

# **Guide to the implementation of SCSI in Linux 2.6.X kernel**

By  
Samdeep Nayak

**Revision History:**

Rev No	Date	Comments	Author
0.1	12/11/2004	Initial Draft	Samdeep Nayak With feedback from Amir A Vetry and Randy Dunlap
0.2	01/08/05	Implemented Machtelt Garrels suggestions on the document style	Samdeep nayak with feedback from Machtelt Garrels

## **Abstract**

SCSI has emerged as a popular protocol in the storage world. This document is a guide to understanding the implementation of SCSI in the Linux 2.6.X kernel. This project analyzes the code implementing the SCSI protocol, provides the big picture involving the IO, explains the entry points, data structures and the exported functions. The project is intended as a reference for experimenters and developers who would be working on the SCSI module in Linux.

## Table of Contents

1.0 Introduction.....	6
1.1 Document Assumptions and Conventions.....	6
1.2 Copyright, License and Disclaimer.....	6
1.3 Acknowledgments.....	7
1.4 Feedback.....	7
2.0 Linux Overview of Block Data Transfer.....	8
2.1 Block Device Driver Upper Entry points.....	9
2.2 Block Device Driver Lower Entry points.....	10
2.3 Flow of an IO in the block device driver.....	11
2.4 Block Device Driver Data Structures.....	11
2.4.1 Struct block_device_operations.....	11
2.4.2 Struct request_queue.....	12
3.0 Overview of Linux SCSI Subsystem – Layered Architecture.....	14
4.0 Overview of SCSI Disk Driver (SD) – Upper Layer.....	16
4.1 SD functionality.....	16

4.1.1	sd_probe	17
4.1.2	sd_remove	17
4.1.3	sd_shutdown	17
4.1.4	sd_init_command	18
4.1.5	sd_open	18
4.1.6	sd_release	18
4.1.7	sd_ioctl	18
4.1.8	sd_media_change	19
4.1.9	sd_revalidate_disk	19
4.1.10	sd_rw_intr	19
4.2	SD Provided Entry points for the mid layer SCSI	19
4.3	SD Provided Entry points for the block device	19
4.4	SD used Entry points from the mid layer SCSI	20
4.5	SD used Entry points from the block device	20
4.6	SD data structures	20
5.0	Overview of SCSI Unifying Mid Layer	22
5.1	SCSI Initialization	22
5.2	SCSI BUS Scan	22
5.3	IO Functionality	23
5.3.1	scsi_prep_fn	23
5.3.2	scsi_request_function	24
5.3.3	scsi_done	24
5.4	Error Handling Functionality	25
5.5	Data Structures	27
6.0	SCSI Low Level Driver	41
6.1	SCSI Low Level Driver functional entry points	42
6.1.1	bios_param	42
6.1.2	detect	42
6.1.3	eh_timed_out	42
6.1.4	eh_abort_handler	43
6.1.5	eh_bus_reset_handler	43
6.1.6	eh_device_reset_handler	43
6.1.7	eh_host_reset_handler	43
6.1.8	eh_strategy_handler	43
6.1.9	info	44
6.1.10	ioctl	44
6.1.11	proc_info	44
6.1.12	queuecommand	44
6.1.13	release	45
6.1.14	slave_alloc	45
6.1.15	slave_configure	45
6.1.16	slave_destroy	45
6.2	Important Scsi_Host parameters set by the LLD	46
6.3	Important Scsi_Cmd parameters provided by SCSI mid layer	46
6.4	SCSI mid layer functions exported to LLD	48
	References	49

## List of Figures

Figure 1	- Linux File IO Transfer Overview	8
Figure 2	- High-level overview of an IO transfer using block device driver	11
Figure 3	- Linux SCSI SubSystem - Layered architecture	14
Figure 4	- IO transfer using all the three layers	15
Figure 5	- SCSI Bus Scan	23
Figure 6	- SCSI IO Low and Midlevel IO transfer	41

## List of Tables

Table 1 - Block Initialization entry points.....	9
Table 2 - Block IO entry points.....	10
Table 3 - Block Device Driver Entry Points.....	10
Table 4 - SD Provided entry points to Mid Layer.....	19
Table 5 - SD Provided Entry points for the block device.....	20
Table 6 - SD used Entry points from the mid layer SCSI.....	20
Table 7 - SD used Entry points from the block device.....	20
Table 8 - Scsi_Host parameters set by LLD.....	46
Table 9 - Scsi_Cmdnd parameters provided by SCSI mid layer.....	48
Table 10 - MidLayer exported functions to LLD.....	48

## 1.0 Introduction

This document describes the internals of SCSI in 2.6.X Linux kernel. This provides the internal view of the SCSI subsystem as seen from the current implementation. The document covers all the three layers of Linux SCSI implementation and tries to provide a top down view of the modules. The discussion revolves around the overview of the block driver interface, three layer SCSI architecture, SCSI upper layer (SD in specific), SCSI mid layer and Low Level driver interface. Each of the sections begin with a big picture, interfaces to the adjoining layers, functional flow, description of the important functions and the important data structures involved with the layer.

## 1.1 Document Assumptions and Conventions

It is assumed that the reader understands C programming language and is acquainted with bit of SCSI protocol. More documentation describing the SCSI architecture and model can be found at [www.t10.org](http://www.t10.org).

The file references in the document are in italics. The data structures are in a reduced font to identify them from rest of the document.

## 1.2 Copyright, License and Disclaimer

*Copyright © 2005 Samdeep Nayak*

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; The complete licensing terms can be found at <http://www.gnu.org/copyleft/fdl.html#TOC1>

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors and inaccuracies, that could be damaging to your system. Proceed with caution, and although it is highly unlikely that accidents will happen because of following advice or procedures described in this document, the author(s) do not take any responsibility for any damage claimed to be caused by doing so.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

### **1.3 Acknowledgments**

This document was written as part of my Master's project at the San Jose State University. I would like to thank Professor Dr. Ahmed Hambaba for all his help during our project. I am also thankful to our adviser Amir A Vetry for providing numerous pointers, validating our findings and providing encouragement during our project. I am also thankful to Randy Dunlap from the OSDL for providing the valuable inputs during the early stage of the document.

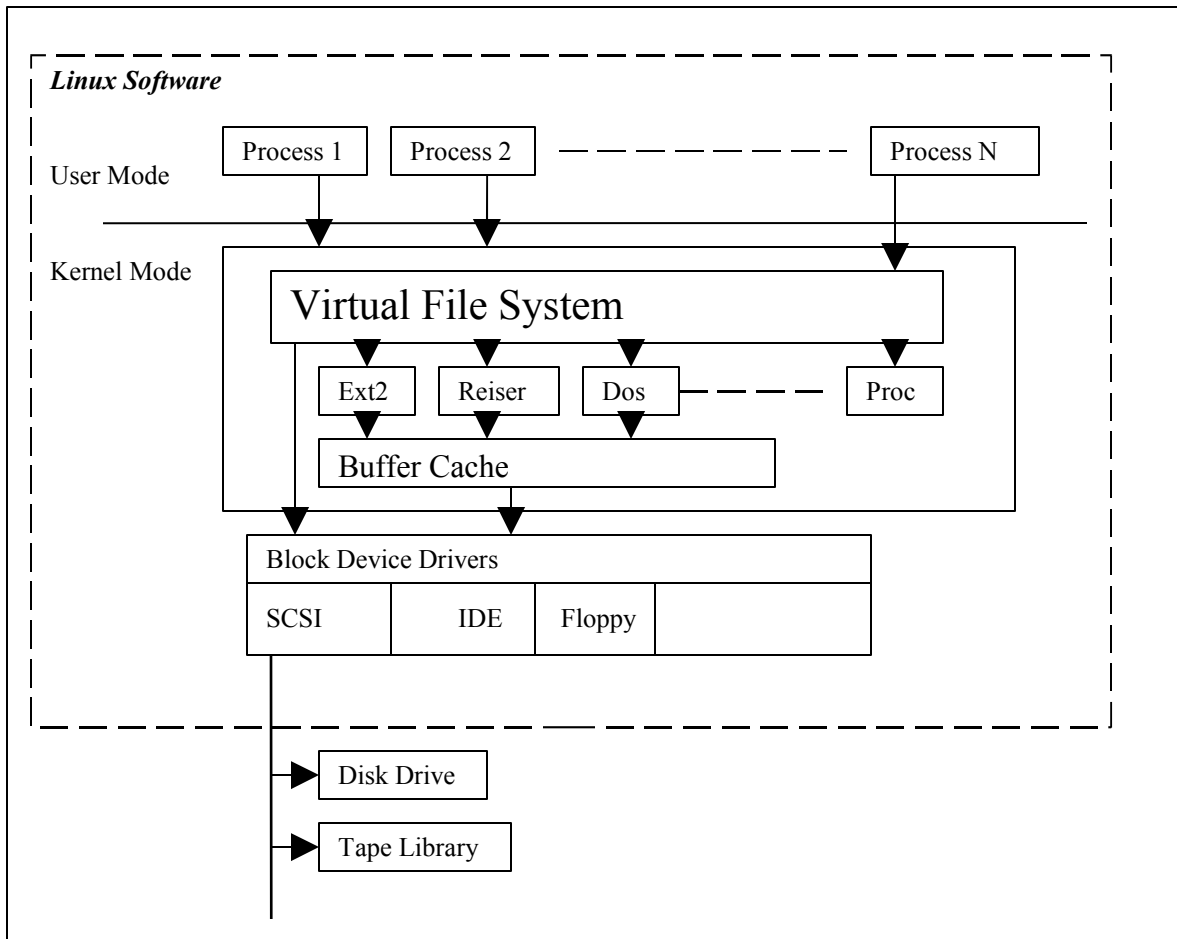
### **1.4 Feedback**

Please e-mail your corrections and suggestions to [samdeep@ieee.org](mailto:samdeep@ieee.org)

## **2.0 Linux Overview of Block Data Transfer**

Linux uses Virtual File System architecture to transfer user requests for IO transfers over to the file systems. VFS provides file level abstraction for all the file systems. This module receives system requests from users and interacts with a specific file systems based on the file system on which the request is traced.

**Figure 1 - Linux File IO Transfer Overview**



Each of the individual file systems provides management of file and directory data to the users. They also help in transferring data to the target devices such as block device nodes and NFS network nodes.

The `ll_rw_blk` sits below the file systems and provides interface to the file systems to send out IO requests. If the IO requests are in the adjacent blocks, the `ll_rw_blk` module merges the IO requests and the new request is forwarded to the block level driver to physically place the data in the storage media.

A block device driver is the one, which transfers data in blocks. Each of the blocks supported by the hardware is called a sector. Some of the commonly used block device drivers include floppy device driver, SCSI device driver. In this document we will concentrate on the SCSI interface only.

## 2.1 Block Device Driver Upper Entry points

IO operation involving a block device driver is conducted through a set of entry points exchanged between the driver and the `ll_rw_blk` module. The upper level entry points are the exported functions used by the block device driver to get the IO request. These requests are completed back to the `ll_rw_blk` module. Some of these entry points that are used by the SCSI block device driver are listed below.

<b>Initialization and Registration</b>	<b>Brief Explanation</b>
int register_blkdev(unsigned int major_no, const char *name)	Register a block device with a given major number and name
int unregister_blkdev(unsigned int major_no, const char *name)	Unregisters the block device driver

**Table 1 - Block Initialization entry points**

<b>IO related functions</b>	<b>Brief Explanation</b>
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)	This is a very important function for the SCSI stack, which prepares a request queue for use with a block device. The block device driver registers a call back function to accept coalesced IO requests.
void blk_queue_prep_rq(request_queue_t *q, prep_rq_fn *pfn)	The block device drivers register a prepare_request callback that gets invoked before the request is handed to the request_fn. This callback function is called before request function is called and this can be used to initialize the IO request such as initializing the cdb from the request data
void blk_queue_max_sectors(request_queue_t *q, unsigned short max_sectors)	This function set a limit for max sectors for a request and thus enables a low level driver to set an upper limit on the size of IO request
void blk_queue_max_hw_segments(request_queue_t *q, unsigned short max_segments)	This function enables a low level driver to set an upper limit on the number of hw data segments in a request.
void blk_queue_max_phys_segments(request_queue_t *q, unsigned short max_segments)	This function enables a low level driver to set an upper limit on the number of hw data segments in a request.
void blk_queue_segment_boundary(request_queue_t *q, unsigned long mask)	This sets the boundary rules for segment merging

**Table 2 - Block IO entry points**

The kernel provides several other interfaces to the block device drivers. However only the interfaces that have been used in the SCSI subsystem have been documented here. For more in depth information on these interfaces pls refer to drivers/block/ll\_rw\_blk.c file in the kernel.

## **2.2 Block Device Driver Lower Entry points**

These are the entry points provided by the block device drivers to the upper kernel module.

<b>Lower level entry points</b>	<b>Brief Explanation</b>
int open (struct inode *, struct file *)	This is the first entry point that gets called in the block device driver. This function is part of the block_device_operations structure and the block device exchanges the pointer of this function with the kernel.
int release(struct inode *, struct file *)	This function will be called when the node is getting closed. This function is part of the block device operations
int ioctl(struct inode *, struct file *, unsigned, unsigned long)	This function provides the user a way to issue device specific commands to the node. This function is part of the block device operations
int media_changed(struct gendisk *)	This function checks to see if the medium changed. This function is part of the block device operations
int revalidate_disk(struct gendisk *)	This function when a disk change is detected. This function is part of the block_device_operations
void io_request_fn(struct request_queue *q)	This function is the main entry point for passing all the IO requests. This callback function is registered using blk_init_queue function during initialization.
int io_prep_fn(struct request_queue *q, struct request *req)	This function provides functionality to initialize an IO request. This is called before io_request_fn is invoked. This callback function is registered using blk_queue_prep_rq during initialization

**Table 3 - Block Device Driver Entry Points**

## 2.3 Flow of an IO in the block device driver

The block device driver loads and registers itself as a block device. The block\_device\_operations structure containing the various block entry points is exchanged with the kernel. It then initializes the io request strategy routine with the kernel to accept the IO requests from ll\_rw\_blk module using blk\_init\_queue function. The ll\_rw\_blk module will create an IO request, populates the fields in the request structure and sends it down to the block device driver. The block device driver, dma's data from the request to the block device. After the IO is completed successfully, the block device will call out an end request to indicate the completion of the IO to the ll\_rw\_blk module. Following diagram briefly describes a high level overview of an IO transfer using the block device.

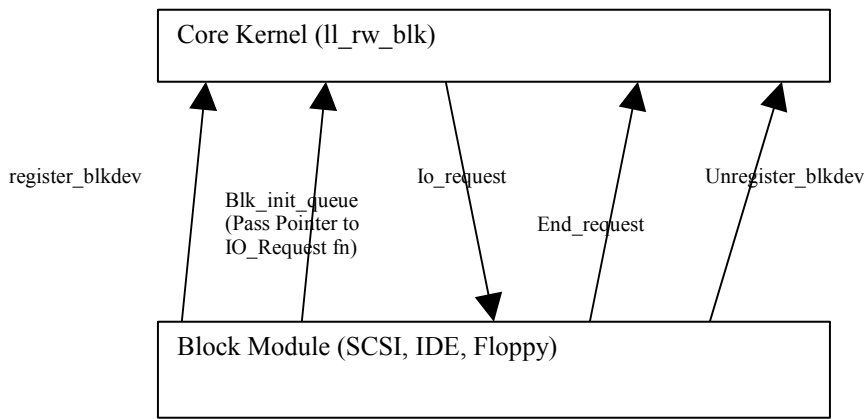


Figure 2 - High-level overview of an IO transfer using block device driver

## 2.4 Block Device Driver Data Structures

This section has various data structures used by the Block device driver.

### 2.4.1 Struct `block_device_operations`

This structure provides various entry points required for the operation of a block device. This is defined in `include/linux/fs.h` directory

```
struct block_device_operations{
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    struct module *owner;
}
```

### 2.4.2 Struct `request_queue`

This structure is used for sending down IO requests to the block device drivers. The requests are converted to SCSI commands and sent down to SCSI targets. This is defined in `include/linux/blkdev.h` file

```
struct request_queue
{
    /*
     * Together with queue_head for cacheline sharing
     */
    struct list_head queue_head;
    struct request *last_merge;
    elevator_t elevator;
    /*
     * the queue request freelist, one for reads and one for writes
     */
    struct request_list rq;

    /* Various Function pointers */
    request_fn_proc *request_fn;
    merge_request_fn*back_merge_fn;
    merge_request_fn*front_merge_fn;
    merge_requests_fn *merge_requests_fn;
}
```

```

make_request_fn      *make_request_fn;
prep_rq_fn          *prep_rq_fn;
unplug_fn           *unplug_fn;
merge_bvec_fn       *merge_bvec_fn;
activity_fn         *activity_fn;
/*
 * Auto-unplugging state
 */
struct timer_list   unplug_timer;
int                 unplug_thresh; /* After this many requests */
unsigned long       unplug_delay; /* After this many jiffies */
struct work_struct  unplug_work;
struct backing_dev_info backing_dev_info;

/*
 * The queue owner gets to use this for whatever they like.
 * ll_rw_blk doesn't touch it.
 */
void                *queuedata;
void                *activity_data;

/*
 * queue needs bounce pages for pages above this limit
 */
unsigned long       bounce_pfn;
int                 bounce_gfp;

/*
 * various queue flags, see QUEUE_* below
 */
unsigned long       queue_flags;

/*
 * protects queue structures from reentrancy
 */
spinlock_t         *queue_lock;

/*
 * queue kobject
 */
struct kobject      kobj;

/*
 * queue settings
 */
unsigned long       nr_requests; /* Max # of requests */
unsigned int        nr_congestion_on;
unsigned int        nr_congestion_off;

unsigned short      max_sectors;
unsigned short      max_phys_segments;
unsigned short      max_hw_segments;
unsigned short      hardsect_size;
unsigned int        max_segment_size;

unsigned long       seg_boundary_mask;
unsigned int        dma_alignment;

```

```

struct blk_queue_tag    *queue_tags;

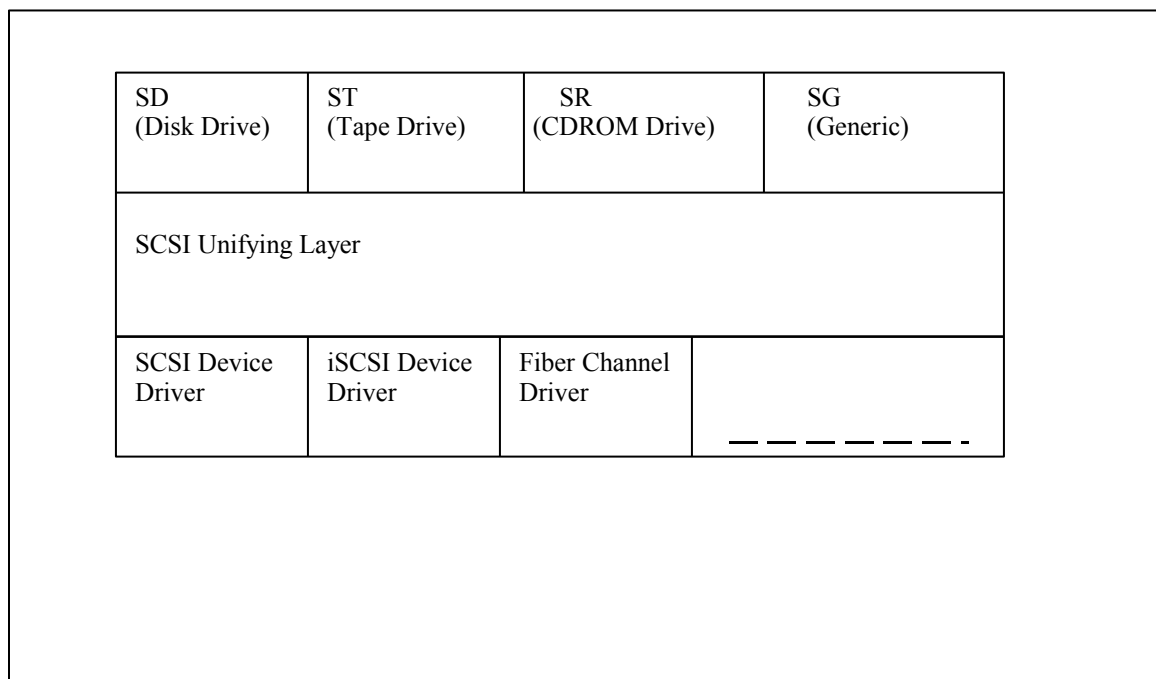
atomic_t                refcnt;

unsigned int            in_flight;

/*
 * sg stuff
 */
unsigned int            sg_timeout;
unsigned int            sg_reserved_size;
};

```

### 3.0 Overview of Linux SCSI Subsystem – Layered Architecture



**Figure 3 - Linux SCSI SubSystem - Layered architecture**

Linux SCSI uses layered architecture to transfer data. All the IO requests are serviced by each of the three layers shown above. The upper layer is the closest one to the kernel and hence ends up providing most of the class specific entry points (i.e. block\_device\_operations for the block device). They also provide wrapper functions required for the given class of device (such as disk, tape etc). Not all the upper layer modules are block based drivers. E.g. the sg driver provides a character device interface to the kernel and lets the user applications send various SCSI commands to the underlying devices.

The mid layer does bulk of the operations that include building an IO request, performing an error recovery, managing the low level controllers, providing proc interface to the scsi drivers, and the bus scan feature. This module also provides internal interface to both the upper and lower level SCSI drivers.

The low level drivers usually interact with the hardware and DMA's the data on to the target device such as Hard Disk, Tape Drive and CDROM drive devices. They also provide SCSI interfaces to the controllers, which typically transfer data over a different media such as TCP, FC etc. They interact with SCSI mid layer and hardware to deliver the IO requests.

**Flow of an IO using all the three layers**

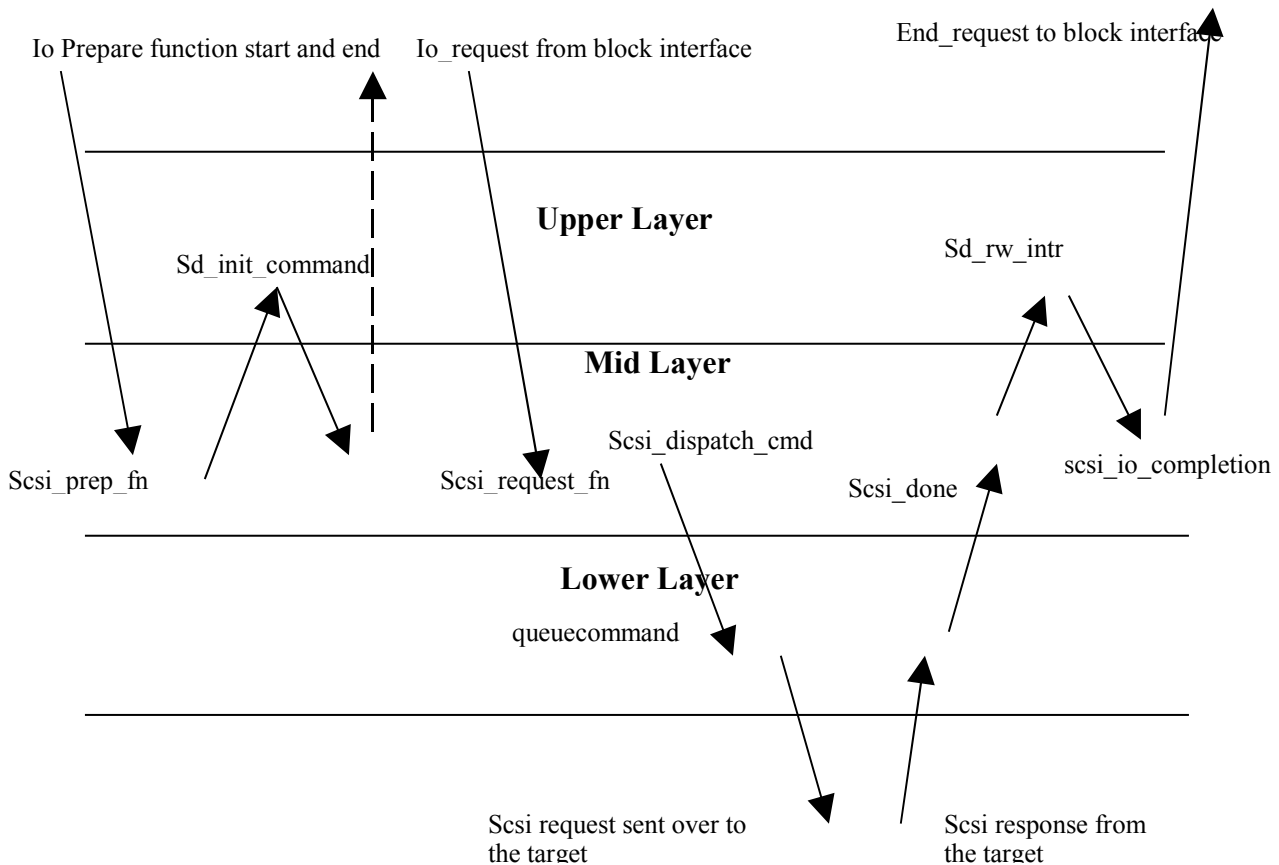
The mid layer registers callback functions for a given device to prepare the IO requests and to accept the IO commands. The block device calls the `scsi_prep_fn` to prepare the io request. This function will validate for any errors, sets up SG lists, fills in the scsi command, and updates upper layer specific parameters such as IO time out before returning back to the calling function in the block interface.

After the `scsi_prep_fn` is completed, the block interface calls the `scsi_request_function` as registered by the midlayer. The `scsi_request_function` will initialize the error handling routines before it dispatches the request to the Lower Level Drivers for the data DMA.

The lower level drivers will send down the IO request to the hardware and waits till the data DMA is completed. When the controller gets an ISR notification about the IO being completed, the LLD call the scsi completion routine to indicate the status of the IO completion to the mid layer. Upon receiving the completion routine, the mid layer will notify the block interface about the completion of the IO.

The below diagrams depicts the IO operation as carried out by the three layers

**Figure 4 - IO transfer using all the three layers**



## 4.0 Overview of SCSI Disk Driver (SD) – Upper Layer

SD is one of the upper layer drivers in the stack and handles SCSI requests that are targeted to the scsi disks. This kernel driver provides block driver interface to the file system at the top, while utilizing the unifying SCSI layer/device driver underneath it to deliver the requests to the intended hard disk device.

The functionalities performed by this driver include

- Device number to disk map
- Build an IO request command
- Error interpretations for the command completions
- Provide information such as retry count and timeout

The SD module essentially has 2 major entry points such as init and shutdown as exposed to the operating system. The module also shares an API interface with the SCSI middle layer.

### 4.1 SD functionality

SD provides internal functions required by the mid layer. It also provides the block interface to the ll\_rw\_blk module. The SD functionalities are implemented in drivers/scsi/sd.c file. The following section provides overview of some of these functions.

#### Init Functionality

The module init entry point is called every time the module is loaded into the kernel. During the load time, the driver registers as a block device for all the major numbers with the Operating System. This would let the module receive all the IO requests directed to the SCSI disks. After registering the block interface, the module will register itself with the underlying SCSI middle layer.

#### Shutdown Functionality

This function unregisters this driver from the scsi mid-level and then unregisters itself from the block device. From now on, the driver ceases to accept any more commands from the kernel.

Data structures used during initialization of the block device with the OS

```
.gendrv = {  
    .name           = "sd",  
    .probe          = sd_probe,  
    .remove         = sd_remove,  
    .shutdown       = sd_shutdown,  
}
```

Data structures used during initialization with the mid layer

```

struct scsi_driver sd_template = {
    .owner          = THIS_MODULE,
    .gendrv = {
        .name       = "sd",
        .probe      = sd_probe,
        .remove     = sd_remove,
        .shutdown   = sd_shutdown,
    },
    .rescan        = sd_rescan,
    .init_command  = sd_init_command,
};

```

Functionalities Performed by each of the functions is listed below

#### 4.1.1 sd\_probe

This function is called during driver initialization and whenever a new scsi device is attached to the system. It is called once for each scsi device (not just disks) present.

Returns: Returns 0 if successful (may be scsi disk or not) and 1 when there is an error

Called From: This function is invoked from the scsi mid-level. This function sets up the mapping between a given <host,channel,id,lun> (found in sdp) and new device name (e.g. /dev/sda).

Input – Pointer to device struct device \*dev

#### 4.1.2 sd\_remove

This function is called whenever a scsi disk (previously recognized by sd\_probe) is detached from the system. It is called (potentially multiple times) during sd module unload.

Returns: Returns 0 if successful (may be scsi disk or not) and 1 when there is an error

Called From: This function is invoked from the scsi mid-level. This function potentially frees up a device name (e.g. /dev/sdc) that could be re-used by a subsequent sd\_probe(). This function is not called when the built-in sd driver is "exit-ed"

Input – Pointer to device struct device \*dev

#### 4.1.3 sd\_shutdown

This function issues a synchronize cache command to the disk and waits till it gets the response from it

Returns: None

Called From: This is called from sd\_remove function

Input – Pointer to device struct device \*dev

#### 4.1.4 sd\_init\_command

This function builds a scsi (read or write) command from information in the request structure.

Returns: 1 if successful and 0 if error  
Called From: This is called from scsi mid layer to initialize disk specific command  
Input – Pointer to struct `scsi_cmnd * SCpnt`

Since `sd` exposes the block level device driver to the operating system, it supports following functions as required by the Operating System.

```
static struct block_device_operations sd_fops = {  
    .owner          = THIS_MODULE,  
    .open           = sd_open,  
    .release        = sd_release,  
    .ioctl          = sd_ioctl,  
    .media_changed  = sd_media_changed,  
    .revalidate_disk = sd_revalidate_disk,  
};
```

#### 4.1.5 `sd_open`

This function open up a scsi disk device for IO operations.  
Returns: 0 if successful and a negative number if failed  
Called From: This is called from a user context (e.g. `fsck(1)` ) or from within the kernel  
Input – Pointer to struct `inode *inode`, struct `file *filp`

#### 4.1.6 `sd_release`

This function releases a scsi disk device from IO operations.  
Returns: 0  
Called From: This is called when the user invokes `close` on the scsi disk  
Input – Pointer to struct `inode *inode`, struct `file *filp`

#### 4.1.7 `sd_ioctl`

This function processes `ioctl` requests from the user. Most `ioctls` are forwarded onto the block subsystem or further down in the scsi subsystem  
Returns: 0 or positive value on success and a negative value for failure  
Called From: This is called from `ioctl` system call  
Input – struct `inode * inode`, struct `file * filp`, unsigned int `cmd`, unsigned long `arg`

#### 4.1.8 `sd_media_change`

This function checks to see if the medium changed. This sends a Test Unit Ready Command to verify if medium is present  
Returns: 0 if not applicable or no change; 1 if change  
Called From: This is called from block sub system  
Input – struct `gendisk *disk`

#### 4.1.9 sd\_revalidate\_disk

This function when a disk change is detected. This issues a command to spin up the disk and issues a Read Capacity Command to the disk.

Returns: 0

Called From: This is called from block sub system

Input – struct gendisk \*disk

#### 4.1.10 sd\_rw\_intr

This function is called whenever a disk IO command is completed. This command updates the disk specific error cases before calling the SCSI mid layer to complete the IO to the OS.

Returns: None

Called From: This is called from SCSI mid layer upon completion of the IO

Input – struct scsi\_cmnd \* SCpnt

### 4.2 SD Provided Entry points for the mid layer SCSI

Entry Point	Brief Explanation
sd_init_command	Builds a scsi command for a given request
sd_rw_intr	Handles disk specific errors during the IO completion
sd_rescan	Spins up the disk

Table 4 - SD Provided entry points to Mid Layer

### 4.3 SD Provided Entry points for the block device

Entry Point	Brief Explanation
sd_open	open a scsi disk device
sd_release	invoked when the last close is called on the disk
sd_ioctl	process an ioctl request to the disk device
sd_media_changed	Checks if the media has been changed
sd_revalidate_disk	Spins up the disk

Table 5 - SD Provided Entry points for the block device

### 4.4 SD used Entry points from the mid layer SCSI

Entry Point	Brief Explanation
scsi_block_when_processing_errors	Prevent commands from being queued
scsi_device_online	Returns the scsi device state
scsi_set_medium_removal	Sends out ALLOW_MEDIUM_REMOVAL command
scsi_ioctl	Sends out various scsi commands to the device

scsi_print_sense	Prints the scsi sense request
scsi_wait_req	Waits till the request is completed for a given timeout and retries
scsi_status_is_good	This returns true for known good conditions of the device
scsi_mode_sense	Issues a SCSI mode sense command
scsi_allocate_request	Allocate a request descriptor
scsi_register_driver	Driver registration
scsi_unregister_driver	Driver unregistration
scsi_print_req_sense	Prints sense information
scsi_io_completion	Called to complete IO request

**Table 6 - SD used Entry points from the mid layer SCSI**

## 4.5 SD used Entry points from the block device

Entry Point	Brief Explanation
register_blkdev	Registers as a block device
unregister_blkdev	Unregisters as a block device

**Table 7 - SD used Entry points from the block device**

## 4.6 SD data structures

This structure provides various function entry points required for the block device driver. This structure also provides entry points and information required for performing SCSI target device specific operations(in this case the hard disk). The following structures are defined in *drivers/scsi/sd.c* file.

```

struct scsi_driver sd_template = {
    .owner                = THIS_MODULE,
    .gendrv = {
        .name             = "sd",
        .probe            = sd_probe,
        .remove           = sd_remove,
        .shutdown         = sd_shutdown,
    },
    .rescan               = sd_rescan,
    .init_command         = sd_init_command,
};

struct scsi_disk {
    struct scsi_driver *driver; /* always &sd_template */
    struct scsi_device *device;
    struct kref          kref;
    struct gendisk       *disk;
    unsigned int         openers; /* protected by BKL for now, yuck */
    sector_t             capacity; /* size in 512-byte sectors */
    u32                  index;
    u8                   media_present;
    u8                   write_prot;
    unsigned             WCE : 1; /* state of disk WCE bit */
};

```

```
};    unsigned    RCD : 1;    /* state of disk RCD bit, unused */
```

## 5.0 Overview of SCSI Unifying Mid Layer

The SCSI mid layer does most of the operations in the SCSI stack. It accepts IO commands from the block device at the `scsi_request_function` entry point. It translates the IO request to appropriate device command format and sends it down to the lower layer driver. Upon the completion of the IO, the mid layer completes it to the kernel block interface. It also performs the error handling operations such as command abort and time out handling. It performs operations for scanning the devices and acts as unifying interface between the upper layer and the lower layer. The module registers with the proc interface and provides management functionalities to the users.

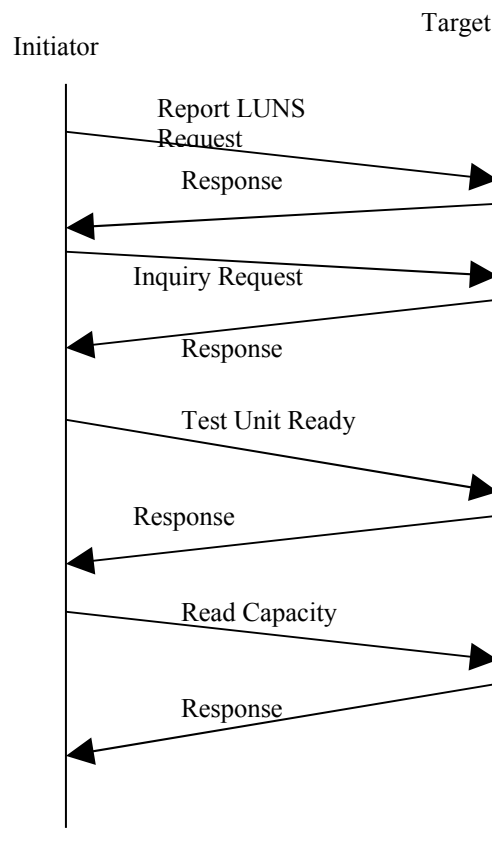
### 5.1 SCSI Initialization

`Init_scsi` is the major entry point and is used to initialize the SCSI sub system. This initializes the queues that are required for SG pools, creates a SCSI proc entry, sets up dynamic device list, registers `scsi_hosts`, and makes devfile system entry for scsi class devices.

## 5.2 SCSI BUS Scan

The SCSI bus scans for a given SCSI initiator is triggered either by the LLD calling the SCSI to scan the bus explicitly during the device initialization or called from the proc interface. The LLD will usually provide the maximum number of channels, id and LUN it supports in the Scsi\_Host structure during the Scsi registration phase. The LLD will initiate `scsi_scan_host` in the mid layer to scan for each of these devices. This function will internally call `scsi_scan_channel` for all the given channels supported by the host, `scsi_scan_target` for all the targets supported and `scsi_probe_and_add_lun` to add the LUNS. This will cause the mid layer to send REPORT LUN on to the bus. The mid layer scans for LUN 0, and if there is some response to it, it will scan further for the remaining LUNS. The LUN 0 would not be configured until all the LUNs are scanned. Depending on the response to the REPORT LUN command, the midlayer scans the resulting list of LUNs by calling `scsi_probe_and_add_lun` function. If the Report LUNS request failed to scan the target, Inquiry will be sent sequentially to scan each of the LUNs using `scsi_probe_and_add_lun` function.

Whenever a scsi device is found to be attached to the system, the probe function of the SCSI upper layer is called. The probe is called for each of the Scsi device present and hence `sd_probe` gets called. The upper layer `sd_probe` sends down TEST\_UNIT\_READY command to the target device and makes sure that the device is ready to accept SCSI commands. After getting a successful response to the Test Unit Ready command, the upper layer driver (SD) sends down READ CAPACITY to the device to read the disk capacity. The probe function also sets up the mapping between a given host, channel, id, lun and the new device name. The block device major and minor numbers are also chosen here.



## Figure 5 - SCSI Bus Scan

The SCSI bus scan functions are located in drivers/scsi/scsi\_scan.c file.

## 5.3 IO Functionality

The IO entry point is called for starting and completing the IO.

### 5.3.1 scsi\_prep\_fn

This function is used to initialize a command before the block interface sends down the command. This function checks to see if the device is online, if not returns BLKPREP\_KILL to indicate failure. This routine will allocate a command structure for this request and initializes it with the data found in the request structure. It sets up the required SG elements and gets appropriate SCSI CDB command from the upper layer before returning successfully to the block interface.

Returns: BLKPREP\_OK for continuing with the command, BLKPREP\_KILL or BLKPREP\_DEFER otherwise

Called From: This is called from the block interface

Input : struct request\_queue \*q, struct request \*req  
int scsi\_prep\_fn(struct request\_queue \*q, struct request \*req)

### 5.3.2 scsi\_request\_function

This function is called to transfer data over to the target device. The routine loops and retrieves the commands until the queue is empty. It then initializes the error handling routines required for this command. It assigns a non zero serial number to the command and starts a command-timeout timer. The command then calls the low level quecommand routine to queue up the command to the LLD.

Returns: Nothing

Called From: This is called from the block interface

Input : struct request\_queue \*q  
void scsi\_request\_fn(struct request\_queue \*q)

### 5.3.3 scsi\_done

This is called by the LLD modules to indicate the completion of a given command. This is often called in an interrupt context. The mid layer will delete the timer involved with this command after it receives this request. It initializes the command serial number to zero and puts the command to the done queue list. If the command was successfully completed, scsi\_finish\_command is called. This will inturn call the upper layer done command. The upper layer turns back and calls scsi\_io\_completion to call the IO completion routine of the block interface

Returns: None

Called From: This is called from the LLD upon the completion of the IO

Input : struct scsi\_cmnd \*cmd

scsi\_done(struct scsi\_cmnd \*cmd)

There are 3 major contexts in which a SCSI command can be executing. The block interface context is essentially the one in which the block interface calls the SCSI layer to send out a command. Scsi\_prep\_fn and scsi\_request\_function are both called from this context. The IO is submitted to the LLD to be delivered to the target in the same context. After the IO is completed, the LLD calls the scsi\_done routine in an interrupt context. This context has to be the most efficient of all. In addition to the two above-mentioned contexts, each host LLD has its own kernel thread to handle all the error conditions. Since all the three threads may be running simultaneously and the same entry points may be called for transferring different IO, a good number of functions are designed to be reentrant and multi thread safe. In addition to these contexts, the modules use soft irq mechanism to transfer data effectively.

## 5.4 Error Handling Functionality

The core of the SCSI error handling is done by the scsi\_error\_handler function. Each of the host (LLD) gets a dedicated kernel thread to handle all the error cases. This thread is started in the scsi\_host\_alloc function during the initialization of the LLD. This function sleeps and waits for an error event to trigger it out. Hence whenever an error happens, this thread is woken up. This function will try to unjam the host by calling the host error handling strategy routine. If for some reason, the host has not provided the strategy routine, it will call scsi\_unjam\_host to take care of the failed command. During this time any new commands to the host are not accepted.

Scsi\_unjam\_host is an important function, which provides the error handling feature for all the low-level drivers. The low level drivers can also provide an equivalent function, which can use scsi\_unjam\_host as their template. In this function, the mid layer gets the sense information for the failed commands, aborts all the commands and finishes processed command or retries them back.

Scsi\_eh\_get\_sense gets the sense information for the commands. This routine will check to see if there is autosense data available in the host buffer. If the target has already sent the autosense data, then request sense command will be issued by the initiator. If the target did not send the autosense data, then the host will send initiate request sense and gets sense data from the target.

scsi\_eh\_abort\_cmds will try to abort the commands that have not been completed. For the commands that have failed, abort request will not be sent. The mid layer calls the abort function of the LLD to abort a running command. The LLD will usually send a Task Management Request to abort the given command and return back to the mid layer. The host sending a Test Unit Ready to the target device usually follows the abort command.

If the device is online, `scsi_ah_flush_done_q` will try to retry the command until the maximum permissible limit. If not it would complete the command and copies the sense information by calling `scsi_finish_command`. This routine in turn calls the upper layer `rw_intr` function which would check for errors and calls the `scsi_io_completion` to complete the command back to the block interface

The SCSI errors are usually triggered by a Command Timing Out or by Command failing at the target. When a command times out `scsi_times_out` function is called. This will check if the low level driver can handle the time out request. If so it would let the LLD handle the request and decipher the status on return. If the low level driver requested to reset the timer, then the midlayer will give another try to the command and resends it down to the LLD. If the LLD did not handle this situation, then it queues up the request to cancel this command in `scsi_ah_scmd_add` function. This function will set the new owner of the command as the Error Handler and wakes up the `scsi_error_handler` to further treat the command.

`scsi_decide_disposition` is used to decide the fate of the command that was returned from the LLD. However not all the failed commands trigger an error handling. E.g. If a command was completed with check condition, and the sense data is stored in the buffer, then the command is completed without triggering any error handling. If for some reason, there was no sense data, then the mid layer will push the command for error handling and will wake up the error handling function. For some of the other errors, the command will be retried until the maximum permissible limit set by the upper layer driver on the device.

## 5.5 Data Structures

Most of the SCSI data structures are defined in the *include/scsi* directory and in *drivers/scsi* directory in the source tree.

The upper layer devices register type of the device they support with the mid layer using this structure. The SCSI mid layer appends the bus type as *scsi* before it registers with the kernel.

```
Struct device_driver{
    char                * name;
    struct bus_type     * bus;

    struct semaphore    unload_sem;
    struct kobject      kobj;
    struct list_head    devices;

    int      (*probe)(struct device * dev);
    int      (*remove)      (struct device * dev);
    void     (*shutdown)    (struct device * dev);
    int      (*suspend)     (struct device * dev, u32 state, u32 level);
    int      (*resume)      (struct device * dev, u32 level);
};
```

The lower layer devices register with the mid layer using this function. There is one "struct *scsi\_host\_template*" instance per LLD. The structure is defined in the *include/scsi/scsi\_host.h* file

```
struct scsi_host_template {
    struct module *module;
    const char *name;

    /*
     * Used to initialize old-style drivers. For new-style drivers
     * just perform all work in your module initialization function.
     *
     * Status: OBSOLETE
     */
    int (* detect)(struct scsi_host_template *);

    /*
     * Used as unload callback for hosts with old-style drivers.
     *
     * Status: OBSOLETE
     */
    int (* release)(struct Scsi_Host *);

    /*
     * The info function will return whatever useful information the
```

```

* developer sees fit. If not provided, then the name field will
* be used instead.
*
* Status: OPTIONAL
*/
const char *(* info)(struct Scsi_Host *);

/*
* Ioctl interface
*
* Status: OPTIONAL
*/
int (* ioctl)(struct scsi_device *dev, int cmd, void __user *arg);

/*
* The queuecommand function is used to queue up a scsi
* command block to the LLDD. When the driver finished
* processing the command the done callback is invoked.
*
* If queuecommand returns 0, then the HBA has accepted the
* command. The done() function must be called on the command
* when the driver has finished with it. (you may call done on the
* command before queuecommand returns, but in this case you
* *must* return 0 from queuecommand).
*
* Queuecommand may also reject the command, in which case it may
* not touch the command and must not call done() for it.
*
* There are two possible rejection returns:
*
* SCSI_MLQUEUE_DEVICE_BUSY: Block this device temporarily, but
* allow commands to other devices serviced by this host.
*
* SCSI_MLQUEUE_HOST_BUSY: Block all devices served by this
* host temporarily.
*
* For compatibility, any other non-zero return is treated the
* same as SCSI_MLQUEUE_HOST_BUSY.
*
* NOTE: "temporarily" means either until the next command for#
* this device/host completes, or a period of time determined by
* I/O pressure in the system if there are no other outstanding
* commands.
*
* STATUS: REQUIRED
*/
int (* queuecommand)(struct scsi_cmnd *,
                    void (*done)(struct scsi_cmnd *));

/*
* This is an error handling strategy routine. You don't need to
* define one of these if you don't want to - there is a default
* routine that is present that should work in most cases. For those
* driver authors that have the inclination and ability to write their
* own strategy routine, this is where it is specified. Note - the
* strategy routine is *ALWAYS* run in the context of the kernel eh
* thread. Thus you are guaranteed to *NOT* be in an interrupt
* handler when you execute this, and you are also guaranteed to

```

```

* *NOT* have any other commands being queued while you are in the
* strategy routine. When you return from this function, operations
* return to normal.
*
* See scsi_error.c scsi_unjam_host for additional comments about
* what this function should and should not be attempting to do.
*
* Status: REQUIRED      (at least one of them)
*/
int (* eh_strategy_handler)(struct Scsi_Host *);
int (* eh_abort_handler)(struct scsi_cmnd *);
int (* eh_device_reset_handler)(struct scsi_cmnd *);
int (* eh_bus_reset_handler)(struct scsi_cmnd *);
int (* eh_host_reset_handler)(struct scsi_cmnd *);

/*
* This is an optional routine to notify the host that the scsi
* timer just fired. The returns tell the timer routine what to
* do about this:
*
* EH_HANDLED:           I fixed the error, please complete the command
* EH_RESET_TIMER:      I need more time, reset the timer and
*                      begin counting again
* EH_NOT_HANDLED       Begin normal error recovery
*
* Status: OPTIONAL
*/
enum scsi_eh_timer_return (* eh_timed_out)(struct scsi_cmnd *);

/*
* Old EH handlers, no longer used. Make them warn the user of old
* drivers by using a wrong type
*
* Status: MORE THAN OBSOLETE
*/
int (* abort)(int);
int (* reset)(int, int);

/*
* Before the mid layer attempts to scan for a new device where none
* currently exists, it will call this entry in your driver. Should
* your driver need to allocate any structs or perform any other init
* items in order to send commands to a currently unused target/lun
* combo, then this is where you can perform those allocations. This
* is specifically so that drivers won't have to perform any kind of
* "is this a new device" checks in their queuecommand routine,
* thereby making the hot path a bit quicker.
*
* Return values: 0 on success, non-0 on failure
*
* Deallocation: If we didn't find any devices at this ID, you will
* get an immediate call to slave_destroy(). If we find something
* here then you will get a call to slave_configure(), then the
* device will be used for however long it is kept around, then when
* the device is removed from the system (or * possibly at reboot
* time), you will then get a call to slave_destroy(). This is
* assuming you implement slave_configure and slave_destroy.
* However, if you allocate memory and hang it off the device struct,

```

```

* then you must implement the slave_destroy() routine at a minimum
* in order to avoid leaking memory
* each time a device is tore down.
*
* Status: OPTIONAL
*/
int (* slave_alloc)(struct scsi_device *);

/*
* Once the device has responded to an INQUIRY and we know the
* device is online, we call into the low level driver with the
* struct scsi_device *. If the low level device driver implements
* this function, it *must* perform the task of setting the queue
* depth on the device. All other tasks are optional and depend
* on what the driver supports and various implementation details.
*
* Things currently recommended to be handled at this time include:
*
* 1. Setting the device queue depth. Proper setting of this is
* described in the comments for scsi_adjust_queue_depth.
* 2. Determining if the device supports the various synchronous
* negotiation protocols. The device struct will already have
* responded to INQUIRY and the results of the standard items
* will have been shoved into the various device flag bits, eg.
* device->sdtr will be true if the device supports SDTR messages.
* 3. Allocating command structs that the device will need.
* 4. Setting the default timeout on this device (if needed).
* 5. Anything else the low level driver might want to do on a device
* specific setup basis...
* 6. Return 0 on success, non-0 on error. The device will be marked
* as offline on error so that no access will occur. If you return
* non-0, your slave_destroy routine will never get called for this
* device, so don't leave any loose memory hanging around, clean
* up after yourself before returning non-0
*
* Status: OPTIONAL
*/
int (* slave_configure)(struct scsi_device *);

/*
* Immediately prior to deallocating the device and after all activity
* has ceased the mid layer calls this point so that the low level
* driver may completely detach itself from the scsi device and vice
* versa. The low level driver is responsible for freeing any memory
* it allocated in the slave_alloc or slave_configure calls.
*
* Status: OPTIONAL
*/
void (* slave_destroy)(struct scsi_device *);

/*
* This function determines the bios parameters for a given
* harddisk. These tend to be numbers that are made up by
* the host adapter. Parameters:
* size, device, list (heads, sectors, cylinders)
*
* Status: OPTIONAL
*/

```

```

int (* bios_param)(struct scsi_device *, struct block_device *,
                  sector_t, int []);

/*
 * Can be used to export driver statistics and other infos to the
 * world outside the kernel ie. userspace and it also provides an
 * interface to feed the driver with information.
 *
 * Status: OBSOLETE
 */
int (*proc_info)(struct Scsi_Host *, char *, char **, off_t, int, int);

/*
 * Name of proc directory
 */
char *proc_name;

/*
 * Used to store the procfs directory if a driver implements the
 * proc_info method.
 */
struct proc_dir_entry *proc_dir;

/*
 * This determines if we will use a non-interrupt driven
 * or an interrupt driven scheme, It is set to the maximum number
 * of simultaneous commands a given host adapter will accept.
 */
int can_queue;

/*
 * In many instances, especially where disconnect / reconnect are
 * supported, our host also has an ID on the SCSI bus. If this is
 * the case, then it must be reserved. Please set this_id to -1 if
 * your setup is in single initiator mode, and the host lacks an
 * ID.
 */
int this_id;

/*
 * This determines the degree to which the host adapter is capable
 * of scatter-gather.
 */
unsigned short sg_tablesize;

/*
 * If the host adapter has limitations beside segment count
 */
unsigned short max_sectors;

/*
 * dma scatter gather segment boundary limit. a segment crossing this
 * boundary will be split in two.
 */
unsigned long dma_boundary;

/*
 * This specifies "machine infinity" for host templates which don't

```

```

* limit the transfer size. Note this limit represents an absolute
* maximum, and may be over the transfer limits allowed for
* individual devices (e.g. 256 for SCSI-1)
*/
#define SCSI_DEFAULT_MAX_SECTORS1024

/*
* True if this host adapter can make good use of linked commands.
* This will allow more than one command to be queued to a given
* unit on a given host. Set this to the maximum number of command
* blocks to be provided for each device. Set this to 1 for one
* command block per lun, 2 for two, etc. Do not set this to 0.
* You should make sure that the host adapter will do the right thing
* before you try setting this above 1.
*/
short cmd_per_lun;

/*
* present contains counter indicating how many boards of this
* type were found when we did the scan.
*/
unsigned char present;

/*
* true if this host adapter uses unchecked DMA onto an ISA bus.
*/
unsigned unchecked_isa_dma:1;

/*
* true if this host adapter can make good use of clustering.
* I originally thought that if the tablesize was large that it
* was a waste of CPU cycles to prepare a cluster list, but
* it works out that the Buslogic is faster if you use a smaller
* number of segments (i.e. use clustering). I guess it is
* inefficient.
*/
unsigned use_clustering:1;

/*
* True for emulated SCSI host adapters (e.g. ATAPI)
*/
unsigned emulated:1;

/*
* True if the low-level driver performs its own reset-settle delays.
*/
unsigned skip_settle_delay:1;

/*
* Countdown for host blocking with no commands outstanding
*/
unsigned int max_host_blocked;

/*
* Default value for the blocking. If the queue is empty,
* host_blocked counts down in the request_fn until it restarts
* host operations as zero is reached.
*

```

```

        * FIXME: This should probably be a value in the template
        */
#define SCSI_DEFAULT_HOST_BLOCKED      7

/*
 * Pointer to the sysfs class properties for this host, NULL terminated.
 */
struct class_device_attribute **shost_attrs;

/*
 * Pointer to the SCSI device properties for this host, NULL terminated.
 */
struct device_attribute **sdev_attrs;

/*
 * List of hosts per template.
 *
 * This is only for use by scsi_module.c for legacy templates.
 * For these access to it is synchronized implicitly by
 * module_init/module_exit.
 */
struct list_head legacy_hosts;
};

```

There is one struct Scsi\_Host instance per host (HBA) that an LLD controls. The struct Scsi\_Host structure has many members in common with "struct scsi\_host\_template". When a new struct Scsi\_Host instance is created (in scsi\_host\_alloc() in hosts.c) those common members are initialized from the driver's struct scsi\_host\_template instance. The structure is defined in the *include/scsi/scsi\_host.h* file

```

struct Scsi_Host {
/*
 * __devices is protected by the host_lock, but you should
 * usually use scsi_device_lookup / shost_for_each_device
 * to access it and don't care about locking yourself.
 * In the rare case of being in irq context you can use
 * their __prefixed variants with the lock held. NEVER
 * access this list directly from a driver.
 */
struct list_head __devices;

struct scsi_host_cmd_pool *cmd_pool;
spinlock_t free_list_lock;
struct list_head free_list; /* backup store of cmd structs */
struct list_head starved_list;

spinlock_t default_lock;
spinlock_t *host_lock;

struct semaphore scan_mutex; /* serialize scanning activity */

struct list_head eh_cmd_q;
struct task_struct *ehandler; /* Error recovery thread. */
struct semaphore *eh_wait; /* The error recovery thread waits
                           on this. */

struct completion *eh_notify; /* wait for eh to begin or end */
struct semaphore *eh_action; /* Wait for specific actions on the

```

```

        host. */
unsigned int      eh_active:1; /* Indicates the eh thread is awake and active if
        this is true. */
unsigned int      eh_kill:1; /* set when killing the eh thread */
wait_queue_head_t host_wait;
struct scsi_host_template *hostt;
struct scsi_transport_template *transportt;
volatile unsigned short host_busy; /* commands actually active on low-level */
volatile unsigned short host_failed; /* commands that failed. */

unsigned short host_no; /* Used for IOCTL_GET_IDLUN, /proc/scsi et al. */
int resetting; /* if set, it means that last_reset is a valid value */
unsigned long last_reset;

/*
 * These three parameters can be used to allow for wide scsi,
 * and for host adapters that support multiple busses
 * The first two should be set to 1 more than the actual max id
 * or lun (i.e. 8 for normal systems).
 */
unsigned int max_id;
unsigned int max_lun;
unsigned int max_channel;

/*
 * This is a unique identifier that must be assigned so that we
 * have some way of identifying each detected host adapter properly
 * and uniquely. For hosts that do not support more than one card
 * in the system at one time, this does not need to be set. It is
 * initialized to 0 in scsi_register.
 */
unsigned int unique_id;

/*
 * The maximum length of SCSI commands that this host can accept.
 * Probably 12 for most host adapters, but could be 16 for others.
 * For drivers that don't set this field, a value of 12 is
 * assumed. I am leaving this as a number rather than a bit
 * because you never know what subsequent SCSI standards might do
 * (i.e. could there be a 20 byte or a 24-byte command a few years
 * down the road?).
 */
unsigned char max_cmd_len;

int this_id;
int can_queue;
short cmd_per_lun;
short unsigned int sg_tablesize;
short unsigned int max_sectors;
unsigned long dma_boundary;

unsigned unchecked_isa_dma:1;
unsigned use_clustering:1;
unsigned use_blk_tq:1;

/*
 * Host has requested that no further requests come through for the
 * time being.

```

```

    */
    unsigned host_self_blocked:1;

    /*
    * Host uses correct SCSI ordering not PC ordering. The bit is
    * set for the minority of drivers whose authors actually read
    * the spec ;)
    */
    unsigned reverse_ordering:1;

    /*
    * Host has rejected a command because it was busy.
    */
    unsigned int host_blocked;

    /*
    * Value host_blocked counts down from
    */
    unsigned int max_host_blocked;

    /* legacy crap */
    unsigned long base;
    unsigned long io_port;
    unsigned char n_io_port;
    unsigned char dma_channel;
    unsigned int irq;

    unsigned long shost_state;

    /* Idm bits */
    struct device          shost_gendev;
    struct class_device    shost_classdev;

    /*
    * List of hosts per template.
    *
    * This is only for use by scsi_module.c for legacy templates.
    * For these access to it is synchronized implicitly by
    * module_init/module_exit.
    */
    struct list_head sht_legacy_list;

    /*
    * We should ensure that this is aligned, both for better performance
    * and also because some compilers (m68k) don't automatically force
    * alignment to a long boundary.
    */
    unsigned long hostdata[0] /* Used for storage of host specific stuff */
        __attribute__((aligned (sizeof(unsigned long))));
};

```

This is the command data structure which is used to send down commands. This structure is defined in the *include/scsi/scsi\_cmnd.h*

```

struct scsi_cmnd {
    int    sc_magic;

```

```

struct scsi_device *device;
unsigned short state;
unsigned short owner;
struct scsi_request *sc_request;

struct list_head list; /* scsi_cmnd participates in queue lists */

struct list_head eh_entry; /* entry for the host eh_cmd_q */
int eh_state; /* Used for state tracking in error handlr */
int eh_eflags; /* Used by error handlr */
void (*done) (struct scsi_cmnd *); /* Mid-level done function */

/*
 * A SCSI Command is assigned a nonzero serial_number when internal_cmnd
 * passes it to the driver's queue command function. The serial_number
 * is cleared when scsi_done is entered indicating that the command has
 * been completed. If a timeout occurs, the serial number at the moment
 * of timeout is copied into serial_number_at_timeout. By subsequently
 * comparing the serial_number and serial_number_at_timeout fields
 * during abort or reset processing, we can detect whether the command
 * has already completed. This also detects cases where the command has
 * completed and the SCSI Command structure has already being reused
 * for another command, so that we can avoid incorrectly aborting or
 * resetting the new command.
 */
unsigned long serial_number;
unsigned long serial_number_at_timeout;

int retries;
int allowed;
int timeout_per_command;
int timeout_total;
int timeout;

/*
 * We handle the timeout differently if it happens when a reset,
 * abort, etc are in process.
 */
unsigned volatile char internal_timeout;

unsigned char cmd_len;
unsigned char old_cmd_len;
enum dma_data_direction sc_data_direction;
enum dma_data_direction sc_old_data_direction;

/* These elements define the operation we are about to perform */
#define MAX_COMMAND_SIZE 16
unsigned char cmd[MAX_COMMAND_SIZE];
unsigned request_bufflen; /* Actual request size */

struct timer_list eh_timeout; /* Used to time out the command. */
void *request_buffer; /* Actual requested buffer */

/* These elements define the operation we ultimately want to perform */
unsigned char data_cmnd[MAX_COMMAND_SIZE];
unsigned short old_use_sg; /* We save use_sg here when requesting
 * sense info */
unsigned short use_sg; /* Number of pieces of scatter-gather */

```

```

unsigned short sglst_len; /* size of malloc'd scatter-gather list */
unsigned short abort_reason; /* If the mid-level code requests an
                             * abort, this is the reason. */
unsigned buflen; /* Size of data buffer */
void *buffer; /* Data buffer */

unsigned underflow; /* Return error if less than
                    * this amount is transferred */
unsigned old_underflow; /* save underflow here when reusing the
                        * command for error handling */

unsigned transfersize; /* How much we are guaranteed to
                       * transfer with each SCSI transfer
                       * (ie, between disconnect /
                       * reconnects. Probably == sector
                       * size */

int resid; /* Number of bytes requested to be
            * transferred less actual number
            * transferred (0 if not supported) */

struct request *request; /* The command we are
                          * working on */

#define SCSI_SENSE_BUFFERSIZE 96
unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE];
/* obtained by REQUEST SENSE
 * when CHECK CONDITION is
 * received on original command
 * (auto-sense) */

/* Low-level done function - can be used by low-level driver to point
 * to completion function. Not used by mid/upper level code. */
void (*scsi_done) (struct scsi_cmnd *);

/*
 * The following fields can be written to by the host specific code.
 * Everything else should be left alone.
 */
struct scsi_pointer SCp; /* Scratchpad used by some host adapters */

unsigned char *host_scribble; /* The host adapter is allowed to
 * call scsi_malloc and get some memory
 * and hang it here. The host adapter
 * is also expected to call scsi_free
 * to release this memory. (The memory
 * obtained by scsi_malloc is guaranteed
 * to be at an address < 16Mb). */

int result; /* Status code from lower level driver */

unsigned char tag; /* SCSI-II queued command tag */
unsigned long pid; /* Process ID, starts at 0 */
};

```

This structure holds information about a scsi device such as a hard disk, cd-rom etc. This structure is defined in the *include/scsi/scsi\_device.h* file

```
struct scsi_device {
```

```

struct Scsi_Host *host;
struct request_queue *request_queue;

/* the next two are protected by the host->host_lock */
struct list_head siblings; /* list of all devices on this host */
struct list_head same_target_siblings; /* just the devices sharing same target id */

volatile unsigned short device_busy; /* commands actually active on low-level */
spinlock_t sdev_lock; /* also the request_queue_lock */
spinlock_t list_lock;
struct list_head cmd_list; /* queue of in use SCSI Command structures */
struct list_head starved_entry;
struct scsi_cmnd *current_cmnd; /* currently active command */
unsigned short queue_depth; /* How deep of a queue we want */
unsigned short last_queue_full_depth; /* These two are used by */
unsigned short last_queue_full_count; /* scsi_track_queue_full() */
unsigned long last_queue_full_time; /* don't let QUEUE_FULLs on the same
                                     jiffie count on our counter, they
                                     could all be from the same event. */

unsigned int id, lun, channel;

unsigned int manufacturer; /* Manufacturer of device, for using
                             * vendor-specific cmd's */
unsigned sector_size; /* size in bytes */

void *hostdata; /* available to low-level driver */
char devfs_name[256]; /* devfs junk */
char type;
char scsi_level;
char inq_periph_qual; /* PQ from INQUIRY data */
unsigned char inquiry_len; /* valid bytes in 'inquiry' */
unsigned char *inquiry; /* INQUIRY response data */
char *vendor; /* [back_compat] point into 'inquiry' ... */
char *model; /* ... after scan; point to static string */
char *rev; /* ... "nullnullnullnull" before scan */
unsigned char current_tag; /* current tag */
struct scsi_target *sdev_target; /* used only for single_lun */

unsigned int sdev_bflags; /* black/white flags as also found in
                           * scsi_devinfo.[hc]. For now used only to
                           * pass settings from slave_alloc to scsi
                           * core. */

unsigned writeable:1;
unsigned removable:1;
unsigned changed:1; /* Data invalid due to media change */
unsigned busy:1; /* Used to prevent races */
unsigned lockable:1; /* Able to prevent media removal */
unsigned locked:1; /* Media removal disabled */
unsigned borken:1; /* Tell the Seagate driver to be
                   * painfully slow on this device */
unsigned disconnect:1; /* can disconnect */
unsigned soft_reset:1; /* Uses soft reset option */
unsigned sdtr:1; /* Device supports SDTR messages */
unsigned wdtr:1; /* Device supports WDTR messages */
unsigned ppr:1; /* Device supports PPR messages */
unsigned tagged_supported:1; /* Supports SCSI-II tagged queuing */
unsigned simple_tags:1; /* simple queue tag messages are enabled */

```

```

unsigned ordered_tags:1; /* ordered queue tag messages are enabled */
unsigned single_lun:1; /* Indicates we should only allow I/O to
                        * one of the luns for the device at a
                        * time. */
unsigned was_reset:1; /* There was a bus reset on the bus for
                       * this device */
unsigned expecting_cc_ua:1; /* Expecting a CHECK_CONDITION/UNIT_ATTEN
                             * because we did a bus reset. */
unsigned use_10_for_rw:1; /* first try 10-byte read / write */
unsigned use_10_for_ms:1; /* first try 10-byte mode sense/select */
unsigned skip_ms_page_8:1; /* do not use MODE SENSE page 0x08 */
unsigned skip_ms_page_3f:1; /* do not use MODE SENSE page 0x3f */
unsigned use_192_bytes_for_3f:1; /* ask for 192 bytes from page 0x3f */
unsigned no_start_on_add:1; /* do not issue start on add */
unsigned allow_restart:1; /* issue START_UNIT in error handler */

unsigned int device_blocked; /* Device returned QUEUE_FULL. */

unsigned int max_device_blocked; /* what device_blocked counts down from */
#define SCSI_DEFAULT_DEVICE_BLOCKED 3

int timeout;

struct device          sdev_gendev;
struct class_device    sdev_classdev;

struct class_device    transport_classdev;

enum scsi_device_state sdev_state;
unsigned long          transport_data[0];
} __attribute__((aligned(sizeof(unsigned long))));

```

## 6.0 SCSI Low Level Driver

The lower level drivers are the ones that support and control the SCSI hardware devices. The hardware devices are often called Host Bus Adapters (HBA). Almost all the lower level drivers can be built either as modules or can be built into the kernel.

The functionalities performed by these drivers include

- Sending out the IO requests to the target system and completing them to the OS
- Performing bus scan functionalities and updating the new devices to the os
- Interact with the custom hardware
- Provides interface functions to the SCSI mid layer

During the initialization time, the low level drivers initialize the hardware, register interrupt routine and to the SCSI interface. Some of the important data that are exchanged with the SCSI mid layer using the host structure include the IO queue depth, total number of SCSI bus, target and LUNS supported. The host template that is exchanged with the mid layer also includes pointers to functions to submit IO's, error handling functions etc. The midlayer will not forward more IO requests than that is supported from the given host.

The SCSI midlayer scans the reported buses for any scsi devices and adds them to the list. All the bus scan functions from the midlayer invoke IO entry point (queuecommand) in the LLD and LLD does not usually distinguish between bus scan CDB or the IO CDB. The commands are placed in Scsi\_cmnd structure and forwarded to the queuecommand. After the bus scan, the SCSI devices are mapped on to the /dev/sdX interface and are ready to accept IO from the kernel.

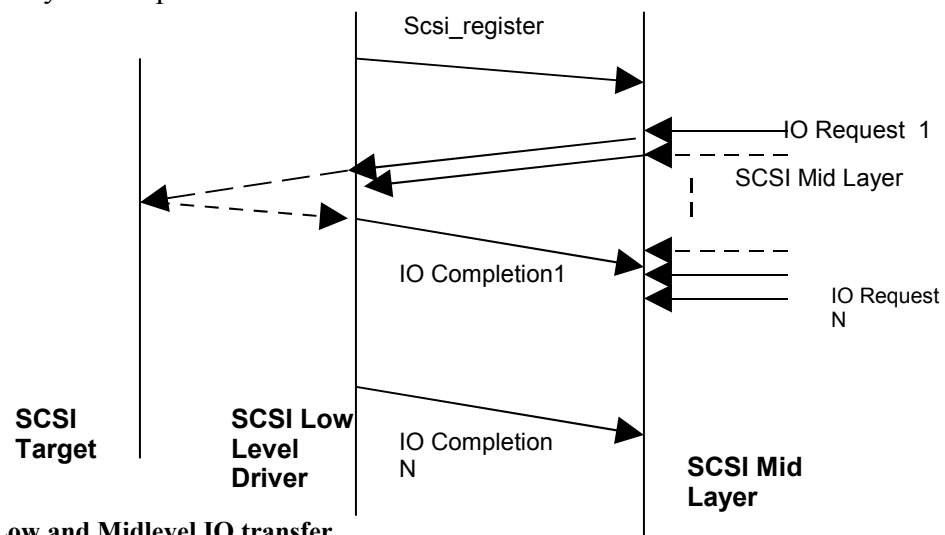


Figure 6 - SCSI IO Low and Midlevel IO transfer

The kernel IO is sent down to the device and is forwarded to the LLD for transferring the data. The hardware performs the DMA of the data and triggers an interrupt to indicate the completion of the IO. The midlayer interface does not distinguish between different types of transport mechanism used by the underlying LLD while delivering the IO command. It is the responsibility of the LLD to transfer the IO to the supported interface over a given transport. The LLD completes the IO using the scsi\_done pointer stored in the command request.

The LLD provides interface to the proc interface, which helps in managing the LLD interface. The LLD also provides interfaces for the Task Management functionality such as aborting a given command, device reset, bus reset and host reset etc.

The low level SCSI drivers provide following entry points to the Mid Layer using the template structure. A good number of these entry points are optional.

## 6.1 SCSI Low Level Driver functional entry points

These are the function implemented by the LLD and the SCSI midlayer and the top layer uses these functions for the data transfer.

### 6.1.1 bios\_param

This function fetches head, sector, and cylinder information of the disk.

Returns: 0 on success

Called From: This is called from sd\_hdio\_getgeo to get the disk geometry by the SD layer

Input: pointer to scsi device, pointer to block device, device size, and params that will hold the data for number of heads, number of sectors and number of cylinders

```
int bios_param(struct scsi_device * sdev, struct block_device *bdev,  
              sector_t capacity, int params[3])
```

### 6.1.2 detect

This function returns number of hosts this driver wants to control. This function is also used by the LLD modules to initialize each of the controllers it would like to control. Each of the controllers also registers with the scsi mid layer using scsi\_register function. This is usually used in LLD modules which use passive mode of initialization and include scsi\_module.c. This scsi\_module.c function calls the detect function in its module\_init

Returns: total number of controllers it would like to control

Called From: init\_this\_scsi\_driver from scsi\_module.c

Input: scsi\_host\_template \* shtp

```
int detect(struct scsi_host_template * shtp)
```

### 6.1.3 eh\_timed\_out

The function gets called when the timeout on the command has triggered.

Returns: This function returns the current state of the command and conveys if the normal error recovery should start now on this command or not depending on the state.

Called From: scsi\_times\_out in the scsi mid layer

Input: struct scsi\_cmnd \* scp

```
int eh_timed_out(struct scsi_cmnd * scp)
```

### 6.1.4 eh\_abort\_handler

The function gets called to abort a command submitted to the LLD.

Returns: SUCCESS if command aborted else FAILED

Called From: scsi\_send\_eh\_cmnd and scsi\_try\_abort\_cmd in the scsi mid layer

Input: struct scsi\_cmnd \* scp

```
int eh_abort_handler(struct scsi_cmnd * scp)
```

### 6.1.5 eh\_bus\_reset\_handler

This function asks the LLD to send bus reset TMF

Returns: SUCCESS if reset else returns FAILED

Called From: scsi\_try\_bus\_reset in the scsi mid layer

Input: struct scsi\_cmnd \* scp

```
int eh_bus_reset_handler(struct scsi_cmnd * scp)
```

### **6.1.6 eh\_device\_reset\_handler**

This function will reset the device using a TMF

Returns: SUCCESS if reset else returns FAILED

Called From: scsi\_try\_bus\_device\_reset in the scsi mid layer

Input: struct scsi\_cmnd \* scp

```
int eh_device_reset_handler(struct scsi_cmnd * scp)
```

### **6.1.7 eh\_host\_reset\_handler**

This function will reset the host. The device will be put to offline if all the error handling functions returned a failure.

Returns: SUCCESS if reset else returns FAILED

Called From: scsi\_try\_host\_reset in the scsi mid layer

Input: struct scsi\_cmnd \* scp

```
int eh_device_reset_handler(struct scsi_cmnd * scp)
```

### **6.1.8 eh\_strategy\_handler**

This function will try to get the host online in the given circumstances.

Returns: TRUE if host unjammed, else FALSE

Called From: scsi\_error\_handler in the scsi mid layer

Input: struct Scsi\_Host \* shp

```
int eh_strategy_handler(struct Scsi_Host * shp)
```

### **6.1.9 info**

This function will try to get the host information

Returns: Null terminated string of information

Called From: ioctl\_probe in the scsi mid layer

Input: struct Scsi\_Host \* shp

```
const char * info(struct Scsi_Host * shp)
```

### **6.1.10 ioctl**

This function provides ioctl entry point to get the host information

Returns: Negative "errno" value when there is a problem. 0 or a positive value indicates success

Called From: ioctl in the scsi mid layer

Input: struct scsi\_device \*sdp, int cmd, void \*arg

```
int ioctl(struct scsi_device *sdp, int cmd, void *arg)
```

### 6.1.11 proc\_info

This function provides ioctl entry point to get the host information

Returns: The length of the character array it has output on the proc file system

Called From: SCSI proc interface

Input: char \* buffer, char \*\* start, off\_t offset, int length, int hostno, int

writeto1\_read0

int proc\_info(char \* buffer, char \*\* start, off\_t offset, int length, int hostno, int writeto1\_read0)

### 6.1.12 queuecommand

This function is the main entry point for all the IO operations that an LLD supports. This function is relatively fast and will not wait for the IO to complete before it returns. The done call back routine will be usually be called after the IO routine is completed in the ISR.

Returns: 0 on success, SCSI\_MLQUEUE\_DEVICE\_BUSY if the target device on which IO is being performed is busy and SCSI\_MLQUEUE\_HOST\_BUSY if the entire host is busy. If the command was returned with SUCCESS then the driver will call the done callback function. If not it will not call the callback done function.

Called From: scsi\_dispatch\_cmnd and scsi\_send\_eh\_cmnd in the scsi mid layer  
struct scsi\_cmnd \* scp, void (\*done)(struct scsi\_cmnd \*)

int queuecommand(struct scsi\_cmnd \* scp, void (\*done)(struct scsi\_cmnd \*))

### 6.1.13 release

This function releases all resources that are tied to a given host. This is the counterpart of detect and is used in the passive initialization model. This function will call scsi\_unregister before it returns.

Returns: Ignored

Called From: scsi\_module.c during the module exit

Input: struct Scsi\_Host \* shp

int release(struct Scsi\_Host \* shp)

### 6.1.14 slave\_alloc

This function allows the driver to allocate any resources for a device before it is going to be scanned.

Returns: 0 on SUCCESS and any other value to indicate failure on this device

Called From: scsi\_alloc\_sdev in scsi\_scan.c during the device scan

Input: struct scsi\_device \*sdp

int slave\_alloc(struct scsi\_device \*sdp)

### 6.1.15 slave\_configure

This function allows the driver to inspect the response to the initial INQUIRY done by the scanning code to take care of appropriate action

Returns: 0 on SUCCESS and any other value to indicate failure on this device and hence taken offline

Called From: scsi\_alloc\_sdev in scsi\_scan.c during the device scan

Input: struct scsi\_device \*sdp

int slave\_alloc(struct scsi\_device \*sdp)

### 6.1.16 slave\_destroy

This function allows the driver to release any resources that were allocated on the device.

Returns: 0 on SUCCESS and any other value to indicate failure on this device and hence taken offline

Called From: scsi\_free\_sdev and any other failures on the scsi device in scsi\_scan.c Input: struct scsi\_device \*sdp

void slave\_destroy(struct scsi\_device \*sdp)

## 6.2 Important Scsi\_Host parameters set by the LLD

Parameters	Brief Explanation
unsigned int max_id	This is the maximum SCSI ID for devices attached to this bus
unsigned int max_lun	This is the maximum SCSI LUN for devices attached to this bus
unsigned int max_channel	This is the maximum channel for devices attached to this bus.
unsigned int unique_id	This is used to identify the device
unsigned char max_cmd_len	Maximum length of the SCSI command (CDB) length. Default is set to 12
int this_id	SCSI id of this initiator
int can_queue	Max outstanding commands that can be sent to the initiator device
short cmd_per_lun	Max outstanding commands to a given LUN
short unsigned int sg_tablesize	The maximum number of sg elements supported by this device
unsigned use_clustering:1	If set, the host will try to merge multiple IO request blocks into one. This should be not set for a device that supports fixed size IO
unsigned host_self_blocked:1	Host sets this if it does not like to receive any more requests
unsigned int host_blocked	This flag is set if the host adapter rejected a command
unsigned int max_host_blocked	Value host_blocked counts down from

Table 8 - Scsi\_Host parameters set by LLD

### 6.3 Important Scsi\_Cmnd parameters provided by SCSI mid layer

Parameters	Brief Explanation
void (*done) (struct scsi_cmnd *)	Function pointer to be called when the IO has completed
unsigned long serial_number	A nonzero serial_number is assigned by the scsi_dispatch_cmd for a given command. The serial_number is cleared when scsi_done is entered indicating that the command has been completed.
unsigned long serial_number_at_timeout	If a timeout occurs, the serial number at the moment of timeout is copied into serial_number_at_timeout field. This is used by the error handling process to know if the IO was already completed to the host.
int retries	Total number of command retries tried on this command.
int allowed	Maximum number of retries that can be used on a given command. This is filled in by the upper layer. For sd it is set to 5
int timeout_per_command	The command time out period
unsigned char cmd_len	Command length
unsigned char old_cmd_len	Saved copy of the command length. Used in the error recovery
unsigned char cmd[MAX_COMMAND_SIZE]	SCSI Command
unsigned request_bufflen	This is the saved buffer length associated with the request.
struct timer_list eh_timeout	Timer structure used for timeout error handling
void *request_buffer	This is the saved buffer pointer associated with the request
unsigned char data_cmnd[MAX_COMMAND_SIZE]	This is a saved copy of the cmd array
unsigned short old_use_sg	This is a saved copy of use_sg
unsigned short use_sg	If 0, it means that the buffer field indicates the actual I/O buffer, if non-zero, the buffer field indicates the address of the scatter-gather table, and the use_sg field will indicate the number of entries in the scatter-gather table.
unsigned short sglst_len	Amount of memory allocated for the scatter-gather list
unsigned short abort_reason	indicates the reason that the command is being aborted
unsigned bufflen	Data transfer length in bytes
void *buffer	Pointer to buffer or SG table

unsigned underflow	Minimum number of bytes for data transfer
unsigned transfersize	Amount of data that we are guaranteed to transfer with each SCSI transfer
struct request *request	Pointer to request structure sent down by the block interface
unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE]	Place to hold sense data
void (*scsi_done) (struct scsi_cmnd *)	Function pointer to be called when the IO has completed
struct scsi_pointer SCp	this is used as a scratchpad
int result	Status code from lower level driver
Unsigned char *host_scribble	This field is used as a pointer to a scratchpad area that low-level drivers can use

**Table 9 - Scsi\_Cmnd parameters provided by SCSI mid layer**

## 6.4 SCSI mid layer functions exported to LLD

<b>Functions</b>	<b>Brief Explanation</b>
scsi_add_device	creates new scsi device (lu) instance
scsi_add_host	Perform sysfs registration and SCSI bus scan
scsi_add_timer	(re-)start timer on a SCSI command
scsi_adjust_queue_depth	change the queue depth on a SCSI device
scsi_assign_lock	replace default host lock with given lock
scsi_bios_ptable	return copy of block device's partition table
scsi_block_requests	prevent further commands being queued to given host
scsi_delete_timer	cancel timer on a SCSI command
scsi_host_alloc	return a new scsi_host instance whose refcount==1
scsi_host_get	increments Scsi_Host instance's refcount
scsi_host_put	decrements Scsi_Host instance's refcount (free if 0)
scsi_partsize	parse partition table into cylinders, heads + sectors
scsi_register	create and register a scsi host adapter instance
scsi_remove_device	detach and remove a SCSI device
scsi_remove_host	detach and remove all SCSI devices owned by host
scsi_report_bus_reset -	report scsi_bus_reset observed
scsi_set_device -	place device reference in host structure
scsi_to_pci_dma_dir -	convert SCSI subsystem direction flag to PCI
scsi_to_sbus_dma_dir -	convert SCSI subsystem direction flag to SBUS
scsi_track_queue_full -	track successive QUEUE_FULL events
scsi_unblock_requests -	allows further commands to be queued to given host
Scsi_unregister -	Unregister scsi interface

**Table 10 - MidLayer exported functions to LLD**

## References

### Internet References

- 1) Douglas, Gilbert (2004) The Linux 2.4 SCSI subsystem HOWTO, Rev 2.0, Retrieved 06/17/2004 from <http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/>
- 2) Douglas, Gilbert (2002) The Linux SCSI Generic (sg) HOWTO Rev 1.2, Retrieved 06/17/2004 from <http://tldp.org/HOWTO/SCSI-Generic-HOWTO/>
- 3) YoungDale, Eric (2002) Linux SCSI resources, Retrieved 06/17/2004 from <http://www.andante.org/scsi.html>
- 4) T10 committee, SCSI (draft) standards (SAM-3, SBC-2, SPC3), Retrieved 02/15/2004 from [www.t10.org](http://www.t10.org)
- 5) Linux Kernel Developers, Linux Kernel Source and Documentation 2.6.X, Retrieved 03/10/2004 from [www.kernel.org](http://www.kernel.org)

### Book References

- 1) Beck M, Bohme H, Dziadzka M, Kunitz U, Magnus R, Verworner D; Addison-Wesley - Second Edition (1998), Linux Kernel Internals
- 2) Rubini, Alessandro and Corbet, Jonathan; O'Reilly and Associates, Second Edition (2002) - Linux Device Drivers

### Credits

- 1) Amir A Vetry
- 2) Randy Dunlap

### Corrections and Suggestions

Pls e-mail your corrections and suggestions to [samdeep@ieee.org](mailto:samdeep@ieee.org)