Douglas Gilbert

dgilbert@interlog.com

2002−01−26

Revision History

Revision 1.1	2002−01−26	Revised by: dpg
corrections, host_status, odd dxfer_len		
Revision 1.0	2001−12−21	Revised by: dpg
original, displace SCSI−PROGRAMMING−HOWTO		

This HOWTO describes the SCSI Generic driver (sg) found in the Linux 2.4 production series of kernels. It focuses on the the interface and characteristics of the driver that application writers may need to know. The driver's theory of operations is covered and some brief examples are included.

Table of Contents

Table of Contents

Chapter 1. Introduction

This document outlines the Linux SCSI Generic (sg) driver interface as found in the 2.4 series kernels. The driver's purpose is to allow SCSI commands to be sent directly to SCSI devices. The responses of those commands can then be obtained. This type of driver is sometimes termed as a "pass through". In the case of SCSI disks, the block subsystem which is normally used to mount and access a disk, is bypassed permitting low level operations such as formatting to be performed. Various specialized applications for writing CD−Rs and document scanning use the sg driver.

Many devices that use other physical buses (e.g. ATAPI cdroms, USB mass storage devices and IEEE 1394 sbp2 devices) utilize the SCSI command set. By using Linux pseudo SCSI device drivers which bridge between the native protocol stack and the SCSI subsystem, the upper level SCSI device drivers, including sg, can be used to control "non−SCSI" devices.

This is the third major version of the sg driver. A summary of the sg driver history is as follows:

- sg version 1 (original) from 1992 to early 1999 (lk 2.2.5) . A copy of the original HOWTO (in plain text) is at `www.torque.net/sg/p/original/SCSI-Programming-HOWTO.txt`
- sg version 2 from lk 2.2.6 in the 2.2 series. Its documentation is available in abridged form [`www.torque.net/sg/p/scsi-generic.txt`] and a longer form [`www.torque.net/sg/p/scsi-generic_long.txt`].
- sg version 3 in the linux kernel 2.4 series.

This document can be found at the Linux Documentation Project's site at `www.linuxdoc.org/HOWTO/SCSI-Generic-HOWTO/` . It is available in plain text and pdf renderings at that site. A (possibly later) version of this document can be found at `www.torque.net/sg/p/sg_v3_ho.html`. That is a single html page; drop the ".html" extension for multi−page html. There are also postscript, pdf and rtf renderings from the original SGML (docbook) file at the same location.

A more general description of the Linux SCSI subsystem of which sg is a part can be found in the `SCSI-2.4-HOWTO`.

This document was last modified on 26th January 2002.

Chapter 2. What the sg driver does

The sg driver permits user applications to send SCSI commands to devices that understand them. SCSI commands are 6, 10, 12 or 16 bytes long [1]. The SCSI disk driver (sd), once device initialization is complete, only sends SCSI READ and WRITE commands. There a several other interesting things one might want to do, for example, perform a low level format or turn on write caching.

Associated with some SCSI commands there is data to be written to the device. A SCSI WRITE command is one obvious example. When instructed, the sg driver arranges for data to be transferred to the device along with the SCSI command. It is possible that the lower level driver (often known as the "Host Bus Adapter" [HBA] or simply "adapter" driver) is unable to send the command to the device. An example of this occurs when the device does not respond in which case a 'host_status' or 'driver−status' error will be conveyed back to the user application.

All going well the SCSI command (and optionally some data) are conveyed to the device. The device will respond with a single byte value called the 'scsi_status'. GOOD is the scsi status indicating everything has gone well. The most common other status is CHECK CONDITION. In this latter case, the SCSI mid level issues a REQUEST SENSE SCSI command The response of the REQUEST SENSE is 18 bytes or more in length and is called the "sense buffer". It will indicate why the original command may not have been executed. It is important to realize that a CHECK CONDITION may vary in severity from informative (e.g. command needed to be retried before succeeding) to fatal (e.g. "medium error" which often indicates it is time to replace the disk).

So in all cases a user application should check the various status values. If necessary the "sense buffer" will be copied back to the user application. SCSI commands like READ convey data back to the user application (if they succeed). The sg driver arranges for this data transfer from the device to the user space, if necessary.

The description so far has concentrated on a disk device, but in reality the sg driver is not needed very often for disks because there already is a purpose built device driver for that: sd. The same is true of reading audio and data CDs (sr [scd]) and tapes (st). However scanners that understand the SCSI command set and CDR "burning" programs tend to use the sg driver. Other applications include tape "robots" and music CD "ripping".

To find out more about SCSI (draft) standards and resources visit `www.t10.org`. To use the sg device driver you should be familiar with the SCSI commands supported by the device that you wish to control. Getting hold of such information for devices like scanners can be quite challenging (if the vendor does not provide it).

The first SCSI command sent to a SCSI device when it is initialized is an INQUIRY. All SCSI devices should respond promptly to an INQUIRY supplying information such as the vendor, product designation and revision. Appendix C shows the sg driver being used to send an INQUIRY and print out some of the information in the response.

Chapter 3. Identifying the version of the SG driver

Earlier versions of the sg device driver either have no version number (e.g. the original driver) or a version number starting with "2". The drivers that support this new interface have a major version number of "3". The sg version numbers are of the form "x.y.z" and the single number given by the SG_GET_VERSION_NUM ioctl() is calculated by (x * 10000 + y * 100 + z). The sg driver discussed here will yield a number greater than or equal to 30000 from SG_GET_VERSION_NUM. The version number can also be seen using **cat /proc/scsi/sg/version** in the new driver. This document describes sg version 3.1.22 for the lk 2.4 series. Where some facility has been added during the lk 2.4 series (e.g. mmap−ed IO) and hence is not available in all versions of the lk 2.4 series, this is noted. [2]

Here is a list of sg versions that have appeared to date during the lk 2.4 series.

- lk 2.4.0 : sg version 3.1.17
- lk 2.4.7 : sg version 3.1.19 [see `include/scsi/sg.h` in that or a later version for the changelog]
- lk 2.4.10 : sg version 3.1.20 [This version had several changes put into it by third parties over the next 6 release kernel versions.]
- lk 2.4.17 : sg version 3.1.22

Chapter 4. Interface

This driver supports the following system calls, most of which are typical for a character device driver in Linux. They are:

- open()
- close()
- write()
- read()
- ioctl()
- poll()
- fcntl(sg_fd, F_SETFL, oflags | FASYNC)
- mmap()

The interface to these calls as seem from Linux applications is well documented in the "man" pages (in section 2).

A user application accesses the sg driver by using the open() system call on sg device file name. Each sg device file name corresponds to one (potentially) attached SCSI device. These are usually found in the `/dev` directory. Here are some sg device file names:

```
$ ls -l /dev/sg[01]
crw-rw----    1 root     disk      21,   0 Aug 30 16:30 /dev/sg0
crw-rw----    1 root     disk      21,   1 Aug 30 16:30 /dev/sg1
```

The leading "c" at the front of the permissions indicates a character device. The absence of read or write permissions for "others" is prudent security. The major number of all sg device names is 21 while the minor number is the same as the number following "sg" in the device file name. When the device file system (devfs) is active on a system then the primarily sg device file names are found at the bottom of an informative subtree:

```
$ cd /dev/scsi/host1/bus0/target0/lun0
$ ls -l generic
crw-r-----    1 root     root      21,   1 Dec 31  1969 generic
```

Under devfs (when its daemon [devfsd] is running) there would usually be a symbolic link from `/dev/sg1` to `/dev/scsi/host1/bus0/target0/lun0/generic`. This is so existing applications looking for the abridged device file name will not be surprised. One advantage of devfs is that only attached SCSI devices appear in the `/dev/scsi` subtree.

A significant addition in sg v3 is an ioctl() called SG_IO which is functionally equivalent to a write() followed by a blocking read(). In certain contexts the write()/read() combination have advantages over SG_IO (e.g. command queuing) and continue to be supported.

The existing (and original) sg interface based on the sg_header structure is still available using a write()/read() sequence as before. The SG_IO ioctl will only accept the new interface based on the sg_io_hdr_t structure.

The sg v3 driver thus has a write() call that can accept either the older sg_header structure or the new sg_io_hdr_t structure. The write() calls decides which interface is being used based on the second integer position of the passed header (i.e. sg_header::reply_len or sg_io_hdr_t::dxfer_direction). If it is a positive number then the old interface is assumed. If it is a negative number then the new interface is assumed. The direction constants placed in 'dxfer_direction' in the new interface have been chosen to have negative values.

If a request is sent to a write() with the sg_io_hdr_t interface then the corresponding read() that fetches the response must also use the sg_io_hdr_t interface. The same rule applies to the sg_header interface.

This document concentrates on the sg_io_hdr_t interface introduced in the sg version 3 driver. For the definition of the older sg_header interface see the sg version 2 documentation. A brief description is given in Appendix B.

Chapter 5. Theory of operation

The path of a request through the sg driver can be broken into 3 distinct stages:

1. The request is received from the user, resources are reserved as required (e.g. kernel buffer for indirect IO). If necessary, data in the user space is transferred into kernel buffers. Then the request is submitted to the SCSI mid level (and then onto the adapter) for execution. The SCSI mid level maintains a queue so the request may have to wait. If a SCSI device supports command queuing then it may be able to accommodate multiple outstanding requests.
2. Assuming the SCSI adapter supports interrupts, then an interrupt is received when the request is completed. When this interrupt arrives the data transfer is complete. This means that if the SCSI command was a READ then the data is in kernel buffers (indirect IO) or in user buffers (direct or mmap−ed IO). The sg driver is informed of this interrupt via a kernel mechanism called a "bottom half" handler. Some kernel resources are freed up.
3. The user makes a call to fetch the result of the request. If necessary, data in kernel buffers is transferred to the user space. If necessary, the sense buffer is written out to the user space. The remaining kernel resources associated with this request are freed up.

The write() call performs stage 1 while the read() call performs stage 3. If the read() call is made before stage 2 is complete then it will either wait or yield EAGAIN (depending on whether the file descriptor is blocking or not). If asynchronous notification is being used then stage 2 will send a SIGPOLL signal to the user process. The poll() system call will show this file descriptor is now readable (unless it was sent by the SG_IO ioctl()).

The SG_IO ioctl() performs stage 1, waits for stage 2 and then performs stage 3. If the file descriptor in question is set O_NONBLOCK then SG_IO will ignore this and still block! Also a SG_IO call will not effect the poll() state nor cause a SIGPOLL signal to be sent. If you really want non−blocking operation (e.g. for command queuing) then don't use SG_IO; use the write() read() sequence instead.

For more information about normal (or indirect), direct and mmap−ed IO see Chapter 9 .

Currently the sg driver uses one Linux major device number (char 21) which in the lk 2.4 series limits it to handling 256 SCSI devices. Any attempt to attach more than this number will rejected with a message being sent to the console and the log file. [3]

Chapter 6. The sg_io_hdr_t structure in detail

The main control structure for the version 3 SCSI generic driver has a struct tag name of "sg_io_hdr" and a typedef name of "sg_io_hdr_t". The structure is shown in abridged form below. The "[i]" notation indicates an input value while "[o]" indicates a value that is output. The "[i−>o]" indicates a value that is conveyed from input to output and apart from one special case, is not used by the driver. The "[i−>o]" members are meant to aid an application matching the request sent to a write() to the corresponding response received by a read(). For pointers the "[*i]" indicates a pointer that is used for reading from user memory into the driver, "[*o]" is a pointer used for writing, and "[*io]" indicates a pointer used for either reading or writing.

```
typedef struct sg_io_hdr
{
    int interface_id;           /* [i] 'S' (required) */
    int dxfer_direction;        /* [i] */
    unsigned char cmd_len;      /* [i] */
    unsigned char mx_sb_len;    /* [i] */
    unsigned short iovec_count; /* [i] */
    unsigned int dxfer_len;     /* [i] */
    void * dxferp;              /* [i], [*io] */
    unsigned char * cmdp;       /* [i], [*i]  */
    unsigned char * sbp;        /* [i], [*o]  */
    unsigned int timeout;       /* [i] unit: millisecs */
    unsigned int flags;         /* [i] */
    int pack_id;                /* [i->o] */
    void * usr_ptr;             /* [i->o] */
    unsigned char status;       /* [o] */
    unsigned char masked_status;/* [o] */
    unsigned char msg_status;   /* [o] */
    unsigned char sb_len_wr;    /* [o] */
    unsigned short host_status; /* [o] */
    unsigned short driver_status;/* [o] */
    int resid;                  /* [o] */
    unsigned int duration;      /* [o] */
    unsigned int info;          /* [o] */
} sg_io_hdr_t;  /* 64 bytes long (on i386) */
```

6.1. interface_id

This must be set to 'S' (capital ess). If not, the ENOSYS error message is placed in errno. The idea is to allow interface variants in the future that identify themselves with a different value. [The parallel port generic driver (pg) uses the letter 'P' to identify itself.] The type of interface_id is int.

6.2. dxfer_direction

The type of dxfer_direction is int. This is required to be one of the following:

- SG_DXFER_NONE /* e.g. a SCSI Test Unit Ready command */
- SG_DXFER_TO_DEV /* e.g. a SCSI WRITE command */
- SG_DXFER_FROM_DEV /* e.g. a SCSI READ command */
- SG_DXFER_TO_FROM_DEV
- SG_DXFER_UNKNOWN

The value SG_DXFER_NONE should be used when there is no data transfer associated with a command (e.g. TEST UNIT READY). The value SG_DXFER_TO_DEV should be used when data is being moved from user memory towards the device (e.g. WRITE). The value SG_DXFER_FROM_DEV should be used when data is being moved from the device towards user memory (e.g. READ).

The value SG_DXFER_TO_FROM_DEV is only relevant to indirect IO (otherwise it is treated like SG_DXFER_FROM_DEV). Data is moved from the user space to the kernel buffers. The command is then performed and most likely a READ−like command transfers data from the device into the kernel buffers. Finally the kernel buffers are copied back into the user space. This technique allows application writers to initialize the buffer and perhaps deduce the number of bytes actually read from the device (i.e. detect underrun). This is better done by using 'resid' if it is supported.

The value SG_DXFER_UNKNOWN is for those (rare) situations where the data direction is not known. It may be useful for backward compatibility of existing applications when the relevant direction information is not available in the sg interface layer. There is a (minor) performance "hit" associated with choosing this option (e.g. on the PCI bus). Some recent pseudo device drivers (e.g. USB mass storage) may have problems handling this value (especially on vendor−specific SCSI commands).

N.B. 'dxfer_direction' must have one of the five indicated values and cannot be uninitialized or zero.

If 'dxfer_len' is zero then all values are treated like SG_DXFER_NONE.

6.3. cmd_len

This is the length in bytes of the SCSI command that 'cmdp' points to. As a SCSI command is expected an EMSGSIZE error number is produced if the value is less than 6 or greater than 16. Further, if the SCSI mid level has a further limit then EMSGSIZE is produced in this case as well. [4] The type of cmd_len is unsigned char.

6.4. mx_sb_len

This is the maximum size that can be written back to the 'sbp' pointer when a sense_buffer is output which is usually in an error situation. The actual number written out is given by 'sb_len_wr'. In all cases 'sb_len_wr' <= 'mx_sb_len' . The type of mx_sb_len is unsigned char.

6.5. iovec_count

This is the number of scatter gather elements in an array pointed to by 'dxferp'. If the value is zero then scatter gather (in the user space) is _not_ being used and 'dxferp' points to the data transfer buffer. If the value is greater than zero then each element of the array is assumed to be of the form:

```
            typedef struct sg_iovec
            {
                void * iov_base; /* starting address */
                size_t iov_len;  /* length in bytes */
            } sg_iovec_t;
```

Note that this structure has been named and defined in such a way to parallel "struct iovec" used by the readv() and writev() system calls in Linux. See "man 2 readv".

Note that the scatter gather capability offered by 'iovec_count' is unrelated to the scatter gather capability (often associated with DMA) offered by most modern SCSI adapters. Furthermore iovec_count's variety of scatter gather (into the user space) is only available when normal (or "indirect") IO is being used. Hence when the SG_FLAG_DIRECT_IO or SG_FLAG_MMAP_IO are set in 'flags' then 'iovec_count' should be zero.

The type of iovec_count is unsigned short.

6.6. dxfer_len

This is the number of bytes to be moved in the data transfer associated with the command. The direction of the transfer is indicated by 'dxfer_direction'. If 'dxfer_len' is zero then no data transfer takes place. [5]

If iovec_count is non−zero then 'dxfer_len' should be equal to the sum of iov_len lengths. If not, the minimum of the two is the transfer length. The type of dxfer_len is unsigned int.

6.7. dxferp

If 'iovec_count' is zero then this value is a pointer to user memory of at least 'dxfer_len' bytes in length. If there is a data transfer associated with the command then the data will be transferred to or from this user memory. If 'iovec_count' is greater than zero then this value points to a scatter−gather array in user memory. Each element of this array should be an object of type sg_iovec_t. Note that data is sometimes written to user memory (e.g. from a failed SCSI READ) even when an error has occurred.

If mmap−ed IO is selected then the value in 'dxferp' is ignored and any data transfers will be to and from the address returned by the prior mmap() call.

The type of dxferp is void * .

6.8. cmdp

This value points to the SCSI command to be executed. The command is assumed to be 'cmd_len' bytes long. If cmdp is NULL then the system call yields an EMSGSIZE error number. The user memory pointed to is only read (not written to). The type of cmdp is unsigned char * .

6.9. sbp

This value points to user memory of at least 'mx_sb_len' bytes length where the SCSI sense buffer will be output. Most successful commands do not output a sense buffer and this will be indicated by 'sb_len_wr' being zero. Note that there are error conditions that don't result in a sense buffer be generated. The sense buffer results from the "auto−sense" mechanism in the SCSI mid−level driver. This mechanism detects a

CHECK_CONDITION status and issues a REQUEST SENSE command and conveys its response back as the "sense buffer". The type of sbp is unsigned char * .

6.10. timeout

This value is used to timeout the given command. The units of this value are milliseconds. The time being measured is from when a command is sent until when sg is informed the request has been completed. A following read() can take as long as the user likes. Timeouts are best avoided, especially if SCSI bus resets will adversely effect other devices on that SCSI bus. When the timeout expires, the SCSI mid level attempts error recovery. Error recovery completes when the first action in the following list is successful. Note that a more extreme measure is being taken at each step.

- the SCSI command that has timed out is aborted [6]
- a SCSI device reset is attempted
- a SCSI bus reset is attempted. Note this may have an adverse effect on other devices sharing that SCSI bus.
- a SCSI host (bus adapter) reset is attempted. This is an attempt to re−initialize the adapter card associated with the SCSI device that has the timed out command.

If all these fail then the device may be set "offline" which means that it is no longer accessible (except by this driver when open()−ed O_NONBLOCK) until the machine is rebooted. Offline devices still appear in the **cat /proc/scsi/scsi** listing. The last column of the **cat /proc/scsi/sg/devices** listing shows the online/offline status of a device ("1" means online while "0" is offline). The exact status returned depends on which level of error recovery succeeded. Most likely the 'host_status' will be set to DID_ABORT or DID_RESET.

The two error statuses containing the word "TIME(_)OUT" are typically _not_ related to a command timing out. DID_TIME_OUT in the 'host_status' usually means an (unexpected) device selection timeout. DRIVER_TIMEOUT in the 'driver_status' byte means the SCSI adapter is unable to control the devices on its SCSI bus (and has given up).

The type of timeout is unsigned int (and it represents milliseconds).

6.11. flags

These are single or multi−bit values that can be "or−ed" together:

- **SG_FLAG_DIRECT_IO** This is a request for direct IO on the data transfer. If it cannot be performed then the driver automatically performs indirect IO instead. If it is important to find out which type of IO was performed then check the values from the SG_INFO_DIRECT_IO_MASK in 'info' when the request packet is completed (i.e. after read() or ioctl(,SG_IO,)). The default action is to do indirect IO.
- **SG_FLAG_LUN_INHIBIT** The default action of the sg driver to overwrite internally the top 3 bits of the second SCSI command byte with the LUN associated with the file descriptor's device. To inhibit this action set this flag. For SCSI 3 (or later) devices, this internal LUN overwrite does not occur.
- **SG_FLAG_MMAP_IO** When set the driver will attempt to procure the reserved buffer. If the reserved buffer is occupied (EBUSY) or too small (ENOMEM) then the operation (write() or ioctl(SG_IO)) fails. No data transfers occur between the dxferp pointer and the reserved buffer

(dxferp is ignored). In order for a user application to access mmap−ed IO, it must have successfully executed an appropriate mmap() system call on this sg file descriptor. This precondition is not checked by write() or ioctl(SG_IO) when this flag is set. Setting this flag and SG_FLAG_DIRECT_IO results in a EINVAL error.
- **SG_FLAG_NO_DXFER** When set user space data transfers to or from the kernel buffers do not take place. This only has effect during indirect IO. This flag is for testing bus speed (e.g. the "sg_rbuf" utility uses it).

The type of flags is unsigned int.

6.12. pack_id

This value is not normally acted upon by the sg driver. It is provided so the user can identify the request. This is useful when command queuing is being used. The "abnormal" case is when SG_SET_FORCE_PACK_ID is set and a 'pack_id' other than −1 is given to read(). In this case the read() will wait to fetch a request that matches this 'pack_id'. If this mode is used be careful to set 'dxfer_direction' to a valid value (actually any of the SG_DXFER_* values will do) on input to the read(), together with the wanted pack_id. The type of pack_id is int.

6.13. usr_ptr

This value is not acted upon by the sg driver. It is meant to allow the user to associate some object with this request (e.g. to maintain state information). The type of usr_ptr is void * .

6.14. status

This is the SCSI status byte as defined by the SCSI standard. Note that it can have vendor information set in bits 0, 6 and 7 (although this is uncommon). Further note that this 'status' data does _not_ match the definitions in <scsi/scsi.h> (e.g. CHECK_CONDITION). The following 'masked_status' does match those definitions. The type of status is unsigned char .

6.15. masked_status

Logically: masked_status == ((status & 0x3e) >> 1) . So 'masked_status' strips the vendor information bits off 'status' and then shifts it right one position. This makes it easier to do things like "if (CHECK_CONDITION == masked_status) ..." using the definitions in <scsi/scsi.h>. The defined values in this file are:

- GOOD [0x00]
- CHECK_CONDITION [0x01]
- CONDITION_GOOD [0x02]
- BUSY 0x04
- INTERMEDIATE_GOOD 0x08
- INTERMEDIATE_C_GOOD 0x0a
- RESERVATION_CONFLICT 0x0c

- COMMAND_TERMINATED 0x11
- QUEUE_FULL 0x14

N.B. 1 bit offset from usual SCSI status values

Note that SCSI 3 defines some additional status codes. The type of masked_status is unsigned char .

6.16. msg_status

The messaging level in SCSI is under the command level and knowledge of what is happening at the messaging level is very rarely needed. Furthermore most modern chip−sets used in SCSI adapters completely hide this value. Nearly all adapters will return zero in 'msg_status' all the time. The type of msg_status is unsigned char .

6.17. sb_len_wr

This is the actual number of bytes written to the user memory pointed to by 'sbp'. 'sb_len_wr' is always <= 'mx_sb_len'. Linux 2.2 series kernels (and earlier) truncate this value to a maximum of 16 bytes. The actual number of bytes written will not exceed the length indicated by "Additional Sense Length" field (byte 7) of the Request Sense response. The type of sb_len_wr is unsigned char .

6.18. host_status

These codes potentially come from the firmware on a host adapter or from one of several hosts that an adapter driver controls. The 'host_status' field has the following values whose #defines mimic those which are only visible within the kernel (with the "SG_ERR_" removed from the front of each define). A copy of these defines can be found in sg_err.h (see Appendix A):

- SG_ERR_DID_OK [0x00] NO error
- SG_ERR_DID_NO_CONNECT [0x01] Couldn't connect before timeout period
- SG_ERR_DID_BUS_BUSY [0x02] BUS stayed busy through time out period
- SG_ERR_DID_TIME_OUT [0x03] TIMED OUT for other reason (often this an unexpected device selection timeout)
- SG_ERR_DID_BAD_TARGET [0x04] BAD target, device not responding?
- SG_ERR_DID_ABORT [0x05] Told to abort for some other reason. From lk 2.4.15 the SCSI subsystem supports 16 byte commands however few adapter drivers do. Those HBA drivers that don't support 16 byte commands will yield this error code if a 16 byte command is passed to a SCSI device they control.
- SG_ERR_DID_PARITY [0x06] Parity error. Older SCSI parallel buses have a parity bit for error detection. This probably indicates a cable or termination problem.
- SG_ERR_DID_ERROR [0x07] Internal error detected in the host adapter. This may not be fatal (and the command may have succeeded). The aic7xxx and sym53c8xx adapter drivers sometimes report this for data underruns or overruns. [7]
- SG_ERR_DID_RESET [0x08] The SCSI bus (or this device) has been reset. Any SCSI device on a SCSI bus is capable of instigating a reset.
- SG_ERR_DID_BAD_INTR [0x09] Got an interrupt we weren't expecting
- SG_ERR_DID_PASSTHROUGH [0x0a] Force command past mid−layer

* SG_ERR_DID_SOFT_ERROR [0x0b] The low level driver wants a retry

The type of host_status is unsigned short .

6.19. driver_status

One driver can potentially control several host adapters. For example Advansys provide one Linux adapter driver that controls all adapters made by that company − if 2 of more Advansys adapters are in 1 machine, then 1 driver controls both. When ('driver_status' & SG_ERR_DRIVER_SENSE) is true the 'sense_buffer' is also output. The 'driver_status' field has the following values whose #defines mimic those which are only visible within the kernel (with the "SG_ERR_" removed from the front of each define). A copy of these defines can be found in sg_err.h (see the utilities section):

* SG_ERR_DRIVER_OK [0x00] Typically no suggestion
* SG_ERR_DRIVER_BUSY [0x01]
* SG_ERR_DRIVER_SOFT [0x02]
* SG_ERR_DRIVER_MEDIA [0x03]
* SG_ERR_DRIVER_ERROR [0x04]
* SG_ERR_DRIVER_INVALID [0x05]
* SG_ERR_DRIVER_TIMEOUT [0x06] Adapter driver is unable to control the SCSI bus to its is setting its devices offline (and giving up)
* SG_ERR_DRIVER_HARD [0x07]
* SG_ERR_DRIVER_SENSE [0x08] Implies sense_buffer output
* above status 'or'ed with one of the following suggestions
* SG_ERR_SUGGEST_RETRY [0x10]
* SG_ERR_SUGGEST_ABORT [0x20]
* SG_ERR_SUGGEST_REMAP [0x30]
* SG_ERR_SUGGEST_DIE [0x40]
* SG_ERR_SUGGEST_SENSE [0x80]

The type of driver_status is unsigned short .

6.20. resid

This is the residual count from the data transfer. It is 'dxfer_len' less the number of bytes actually transferred. In practice it only reports underruns (i.e. positive number) as data overruns should never happen. This value will be zero if there was no underrun or the SCSI adapter doesn't support this feature. [8] The type of resid is int .

6.21. duration

This value will be the number of milliseconds from when a SCSI command was sent until sg is informed that it is complete. For i386 machines the granularity is 10ms while on alpha machines it is 1ms. This value is rounded toward zero. The type of duration is unsigned int .

6.22. info

This value is designed to convey useful information back to the user about the associated request. This information does not necessarily indicate an error. Several single bit and multi−bit fields are "or−ed" together to make this value.

A single bit component contained in SG_INFO_OK_MASK indicates whether some error or status field is non−zero. If either 'masked_status', 'host_status' or 'driver_status' are non−zero then SG_INFO_CHECK is set. The associated values are:

- SG_INFO_OK_MASK [0x1]
- SG_INFO_OK [0x0] no sense, host nor driver "noise"
- SG_INFO_CHECK [0x1] something abnormal happened. In most but not all cases, the sense buffer will be written. If the sense buffer has not been written than 'sb_len_wr' will be zero. This flag indicates either 'masked_status', 'host_status' or 'driver_status' is non−zero.

A multi bit component contained in SG_INFO_DIRECT_IO_MASK indicates what type of data transfer has just taken place. If indirect IO (or no data transfer) has taken place then SG_INFO_INDIRECT_IO is matched. Note that even if direct IO was requested in 'flags' the driver may choose to do indirect IO instead. If direct IO was requested and performed then SG_INFO_DIRECT_IO will be matched. Currently SG_INFO_MIXED_IO is never set. The associated values are:

- SG_INFO_DIRECT_IO_MASK [0x6]
- SG_INFO_INDIRECT_IO [0x0] data xfer via kernel buffers (or no xfer)
- SG_INFO_DIRECT_IO [0x2]
- SG_INFO_MIXED_IO [0x4] part direct, part indirect IO

The type of info is unsigned int .

Chapter 7. System calls

System calls that can be used on sg devices are discussed in this chapter. The ioctl() system call is discussed in the following chapter [see Chapter 8].

Successfully opening a sg device file name (e.g. `/dev/sg0`) establishes a link between a file descriptor and an attached SCSI device. The sg driver maintains state information and resources at both the SCSI device (e.g. exclusive lock) and the file descriptor (e.g. reserved buffer) levels.

A SCSI device can be detached while an application has a sg file descriptor open. An example of this is a "hotplug" device such as a USB mass storage device that has just been unplugged. Most subsequent system calls that attempt to access the detached SCSI device will yield ENODEV. The close() call will complete silently while the poll() call will "or" in POLLHUP to its result. A subsequent attempt to open() that device name will yield ENODEV.

7.1. open()

open(const char * filename, int flags). The filename should be a sg device file name as discussed in the Chapter 4. Flags can be a number of the following or−ed together:

- O_RDONLY restricts operations to read()s and ioctl()s (i.e. can't use write()).
- O_RDWR permits all system calls to be executed.
- O_EXCL waits for other opens on the associated SCSI device to be closed before proceeding. If O_NONBLOCK is set then yields EBUSY when someone else has the SCSI device open. The combination of O_RDONLY and O_EXCL is disallowed.
- O_NONBLOCK Sets non−blocking mode. Calls that would otherwise block yield EAGAIN (e.g. read()) or EBUSY (e.g. open()). This flag is ignored by ioctl(SG_IO) .

Either O_RDONLY or O_RDWR must be set in flag. Either of the other 2 flags (but not both) can be or−ed in.

Note that multiple file descriptors may be open to the same SCSI device. [This is a way of side stepping the SG_MAX_QUEUE limit.] At the sg level separate state information is maintained. This means that even if multiple file descriptors are open to a single SCSI device their write() read() sequences are essentially independent.

Open() calls may be blocked due to exclusive locks (i.e. O_EXCL). An exclusive lock applies to a single SCSI device and only to sg's use of that device (i.e. it has no effect on access via sd, sr or st to that device). If the O_NONBLOCK flag is used then open() calls that would have otherwise blocked, yield EBUSY. Applications that scan sg devices trying to determine their identity (e.g. whether one is a scanner) should use the O_NONBLOCK flag otherwise they run the risk of blocking.

The driver will attempt to reserve SG_DEF_RESERVED_SIZE bytes (32KBytes in the current sg.h) on open(). The size of this reserved buffer can subsequently be modified with the SG_SET_RESERVED_SIZE ioctl(). In both cases these are requests subject to various dynamic constraints. The actual amount of memory obtained can be found by the SG_GET_RESERVED_SIZE ioctl(). The reserved buffer will be used if:

- it is not already in use (e.g. when command queuing is in use)
- a write() or ioctl(SG_IO) requests a data transfer size that is less than or equal to the reserved buffer

size.

Returns a file descriptor if >= 0 , otherwise −1 implies an error.

7.2. write()

write(int sg_fd, const void * buffer, size_t count). The action of write() with a control block based on struct sg_header is discussed in the earlier document: `www.torque.net/sg/p/scsi-generic.txt` (i.e the sg version 2 documentation). This section describes the action of write() when it is given a control block based on struct sg_io_hdr.

The 'buffer' should point to an object of type sg_io_hdr_t and 'count' should be sizeof(sg_io_hdr_t) [it can be larger but the excess is ignored]. If the write() call succeeds then the 'count' is returned as the result.

Up to SG_MAX_QUEUE (16) write()s can be queued up before any finished requests are completed by read(). An attempt to queue more than that will result in an EDOM error. [9] The write() command should return more or less immediately. [10]

The version 2 sg driver defaulted the maximum queue length to 1 (and made available the SG_SET_COMMAND_Q ioctl() to switch it to SG_MAX_QUEUE). So for backward compatibility a file descriptor that only receives sg_header structures in its write() will have a default "max" queue length of 1. As soon as a sg_io_hdr_t structure is seen by a write() then the maximum queue length is switched to SG_MAX_QUEUE on that file descriptor.

The "const" on the 'buffer' pointer is respected by the sg driver. Data is read in from the sg_io_hdr object that is pointed to. Significantly this is when the 'sbp' and the 'dxferp' are recorded internally (i.e. not from the sg_io_hdr object given to the corresponding read()).

7.3. read()

read(int sg_fd, void * buffer, size_t count). The action of read() with a control block based on struct sg_header is discussed in the earlier document: `www.torque.net/sg/p/scsi-generic.txt` (i.e. the sg version 2 documentation). This section describes the action of read() when it is given a control block based on struct sg_io_hdr.

The 'buffer' should point to an object of type sg_io_hdr_t and 'count' should be sizeof(sg_io_hdr_t) [it can be larger but the excess is ignored]. If the read() call succeeds then the 'count' is returned as the result.

By default, read() will return the oldest completed request that is queued up. A read() will not interfere with any request associated with the SG_IO ioctl() on this file descriptor except in a special case when a SG_IO ioctl() is interrupted by a signal.

If the SG_SET_FORCE_PACK_ID,1 ioctl() is active then read() will attempt to fetch the packet whose pack_id (given earlier to write()) matches the sg_io_hdr_t::pack_id given to this read(). If not available it will either wait or yield EAGAIN. As a special case, −1 in sg_io_hdr_t::pack_id given to read() will match the request whose response has been waiting for the longest time. Take care to also set 'dxfer_direction' to any valid value (e.g. SG_DXFER_NONE) when in this mode. The 'interface_id' member should also be set appropriately.

Apart from the SG_SET_FORCE_PACK_ID case (and then only for the 3 indicated fields), the sg_io_hdr_t object given to read() can be uninitialized. Note that the 'sbp' pointer value for optionally outputting a sense buffer was recorded from the earlier, corresponding write().

7.4. poll()

poll(struct pollfd *ufds, unsigned int nfds, int timeout). This call can be used to check the state of a sg file descriptor. It will always respond immediately. Typical usages are to periodically poll the state of a sg file descriptor and to determine why a SIG_IO signal was received.

For file descriptors associated with sg devices:

- POLLIN one or more responses is awaiting a read()
- POLLOUT command can be sent to write() without causing an EDOM error (i.e. sufficient space on sg's queues)
- POLLHUP SCSI device has been detached, awaiting cleanup
- POLLERR internal structures are inconsistent

POLLOUT indicates the sg will not block a new write() or SG_IO ioctl(). However it is still possible (but unlikely) that the mid level or an adapter may block (or yield EAGAIN).

7.5. close()

close(int sg_fd). Preferably a close() should be done after all issued write()s have had their corresponding read() calls completed. Unfortunately this is not always possible (e.g. the user may choose to send a kill signal to a running process). The sg driver implements "fast" close semantics and thus will return more or less immediately (i.e. not wait on any event). This is application friendly but requires the sg driver to arrange for an orderly cleanup of those packets that are still "in flight".

When close() leaves outstanding SCSI commands still awaiting responses, the sg driver maintains its internal structures for the now defunct file descriptor. These internal structures are maintained until all outstanding responses (some might be timeouts) are received. When the sg driver is loaded as a module and has any open file descriptors or "defunct" file descriptors then it cannot be unloaded. An attempt to call **rmmod sg** will report the driver is busy. Defunct file descriptors that remain for some time, perhaps awaiting a timeout, can be observed with the **cat /proc/scsi/sg/debug** command. In this case "closed=1" will be set on the defunct file descriptor [see Section 11.1]. Defunct file descriptors do not impede attempts by applications to open() new file descriptors on the same SCSI device.

The kernel arranges for only the last close() on a file descriptor to be seen by a driver (and to emphasize this, the corresponding sg driver call is named sg_release() rather than sg_close()). This is only significant when an application uses fork() or dup().

Returns 0 if successful, otherwise −1 implies an error.

7.6. mmap()

mmap(void * start, size_t length, int prot, int flags, int sg_fd, off_t offset). This system call returns a pointer to the beginning of the reserved buffer associated with the sg file descriptor 'sg_fd'. The 'start' argument is a hint to the kernel and is ignored by this driver; best set it to 0. The 'length' argument should be less than or equal to the size of the reserved buffer associated with 'sg_fd'. If it exceeds the reserved buffer size (after 'length' has been rounded up to a page size multiple) then MAP_FAILED is returned and ENOMEM is placed in errno. The 'prot' argument should either be PROT_READ or (PROT_READ | PROT_WRITE). The 'flags' argument should contain MAP_SHARED. In a sense, the user application is "sharing" data with the sg driver. The MAP_PRIVATE flag does not play well with compiler optimization flags such as '−O2'. The 'offset' argument must be set to 0 (or NULL).

The mmap() system call can be made multiple times on the same sg_fd. The munmap() system call is not required if close() is called on sg_fd. Mmap−ed IO is well−behaved when a process is fork()−ed (or the equivalent finer grained clone() system call is made). In the case of a fork(), 2 processes will be sharing the same memory mapped area together with the sg driver for a sg_fd and the last one to close the sg_fd (or exit) will cause the shared memory to be freed.

It is assumed that if the default reserved buffer size of 32 KB is not sufficient then a ioctl(SG_SET_RESERVED_SIZE) call is made prior to any calls to mmap(). If the required size is not a multiple of the kernel's page size (returned by getpagesize() system call) then the size passed to ioctl(SG_SET_RESERVED_SIZE) should be rounded up to the next page size multiple.

Mmap−ed IO is requested by setting (or or−ing in) the SG_FLAG_MMAP_IO constant into the flag member of the the sg_io_hdr structure prior to a call to write() or ioctl(SG_IO). The logic to do mmap−ed IO _assumes_ that an appropriate mmap() call has been made by the application. In other words it does not check. [11]

7.7. fcntl(sg_fd, F_SETFL, oflags | FASYNC)

fcntl(int sg_fd, int cmd, long arg). There are several uses for this system call in association with a sg file descriptor. The following pseudo code shows code that is useful for scanning the sg devices, taking care not to be caught in a wait for an O_EXCL lock by another process, and when the appropriate device is found, switching to normal blocked io. A working example of this logic is in the sg_scan utility program.

```
open("/dev/sg0", O_RDONLY | O_NONBLOCK)
/* check device, EBUSY means some other process has O_EXCL lock on it */
/* when the device you want is found then ... */
flags = fcntl(sg_fd, F_GETFL)
fcntl(sg_fd, F_SETFL, flags & (~ O_NONBLOCK))
/* since, with simple apps, it is easier to use normal blocked io */
```

The sg driver supports asynchronous notification. This is a non−blocking mode of operation in which, when the driver receives data back from a device so that a read() can be done, it sends a SIGPOLL (aka SIGIO) signal to the owning process. Here is a code snippet from the sg_poll test program.

```
sigemptyset(&sig_set)
sigaddset(&sig_set, SIGPOLL)
sigaction(SIGPOLL, &s_action, 0)
fcntl(sg_fd, F_SETOWN, getpid())
```

```
flags = fcntl(sg_fd, F_GETFL);
fcntl(sg_fd, F_SETFL, flags | O_ASYNC)
```

7.8. Errors reported in errno

With the original interface almost any string could be accidentally given to write() and potentially (but rarely) something nasty could happen. If some error was detected then more than likely EIO was placed in errno.

Unfortunately this can still happen with write() since it can accept both the original struct sg_header or the newer sg_io_hdr_t described in this note. However since the SG_IO ioctl() will only accept the sg_io_hdr_t structure there is less chance of a random string being interpreted as a command. Since the sg_io_hdr_t interface does a lot more error checking, it attempts to give out more precise errno values to help the user pinpoint the problem. [Admittedly some of these errno values are picked in an arbitrary way from the large set of available values.]

In most cases when a system call on a sg file descriptor fails, the call in question will return −1. After an application detects that a system call has failed it should read the value in the "errno" variable (prior to do any more system calls). Applications should include the <errno.h> header.

Below is a table of errno values indicating which calls to sg will generate them and the meaning of the error. A write() call is indicated by "w", a read() call by "r" and an open() call by "o".

```
errno    which_calls    Meaning
-----    -----------    ----------------------------------------------
EACCES    o, w,r,SG_IO  User does not have permissions to do this. They will
          <some ioctls> need read,write permissions currently. This has
                        been relaxed for certain SCSI commands via SG_IO.
EAGAIN    r             The file descriptor is non-blocking and the request
                        has not been completed yet.
EAGAIN    w,SG_IO       SCSI sub-system has (temporarily) run out of
                        command blocks.
EBADF     w             File descriptor was not open()ed O_RDWR.
EBUSY     o             Someone else has an O_EXCL lock on this device.
EBUSY     w             With mmap-ed IO, the reserved buffer already in use.
EBUSY     <some ioctls> Attempt to change something (e.g. reserved buffer
                        size) when the resource was in use.
EDOM      w,SG_IO       Too many requests queued against this file
                        descriptor. Limit is SG_MAX_QUEUE active requests.
                        If sg_header interface is being used then the
                        default queue depth is 1. Use SG_SET_COMMAND_Q
                        ioctl() to increase it.
EFAULT    w,r,SG_IO     Pointer to user space invalid.
          <most ioctls>
EINVAL    w,r           Size given as 3rd argument not large enough for the
                        sg_io_hdr_t structure. Both direct and mmap-ed IO
                        selected.
EIO       w             Size given as 3rd argument less than size of old
                        header structure (sg_header). Additionally a write()
                        with the old header will yield this error for most
                        detected malformed requests.
EIO       r             A read() with the older sg_header structure yields
                        this value for some errors that it detects.
EINTR     o             While waiting for the O_EXCL lock to clear this call
                        was interrupted by a signal.
EINTR     r,SG_IO       While waiting for the request to finish this call
```

```
                              was interrupted by a signal.
EINTR     w                   [Very unlikely] While waiting for an internal SCSI
                              resource this call was interrupted by a signal.
EMSGSIZE  w,SG_IO             SCSI command size ('cmd_len') was too small
                              (i.e. < 6) or too large
ENODEV    o                   Tried to open() a file with no associated device.
                              [Perhaps sg has not been built into the kernel or
                              is not available as a module?]
ENODEV    o,w,r,SG_IO         SCSI device has detached, awaiting cleanup.
                              User should close fd. Poll() will yield POLLHUP.
ENOENT    o                   Given filename not found.
ENOMEM    o                   [Very unlikely] Kernel was not even able to find
                              enough memory for this file descriptor's context.
ENOMEM    w,SG_IO             Kernel unable to find memory for internal buffers.
                              This is usually associated with indirect IO.
                              For mmap-ed IO 'dxfer_len' greater than reserved
                              buffer size.
ENOSYS    w,SG_IO             'interface_id' of a sg_io_hdr_t object was _not_ 'S'.
ENXIO     o                   "remove-single-device" may have removed this device.
ENXIO     o, w,r,SG_IO        Internal error (including SCSI sub-system busy doing
                              error processing - e.g. SCSI bus reset). When a
                              SCSI device is offline, this is the response. This
                              can be bypassed by opening O_NONBLOCK.
```

Chapter 8. Ioctl()s

The Linux SCSI upper level drivers, including sg, have a "trickle down" ioctl() architecture. This means that ioctl()s whose request value (i.e. the second argument) is not understood by the upper level driver, are passed down to the SCSI mid−level. Those ioctl()s that are not understood by the mid level driver are passed down to the lower level (adapter) driver. If none of the 3 levels understands the ioctl() request value then −1 is returned and EINVAL is placed in errno. By convention the beginning of the request value's symbolic name indicates which level will respond to the ioctl(). For example, request values starting with "SG_" are processed by the sg driver while those starting with "SCSI_" are processed by the mid level.

Most of the sg ioctl()s read or write information via a pointer given as the third argument to the ioctl() call and return 0 on success. A few of the older ioctl()s that get a value from the driver return that value as the result of the ioctl() call (e.g. ioctl(SG_GET_TIMEOUT)).

All sg driver ioctl()s are listed below. They all start with "SG_". They are followed by several interesting SCSI mid level ioctl()s which start with "SCSI_IOCTL_". The sg ioctl()s are roughly in alphabetical order (with _SET_, _GET_ and _FORCE_ ignored). Since ioctl(SG_IO) is a complete SCSI command request/response sequence then it is listed first.

8.1. SG_IO

SG_IO 0x2285. The idea is deceptively simple: just hand a sg_io_hdr_t object to an ioctl() and it will return when the SCSI command is finished. It is logically equivalent to doing a write() followed by a blocking read(). The word "blocking" here implies the read() will wait until the SCSI command is complete.

The same file descriptor can be used both for SG_IO synchronous calls and the write() read() sequences at the same time. The sg driver makes sure that the response to a SG_IO call will never accidentally be fetched by a read(). Even though a single file descriptor can be shared in this manner, it is probably more sensible (and results in cleaner code) if separate file descriptors to the same SCSI device are used in this case.

It is possible that the wait for the command completion is interrupted by a signal. In this case the SG_IO call will yield an EINTR error. This is reasonably complex to handle and is discussed in the ioctl(SG_SET_KEEP_ORPHAN) description below. The following SCSI commands will be permitted by SG_IO when the sg file descriptor was opened O_RDONLY:

- TEST UNIT READY
- INQUIRY
- READ CAPACITY
- READ BUFFER
- READ(6) (10) and (12)
- MODE_SENSE(6) and (10)

All commands to SCSI device type SCANNER are accepted. Other cases yield an EACCES error. Note that the write() read() interface must have the sg file descriptor open()−ed with O_RDWR as write permission is required by Linux to execute a write() system call.

The ability of the SG_IO ioctl() to issue certain SCSI commands has led to some relaxation on file descriptors open()ed "read−only" compared with the version 2 sg driver. The open() call will now attempt to allocate a reserved buffer for all newly opened file descriptors. The ioctl(SG_SET_RESERVED_SIZE) will now work on "read−only" file descriptors.

8.2. SG_GET_ACCESS_COUNT

SG_GET_ACCESS_COUNT 0x2289. This ioctl() yields the access count maintained by the mid level for this SCSI device. This number is incremented by each open() call done by the upper level SCSI drivers (i.e. sd, sr, st and sg) and decremented by those drivers' release(). [A driver's release() corresponds to the last close() on a file descriptor, or is supplied by the kernel when a process is aborted.] Each SCSI device has a separate access count.

8.3. SG_SET_COMMAND_Q (and _GET_)

SG_SET_COMMAND_Q 0x2271 [_GET_ 0x2270] . The default it the original sg driver was not to allow commands to be queued on the same file descriptor (actually it was more restrictive, commands could not be queued on a SCSI device). The version 2 sg driver kept this action as its default (for backward compatibility) and offered these ioctl()s to change and monitor the command queuing state.

8.4. SG_SET_DEBUG

SG_SET_DEBUG 0x227e. The third argument is assumed to point to an int. The default value is 0. If this call is made pointing to an int greater than 0 then any SCSI request that is issued that results in the SCSI status of CHECK_CONDITION (or COMMAND_TERMINATED) will cause a message to be sent to the log (and perhaps the console). The message is information derived from the sense buffer (i.e. the SCSI error message) and it is prefixed with "sg_cmd_done_bh".

The other actions of debug mode performed in version 2 of the sg driver have been removed as they are no longer needed. The internal state of the sg driver can now be found by viewing the output of **cat /proc/scsi/sg/debug**.

8.5. SG_EMULATED_HOST

SG_EMULATED_HOST 0x2203. Assumes 3rd argument points to an int and outputs a flag indicating whether the host (adapter) is connected to a "real" SCSI bus or is an emulated one (e.g. ide−scsi or usb storage device driver). A value of 1 means emulated while 0 is not. [To check: is IEEE1394 a "real" SCSI serial bus?]

8.6. SG_SET_KEEP_ORPHAN (and _GET_)

SG_SET_KEEP_ORPHAN 0x2287 [_GET_ 0x2288]. These ioctl()s allow the setting and reading of the "keep_orphan" flag. This controls what happens to the request associated with a SG_IO ioctl() that is interrupted (i.e. errno is EINTR). The default action is to drop the response as soon as it is received. This corresponds to the "keep_orphan" flag being 0. When the "keep_orphan" flag is 1 then the response is transformed in such a way that it can be fetched by a read(). This is the only circumstance in which a request sent by a SG_IO ioctl() can have the associated response fetched by a read().

8.7. SG_SET_FORCE_LOW_DMA

SG_SET_FORCE_LOW_DMA 0x2279. Assumes 3rd argument points to an int containing 0 or 1. 0 (default) means sg decides whether to use memory above 16 Mbyte level (on i386) based on the host adapter being used by this SCSI device. Typically PCI SCSI adapters will indicate they can DMA to the whole 32 bit address space. If 1 is given then the host adapter is overridden and only memory below the 16MB level is used for DMA. A requirement for this should be extremely rare. If the "reserved" buffer allocated on open() is not in use then it will be de−allocated and re−allocated under the 16MB level (and the latter operation could fail yielding ENOMEM). Only the current file descriptor is affected.

8.8. SG_GET_LOW_DMA

SG_GET_LOW_DMA 0x227a. Assumes 3rd argument points to an int and places 0 or 1 in it. 0 indicates the whole 32 bit address space is being used for DMA transfers on this file descriptor. 1 indicates the memory below the 16MB level (on i386) is being used (and this may be the case because the host adapters setting has been overridden by SG_SET_FORCE_LOW_DMA,1 .

8.9. SG_NEXT_CMD_LEN

SG_NEXT_CMD_LEN 0x2283. This ioctl() is not required with sg_io_hdr structure since command length is set explicitly for every command. Assumes 3rd argument is pointing to an int. The value of the int (if > 0) will be used as the SCSI command length of the next SCSI command sent to a write() using the sg_header interface. After that write() the SCSI command length logic is reset to use automatic length detection (i.e. depending on SCSI command group and the 'twelve_byte' field). If the current SCSI command length maximum of 16 is exceeded then the affected write() will yield an EDOM error. Giving this ioctl() a value of 0 will set automatic length detection for the next write(). N.B. Only the following write() on this fd is affected by this ioctl().

8.10. SG_GET_NUM_WAITING

SG_GET_NUM_WAITING 0x227d. Assumes 3rd argument points to an int and places the number of packets waiting to be read in it. Only those requests that have been issued by a write() and are now available to be read() are counted. In other words any ioctl(SG_IO) operations underway on this file descriptor will not effect this count [12].

8.11. SG_SET_FORCE_PACK_ID

SG_SET_FORCE_PACK_ID 0x227b. Assumes 3rd argument is pointing to an int. 0 (default) instructs read() to return the oldest (written) packet if multiple packets are waiting to be read. 1 instructs read() to view the sg_io_hdr::pack_id (or sg_header::pack_id) as input and return the oldest packet matching that pack_id or wait until it arrives. If the file descriptor is in O_NONBLOCK state, rather than wait this ioctl() will yield EAGAIN. As a special case the pack_id of −1 given to read() in the mode will match the oldest packet. Only the current file descriptor is affected by this command.

8.12. SG_GET_PACK_ID

SG_GET_PACK_ID 0x227c. Assumes 3rd argument points to an int and places the pack_id of the oldest (written) packet in it. If no packet is waiting to be read then yields −1.

8.13. SG_GET_REQUEST_TABLE

SG_GET_REQUEST_TABLE 0x2286. This ioctl outputs an array of information about the status of requests associated with the current file descriptor. Its 3rd argument should point to memory large enough to receive SG_MAX_QUEUE objects of the sg_req_info_t structure. This structure has the following members:

```
        req_state
            0 -> request not in use
            1 -> request has been sent, but is not finished (i.e. it is
                 between stages 1 and 2 in the "theory of operation")
            2 -> request is ready to be read() (i.e. it is between stages
                 2 and 3 in the "theory of operation")
        orphan
            0 -> normal request
            1 -> request sent by SG_IO ioctl() which has been interrupted
                 by a signal
        sg_io_owned
            0 -> request sent by a write()
            1 -> request sent by a SG_IO ioctl()
        problem
            0 -> no problem (or 1 == req_state)
            1 -> req_state is 2 and either masked_status, host_status or
                 driver_status is non-zero
        duration
            [if 1 == req_state] time since request was sent (in millisecs)
            [if 2 == req_state] duration of request (in millisecs). Clock
                 is stopped when stage 2 in "theory of operation" is
                 reached
        pack_id
        usr_ptr
            these are user provided values in the sg_io_hdr_t (or
            struct sg_header) that sent the request
```

8.14. SG_SET_RESERVED_SIZE (and _GET_)

SG_SET_RESERVED_SIZE 0x2275 [_GET_ 0x2272]. Both ioctl()s assume the 3rd argument is pointing to an int.

For ioctl(SG_SET_RESERVED_SIZE) the value will be used to request a new reserved buffer of that size. The previous reserved buffer is freed (if it is not in use; if it was in use then the ioctl() fails and EBUSY is placed in errno). A new reserved buffer is then allocated and its actual size can be found by calling the ioctl(SG_GET_RESERVED_SIZE). The reserved buffer is then used for DMA purposes by subsequent write() and ioctl(SG_IO) commands if it is not already in use and if the write() is not calling for a buffer size larger than that reserved. The reserved buffer may well be a series of kernel buffers if the adapter supports scatter−gather. Large buffers can be requested (e.g. 4 MB) but not necessarily granted. Once a mmap() call has been made on a sg file descriptor, subsequent calls to this ioctl() will fail with EBUSY placed in errno.

In the case of ioctl(SG_GET_RESERVED_SIZE) the size in bytes of the reserved buffer from open() or the most recent SG_SET_RESERVED_SIZE ioctl() call on this fd. The result can be 0 if memory is very tight. In this case it may not be wise to attempt something like burning a CD on this file descriptor.

8.15. SG_SCSI_RESET

SG_SCSI_RESET 0x2284. Assumes 3rd argument points to an int. That int should be one of the following defined in the `sg.h` header:

- SG_SCSI_RESET_NOTHING (0x0): can be used to poll the device after a reset has been issued to see if it has returned to the normal state. If it is still being reset or it is offline then EBUSY will be placed in errno,
- SG_SCSI_RESET_DEVICE (0x1): issues a reset to the SCSI device associated with the current sg file descriptor,
- SG_SCSI_RESET_BUS (0x2): issues a reset to the SCSI bus that contains the device associated with the current sg file descriptor. This will usually have an adverse effect on any other SCSI device sharing this SCSI bus, especially if it was in the middle of an operation,
- SG_SCSI_RESET_HOST (0x3): issues a reset to the host that controls the SCSI bus that contains the device associated with the current sg file descriptor. This operation can have an adverse effect on any SCSI device that is connected to this host.

The reset options are in ascending order of severity. Not all levels are supported by all linux lower level drivers. Most lower level (adapter) drivers support the SCSI bus reset. These boards often issue a SCSI bus reset during their initialization.

Unfortunately this ioctl() doesn't currently do much (but may in the future after other issues are resolved). Yields an EBUSY error if the SCSI bus or the associated device is being reset when this ioctl() is called, otherwise returns 0. N.B. In some recent distributions there is a patch to the SCSI mid level code that activates this ioctl. Check your distribution.

8.16. SG_GET_SCSI_ID

SG_GET_SCSI_ID 0x2276. Assumes 3rd argument is pointing to an object of type Sg_scsi_id (see sg.h) and populates it. That structure contains ints for host_no, channel, scsi_id, lun, scsi_type, allowable commands per lun and queue_depth. Most of this information is available from other sources (e.g. SCSI_IOCTL_GET_IDLUN and SCSI_IOCTL_GET_BUS_NUMBER) but tends to be awkward to collect. Allowable commands per lun and queue_depth give an insight to the command queuing capabilities of the adapters and the device. The latter overrides the former (logically) and the former is only of interest if it is equal to queue_depth which probably indicates the device does not support queuing commands (e.g. most scanners).

```
typedef struct sg_scsi_id { /* used by SG_GET_SCSI_ID ioctl() */
    int host_no;        /* as in "scsi<n>" where 'n' is one of 0, 1, 2 etc */
    int channel;
    int scsi_id;        /* scsi id of target device */
    int lun;
    int scsi_type;      /* TYPE_... defined in scsi/scsi.h */
    short h_cmd_per_lun;/* host (adapter) maximum commands per lun */
    short d_queue_depth;/* device (or adapter) maximum queue length */
    int unused[2];      /* probably find a good use, set 0 for now */
} sg_scsi_id_t;
```

8.17. SG_GET_SG_TABLESIZE

SG_GET_SG_TABLESIZE 0x227F. Assumes 3rd argument points to an int and places the maximum number of scatter gather elements supported by the host adapter associated with the current SCSI device. 0 indicates that the adapter does support scatter gather.

8.18. SG_GET_TIMEOUT

SG_GET_TIMEOUT 0x2202. Ignores its 3rd argument and _returns_ the timeout value (which will be >= 0). The unit of this timeout is "jiffies" which are currently 10 millisecond intervals on i386 (less on an alpha). Linux supplies a manifest constant HZ which is the number of "jiffies" in 1 second. This ioctl() is not relevant to the sg version 3 driver because timeouts are specified explicitly for each command in the sg_io_hdr structure.

8.19. SG_SET_TIMEOUT

SG_SET_TIMEOUT 0x2201. Assumes 3rd argument points to an int containing the new timeout value for this file descriptor. The unit is a "jiffy". Packets that are already "in flight" will not be affected. The default value is set on open() and is SG_DEFAULT_TIMEOUT (defined in sg.h). This default is currently 1 minute and may not be long enough for formats. Negative values will yield an EIO error. This ioctl() is not relevant to the sg version 3 driver because timeouts are specified explicitly for each command in the sg_io_hdr structure. Only when the sg_header structure is used is the timeout inherited from this value (help on a per file descriptor basis).

8.20. SG_SET_TRANSFORM

SG_SET_TRANSFORM 0x2204. Only is meaningful when SG_EMULATED host has yielded 1 (i.e. the low−level is the ide−scsi device driver); otherwise an EINVAL error occurs. The default state is to _not_ transform SCSI commands to the corresponding ATAPI commands but pass them straight through as is. [Only certain classes of SCSI commands need to be transformed to their ATAPI equivalents.] The third argument is interpreted as an integer. When it is non−zero then a flag is set inside the ide−scsi driver that transforms subsequent commands sent to this driver. When zero is passed as the 3rd argument to this ioctl then the flag within the ide−scsi driver is cleared and subsequent commands are not transformed. Beware, this state will affect all devices (and hence all related sg file descriptors) associated with this ide−scsi "bus".

8.21. SG_GET_TRANSFORM

SG_GET_TRANSFORM 0x2205. Third argument is ignored. Only is meaningful when SG_EMULATED host has yielded 1 (i.e. the low−level is the ide−scsi device driver); otherwise an EINVAL error occurs. Returns 0 to indicate _not_ transforming SCSI to ATAPI commands (default). Returns 1 when it is transforming them.

8.22. Sg ioctls removed in version 3

Some seldom used ioctl()s introduced in the sg 2.x series drivers have been withdrawn. They are:

- SG_SET_UNDERRUN_FLAG (and _GET_) [use 'resid' in this new interface]
- SG_SET_MERGE_FD (and _GET) [added complexity with little benefit]

8.23. SCSI_IOCTL_GET_IDLUN

SCSI_IOCTL_GET_IDLUN 0x5382. This ioctl takes a pointer to a "struct scsi_idlun" object as its third argument. The "struct scsi_idlun" is not visible to user applications. To use this, that structure needs to be replicated in the user's program. Something like:

```
typedef struct my_scsi_idlun {
    int four_in_one;    /* 4 separate bytes of info compacted into 1 int */
    int host_unique_id; /* distinguishes adapter cards from same supplier */
} My_scsi_idlun;
```

"four_in_one" is made up as follows:

```
(scsi_device_id | (lun << 8) | (channel << 16) | (host_no << 24))
```

These 4 components are assumed (or masked) to be 1 byte each. These are the four numbers that the SCSI subsystem uses to index devices, often written as "<host_no, channel, scsi_id, lun>". The 'host_unique_id' assigns a different number to each controller from the same manufacturer/low−level device driver. Most of the information provided by this command is more easily obtained from SG_GET_SCSI_ID.

The 'host_no' element is a change in lk 2.4 kernels. [In the lk 2.2 series and earlier, it was 'low_inode & 0xff' from the procfs entry corresponding to the host.] This change makes the use of the SCSI_IOCTL_GET_BUS_NUMBER ioctl() superfluous.

The advantage of this ioctl() is that it can be called on any SCSI file descriptor.

8.24. SCSI_IOCTL_GET_PCI

SCSI_IOCTL_GET_PCI 0x5387. Yields the PCI slot name (pci_dev::slot_name) associated with the lower level (adapter) driver that controls the current device. Up to 8 characters are output to the location pointed to by 'arg'. If the current device is not controlled by a PCI device then errno is set to ENXIO. [This ioctl() was introduced in lk 2.4.4]

8.25. SCSI_IOCTL_PROBE_HOST

SCSI_IOCTL_PROBE_HOST 0x5385. This command should be given a pointer to a 'char' array as its 3rd argument. That array should be at least sizeof(int) long and have the length of the array as an 'int' at the beginning of the array! An ASCII string of no greater than that length containing "information" (or the name) of SCSI host (i.e. adapter) associated with this file descriptor is then placed in the given byte array. N.B. A trailing '\0' may need to be put on the output string if it has been truncated by the input length. Returns 1 if host is present, 0 if it is not and a negative value if there is an error.

8.26. SCSI_IOCTL_SEND_COMMAND

SCSI_IOCTL_SEND_COMMAND 0x1. This ioctl() also offers a "pass through" SCSI command capability which is a subset of what is offered by the sg driver.

The structure that we are passed should look like:

```
struct sdata {
 unsigned int inlen;      [i] Length of data written to device
 unsigned int outlen;     [i] Length of data read from device
 unsigned char cmd[x];    [i] SCSI command (6 <= x <= 16)
                          [o] Data read from device starts here
                          [o] On error, sense buffer starts here
 unsigned char wdata[y];  [i] Data written to device starts here
};
```

Notes:

- The SCSI command length is determined by examining the 1st byte of the given command [13] . There is no way to override this.
- Data transfers are limited to PAGE_SIZE (4K on i386, 8K on alpha).
- The length (x + y) must be at least OMAX_SB_LEN bytes long to accommodate the sense buffer when an error occurs. The sense buffer is truncated to OMAX_SB_LEN (16) bytes so that old code will not be surprised.
- If a Unix error occurs (e.g. ENOMEM) then the user will receive a negative return and the Unix error code in 'errno'. If the SCSI command succeeds then 0 is returned. Positive numbers returned are the compacted SCSI error codes (4 bytes in one int) where the lowest byte is the SCSI status. See the drivers/scsi/scsi.h file for more information on this.

Chapter 9. Direct and Mmap−ed IO

The normal action of the sg driver for a read operation (from a device) is to request the lower level (adapter) driver to DMA [14] data into kernel buffers that the sg driver manages. The sg driver will then copy the contents of its buffers into the user space. [This sequence is reversed for a write operation (towards a device)]. While this double handling of data is obviously inefficient it does decouple some hardware issues from user applications. For these and historical reasons the "double−buffered" IO remains the default for the sg driver.

Both "direct" and "mmap−ed" IO are techniques that permit the data to be DMA−ed directly from the lower level (adapter) driver into the user application (vice versa for write operations). Both techniques result in faster speed, smaller latencies and lower CPU utilization but come at the expense of complexity (as always). For example the Linux kernel must not attempt to swap out pages in a user application that a SCSI adapter is busy DMA−ing data into.

9.1. Direct IO

Direct IO uses the kiobuf mechanism [see the Linux Device Drivers book] to manipulate memory allocated within the user space so that a lower level (adapter) driver can DMA directly to or from that user space memory. Since the user can give a different data buffer to each SCSI command passed through the sg interface then the kiobuf mechanism needs to setup its structures (and undo that setup) for each SCSI command. [15] Direct IO is available as an option in sg 3.1.18 (before that the sg driver needed to be recompiled with an altered define). Direct IO support is designed in such a way that if it is requested and cannot be performed then the command will still be performed using indirect IO. If direct IO is requested and has been performed then the SG_INFO_DIRECT_IO bit will be set in the 'info' member of the sg_io_hdr_t control structure after the request has been completed. Direct IO is not supported on ISA SCSI adapters since they only can address a 24 bit address space.

One limit on direct IO is that sg_io_hdr_t::iovec_count==0. So the user cannot (currently) use application level scatter gather and direct IO on the same request.

For direct IO to be worthwhile, a reasonable amount of data should be requested for data transfer. For transfers less than 8 KByte it is probably not worth the trouble. On the other hand "locking down" a multiple 512 KB blocks of data for direct IO could adversely impact overall system performance. Remember that for the duration of a direct IO request, the data transfer buffer is mapped to a fixed memory location and locked in such a way that it won't be swapped out. This can "cramp the style" of the kernel if it is overdone.

Prior to sg 3.1.18 the direct IO code was commented out with the "SG_ALLOW_DIO" define. In sg 3.1.18 (available for lk 2.4.2 and later) the direct IO code is active but is defaulted off by a run time value. This value can be accessed via the "proc" file system at `/proc/scsi/sg/allow_dio` . Direct IO is enabled when a user with root permissions writes "1" to that file: **echo 1 > /proc/scsi/sg/allow_dio** . If SG_FLAG_DIRECT_IO is set in sg_io_hdr::flags but `/proc/scsi/sg/allow_dio` holds "0" then indirect IO will be performed (and this is indicated by ((sg_io_hdr::info & SG_INFO_DIRECT_IO_MASK) == SG_INFO_INDIRECT_IO) after the request is completed).

9.2. Mmap−ed IO

Memory−mapped IO takes a different approach from direct IO to removing the extra data copy performed by normal ("indirect") IO. With mmap−ed IO the application calls the mmap() system call to memory map sg's reserved buffer. The sg driver maintains one reserved buffer per file descriptor. The default size of the reserved buffer is 32 KB and it can be changed with the ioctl(SG_SET_RESERVED_SIZE). The mmap() system call only needs to be called once prior [16] to doing mmap−ed IO. For more details on the mmap() see Section 7.6. An application indicates that it wants mmap−ed on a SCSI request by setting the SG_FLAG_MMAP_IO value in 'flags'.

Since there is only reserved buffer per sg file descriptor then only one mmap−ed IO command can be active at one time. In order to perform command queuing with mmap−ed IO, an application will need to open() multiple file descriptors to the same SCSI device. With mmap−ed IO the various status values and the sense buffer (if required) are conveyed back to an application in the same fashion as normal ("indirect") IO.

Mmap−ed has very low per command latency since the reserved buffer mapping only needs to be done once per file descriptor. Also the reserved buffer is set up by the sg driver to aid the efficient construction of the internal scatter gather list used by the lower level (adapter) driver for DMA purposes. This tends to be more efficient than the user memory that direct IO requires the sg driver to process into an internal scatter gather list. So on both these counts, mmap−ed IO has the edge over direct IO.

Chapter 10. Driver and module initialization

The size of the default reserved buffer can be specified when the sg driver is loaded. If it is built into the kernel then use:

```
    sg_def_reserved_size=<n>
```

on the boot line (only supported in 2.4 kernels).

If sg is a module, it can be loaded with **modprobe** in either manner:

```
    modprobe sg
    modprobe sg def_reserved_size=<n>
```

In the second case "<n>" is an integer (non negative). The default value is the value of the SG_DEF_RESERVED_SIZE defined in sg.h . This is currently 32768.

If sg is a module, it can be unloaded with **rmmod** like this:

```
    rmmod sg
```

However if there is a file descriptor still open with the sg driver (or there is an outstanding request awaiting a response) then the sg module is considered to be busy and can't be unloaded.

Chapter 11. Sg and the "proc" file system

The sg driver provides information about the SCSI subsystem and the current internal state of the sg driver in the `/proc/scsi/sg` directory. Some sg driver defaults can be changed by super user writing values to these "pseudo" files [17].

The following files which are readable by all:

```
allow_dio       0 indicates direct IO disable, 1 for enabled
debug           debug information including active request data
def_reserved_size  default buffer size reserved for each file descriptor
devices         one line of numeric data per device
device_hdr      single line of column names corresponding to 'devices'
device_strs     one line of vendor, product and rev info per device
hosts           one line of numeric data per host
host_hdr        single line of column names corresponding to 'hosts'
host_strs       one line of host information (string) per host
version         sg version as a number followed by a string representation
```

Each line in 'devices' and 'device_strs' corresponds to an sg device. For example the first line corresponds to `/dev/sg0`. The line number (origin 0) also corresponds to the sg minor device number. This mapping is local to sg and is normally the same as given by th **cat /proc/scsi/scsi** command which is reported by the SCSI mid level driver. The two mappings may diverge when 'remove−single−device' and 'add−single−device' are used (see the SCSI−2.4−HOWTO for more information).

Each line in 'hosts' and 'host_strs' corresponds to a SCSI host. For example the first line corresponds to the host normally represented as "scsi0". This mapping is invariant across the SCSI sub system. [So these entries could arguably be migrated to the mid level.]

The column headers in 'device_hdr' are given below. If the device is not present (and one is present after it) then a line of "−1" entries is output. Each entry is separated by a whitespace (currently a tab):

```
host            host number (indexes 'hosts' table, origin 0)
chan            channel number of device
id              SCSI id of device
lun             Logical Unit number of device
type            SCSI type (e.g. 0->disk, 5->cdrom, 6->scanner)
opens           number of opens (by sd, sr, sr and sg) at this time
depth           maximum queue depth supported by device
busy            number of commands being processed by host for this device
online          1 indicates device is in normal online state, 0->offline
```

A SCSI device is set offline by the SCSI mid level when it decides that a device is no longer responding (e.g. the device does not respond to an SCSI INQUIRY command after it has been reset).

The column headers in 'host_hdr' are given below. Each entry is separated by a whitespace (currently a tab):

```
uid             unique id (non−zero if multiple hosts of same type)
busy            number of commands being processed for this host
cpl             maximum number of command per lun (may be 0 if "device depth"
                is given
sgat            maximum elements of scatter gather the adapter (pseudo)
                DMA can accommodate
isa             0 -> non−ISA adapter, 1 -> ISA adapter. ISA adapters are
                assumed to have a 24 bit address bus limit (16 MB).
emu             0 -> real SCSI adapter, 1 -> emulated SCSI adapter
```

```
                (e.g. ide-scsi device driver)
```

The 'def_reserved_size' is both readable and writable. It is only writable by root. It is initialized to the value of DEF_RESERVED_SIZE in the "sg.h" file. Values between 0 and 1048576 (which is 2 ** 20) are accepted and can be set from the command line with the following syntax:

```
$ echo "262144" > /proc/scsi/sg/def_reserved_size
```

Note that the actual reserved buffer associated with a file descriptor could be less than 'def_reserved_size' if appropriate memory is not available. If the sg driver is compiled into the kernel (but not when it is a module) this value can also be read at /proc/sys/kernel/sg−big−buff . This latter feature is deprecated.

The 'allow_dio' is both readable and writable. It is only writable by root. When it is 0 (default) any request to do direct IO (i.e. by setting SG_FLAG_DIRECT_IO) will be ignored and indirect IO will be done instead.

11.1. /proc/scsi/sg/debug

This appendix explains the output from the `/proc/scsi/sg/debug` which is typically viewed by the command **cat /proc/scsi/sg/debug**. Below is the (slightly abridged) output while this command: **sgp_dd if=/dev/sg0 of=/dev/null bs=512** is executing on the system. That sgp_dd command is using command queuing to read a disk (and the data is written to `/dev/null` which forgets it).

```
$ cat /proc/scsi/sg/debug
dev_max(currently)=7 max_active_device=1 (origin 1)
 scsi_dma_free_sectors=416 sg_pool_secs_aval=320 def_reserved_size=32768
 >>> device=sg0 scsi0 chan=0 id=0 lun=0   em=0 sg_tablesize=255 excl=0
   FD(1): timeout=60000ms bufflen=65536 (res)sgat=2 low_dma=0
   cmd_q=1 f_packid=1 k_orphan=0 closed=0
     fin: id=3949312 blen=65536 dur=10ms sgat=2 op=0x28
     act: id=3949440 blen=65536 t_o/elap=60000/10ms sgat=2 op=0x28
     rb>> act: id=3949568 blen=65536 t_o/elap=60000/10ms sgat=2 op=0x28
     act: id=3949696 blen=65536 t_o/elap=60000/0ms sgat=2 op=0x28
```

Those items output above that are significant to user applications are described below.

Broadly speaking the above output shows everything is going fine. Four SCSI READ(10) commands (SCSI opcode 0x28) for different ids are underway. Three commands are active while one is finished with its status and data read() and the request structure is pending deletion. The "id" corresponds to the pack_id given in the sg_io_hdr structure (or the sg_header structure). In the case if sgp_dd the pack_id value is the block number being given to the SCSI READ (or WRITE). You will notice the 4 ids are 128 apart.

The ">>>" line shows the sg device name followed by the linux scsi adapter, channel, scsi id and lun numbers. The "em=" argument indicates whether the driver emulates a SCSI HBA. The ide−scsi driver would set "em=1". The "sg_tablesize" is the maximum number of scatter gather elements supported by the adapter driver. The "excl=0" indicates no sg open() on this device is currently using the O_EXCL flag.

The next two lines starting with "FD(1)" supply data about the first (and only in this case) open file descriptor on `/dev/sg0`. The default timeout is 60 seconds however this is only significant if the sg_header interface is being used since the sg_io_hdr interface explicits sets the timeout on a per command basis. "bufflen=65536" is the reserved buffer size for this file descriptor. The "(res)sgat=2" indicates that this reserved buffer requires 2 scatter gather elements. The "low_dma" will be set to 1 for ISA HBAs indicating only the bottom 16 MB of RAM can be used for its kernel buffers. The "cmd_q=1" indicates command queuing is being allowed. The "f_packid=1" indicates the SG_SET_FORCE_PACK_ID mode is on. The "k_orphan" value is 1 in the rare cases when a SG_IO is interrupted while a SCSI command is "in flight". The "closed" value is 1 in the rare cases the file descriptor has been closed while a SCSI command is "in flight".

Each line indented with 5 spaces represents a SCSI command. The state of the command is either:

- prior: command hasn't been sent to mid level (rare)
- act: mid level (adapter driver or device) has command
- rcv: sg bottom half handler has received response to this command (awaiting read() or SG_IO ioctl to complete
- fin: SCSI response (and optionally data) has been or is being read but the command data structures have not been removed

These states can be optionally prefixed by "rb>>" which means the reserved buffer is being used, "dio>>" which means this command is using direct IO, or "mmap>>" which means that mmap−ed IO is being used by this command. The "id" is the pack_id from this command's interface structure. The "blen" is the buffer length used by the data transfer associated with this command. For commands that a response has been received "dur" shows its duration in milliseconds. For commands still "in flight" an indication of "t_o/elap=60000/10ms" means this command has a timeout of 60000 milliseconds of which 10 milliseconds has already elapsed. The "sgat=2" argument indicates that this command's "blen" requires 2 scatter gather elements. The "op" value is the hexadecimal value of the SCSI command being executed.

If sg has lots of activity then the "debug" output may span many lines and in some cases appear to be corrupted. This occurs because procfs requests fixed buffer sizes of information and, if there is more data to output, returns later to get the remainder. The problem with this strategy is that sg's internal state may have changed. Rather than double buffering, the sg driver just continues from the same offset. While procfs is very useful, ioctl()s (such as SG_GET_REQUEST_TABLE) still have their place.

Chapter 12. Asynchronous usage of sg

It is recommended that synchronous sg−based applications use the new SG_IO ioctl() command. Existing applications (which are mainly synchronous) can continue to use the older sg_header based interface which is still supported.

Asynchronous usage allows multiple SCSI commands to be queued up to the device. If the device supports command queuing then there can be a major performance gain. Even if the device doesn't support command queuing (or is temporarily busy) then queuing up commands in the mid level or the host driver can be a minor performance win (since there will be a lower latency to transmit the next command when the device becomes free).

Asynchronous usage usually starts with setting the O_NONBLOCK flag on open() [or thereafter by using the fcntl(fd, SETFD, old_flags | O_NONBLOCK) system call]. A similar effect can be obtained without using O_NONBLOCK when POSIX threads are used. There are several strategies that can then be followed:

1. set O_NONBLOCK and use a poll() loop
2. set O_NONBLOCK and use SIGPOLL signal to alert app when readable
3. use POSIX threads and a single sg file descriptor
4. use POSIX threads and multiple sg file descriptors to same device

The O_NONBLOCK flag also permits open(), write() and read() [but not the ioctl(SG_IO)] to access a SCSI device even though it has been marked offline. SCSI devices are marked offline when they are detected and don't respond to the initial SCSI commands as expected, or, some SCSI error condition is detected on that device and the mid level error recovery logic is unable to "resurrect" the device. A SCSI device that is being reset (and still settling) could be accessed during this period by using the O_NONBLOCK flag; this could lead to unexpected behaviour so the sg user should take care.

In Linux SIGIO and SIGPOLL are the same signal. If POSIX real time signals are used (e.g. when SA_SIGINFO is used with sigaction() and fcntl(fd, F_SETSIG, SIGRTMIN + <n>)) then the file descriptor with which the signal is associated is available to the signal handler. The associated file descriptor is in the si_fd member of the siginfo_t structure. The poll() system call that is often used after a signal is received can thus be bypassed.

Appendix A. Sg3_utils package

The sg3_utils package is a collection of programs that use the sg interface. The utilities can be categorized as follows:

- variants of the Unix **dd** command: sg_dd, sgp_dd, sgq_dd and sgm_dd,
- scanning and mapping utilities: sg_scan, sg_map and scsi_devfs_scan,
- SCSI support: sg_inq, scsi_inquiry, sginfo, sg_readcap, sg_start and sg_reset,
- timing and testing: sg_rbuf, sg_test_rwbuf, sg_read, sg_turs and sg_debug,
- example programs: sg_simple1..4 and sg_simple16,

The "dd" family of utilities take a sg device file name as input (i.e. if=<sg_dev_filen_name>), as output of both. They can also take raw device file names [18] instead of sg device file names. One important difference from the standard **dd** command is that the value given to the block size (bs=) argument must be the exact block size of that device and not a integral multiple as allowed by **dd**. These "dd" variants are suitable for SCSI Direct Access Devices such as disk and CDROMs (but are not suitable for SCSI tape devices).

The sg3_utils package is designed to be used with the sg version 3 driver found in the lk 2.4 series. There is also a sg_utils package that supports a subset of these commands for the sg version 2 driver (with some support for the original sg driver) which is found in the lk 2.2 series (from and after lk 2.2.6). There are links to the most recent sg3_utils (and sg_utils) packages at the sg website at `www.torque.net/sg`. There are tarballs and both source and binary rpm packages. At the time of writing the latest sg3_utils tarball is at `www.torque.net/sg/p/sg3_utils-0.97.tgz`. There is a README file in that tarball that should be examined for up to date information. The more important utility commands (e.g. sg_dd) have "man" pages. [19]

Almost all of the sg device driver capabilities discussed in this document appear in code in one or more of these programs. For example the recently added mmap–ed IO can be found in sgm_dd, sg_read and sg_rbuf.

The sg3_utils package also provides some functions that may be useful for applications that use sg. The functions declared in `sg_err.h` and defined in `sg_err.c` categorize SCSI subsystem errors that are returned to an application in a read() or a ioctl(SG_IO). In the case of sense buffers, they are decoded into text message (as per SCSI 2 definitions). There is also a function to do a 64 bit seek (`llseek.h`).

Appendix B. sg_header, the original sg control structure

Following is the original interface structure of the sg driver that dates back to 1991. Those field elements with a "[o]+" are added by the sg version 2 driver which was first placed in lk 2.2.6 in April 1999.

```
struct sg_header
{
    int pack_len;    /* [o] */
    int reply_len;   /* [i] */
    int pack_id;     /* [i->o] */
    int result;      /* [o] */
    unsigned int twelve_byte:1;     /* [i] */
    unsigned int target_status:5;   /* [o]+ */
    unsigned int host_status:8;     /* [o]+ */
    unsigned int driver_status:8;   /* [o]+ */
    unsigned int other_flags:10;    /* unused */
    unsigned char sense_buffer[SG_MAX_SENSE]; /* [o] */
};      /* This structure is 36 bytes long on i386 */
```

SCSI commands are sent via write() calls to an sg device name (e.g. /dev/sg0). The data written to write() is of the form <a_sg_header_obj + scsi_command [+ data_to_write]>. The "data_to_write" component is only needed for SCSI commands that transfer data towards the SCSI device. The corresponding read() to the sg device name will yield data of the form <a_sg_header_obj [+ data_to_read]>.

This interface is fully described in the `www.torque.net/sg/p/scsi-generic.txt` file which documents the sg version 2 driver.

Since many Linux applications use this interface, it is still supported in this version (i.e. version 3) of the driver. Only its most perverse idiosyncrasies have been modified and no major applications have reported any problems running old applications atop this newer driver.

Appendix C. Programming example

This appendix contains an example program. It is an abridged version of sg_simple2.c found in the sg3_utils package. It send a SCSI INQUIRY command to the nominated sg device and prints out some of the response or outputs error information. Hopefully showing the error processing does not cloud what is being illustrated.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <scsi/sg.h> /* take care: fetches glibc's /usr/include/scsi/sg.h */

/* This is a simple program executing a SCSI INQUIRY command using the
   sg_io_hdr interface of the SCSI generic (sg) driver.

*  Copyright (C) 2001 D. Gilbert
*  This program is free software.   Version 1.00 (20011204)
*/

#define INQ_REPLY_LEN 96
#define INQ_CMD_CODE 0x12
#define INQ_CMD_LEN 6

int main(int argc, char * argv[])
{
    int sg_fd, k;
    unsigned char inqCmdBlk[INQ_CMD_LEN] =
                    {INQ_CMD_CODE, 0, 0, 0, INQ_REPLY_LEN, 0};
/* This is a "standard" SCSI INQUIRY command. It is standard because the
 * CMDDT and EVPD bits (in the second byte) are zero. All SCSI targets
 * should respond promptly to a standard INQUIRY */
    unsigned char inqBuff[INQ_REPLY_LEN];
    unsigned char sense_buffer[32];
    sg_io_hdr_t io_hdr;

    if (2 != argc) {
        printf("Usage: 'sg_simple0 <sg_device>'\n");
        return 1;
    }
    if ((sg_fd = open(argv[1], O_RDONLY)) < 0) {
        perror("error opening given file name");
        return 1;
    }
    /* It is prudent to check we have a sg device by trying an ioctl */
    if ((ioctl(sg_fd, SG_GET_VERSION_NUM, &k) < 0) || (k < 30000)) {
        printf("%s is not an sg device, or old sg driver\n", argv[1]);
        return 1;
    }
    /* Prepare INQUIRY command */
    memset(&io_hdr, 0, sizeof(sg_io_hdr_t));
    io_hdr.interface_id = 'S';
    io_hdr.cmd_len = sizeof(inqCmdBlk);
    /* io_hdr.iovec_count = 0; */  /* memset takes care of this */
    io_hdr.mx_sb_len = sizeof(sense_buffer);
    io_hdr.dxfer_direction = SG_DXFER_FROM_DEV;
    io_hdr.dxfer_len = INQ_REPLY_LEN;
    io_hdr.dxferp = inqBuff;
```

```
    io_hdr.cmdp = inqCmdBlk;
    io_hdr.sbp = sense_buffer;
    io_hdr.timeout = 20000;     /* 20000 millisecs == 20 seconds */
    /* io_hdr.flags = 0; */     /* take defaults: indirect IO, etc */
    /* io_hdr.pack_id = 0; */
    /* io_hdr.usr_ptr = NULL; */

    if (ioctl(sg_fd, SG_IO, &io_hdr) < 0) {
        perror("sg_simple0: Inquiry SG_IO ioctl error");
        return 1;
    }

    /* now for the error processing */
    if ((io_hdr.info & SG_INFO_OK_MASK) != SG_INFO_OK) {
        if (io_hdr.sb_len_wr > 0) {
            printf("INQUIRY sense data: ");
            for (k = 0; k < io_hdr.sb_len_wr; ++k) {
                if ((k > 0) && (0 == (k % 10)))
                    printf("\n  ");
                printf("0x%02x ", sense_buffer[k]);
            }
            printf("\n");
        }
        if (io_hdr.masked_status)
            printf("INQUIRY SCSI status=0x%x\n", io_hdr.status);
        if (io_hdr.host_status)
            printf("INQUIRY host_status=0x%x\n", io_hdr.host_status);
        if (io_hdr.driver_status)
            printf("INQUIRY driver_status=0x%x\n", io_hdr.driver_status);
    }
    else {  /* assume INQUIRY response is present */
        char * p = (char *)inqBuff;
        printf("Some of the INQUIRY command's response:\n");
        printf("    %.8s  %.16s  %.4s\n", p + 8, p + 16, p + 32);
        printf("INQUIRY duration=%u millisecs, resid=%d\n",
               io_hdr.duration, io_hdr.resid);
    }
    close(sg_fd);
    return 0;
}
```

The sg_simple4.c program is an example of using mmap−ed IO in the sg3_utils package. An example of using direct IO can be found in sg_rbuf.c in the same package.

Appendix D. Debugging

There are various ways to debug what is happening with the sg driver. The information provided in the `/proc/scsi/sg` directory can be useful, especially the `debug` pseudo file. It outputs the state of the sg driver when it is called. Invoking it at the right time can be a challenge. One approach (used in SANE) is to invoke the system() system call like this:

```
    system("cat /proc/scsi/sg/debug");
```

at appropriate times within an application that is using the sg driver.

Another debugging technique is to trace all system calls a program makes with the **strace** command (see its "man" page). This command can also be used to obtain timing information (with the "−r" and "t" options).

To debug the sg driver itself then the kernel needs to be built with CONFIG_SCSI_LOGGING selected. Then copious output will be sent by the sg driver whenever it is invoked to the log (normally `/var/log/messages`) and/or the console. This debug output is turned on by:

```
 $ echo "scsi log timeout 7" > /proc/scsi/scsi
```

As the number (i.e. 7) is reduced, less output is generated. To turn off this type of debugging use:

```
 $ echo "scsi log timeout 0" > /proc/scsi/scsi
```

If you want the system to log SCSI (CHECK_CONDITION related) errors that sg detects rather than process them within the application using sg then set ioctl(SG_SET_DEBUG) to a value greater than zero. Processing SCSI errors within the application using sg is my preference.

Appendix E. Other references

The primary site for SCSI information, standards (draft and emerging) and related reseources is `www.t10.org`.

The most recent news on the sg driver can be found at: `www.torque.net/sg` .

Some notes on the sg v3 driver can be found at: `www.torque.net/sg/s_packet.html` . For some timings (and CPU utilizations) comparisons between direct and indirect IO see: `www.torque.net/sg/rbuf_tbl.html`

The Linux Documentation Project's SCSI–2.4–HOWTO may help to put this driver into perspective: `linuxdoc.org/HOWTO/SCSI-2.4-HOWTO` . The most recent version of that document can be found at `www.torque.net/scsi/SCSI-2.4-HOWTO` .

To understand the inner workings of device drivers there is a fine book called "Linux Device Drivers", second edition by Alessandro Rubini and Jonathan Corbet published by O'Reilly [ISBN 0–596–00008–1]. The authors and the publisher have unselfishly made this book available under the GNU Free Documentation License (version 1.1). It can be found in html at `www.oreilly.com/catalog/linuxdrive2/chapter/book` .

Notes

[1]
: SCSI command opcode 0x7f does allow for variable length commands but that is not supported in Linux currently.

[2]
: There is an sg version 3.0.19 which is an optional driver for the lk 2.2 series. It has the following limitations:

  - maximum size of SCSI commands is 12 bytes
  - sense buffer limited to 16 bytes
  - resid (residual data transfer count) is always 0
  - direct and mmap–ed IO not supported (defaults to indirect IO)

[3]
: Patches exist for sg to extend the number of SCSI devices past the 256 limit when the device file system (devfs) is being used.

[4] Linux kernel prior to 2.4.15 limited SCSI commands to a length of 12 bytes. In lk 2.4.15 this was raised to 16 bytes. However unless lower level drivers (e.g. aic7xxx) indicate that they can handle 16 byte commands (and few currently do) then the command is aborted with a DID_ABORT host status.

[5]
Some HBA − SCSI device combinations have difficulties with an odd valued dxfer_len . In some cases the operation succeeds but a DID_ERROR host status is returned. So unless there is a good reason, applications that want maximum portability should avoid an odd valued dxfer_len .

[6]
Whether aborting individual commands is supported or not is left to the adapter. Many adapters are unable to abort SCSI commands "in flight" because these details are handled in silicon by embedded processors in hardware. SCSI device or bus resets are required.

[7]
In some cases the sym53cxx driver reports a DID_ERROR when it internally rounds up an odd transfer length by 1. This is an example of a "non−error".

[8]
Unfortunately some adapters drivers report an incorrect number for 'resid'. This is due to some "fuzziness" in the internal interface definitions within the Linux scsi subsystem concerning the _exact_ number of bytes to be transferred. Therefore only applications tied to a specific adapter that is known to give the correct figure should use this feature. Hopefully this will be cleared up in the near future.

[9]
The command queuing capabilities of the SCSI device and the adapter driver should also be taken into account. To this end the sg_scsi_id::h_cmd_per_lun and sg_scsi_id::d_queue_depth values returned bu ioctl(SG_GET_SCSI_ID) may be useful. Also some devices that indicate in their INQUIRY response that they can accept command queuing react badly when queuing is actually attempted.

[10]
There is a small probability it will spend some time waiting for a command block to become available. In this case the wait is interruptible. If O_NONBLOCK is active then this scenario will cause a EAGAIN.

[11]
The sg driver does record that the mmap() system call has been invoked at least once on a file descriptor. This is not sufficient because the given 'length' may be too short for the current IO. Also the driver is unaware of munmap() calls so it could easily be tricked.

[12]
If ioctl(SG_SET_KEEP_ORPHAN) is set to 1 and a ioctl(SG_IO) operation is interrupted (e.g. by control−C by the user) then when the response arrives then the "num_waiting" will be incremented to indicate a read() can now pick up the response.

[13]
Here is the mapping from the SCSI opcode "group" (top 3 bits of opcode) to the assumed length (in lk 2.4.15):

```
unsigned char scsi_command_size[8] =
{
        6, 10, 10, 12,
        16, 12, 10, 10
};
```

The assumed length of group 4 commands changed from 12 to 16 in lk 2.4.15 reflecting support for 16 byte SCSI commands being added to the kernel.

[14]

Older SCSI adapters and some pseudo adapter drivers don't have DMA capability in which case the CPU is used to copy the data.

[15]

Unfortunately that setup time is large enough in some versions of the lk 2.4 series to adversely impact direct IO performance. Also memory malloc()−ed in the user space tends to be made up of discontinuous pages seen from the SCSI adapter. This requires the sg driver to build heavily splintered scatter gather lists which is less than desirable. This limits the maximum transfer size to [(max_scsi_adapter_scatter_gather_elements − 1) * PAGE_SIZE]. [This is a _different_ scatter gather mechanism to that which the user sees in the sg interface based on iovec.]

[16]

When a write() or ioctl(SG_IO) attempts mmap−ed IO there is no check performed that a prior mmap() system call has been performed. If no mmap() has been issued then random data is written to the device or data read from the device in inaccessible. Also once mmap() has been called on a file descriptor then all subsequent calls to ioctl(SG_SET_RESERVED_SIZE) will yield EBUSY.

[17]

One strange quirk is that the `/proc/scsi/sg` directory will not appear if there are no SCSI devices (or pseudo devices such as USB mass storage) attached to the system. The reason for this is that in the absence of SCSI devices, the SCSI mid level does not initialize the sg driver (even if it has been loaded as a module). When the sg driver is a module and the **rmmod sg** is successfully executed then the `/proc/scsi/sg` directory and its contents are removed.

[18]

Raw device names are of the form `/dev/raw/raw<n>` and can be bound to block devices (e.g. an IDE disk partition such as `/dev/hda3`). The binding is done with the **raw** command (see "man raw").

[19]

Although the author wrote most of these programs, initially to test facilities within the sg driver, some have been contributed by others. See `www.torque.net/sg/u_index.html` for more information.