

Tour the Linux generic SCSI driver

Dive into the Linux generic SCSI driver API and surface with a usage example

Skill Level: Intermediate

[Mao Yuan Tao \(taomaoy@cn.ibm.com\)](mailto:taomaoy@cn.ibm.com)

Software Engineer

IBM

25 Feb 2009

Computers control and transfer data to SCSI devices via SCSI commands. In this article, the author introduces some of the SCSI commands and methods of executing SCSI commands when using SCSI API in Linux®. He provides background on the SCSI client/server model and the storage SCSI command. Next, he explains the Linux generic SCSI driver API and offers an example of using a system that focuses on executing the inquiry command using the generic driver.

The SCSI client/server model

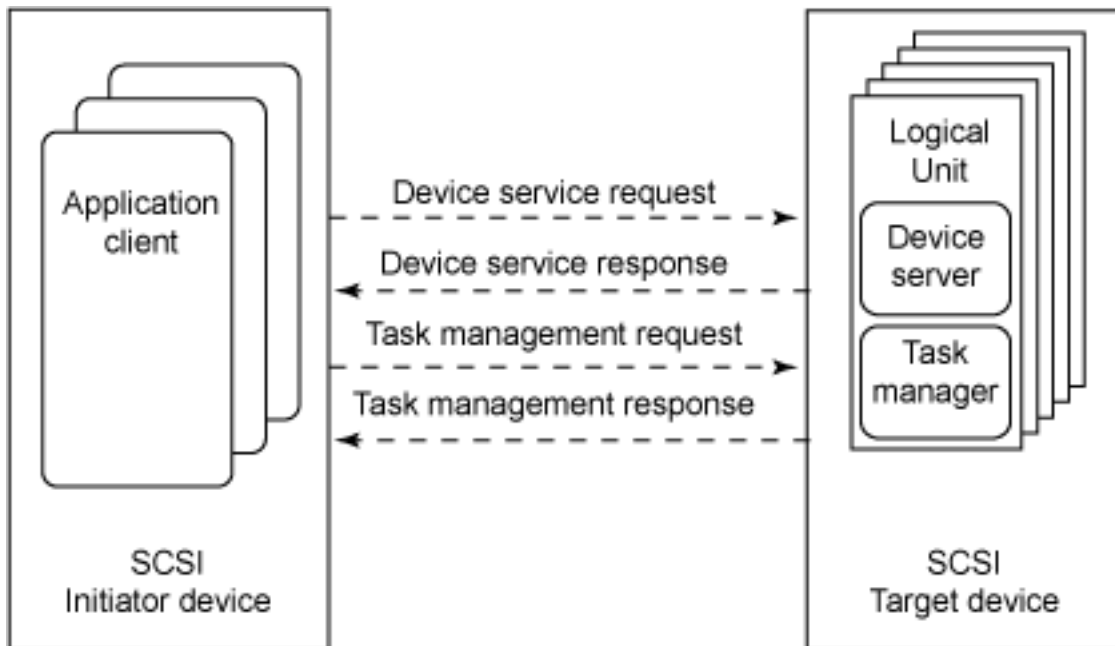
During the communication between a host and storage, a host generally acts as a *SCSI initiator*. In computer storage, the SCSI initiator is the endpoint that initiates a SCSI session, meaning it sends a SCSI command. The storage often acts as a SCSI target that receives and processes SCSI commands. The SCSI target waits for the initiator's commands and then provides required input/output data transfers.

The target usually provides the initiators one or more *logical unit numbers* (LUN). In computer storage, a LUN is simply the number assigned to a logical unit. A logical unit is a SCSI protocol entity, the only one that may be addressed by the actual I/O operations. Each SCSI target provides one or more logical units; it does not perform I/O as itself, but on behalf of a specific logical unit.

In a storage area, a LUN often represents a SCSI disk on which a host can perform

a read and write operation. Figure 1 shows how the SCSI client/server model works.

Figure 1. The SCSI client/server model



The initiator first sends commands to the target, which decodes the command and then may request data from the initiator or send data to the initiator. After that, the target sends the status to the initiator. If the status is bad, then the initiator sends a request sense command to the target. The target returns the sense data to indicate what went wrong.

Now let's focus on storage-related SCSI commands.

Storage-related SCSI commands

Storage-related SCSI commands are defined mainly in the SCSI Architecture Model (SAM), SCSI Primary Commands (SPC), and SCSI Block Commands (SBC):

- **SAM** defines the SCSI systems model, the functional partitioning of the SCSI standard set, and the requirements applicable to all SCSI implementations and implementation standards.
- **SPC** defines behaviors that are common to all SCSI device models.
- **SBC** defines the command set extensions to facilitate operation of SCSI direct-access block devices.

Each SCSI command is described by a Command Descriptor Block (CDB), which defines the operations to be performed by the SCSI device. The SCSI commands

involve data commands that are used for transferring data from or to SCSI devices and non-data commands that request or set the configure parameters of a SCSI device. In Table 1, you can see the most commonly used commands.

Table 1. Most commonly used SCSI commands

Command	Description
Inquiry	Requests general information of the target device
Test/Unit/Ready	Checks whether the target device is ready for the transfer operation
READ	Transfers data from the SCSI target device
WRITE	Transfers data to the SCSI target device
Request Sense	Requests the sense data of the last command
Read Capacity	Requests the storage capacity information

All SCSI commands should begin with an operation code as the first byte. This indicates what operation it represents. All SCSI commands should also contain a control byte. This is typically the last byte of the command, used for vendor-specific information and other uses.

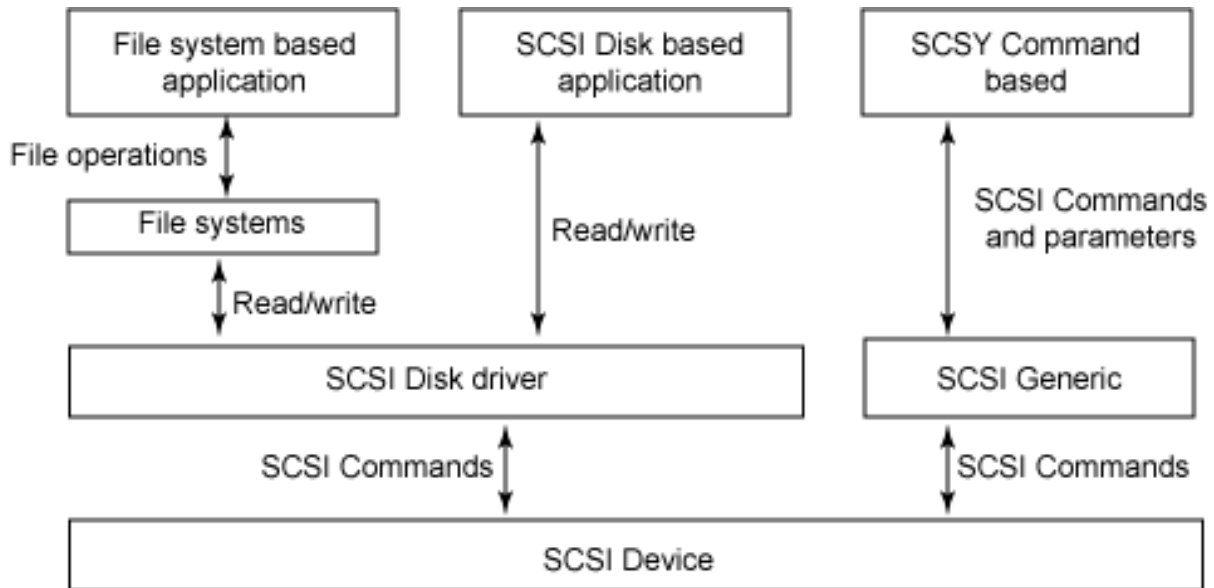
Now on to the generic SCSI driver.

The Linux generic SCSI driver

SCSI devices under Linux are often named to help the user identify the device. For example, the first SCSI CD-ROM is `/dev/scd0`. SCSI disks are labeled `/dev/sda`, `/dev/sdb`, `/dev/sdc`, etc. Once device initialization is complete, the Linux SCSI disk driver interfaces (`sd`) send only `SCSI READ` and `WRITE` commands.

These SCSI devices may also have generic names or interfaces, such as `/dev/sg0`, `/dev/sg1` or `/dev/sga`, `/dev/sgb`, etc. With these generic driver interfaces, you can directly send SCSI commands to SCSI devices, bypassing a file system that is normally created on a SCSI disk and mounted under a directory. In Figure 2, you can see how different applications communicate with SCSI devices.

Figure 2. The myriad ways of communicating with a SCSI device



With a Linux generic driver interface, you can build applications that can send more kinds of SCSI commands to SCSI devices. In other words, you have a choice. To determine which SCSI device stands for which sg interface, you can use the `sg_map` command to list the maps:

```
[root@taomaoy ~]# sg_map -i
/dev/sg0 /dev/sda ATA ST3160812AS 3.AA
/dev/sg1 /dev/scd0 HL-DT-ST RW/DVD GCC-4244N 1.02
```

With Red Hat or Fedora, `sg3_utils` should be installed. Let's take a look at how to perform a typical SCSI system call command.

The typical SCSI generic driver commands

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 is also very simple:

1. Open the SCSI generic device file (such as `sg1`) to get the file descriptor of SCSI device.
2. Prepare the SCSI command.
3. Set related memory buffers.
4. Call the `ioctl()` function to execute the SCSI command.
5. Close the device file.

A typical `ioctl()` function could be written like this:

```
ioctl(fd, SG_IO, p_io_hdr);
```

The `ioctl()` function here requires three parameters:

1. `fd` is the file descriptor of the device file. After the device file is successfully opened by calling `open()`, this parameter could be acquired.
2. `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.
3. The `p_io_hdr` is a pointer to the `sg_io_hdr` object, which contains the SCSI command and other settings.

The most important data structure for the SCSI generic driver is `struct sg_io_hdr`, which is defined in `scsi/sg.h` and contains information on how to use the SCSI command. Listing 1 shows the definition of the structure.

Listing 1. Definition of struct `sg_io_hdr`

```
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) */
```

Not all the fields in this structure are required, so only those that are commonly used are introduced here:

- `interface_id`: Should always be `S`.
- `dxfer_direction`: Used for data transfer direction; could be one of the

following values:

- `SG_DXFER_NONE`: No data transfer is needed. Example: a SCSI Test Unit Ready command.
- `SG_DXFER_TO_DEV`: Data is transferred to device. A SCSI WRITE command.
- `SG_DXFER_FROM_DEV`: Data is transferred from device. A SCSI READ command.
- `SG_DXFER_TO_FROM_DEV`: Data is transferred both ways.
- `SG_DXFER_UNKNOWN`: The data transfer direction is unknown.
- `cmd_len`: The length in bytes of `cmdp` that points to the SCSI command.
- `mx_sb_len`: The maximum size that can be written back to the `sbp` when a `sense_buffer` is output.
- `dxfer_len`: The length of the user memory for data transfer.
- `dxferp`: A pointer to user memory of at least `dxfer_len` bytes in length for data transfer.
- `cmdp`: A pointer to the SCSI command to be executed.
- `sbp`: A pointer to the sense buffer.
- `timeout`: Used to timeout the given command.
- `status`: SCSI status byte as defined by the SCSI standard.

In summary, when data is about to be transferred using this method, `cmdp` must point to SCSI CDB whose length is stored in `cmd_len`; `sbp` points to a user memory whose maximum length is `mx_sb_len`. In case an error occurs, the sense data would be written back to this place. `dxferp` points to a memory; the data would be transferred from or to the SCSI device depending on the `dxfer_direction`.

Finally, let's look at an inquiry command and how it would execute using the generic driver.

Example: Executing an inquiry command

An inquiry command is the most common SCSI command that all SCSI devices implement. This command is used to request the basic information of the SCSI device and is often used as a `ping` operation to test to see if the SCSI device is online. Table 2 shows how the SCSI standard is defined.

Table 2. The inquiry command format definition

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
byte 0	Operation code = 12h							
byte 1	LUN			Reserved			EVPD	
byte 2	Page code							
byte 3	Reserved							
byte 4	Allocation length							
byte 5	Control							

If the EVPD parameter bit (for *enable vital product data*) is zero and the Page Code parameter byte is zero, then the target will return the standard inquiry data. If the EVPD parameter is one, then the target will return vendor-specific data according to the page code fields.

Listing 2 shows the source code clips of using the SCSI generic API. Let's first look at the examples of setting `sg_io_hdr`.

Listing 2. Setting `sg_io_hdr`

```

struct sg_io_hdr * init_io_hdr() {
    struct sg_io_hdr * p_scsi_hdr = (struct sg_io_hdr
*)malloc(sizeof(struct sg_io_hdr));
    memset(p_scsi_hdr, 0, sizeof(struct sg_io_hdr));
    if (p_scsi_hdr) {
        p_scsi_hdr->interface_id = 'S'; /* this is the only choice we
have! */
        /* this would put the LUN to 2nd byte of cdb*/
        p_scsi_hdr->flags = SG_FLAG_LUN_INHIBIT;
    }
    return p_scsi_hdr;
}

void destroy_io_hdr(struct sg_io_hdr * p_hdr) {
    if (p_hdr) {
        free(p_hdr);
    }
}

void set_xfer_data(struct sg_io_hdr * p_hdr, void * data, unsigned
int length) {
    if (p_hdr) {
        p_hdr->dxferp = data;
        p_hdr->dxfer_len = length;
    }
}

void set_sense_data(struct sg_io_hdr * p_hdr, unsigned char * data,
unsigned int length) {
    if (p_hdr) {
        p_hdr->sbp = data;
        p_hdr->mx_sb_len = length;
    }
}

```

These functions are used for setting the `sg_io_hdr` object. Some of the fields point

to user space memory; when the execution is finished, inquiry output data from the SCSI command is copied into the memory where `dxferp` points to. If there is an error and the sense data is required, the sense data would be copied to where `sbp` points to. The example for sending an inquiry command to a SCSI target is shown in Listing 3.

Listing 3. Sending an inquiry command to a SCSI target

```
int execute_Inquiry(int fd, int page_code, int evpd, struct
sg_io_hdr * p_hdr) {
    unsigned char cdb[6];
    /* set the cdb format */
    cdb[0] = 0x12; /*This is for Inquiry*/
    cdb[1] = evpd & 1;
    cdb[2] = page_code & 0xff;
    cdb[3] = 0;
    cdb[4] = 0xff;
    cdb[5] = 0; /*For control filed, just use 0 */

    p_hdr->dxfer_direction = SG_DXFER_FROM_DEV;
    p_hdr->cmdp = cdb;
    p_hdr->cmd_len = 6;

    int ret = ioctl(fd, SG_IO, p_hdr);
    if (ret<0) {
        printf("Sending SCSI Command failed.\n");
        close(fd);
        exit(1);
    }
    return p_hdr->status;
}
```

So the function first prepares the CDB according to the inquiry standard format and then calls the `ioctl()` function, handing file descriptor `SG_IO`, and the `sg_io_hdr` object; the return status is stored in the `status` field of the `sg_io_hdr` object.

Now let's see how the application program uses this function to execute the inquiry command (Listing 4):

Listing 4. Application program executes the inquiry command

```
unsigned char sense_buffer[SENSE_LEN];
unsigned char data_buffer[BLOCK_LEN*256];
void test_execute_Inquiry(char * path, int evpd, int page_code) {
    struct sg_io_hdr * p_hdr = init_io_hdr();
    set_xfer_data(p_hdr, data_buffer, BLOCK_LEN*256);
    set_sense_data(p_hdr, sense_buffer, SENSE_LEN);
    int status = 0;
    int fd = open(path, O_RDWR);
    if (fd>0) {
        status = execute_Inquiry(fd, page_code, evpd, p_hdr);
        printf("the return status is %d\n", status);
        if (status!=0) {
            show_sense_buffer(p_hdr);
        } else{
            show_vendor(p_hdr);
            show_product(p_hdr);
            show_product_rev(p_hdr);
        }
    }
}
```

```

    }
    } else {
        printf("failed to open sg file %s\n", path);
    }
    close(fd);
    destroy_io_hdr(p_hdr);
}

```

The process of sending the SCSI command here is quite simple. First the user space data buffer and sense buffer should be allocated and made to point to the `sg_io_hdr` object. Then, open the device driver and get the file descriptor. With these parameters, the SCSI command could be sent to the target device. The output from the SCSI target would then be copied to the user space buffers when the command is finished.

Listing 5. Using parameters to send SCSI command to target device

```

void show_vendor(struct sg_io_hdr * hdr) {
    unsigned char * buffer = hdr->dxferp;
    int i;
    printf("vendor id:");
    for (i=8; i<16; ++i) {
        putchar(buffer[i]);
    }
    putchar('\n');
}

void show_product(struct sg_io_hdr * hdr) {
    unsigned char * buffer = hdr->dxferp;
    int i;
    printf("product id:");
    for (i=16; i<32; ++i) {
        putchar(buffer[i]);
    }
    putchar('\n');
}

void show_product_rev(struct sg_io_hdr * hdr) {
    unsigned char * buffer = hdr->dxferp;
    int i;
    printf("product ver:");
    for (i=32; i<36; ++i) {
        putchar(buffer[i]);
    }
    putchar('\n');
}

int main(int argc, char * argv[]) {
    test_execute_inquiry(argv[1], 0, 0);
    return EXIT_SUCCESS;
}

```

The standard response of the SCSI Inquiry Command (Page Code and EVPD fields are all set to 0) is complicated. According to the standard, vendor ID extends from the 8th to the 15th byte, product ID from the 16th to the 31st byte, and product version from the 32nd to the 35th byte. This information could be retrieved to check whether the command has been successfully executed.

After building this simple example, run it on `/dev/sg0`, which is normally the local

hard disk. You should get the following:

```
[root@taomaoy scsi_test]# ./scsi_test /dev/sg0
the return status is 0
vendor id:ATA
product id:ST3160812AS
product ver:3.AA
```

The result is the same as the `sg_map` tool reports.

Summary

Linux provides a generic driver for SCSI devices and an application programming interface so you can build applications to send SCSI commands directly to SCSI devices. You can manually make SCSI commands and set other related parameters in an `sg_io_hdr` object, then call `ioctl()` to execute their SCSI commands and get output from the same `sg_io_hdr` object.

Downloads

Description	Name	Size	Download method
Sample code for this article	scsi_test.zip	3KB	HTTP

[Information about download methods](#)

Resources

Learn

- Want more on Linux and the SCSI subsystem? Try these:
 - "[Anatomy of the Linux SCSI subsystem](#)" (developerWorks, November 2007) introduces the Linux SCSI subsystem and discusses where this subsystem is going in the future.
 - "[Anatomy of Linux synchronization methods](#)" (developerWorks, October 2007) outlines atomic synchronization operations (often used by SCSI drivers).
 - "[GCC hacks in the Linux kernel](#)" (developerWorks, November 2008) introduces the GNU Compiler Collection suite; in it you'll find an example of range extension in use in a SCSI switch block ([Listing 2](#)).
- [TC T10 SCSI Storage Interfaces](#) is an excellent repository of knowledge on I/O interfaces, especially SCSI-3 and SAS.
- And here is a wonderful library on the [Linux SCSI generic driver](#) that includes release updates, backgrounder, features list, device driver downloads, utilities, and related Web sites.
- In the [developerWorks Linux zone](#), find more resources for Linux developers (including developers who are [new to Linux](#)), and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and spaces.

About the author

Mao Yuan Tao

Mao Yuan Tao is a software engineer in IBM China System and Technology Lab (CSTL). His engineering background ranges from testing storage products to

configuring SAN environments (including FC switches, back-end storage, and hosts) and even developing test tools.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.