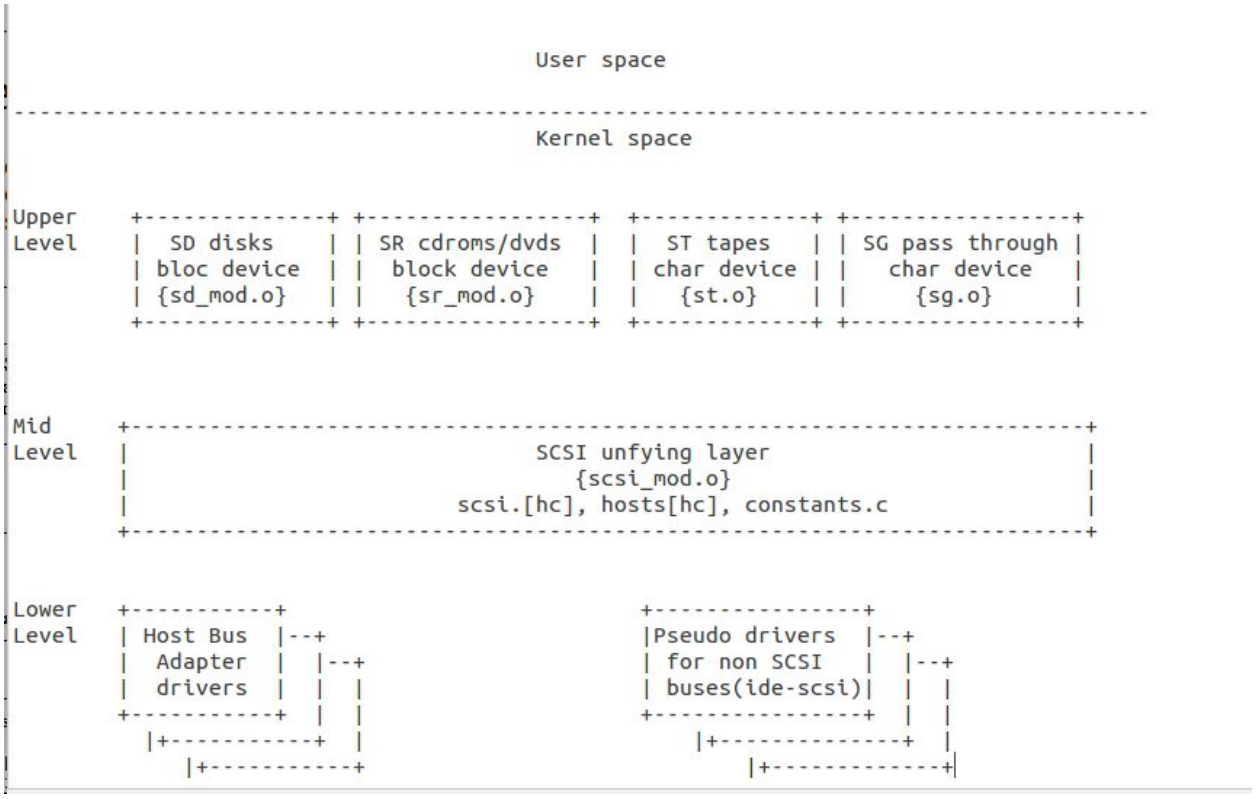
Linux and SCSI

The Linux SCSI subsystem has a 3-level architecture:
"upper" level is closest to the user/kernel interface while the "lower" level is closest to the hardware. The SCSI mid level provides common services to the upper and lower level drivers.

SCSI disks are labeled /dev/sda, /dev/sdb, /dev/sdc, etc. Once device initialization is complete, the Linux SCSI disk driver interfaces (sd) will send only SCSI READ and WRITE commands.

The upper level drivers are commonly known by two letter abbreviation.

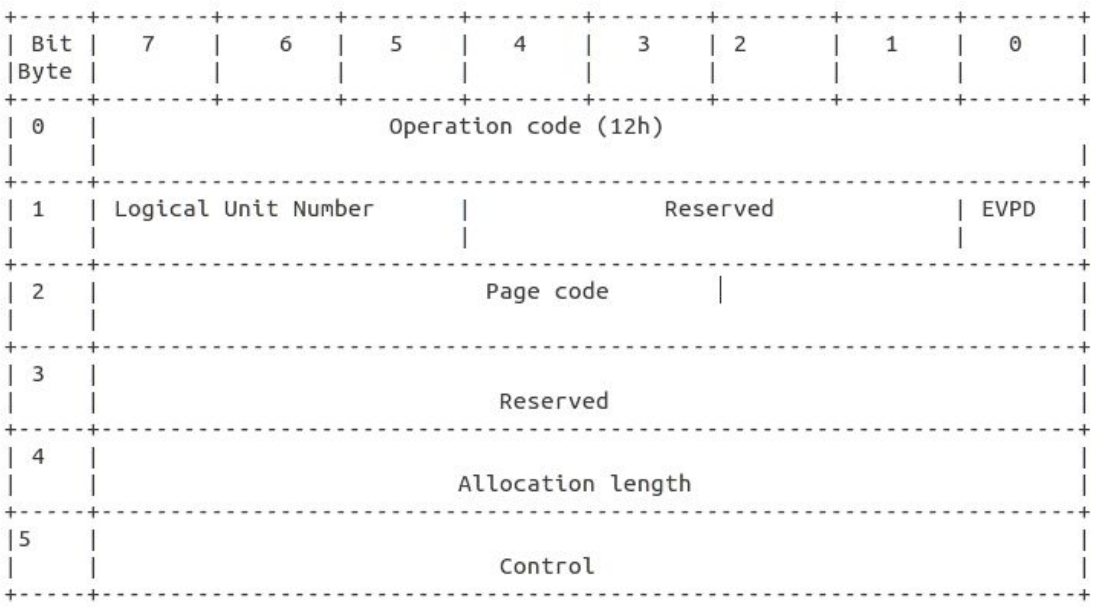


This presentation will focus on generic drivers(sg) - the sg driver permits user applications to send SCSI commands to devices that understand them. Each SCSI command is characterized by a Command Descriptor Block (CDB) - defining the operations that are allowed to be performed by a SCSI device.

Commonly used SCSI commands are: Inquiry, Test/Unit/Ready, READ, WRITE, Request Sense, Read Capacity.

In this presentation, the **INQUIRY** command may be used by a system to determine the configuration of the SCSI bus.

Inquiry command format



An EVPD (Enable Vital Product Data) bit of zero specifies that the target shall return the standard INQUIRY data.

SCSI generic driver supports many typical system calls for character device, such as open(), close(), read(), write, poll(), ioctl().
The procedure for sending SCSI commands to a specific SCSI device consists in few steps:

-open the SCSI generic device file
-prepare the SCSI command.
-set related memory buffer.
-call ioctl() function to execute the SCSI command.
-close the device file.

Ioctl() function could be written like this:

ioctl(sg_fd, SG_IO, point_io_hdr);

Where:
sg_fd is the file descriptor of the device file, and can be acquired after file is successfully opened by calling open()
SG_IO indicates that an sg_io_hdr object is handed as the third parameter of the ioctl() function and will return when the SCSI command is finished.
point_io_hdr is a pointer to the sg_io_hdr object

Beside ioctl(), SCSI generic driver supports many other typical system calls for character device: open() ; write() ; read() ; poll() ; close() ; mmap() ; fcntl(sg_fd, F_SETFL, oflags | FASYNC) ; ioctl(); Errors reported in errno.

A user application can access the sg driver by using the open() system call on sg device file name. Each sg device file name corresponds to one attached SCSI device, usually found under /dev.

```
root@Rinzler:~#
root@Rinzler:~# ls -l /dev/sg[01]
crw-rw----  1 root disk  21, 0 Apr 10 18:54 /dev/sg0
crw-rw----+ 1 root cdrom 21, 1 Apr 10 18:54 /dev/sg1
root@Rinzler:~#
root@Rinzler:~#
```

Before continuing, let's first try few Linux commands for gathering information about SCSI devices. We will need one or two of them for comparing the output of source code.

1) List information about scsi devices

```
root@Rinzler:~#
root@Rinzler:~# lsscsi
[0:0:0:0]    disk    ATA      QEMU HARDDISK    0     /dev/sda
[1:0:0:0]    cd/dvd  QEMU     QEMU DVD-ROM     2.0.  /dev/sr0
root@Rinzler:~#
root@Rinzler:~#
root@Rinzler:~#
```

2) Same as 1), but this time using /proc filesystem

```
root@Rinzler:~#
root@Rinzler:~# ls /proc/scsi
device_info  scsi  sg
root@Rinzler:~#
root@Rinzler:~# ls /proc/scsi/sg
allow_dio  debug  def_reserved_size  device_hdr  devices  device_strs  version
root@Rinzler:~#
root@Rinzler:~# ls -ltr /proc/scsi/sg/device_strs
-r--r--r-- 1 root root 0 Apr 10 23:59 /proc/scsi/sg/device_strs
root@Rinzler:~#
root@Rinzler:~# more  /proc/scsi/sg/device_strs
ATA             QEMU HARDDISK           0
QEMU            QEMU DVD-ROM            2.0.
root@Rinzler:~#
root@Rinzler:~#
root@Rinzler:~#
```

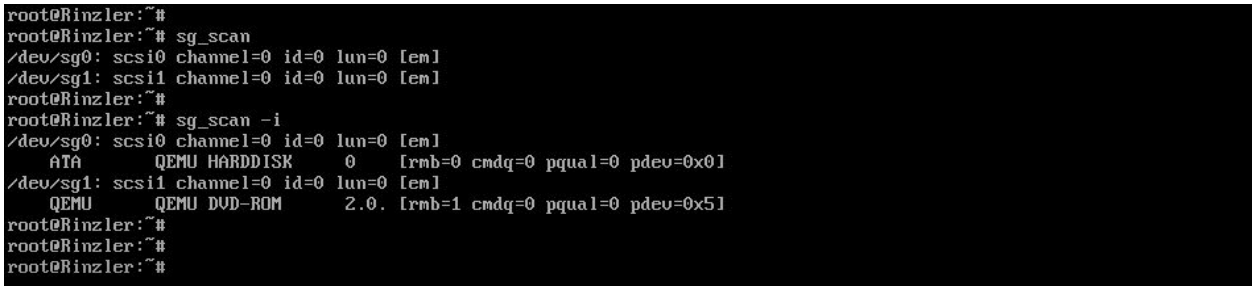
Or…

```
root@Rinzler:~#
root@Rinzler:~# more /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: ATA      Model: QEMU HARDDISK    Rev: 0
  Type:   Direct-Access                    ANSI  SCSI revision: 05
Host: scsi1 Channel: 00 Id: 00 Lun: 00
  Vendor: QEMU     Model: QEMU DVD-ROM     Rev: 2.0.
  Type:   CD-ROM                           ANSI  SCSI revision: 05
root@Rinzler:~#
root@Rinzler:~#
```

3) Information about the device names the Linux kernel uses for a specific scsi ID -- use /sys filesystem.

In this example, the information is listed for scsi device with ID 0:0:0:0

4) Using sg_3 utils



Details on sg_io_hdr_t structure

The main control structure for the SCSI generic driver is **struct sg_io_hdr,** and contains information on how to use the SCSI command.

```
typedef struct sg_io_hdr
 {
      int interface_id;          /* [i] 'S' for SCSI generic (required) */
      int dxfer_direction;       /* [i] data transfer direction  */
      unsigned char cmd_len;     /* [i] SCSI command length */
      unsigned char mx_sb_len;   /* [i] max length to write to sbp */
      unsigned short iovec_count; /* [i] 0 implies no scatter gather */
      unsigned int dxfer_len;    /* [i] byte count of data transfer */
      void __user *dxferp;       /* [i], [*io] points to data transfer memory
                                            or scatter gather list */
      unsigned char __user *cmdp; /* [i], [*i] points to command to perform */
      void __user *sbp;          /* [i], [*o] points to sense_buffer memory */
      unsigned int timeout;      /* [i] MAX_UINT->no timeout (unit: millisec) */
      unsigned int flags;        /* [i] 0 -> default, see SG_FLAG... */
      int pack_id;               /* [i->o] unused internally (normally) */
     void __user * usr_ptr;      /* [i->o] unused internally */
      unsigned char status;      /* [o] scsi status */
      unsigned char masked_status;/* [o] shifted, masked scsi status */
      unsigned char msg_status;   /* [o] messaging level data (optional) */
      unsigned char sb_len_wr;   /* [o] byte count actually written to sbp */
      unsigned short host_status; /* [o] errors from host adapter */
      unsigned short driver_status;/* [o] errors from software driver */
      int resid;                 /* [o] dxfer_len - actual_transferred */
      unsigned int duration;     /* [o] time taken by cmd (unit: millisec) */
      unsigned int info;         /* [o] auxiliary information */
  } sg_io_hdr_t;
```

Where, [*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.

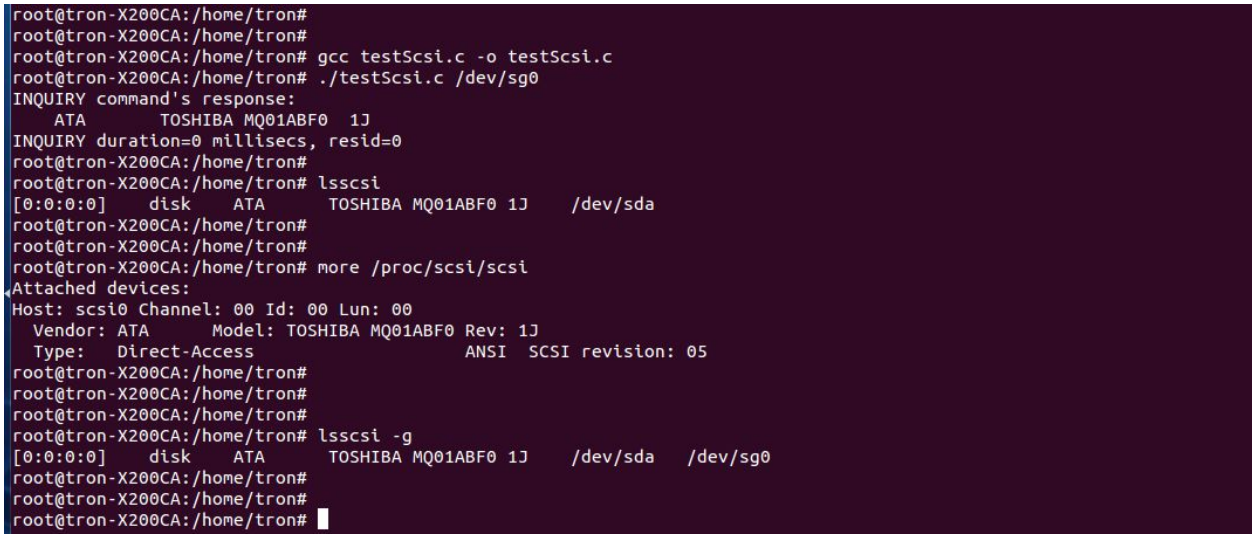
>as defined in Linux Cross Reference:
http://lxr.free-electrons.com/source/include/scsi/sg.h

Source code:

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <scsi/sg.h>

#define TIMEOUT 20000
#define INQ_REPLY_LEN 0xff
#define INQ_CMD_CODE 0x12
#define INQ_CMD_LEN 6
#define MAX 32


int main(int argc, char **argv)
{
      int sg_fd, k, evpd, page_code;
      unsigned char InquiryCMD[INQ_CMD_LEN] = {INQ_CMD_CODE, evpd & 1 ,
page_code & 0xff , 0, INQ_REPLY_LEN , 0};
      unsigned char InquiryBuffer[INQ_REPLY_LEN];
      unsigned char sense_buffer[MAX];

      sg_io_hdr_t point_io_hdr;

      if ((sg_fd = open(argv[1], O_RDONLY)) < 0) {
      perror("error opening given file name");
      return 1;
      }

      /* Prepare INQUIRY command */

      memset(&point_io_hdr, 0, sizeof(sg_io_hdr_t));

      point_io_hdr.interface_id = 'S';
      point_io_hdr.cmd_len = sizeof(InqueryCMD);
      point_io_hdr.mx_sb_len = sizeof(sense_buffer);
      point_io_hdr.dxfer_direction = SG_DXFER_FROM_DEV;
      point_io_hdr.dxfer_len = INQ_REPLY_LEN;
      point_io_hdr.dxferp = InquiryBuffer;
```

```
      point_io_hdr.cmdp = InquiryCMD;
      point_io_hdr.sbp = sense_buffer;
      point_io_hdr.timeout = TIMEOUT;

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

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

Compile, run and compare :)



In the link, a more detailed description of sg_io_hdr_t structure:
http://sg.danny.cz/sg/p/scsi-generic_v3.txt

>>interface_id

This must be set to 'S'. If not, the ENOSYS error message is placed in errno.

>>dxfer_direction

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

```
SG_DXFER_NONE /* a SCSI Test Unit Ready command */
SG_DXFER_TO_DEV /*a SCSI WRITE command */
SG_DXFER_FROM_DEV /* a SCSI READ command */
SG_DXFER_TO_FROM_DEV
SG_DXFER_UNKNOWN
```

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.

>>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.

>>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.

>>iovec_count

This is the number of scatter gather elements in an array pointed to by 'dxferp'.

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;
```

>>dxfer_len

This is the number of bytes to be moved in the data transfer associated with the command.

>>dxferp

If 'iovec_count' is zero then this value is a pointer to user memory of at least 'dxfer_len' bytes in length.

>>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).

>>sbp

This value points to user memory of at least 'mx_sb_len' bytes length where the SCSI sense buffer will be output.

>>timeout

This value is used to timeout the given command. The units of this value are milliseconds.

>>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
```

>>sb_len_wr

This is the actual number of bytes written to the user memory pointed to by 'sbp'.

>>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 defined values are:

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 (
SG_ERR_DID_BAD_TARGET [0x04] BAD target, device not responding?
SG_ERR_DID_ABORT [0x05] Told to abort for some other reason.
SG_ERR_DID_PARITY [0x06] Parity error.
SG_ERR_DID_ERROR [0x07] Internal error detected in the host adapter.
SG_ERR_DID_RESET [0x08] The SCSI bus (or this device) has been 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

>>driver_status

One driver can potentially control several host adapters.

>>unsigned int info;
This value is designed to convey useful information back to the user about the associated request. 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 */
```

SG_INFO_OK_MASK - defined in Linux Cross Reference, as preprocessor macro:
http://lxr.free-electrons.com/source/include/scsi/sg.h#L94
SG_INFO_OK - defined in Linux Cross Reference, as preprocessor macro:
http://lxr.free-electrons.com/source/include/scsi/sg.h#L95
SG_INFO_CHECK - defined in Linux Cross Reference:

http://lxr.free-electrons.com/source/include/scsi/sg.h?v=2.6.33#L133

System Calls

-> open() ; write() ; read() ; poll() ; close() ; mmap() ; fcntl(sg_fd, F_SETFL, oflags | FASYNC) ; ioctl(); Errors reported in errno

>>open(const char * filename, int flags) -- The filename should be a sg device file name. Flags can be as following:

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) .

>>write()

write(int sg_fd, const void * buffer, size_t count).
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.

>>read()

read(int sg_fd, void * buffer, size_t count).
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.

>>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".

>>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.

>>ioctl()

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.

All sg driver ioctl() 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.

In our case: SG_IO
SG_IO 0x2285 -- hands 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 following SCSI commands will be permitted by SG_IO when the sg file descriptor was opened O_RDONLY:

TEST UNIT READY
REQUEST SENSE
INQUIRY
READ CAPACITY
READ BUFFER
READ(6) (10) and (12)
MODE SENSE(6) and (10)
LOG SENSE

Asynchronous usage of sg:
-recommended that synchronous sg-based applications use the new SG_IO ioctl() command
-asynchronous usage allows multiple SCSI commands to be queued up to the device.

Buh-bye for now!