SCSI Interfaces Guide

James Bottomley <James.Bottomley@hansenpartnership.com>
Rob Landley <rob@landley.net>

SCSI Interfaces Guide

by James Bottomley and Rob Landley

Copyright © 2007 Linux Foundation

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. For more details see the file COPYING in the source distribution of Linux.

Table of Contents

1. Introduction 1
    Protocol vs bus 1
    Design of the Linux SCSI subsystem 1
2. SCSI upper layer 2
    sd (SCSI Disk) 2
    sr (SCSI CD-ROM) 2
    st (SCSI Tape) 2
    sg (SCSI Generic) 2
    ch (SCSI Media Changer) 2
3. SCSI mid layer 3
    SCSI midlayer implementation 3
        include/scsi/scsi_device.h 3
        drivers/scsi/scsi.c 6
        drivers/scsi/scsicam.c 19
        drivers/scsi/scsi_error.c 22
        drivers/scsi/scsi_devinfo.c 33
        drivers/scsi/scsi_ioctl.c 39
        drivers/scsi/scsi_lib.c 40
        drivers/scsi/scsi_lib_dma.c 55
        drivers/scsi/scsi_module.c 57
        drivers/scsi/scsi_proc.c 57
        drivers/scsi/scsi_netlink.c 68
        drivers/scsi/scsi_scan.c 71
        drivers/scsi/scsi_sysctl.c 86
        drivers/scsi/scsi_sysfs.c 86
        drivers/scsi/hosts.c 88
        drivers/scsi/constants.c 97
    Transport classes 98
        Fibre Channel transport 98
        iSCSI transport class 108
        Serial Attached SCSI (SAS) transport class 123
        SATA transport class 152
        Parallel SCSI (SPI) transport class 152
        SCSI RDMA (SRP) transport class 155
4. SCSI lower layer 167
    Host Bus Adapter transport types 167
        Debug transport 167
        todo 167

Chapter 1. Introduction

Protocol vs bus

Once upon a time, the Small Computer Systems Interface defined both a parallel I/O bus and a data protocol to connect a wide variety of peripherals (disk drives, tape drives, modems, printers, scanners, optical drives, test equipment, and medical devices) to a host computer.

Although the old parallel (fast/wide/ultra) SCSI bus has largely fallen out of use, the SCSI command set is more widely used than ever to communicate with devices over a number of different busses.

The SCSI protocol [http://www.t10.org/scsi-3.htm] is a big-endian peer-to-peer packet based protocol. SCSI commands are 6, 10, 12, or 16 bytes long, often followed by an associated data payload.

SCSI commands can be transported over just about any kind of bus, and are the default protocol for storage devices attached to USB, SATA, SAS, Fibre Channel, FireWire, and ATAPI devices. SCSI packets are also commonly exchanged over Infiniband, I20 [http://i2o.shadowconnect.com/faq.php], TCP/IP (iSCSI [https://en.wikipedia.org/wiki/ISCSI]), even Parallel ports [http://cyberelk.net/tim/parport/parscsi.html].

Design of the Linux SCSI subsystem

The SCSI subsystem uses a three layer design, with upper, mid, and low layers. Every operation involving the SCSI subsystem (such as reading a sector from a disk) uses one driver at each of the 3 levels: one upper layer driver, one lower layer driver, and the SCSI midlayer.

The SCSI upper layer provides the interface between userspace and the kernel, in the form of block and char device nodes for I/O and ioctl(). The SCSI lower layer contains drivers for specific hardware devices.

In between is the SCSI mid-layer, analogous to a network routing layer such as the IPv4 stack. The SCSI mid-layer routes a packet based data protocol between the upper layer's /dev nodes and the corresponding devices in the lower layer. It manages command queues, provides error handling and power management functions, and responds to ioctl() requests.

Chapter 2. SCSI upper layer

The upper layer supports the user-kernel interface by providing device nodes.

sd (SCSI Disk)

sd (sd_mod.o)

sr (SCSI CD-ROM)

sr (sr_mod.o)

st (SCSI Tape)

st (st.o)

sg (SCSI Generic)

sg (sg.o)

ch (SCSI Media Changer)

ch (ch.c)

Chapter 3. SCSI mid layer

SCSI midlayer implementation

include/scsi/scsi_device.h

Name

shost_for_each_device — iterate over all devices of a host

Synopsis

```
shost_for_each_device ( sdev, shost);
```

Arguments

sdev the struct scsi_device to use as a cursor

shost the struct scsi_host to iterate over

Description

Iterator that returns each device attached to *shost*. This loop takes a reference on each device and releases it at the end. If you break out of the loop, you must call scsi_device_put(sdev).

Name

__shost_for_each_device — iterate over all devices of a host (UNLOCKED)

Synopsis

```
__shost_for_each_device ( sdev, shost);
```

Arguments

sdev the struct scsi_device to use as a cursor

shost the struct scsi_host to iterate over

Description

Iterator that returns each device attached to *shost*. It does _not_ take a reference on the scsi_device, so the whole loop must be protected by shost->host_lock.

Note

The only reason to use this is because you need to access the device list in interrupt context. Otherwise you really want to use shost_for_each_device instead.

Name

scsi_device_supports_vpd — test if a device supports VPD pages

Synopsis

```
int scsi_device_supports_vpd (struct scsi_device * sdev);
```

Arguments

sdev the struct scsi_device to test

Description

If the 'try_vpd_pages' flag is set it takes precedence. Otherwise we will assume VPD pages are supported if the SCSI level is at least SPC-3 and 'skip_vpd_pages' is not set.

drivers/scsi/scsi.c

Main file for the SCSI midlayer.

Name

scsi_cmd_get_serial — Assign a serial number to a command

Synopsis

```
void scsi_cmd_get_serial (struct Scsi_Host * host, struct scsi_cmnd *
cmd);
```

Arguments

host the scsi host

cmd command to assign serial number to

Description

a serial number identifies a request for error recovery and debugging purposes. Protected by the Host_Lock of host.

Name

scsi_change_queue_depth — change a device's queue depth

Synopsis

```
int scsi_change_queue_depth (struct scsi_device * sdev, int depth);
```

Arguments

sdev SCSI Device in question

depth number of commands allowed to be queued to the driver

Description

Sets the device queue depth and returns the new value.

Name

scsi_track_queue_full — track QUEUE_FULL events to adjust queue depth

Synopsis

```
int scsi_track_queue_full (struct scsi_device * sdev, int depth);
```

Arguments

sdev SCSI Device in question

depth Current number of outstanding SCSI commands on this device, not counting the one returned as QUEUE_FULL.

Description

This function will track successive QUEUE_FULL events on a specific SCSI device to determine if and when there is a need to adjust the queue depth on the device.

Returns

0 - No change needed, >0 - Adjust queue depth to this new depth, -1 - Drop back to untagged operation using host->cmd_per_lun as the untagged command depth

Lock Status

None held on entry

Notes

Low level drivers may call this at any time and we will do “The Right Thing.” We are interrupt context safe.

Name

scsi_get_vpd_page — Get Vital Product Data from a SCSI device

Synopsis

```
int scsi_get_vpd_page (struct scsi_device * sdev, u8 page, unsigned char
* buf, int buf_len);
```

Arguments

sdev	The device to ask
page	Which Vital Product Data to return
buf	where to store the VPD
buf_len	number of bytes in the VPD buffer area

Description

SCSI devices may optionally supply Vital Product Data. Each 'page' of VPD is defined in the appropriate SCSI document (eg SPC, SBC). If the device supports this VPD page, this routine returns a pointer to a buffer containing the data from that page. The caller is responsible for calling `kfree` on this pointer when it is no longer needed. If we cannot retrieve the VPD page this routine returns `NULL`.

Name

scsi_report_opcode — Find out if a given command opcode is supported

Synopsis

```
int scsi_report_opcode (struct scsi_device * sdev, unsigned char *
buffer, unsigned int len, unsigned char opcode);
```

Arguments

sdev scsi device to query

buffer scratch buffer (must be at least 20 bytes long)

len length of buffer

opcode opcode for command to look up

Description

Uses the REPORT SUPPORTED OPERATION CODES to look up the given opcode. Returns -EINVAL if RSOC fails, 0 if the command opcode is unsupported and 1 if the device claims to support the command.

Name

scsi_device_get — get an additional reference to a scsi_device

Synopsis

```
int scsi_device_get (struct scsi_device * sdev);
```

Arguments

sdev device to get a reference to

Description

Gets a reference to the scsi_device and increments the use count of the underlying LLDD module. You must hold host_lock of the parent Scsi_Host or already have a reference when calling this.

This will fail if a device is deleted or cancelled, or when the LLD module is in the process of being unloaded.

Name

scsi_device_put — release a reference to a scsi_device

Synopsis

```
void scsi_device_put (struct scsi_device * sdev);
```

Arguments

sdev device to release a reference on.

Description

Release a reference to the scsi_device and decrements the use count of the underlying LLDD module. The device is freed once the last user vanishes.

Name

starget_for_each_device — helper to walk all devices of a target

Synopsis

```
void starget_for_each_device (struct scsi_target * starget, void * data,
void (*fn) (struct scsi_device *, void *));
```

Arguments

starget	target whose devices we want to iterate over.
data	Opaque passed to each function call.
fn	Function to call on each device

Description

This traverses over each device of *starget*. The devices have a reference that must be released by scsi_host_put when breaking out of the loop.

Name

__starget_for_each_device — helper to walk all devices of a target (UNLOCKED)

Synopsis

```
void __starget_for_each_device (struct scsi_target * starget, void *
data, void (*fn) (struct scsi_device *, void *));
```

Arguments

starget target whose devices we want to iterate over.

data parameter for callback *fn*()

fn callback function that is invoked for each device

Description

This traverses over each device of *starget*. It does _not_ take a reference on the scsi_device, so the whole loop must be protected by shost->host_lock.

Note

The only reason why drivers would want to use this is because they need to access the device list in irq context. Otherwise you really want to use starget_for_each_device instead.

Name

__scsi_device_lookup_by_target — find a device given the target (UNLOCKED)

Synopsis

```
struct scsi_device * __scsi_device_lookup_by_target (struct scsi_target
* starget, u64 lun);
```

Arguments

starget SCSI target pointer

lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified *lun* for a given *starget*. The returned scsi_device does not have an additional reference. You must hold the host's host_lock over this call and any access to the returned scsi_device. A scsi_device in state SDEV_DEL is skipped.

Note

The only reason why drivers should use this is because they need to access the device list in irq context. Otherwise you really want to use scsi_device_lookup_by_target instead.

Name

scsi_device_lookup_by_target — find a device given the target

Synopsis

```
struct scsi_device * scsi_device_lookup_by_target (struct scsi_target
* starget, u64 lun);
```

Arguments

starget SCSI target pointer

lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified *lun* for a given *starget*. The returned scsi_device has an additional reference that needs to be released with scsi_device_put once you're done with it.

Name

__scsi_device_lookup — find a device given the host (UNLOCKED)

Synopsis

```
struct scsi_device * __scsi_device_lookup (struct Scsi_Host * shost,
uint channel, uint id, u64 lun);
```

Arguments

shost	SCSI host pointer
channel	SCSI channel (zero if only one channel)
id	SCSI target number (physical unit number)
lun	SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified *channel*, *id*, *lun* for a given host. The returned scsi_device does not have an additional reference. You must hold the host's host_lock over this call and any access to the returned scsi_device.

Note

The only reason why drivers would want to use this is because they need to access the device list in irq context. Otherwise you really want to use scsi_device_lookup instead.

Name

scsi_device_lookup — find a device given the host

Synopsis

```
struct scsi_device * scsi_device_lookup (struct Scsi_Host * shost, uint channel, uint id, u64 lun);
```

Arguments

shost	SCSI host pointer
channel	SCSI channel (zero if only one channel)
id	SCSI target number (physical unit number)
lun	SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified *channel*, *id*, *lun* for a given host. The returned scsi_device has an additional reference that needs to be released with scsi_device_put once you're done with it.

drivers/scsi/scsicam.c

SCSI Common Access Method [http://www.t10.org/ftp/t10/drafts/cam/cam-r12b.pdf] support functions, for use with HDIO_GETGEO, etc.

Name

scsi_bios_ptable — Read PC partition table out of first sector of device.

Synopsis

```
unsigned char * scsi_bios_ptable (struct block_device * dev);
```

Arguments

dev from this device

Description

Reads the first sector from the device and returns `0x42` bytes starting at offset `0x1be`.

Returns

partition table in kmalloc(GFP_KERNEL) memory, or NULL on error.

Name

scsicam_bios_param — Determine geometry of a disk in cylinders/heads/sectors.

Synopsis

```
int scsicam_bios_param (struct block_device * bdev, sector_t capacity,
int * ip);
```

Arguments

bdev	which device
capacity	size of the disk in sectors
ip	return value: ip[0]=heads, ip[1]=sectors, ip[2]=cylinders

Description

determine the BIOS mapping/geometry used for a drive in a SCSI-CAM system, storing the results in ip as required by the HDIO_GETGEO `ioctl`.

Returns

-1 on failure, 0 on success.

Name

scsi_partsize — Parse cylinders/heads/sectors from PC partition table

Synopsis

```
int scsi_partsize (unsigned char * buf, unsigned long capacity, unsigned
int * cyls, unsigned int * hds, unsigned int * secs);
```

Arguments

buf	partition table, see `scsi_bios_ptable`
capacity	size of the disk in sectors
cyls	put cylinders here
hds	put heads here
secs	put sectors here

Description

determine the BIOS mapping/geometry used to create the partition table, storing the results in *cyls, *hds, and *secs

Returns

-1 on failure, 0 on success.

drivers/scsi/scsi_error.c

Common SCSI error/timeout handling routines.

Name

scsi_schedule_eh — schedule EH for SCSI host

Synopsis

```
void scsi_schedule_eh (struct Scsi_Host * shost);
```

Arguments

shost SCSI host to invoke error handling on.

Description

Schedule SCSI EH without scmd.

Name

scsi_block_when_processing_errors — Prevent cmds from being queued.

Synopsis

```
int scsi_block_when_processing_errors (struct scsi_device * sdev);
```

Arguments

sdev Device on which we are performing recovery.

Description

We block until the host is out of error recovery, and then check to see whether the host or the device is offline.

Return value

0 when dev was taken offline by error recovery. 1 OK to proceed.

Name

scsi_check_sense — Examine scsi cmd sense

Synopsis

```
int scsi_check_sense (struct scsi_cmnd * scmd);
```

Arguments

scmd Cmd to have sense checked.

Return value

SUCCESS or FAILED or NEEDS_RETRY or ADD_TO_MLQUEUE

Notes

When a deferred error is detected the current command has not been executed and needs retrying.

Name

scsi_eh_prep_cmnd — Save a scsi command info as part of error recovery

Synopsis

```
void scsi_eh_prep_cmnd (struct scsi_cmnd * scmd, struct scsi_eh_save *
ses, unsigned char * cmnd, int cmnd_size, unsigned sense_bytes);
```

Arguments

scmd	SCSI command structure to hijack
ses	structure to save restore information
cmnd	CDB to send. Can be NULL if no new cmnd is needed
cmnd_size	size in bytes of *cmnd* (must be <= BLK_MAX_CDB)
sense_bytes	size of sense data to copy. or 0 (if != 0 *cmnd* is ignored)

Description

This function is used to save a scsi command information before re-execution as part of the error recovery process. If *sense_bytes* is 0 the command sent must be one that does not transfer any data. If *sense_bytes* != 0 *cmnd* is ignored and this functions sets up a REQUEST_SENSE command and cmnd buffers to read *sense_bytes* into *scmd*->sense_buffer.

Name

scsi_eh_restore_cmnd — Restore a scsi command info as part of error recovery

Synopsis

```
void scsi_eh_restore_cmnd (struct scsi_cmnd * scmd, struct scsi_eh_save
* ses);
```

Arguments

scmd SCSI command structure to restore

ses saved information from a coresponding call to scsi_eh_prep_cmnd

Description

Undo any damage done by above `scsi_eh_prep_cmnd`.

Name

scsi_eh_finish_cmd — Handle a cmd that eh is finished with.

Synopsis

```
void scsi_eh_finish_cmd (struct scsi_cmnd * scmd, struct list_head *
done_q);
```

Arguments

scmd Original SCSI cmd that eh has finished.

done_q Queue for processed commands.

Notes

We don't want to use the normal command completion while we are are still handling errors - it may cause other commands to be queued, and that would disturb what we are doing. Thus we really want to keep a list of pending commands for final completion, and once we are ready to leave error handling we handle completion for real.

Name

scsi_eh_get_sense — Get device sense data.

Synopsis

```
int scsi_eh_get_sense (struct list_head * work_q, struct list_head *
done_q);
```

Arguments

work_q Queue of commands to process.

done_q Queue of processed commands.

Description

See if we need to request sense information. if so, then get it now, so we have a better idea of what to do.

Notes

This has the unfortunate side effect that if a shost adapter does not automatically request sense information, we end up shutting it down before we request it.

All drivers should request sense information internally these days, so for now all I have to say is tough noogies if you end up in here.

XXX

Long term this code should go away, but that needs an audit of all LLDDs first.

Name

scsi_eh_ready_devs — check device ready state and recover if not.

Synopsis

```
void scsi_eh_ready_devs (struct Scsi_Host * shost, struct list_head *
work_q, struct list_head * done_q);
```

Arguments

shost host to be recovered.

work_q list_head for pending commands.

done_q list_head for processed commands.

Name

scsi_eh_flush_done_q — finish processed commands or retry them.

Synopsis

```
void scsi_eh_flush_done_q (struct list_head * done_q);
```

Arguments

done_q list_head of processed commands.

Name

scsi_ioctl_reset —

Synopsis

```
int scsi_ioctl_reset (struct scsi_device * dev, int __user * arg);
```

Arguments

dev scsi_device to operate on

arg reset type (see sg.h)

Name

scsi_get_sense_info_fld — get information field from sense data (either fixed or descriptor format)

Synopsis

```
int scsi_get_sense_info_fld (const u8 * sense_buffer, int sb_len, u64
* info_out);
```

Arguments

sense_buffer	byte array of sense data
sb_len	number of valid bytes in sense_buffer
info_out	pointer to 64 integer where 8 or 4 byte information field will be placed if found.

Return value

1 if information field found, 0 if not found.

drivers/scsi/scsi_devinfo.c

Manage scsi_dev_info_list, which tracks blacklisted and whitelisted devices.

Name

scsi_dev_info_list_add — add one dev_info list entry.

Synopsis

```
int scsi_dev_info_list_add (int compatible, char * vendor, char * model,
char * strflags, int flags);
```

Arguments

compatible	if true, null terminate short strings. Otherwise space pad.
vendor	vendor string
model	model (product) string
strflags	integer string
flags	if strflags NULL, use this flag value

Description

Create and add one dev_info entry for *vendor*, *model*, *strflags* or *flag*. If *compatible*, add to the tail of the list, do not space pad, and set devinfo->compatible. The scsi_static_device_list entries are added with *compatible* 1 and *clfags* NULL.

Returns

0 OK, -error on failure.

Name

scsi_dev_info_list_find — find a matching dev_info list entry.

Synopsis

```
struct scsi_dev_info_list * scsi_dev_info_list_find (const char *
vendor, const char * model, int key);
```

Arguments

vendor vendor string

model model (product) string

key specify list to use

Description

Finds the first dev_info entry matching *vendor*, *model* in list specified by *key*.

Returns

pointer to matching entry, or ERR_PTR on failure.

Name

scsi_dev_info_list_add_str — parse dev_list and add to the scsi_dev_info_list.

Synopsis

```
int scsi_dev_info_list_add_str (char * dev_list);
```

Arguments

dev_list string of device flags to add

Description

Parse dev_list, and add entries to the scsi_dev_info_list. dev_list is of the form “vendor:product:flag,vendor:product:flag”. dev_list is modified via strsep. Can be called for command line addition, for proc or mabye a sysfs interface.

Returns

0 if OK, -error on failure.

Name

scsi_get_device_flags — get device specific flags from the dynamic device list.

Synopsis

```
int scsi_get_device_flags (struct scsi_device * sdev, const unsigned
char * vendor, const unsigned char * model);
```

Arguments

sdev scsi_device to get flags for

vendor vendor name

model model name

Description

Search the global scsi_dev_info_list (specified by list zero) for an entry matching *vendor* and *model*, if found, return the matching flags value, else return the host or global default settings. Called during scan time.

Name

scsi_exit_devinfo — remove /proc/scsi/device_info & the scsi_dev_info_list

Synopsis

```
void scsi_exit_devinfo ( void);
```

Arguments

void no arguments

Name

scsi_init_devinfo — set up the dynamic device list.

Synopsis

```
int scsi_init_devinfo ( void);
```

Arguments

void no arguments

Description

Add command line entries from scsi_dev_flags, then add scsi_static_device_list entries to the scsi device info list.

drivers/scsi/scsi_ioctl.c

Handle ioctl() calls for SCSI devices.

Name

scsi_ioctl — Dispatch ioctl to scsi device

Synopsis

```
int scsi_ioctl (struct scsi_device * sdev, int cmd, void __user * arg);
```

Arguments

sdev scsi device receiving ioctl

cmd which ioctl is it

arg data associated with ioctl

Description

The `scsi_ioctl` function differs from most ioctls in that it does not take a major/minor number as the dev field. Rather, it takes a pointer to a struct scsi_device.

drivers/scsi/scsi_lib.c

SCSI queuing library.

Name

scsi_execute — insert request and wait for the result

Synopsis

```
int scsi_execute (struct scsi_device * sdev, const unsigned char * cmd,
int data_direction, void * buffer, unsigned bufflen, unsigned char *
sense, int timeout, int retries, u64 flags, int * resid);
```

Arguments

sdev	scsi device
cmd	scsi command
data_direction	data direction
buffer	data buffer
bufflen	len of buffer
sense	optional sense buffer
timeout	request timeout in seconds
retries	number of times to retry request
flags	or into request flags;
resid	optional residual length

Description

returns the req->errors value which is the scsi_cmnd result field.

Name

scsi_mode_select — issue a mode select

Synopsis

```
int scsi_mode_select (struct scsi_device * sdev, int pf, int sp, int modepage, unsigned char * buffer, int len, int timeout, int retries, struct scsi_mode_data * data, struct scsi_sense_hdr * sshdr);
```

Arguments

sdev	SCSI device to be queried
pf	Page format bit (1 == standard, 0 == vendor specific)
sp	Save page bit (0 == don't save, 1 == save)
modepage	mode page being requested
buffer	request buffer (may not be smaller than eight bytes)
len	length of request buffer.
timeout	command timeout
retries	number of retries before failing
data	returns a structure abstracting the mode header data
sshdr	place to put sense data (or NULL if no sense to be collected). must be SCSI_SENSE_BUFFERSIZE big.

Description

Returns zero if successful; negative error number or scsi status on error

Name

scsi_mode_sense — issue a mode sense, falling back from 10 to six bytes if necessary.

Synopsis

```
int scsi_mode_sense (struct scsi_device * sdev, int dbd, int modepage,
unsigned char * buffer, int len, int timeout, int retries, struct
scsi_mode_data * data, struct scsi_sense_hdr * sshdr);
```

Arguments

sdev	SCSI device to be queried
dbd	set if mode sense will allow block descriptors to be returned
modepage	mode page being requested
buffer	request buffer (may not be smaller than eight bytes)
len	length of request buffer.
timeout	command timeout
retries	number of retries before failing
data	returns a structure abstracting the mode header data
sshdr	place to put sense data (or NULL if no sense to be collected). must be SCSI_SENSE_BUFFERSIZE big.

Description

Returns zero if unsuccessful, or the header offset (either 4 or 8 depending on whether a six or ten byte command was issued) if successful.

Name

scsi_test_unit_ready — test if unit is ready

Synopsis

```
int scsi_test_unit_ready (struct scsi_device * sdev, int timeout, int
retries, struct scsi_sense_hdr * sshdr_external);
```

Arguments

sdev	scsi device to change the state of.
timeout	command timeout
retries	number of retries before failing
sshdr_external	Optional pointer to struct scsi_sense_hdr for returning sense. Make sure that this is cleared before passing in.

Description

Returns zero if unsuccessful or an error if TUR failed. For removable media, UNIT_ATTENTION sets ->changed flag.

Name

scsi_device_set_state — Take the given device through the device state model.

Synopsis

```
int scsi_device_set_state (struct scsi_device * sdev, enum scsi_device_state state);
```

Arguments

sdev scsi device to change the state of.

state state to change to.

Description

Returns zero if unsuccessful or an error if the requested transition is illegal.

Name

sdev_evt_send — send asserted event to uevent thread

Synopsis

```
void sdev_evt_send (struct scsi_device * sdev, struct scsi_event * evt);
```

Arguments

sdev scsi_device event occurred on

evt event to send

Description

Assert scsi device event asynchronously.

Name

sdev_evt_alloc — allocate a new scsi event

Synopsis

```
struct scsi_event * sdev_evt_alloc (enum scsi_device_event evt_type,
gfp_t gfpflags);
```

Arguments

evt_type type of event to allocate

gfpflags GFP flags for allocation

Description

Allocates and returns a new scsi_event.

Name

sdev_evt_send_simple — send asserted event to uevent thread

Synopsis

```
void sdev_evt_send_simple (struct scsi_device * sdev, enum scsi_device_event evt_type, gfp_t gfpflags);
```

Arguments

sdev scsi_device event occurred on

evt_type type of event to send

gfpflags GFP flags for allocation

Description

Assert scsi device event asynchronously, given an event type.

Name

scsi_device_quiesce — Block user issued commands.

Synopsis

```
int scsi_device_quiesce (struct scsi_device * sdev);
```

Arguments

sdev scsi device to quiesce.

Description

This works by trying to transition to the SDEV_QUIESCE state (which must be a legal transition). When the device is in this state, only special requests will be accepted, all others will be deferred. Since special requests may also be requeued requests, a successful return doesn't guarantee the device will be totally quiescent.

Must be called with user context, may sleep.

Returns zero if unsuccessful or an error if not.

Name

scsi_device_resume — Restart user issued commands to a quiesced device.

Synopsis

```
void scsi_device_resume (struct scsi_device * sdev);
```

Arguments

sdev scsi device to resume.

Description

Moves the device from quiesced back to running and restarts the queues.

Must be called with user context, may sleep.

Name

scsi_internal_device_block — internal function to put a device temporarily into the SDEV_BLOCK state

Synopsis

```
int scsi_internal_device_block (struct scsi_device * sdev);
```

Arguments

sdev device to block

Description

Block request made by scsi lld's to temporarily stop all scsi commands on the specified device. Called from interrupt or normal process context.

Returns zero if successful or error if not

Notes

This routine transitions the device to the SDEV_BLOCK state (which must be a legal transition). When the device is in this state, all commands are deferred until the scsi lld reenables the device with scsi_device_unblock or device_block_tmo fires.

Name

scsi_internal_device_unblock — resume a device after a block request

Synopsis

```
int scsi_internal_device_unblock (struct scsi_device * sdev, enum scsi_device_state new_state);
```

Arguments

sdev device to resume

new_state state to set devices to after unblocking

Description

Called by scsi lld's or the midlayer to restart the device queue for the previously suspended scsi device. Called from interrupt or normal process context.

Returns zero if successful or error if not.

Notes

This routine transitions the device to the SDEV_RUNNING state or to one of the offline states (which must be a legal transition) allowing the midlayer to goose the queue for this device.

Name

scsi_kmap_atomic_sg — find and atomically map an sg-elemnt

Synopsis

```
void * scsi_kmap_atomic_sg (struct scatterlist * sgl, int sg_count,
size_t * offset, size_t * len);
```

Arguments

sgl	scatter-gather list
sg_count	number of segments in sg
offset	offset in bytes into sg, on return offset into the mapped area
len	bytes to map, on return number of bytes mapped

Description

Returns virtual address of the start of the mapped page

Name

scsi_kunmap_atomic_sg — atomically unmap a virtual address, previously mapped with scsi_kmap_atomic_sg

Synopsis

```
void scsi_kunmap_atomic_sg (void * virt);
```

Arguments

virt virtual address to be unmapped

Name

scsi_vpd_lun_id — return a unique device identification

Synopsis

```
int scsi_vpd_lun_id (struct scsi_device * sdev, char * id, size_t id_len);
```

Arguments

sdev SCSI device

id buffer for the identification

id_len length of the buffer

Description

Copies a unique device identification into *id* based on the information in the VPD page 0x83 of the device. The string will be formatted as a SCSI name string.

Returns the length of the identification or error on failure. If the identifier is longer than the supplied buffer the actual identifier length is returned and the buffer is not zero-padded.

drivers/scsi/scsi_lib_dma.c

SCSI library functions depending on DMA (map and unmap scatter-gather lists).

Name

scsi_dma_map — perform DMA mapping against command's sg lists

Synopsis

```
int scsi_dma_map (struct scsi_cmnd * cmd);
```

Arguments

cmd scsi command

Description

Returns the number of sg lists actually used, zero if the sg lists is NULL, or -ENOMEM if the mapping failed.

Name

scsi_dma_unmap — unmap command's sg lists mapped by scsi_dma_map

Synopsis

```
void scsi_dma_unmap (struct scsi_cmnd * cmd);
```

Arguments

cmd scsi command

drivers/scsi/scsi_module.c

The file drivers/scsi/scsi_module.c contains legacy support for old-style host templates. It should never be used by any new driver.

drivers/scsi/scsi_proc.c

The functions in this file provide an interface between the PROC file system and the SCSI device drivers It is mainly used for debugging, statistics and to pass information directly to the lowlevel driver. I.E. plumbing to manage /proc/scsi/*

Name

scsi_proc_hostdir_add — Create directory in /proc for a scsi host

Synopsis

```
void scsi_proc_hostdir_add (struct scsi_host_template * sht);
```

Arguments

sht owner of this directory

Description

Sets sht->proc_dir to the new directory.

Name

scsi_proc_hostdir_rm — remove directory in /proc for a scsi host

Synopsis

```
void scsi_proc_hostdir_rm (struct scsi_host_template * sht);
```

Arguments

sht owner of directory

Name

scsi_proc_host_add — Add entry for this host to appropriate /proc dir

Synopsis

```
void scsi_proc_host_add (struct Scsi_Host * shost);
```

Arguments

shost host to add

Name

scsi_proc_host_rm — remove this host's entry from /proc

Synopsis

```
void scsi_proc_host_rm (struct Scsi_Host * shost);
```

Arguments

shost which host

Name

proc_print_scsidevice — return data about this host

Synopsis

```
int proc_print_scsidevice (struct device * dev, void * data);
```

Arguments

dev A scsi device

data struct seq_file to output to.

Description

prints Host, Channel, Id, Lun, Vendor, Model, Rev, Type, and revision.

Name

scsi_add_single_device — Respond to user request to probe for/add device

Synopsis

```
int scsi_add_single_device (uint host, uint channel, uint id, uint lun);
```

Arguments

host user-supplied decimal integer

channel user-supplied decimal integer

id user-supplied decimal integer

lun user-supplied decimal integer

Description

called by writing “scsi add-single-device” to /proc/scsi/scsi.

does `scsi_host_lookup` and either `user_scan` if that transport type supports it, or else `scsi_scan_host_selected`

Note

this seems to be aimed exclusively at SCSI parallel busses.

Name

scsi_remove_single_device — Respond to user request to remove a device

Synopsis

```
int scsi_remove_single_device (uint host, uint channel, uint id, uint lun);
```

Arguments

host	user-supplied decimal integer
channel	user-supplied decimal integer
id	user-supplied decimal integer
lun	user-supplied decimal integer

Description

called by writing “scsi remove-single-device” to /proc/scsi/scsi. Does a `scsi_device_lookup` and `scsi_remove_device`

Name

proc_scsi_write — handle writes to /proc/scsi/scsi

Synopsis

```
ssize_t proc_scsi_write (struct file * file, const char __user * buf,
size_t length, loff_t * ppos);
```

Arguments

file not used

buf buffer to write

length length of buf, at most PAGE_SIZE

ppos not used

Description

this provides a legacy mechanism to add or remove devices by Host, Channel, ID, and Lun. To use, “echo 'scsi add-single-device 0 1 2 3' > /proc/scsi/scsi” or “echo 'scsi remove-single-device 0 1 2 3' > /proc/scsi/scsi” with “0 1 2 3” replaced by the Host, Channel, Id, and Lun.

Note

this seems to be aimed at parallel SCSI. Most modern busses (USB, SATA, Firewire, Fibre Channel, etc) dynamically assign these values to provide a unique identifier and nothing more.

Name

proc_scsi_open — glue function

Synopsis

```
int proc_scsi_open (struct inode * inode, struct file * file);
```

Arguments

inode not used

file passed to `single_open`

Description

Associates proc_scsi_show with this file

Name

scsi_init_procfs — create scsi and scsi/scsi in procfs

Synopsis

```
int scsi_init_procfs ( void);
```

Arguments

void no arguments

Name

scsi_exit_procfs — Remove scsi/scsi and scsi from procfs

Synopsis

```
void scsi_exit_procfs ( void);
```

Arguments

void no arguments

drivers/scsi/scsi_netlink.c

Infrastructure to provide async events from transports to userspace via netlink, using a single NETLINK_SCSITRANSPORT protocol for all transports. See the original patch submission [http://marc.info/?l=linux-scsi&m=115507374832500&w=2] for more details.

Name

scsi_nl_rcv_msg — Receive message handler.

Synopsis

```
void scsi_nl_rcv_msg (struct sk_buff * skb);
```

Arguments

skb socket receive buffer

Description

Extracts message from a receive buffer. Validates message header and calls appropriate transport message handler

Name

scsi_netlink_init — Called by SCSI subsystem to initialize the SCSI transport netlink interface

Synopsis

```
void scsi_netlink_init ( void);
```

Arguments

void no arguments

Name

scsi_netlink_exit — Called by SCSI subsystem to disable the SCSI transport netlink interface

Synopsis

```
void scsi_netlink_exit ( void);
```

Arguments

void no arguments

Description

drivers/scsi/scsi_scan.c

Scan a host to determine which (if any) devices are attached. The general scanning/probing algorithm is as follows, exceptions are made to it depending on device specific flags, compilation options, and global variable (boot or module load time) settings. A specific LUN is scanned via an INQUIRY command; if the LUN has a device attached, a scsi_device is allocated and setup for it. For every id of every channel on the given host, start by scanning LUN 0. Skip hosts that don't respond at all to a scan of LUN 0. Otherwise, if LUN 0 has a device attached, allocate and setup a scsi_device for it. If target is SCSI-3 or up, issue a REPORT LUN, and scan all of the LUNs returned by the REPORT LUN; else, sequentially scan LUNs up until some maximum is reached, or a LUN is seen that cannot have a device attached to it.

Name

scsi_complete_async_scans — Wait for asynchronous scans to complete

Synopsis

```
int scsi_complete_async_scans ( void);
```

Arguments

void no arguments

Description

When this function returns, any host which started scanning before this function was called will have finished its scan. Hosts which started scanning after this function was called may or may not have finished.

Name

scsi_unlock_floptical — unlock device via a special MODE SENSE command

Synopsis

```
void scsi_unlock_floptical (struct scsi_device * sdev, unsigned char
* result);
```

Arguments

sdev scsi device to send command to

result area to store the result of the MODE SENSE

Description

Send a vendor specific MODE SENSE (not a MODE SELECT) command. Called for BLIST_KEY devices.

Name

scsi_alloc_sdev — allocate and setup a scsi_Device

Synopsis

```
struct scsi_device * scsi_alloc_sdev (struct scsi_target * starget, u64
lun, void * hostdata);
```

Arguments

starget which target to allocate a scsi_device for

lun which lun

hostdata usually NULL and set by ->slave_alloc instead

Description

Allocate, initialize for io, and return a pointer to a scsi_Device. Stores the *shost*, *channel*, *id*, and *lun* in the scsi_Device, and adds scsi_Device to the appropriate list.

Return value

scsi_Device pointer, or NULL on failure.

Name

scsi_target_reap_ref_release — remove target from visibility

Synopsis

```
void scsi_target_reap_ref_release (struct kref * kref);
```

Arguments

kref the reap_ref in the target being released

Description

Called on last put of reap_ref, which is the indication that no device under this target is visible anymore, so render the target invisible in sysfs. Note: we have to be in user context here because the target reaps should be done in places where the scsi device visibility is being removed.

Name

scsi_alloc_target — allocate a new or find an existing target

Synopsis

```
struct scsi_target * scsi_alloc_target (struct device * parent, int channel, uint id);
```

Arguments

parent parent of the target (need not be a scsi host)

channel target channel number (zero if no channels)

id target id number

Description

Return an existing target if one exists, provided it hasn't already gone into STARGET_DEL state, otherwise allocate a new target.

The target is returned with an incremented reference, so the caller is responsible for both reaping and doing a last put

Name

scsi_target_reap — check to see if target is in use and destroy if not

Synopsis

```
void scsi_target_reap (struct scsi_target * starget);
```

Arguments

starget target to be checked

Description

This is used after removing a LUN or doing a last put of the target it checks atomically that nothing is using the target and removes it if so.

Name

scsi_probe_lun — probe a single LUN using a SCSI INQUIRY

Synopsis

```
int scsi_probe_lun (struct scsi_device * sdev, unsigned char *
inq_result, int result_len, int * bflags);
```

Arguments

sdev	scsi_device to probe
inq_result	area to store the INQUIRY result
result_len	len of inq_result
bflags	store any bflags found here

Description

Probe the lun associated with *req* using a standard SCSI INQUIRY;

If the INQUIRY is successful, zero is returned and the INQUIRY data is in *inq_result*; the scsi_level and INQUIRY length are copied to the scsi_device any flags value is stored in **bflags*.

Name

scsi_test_lun — waiting for a LUN to become ready

Synopsis

```
int scsi_test_lun (struct scsi_device * sdev);
```

Arguments

sdev scsi_device to test

Description

Wait for the lun associated with *sdev* to become ready

Send a TEST UNIT READY to detect any unit attention conditions. Retry TEST UNIT READY for up to *scsi_scan_timeout* if the returned sense key is 02/04/01 (Not ready, Logical Unit is in process of becoming ready)

Name

scsi_add_lun — allocate and fully initialze a scsi_device

Synopsis

```
int scsi_add_lun (struct scsi_device * sdev, unsigned char * inq_result,
int * bflags, int async);
```

Arguments

sdev	holds information to be stored in the new scsi_device
inq_result	holds the result of a previous INQUIRY to the LUN
bflags	black/white list flag
async	1 if this device is being scanned asynchronously

Description

Initialize the scsi_device *sdev*. Optionally set fields based on values in **bflags*.

SCSI_SCAN_NO_RESPONSE

could not allocate or setup a scsi_device

SCSI_SCAN_LUN_PRESENT

a new scsi_device was allocated and initialized

Name

scsi_inq_str — print INQUIRY data from min to max index, strip trailing whitespace

Synopsis

```
unsigned char * scsi_inq_str (unsigned char * buf, unsigned char * inq,
unsigned first, unsigned end);
```

Arguments

buf Output buffer with at least end-first+1 bytes of space

inq Inquiry buffer (input)

first Offset of string into inq

end Index after last character in inq

Name

scsi_probe_and_add_lun — probe a LUN, if a LUN is found add it

Synopsis

```
int scsi_probe_and_add_lun (struct scsi_target * starget, u64 lun, int
* bflagsp, struct scsi_device ** sdevp, enum scsi_scan_mode rescan,
void * hostdata);
```

Arguments

starget	pointer to target device structure
lun	LUN of target device
bflagsp	store bflags here if not NULL
sdevp	probe the LUN corresponding to this scsi_device
rescan	if not equal to SCSI_SCAN_INITIAL skip some code only needed on first scan
hostdata	passed to `scsi_alloc_sdev`

Description

Call scsi_probe_lun, if a LUN with an attached device is found, allocate and set it up by calling scsi_add_lun.

SCSI_SCAN_NO_RESPONSE

could not allocate or setup a scsi_device

SCSI_SCAN_TARGET_PRESENT

target responded, but no device is attached at the LUN

SCSI_SCAN_LUN_PRESENT

a new scsi_device was allocated and initialized

Name

scsi_sequential_lun_scan — sequentially scan a SCSI target

Synopsis

```
void scsi_sequential_lun_scan (struct scsi_target * starget, int bflags,
int scsi_level, enum scsi_scan_mode rescan);
```

Arguments

starget	pointer to target structure to scan
bflags	black/white list flag for LUN 0
scsi_level	Which version of the standard does this device adhere to
rescan	passed to `scsi_probe_add_lun`

Description

Generally, scan from LUN 1 (LUN 0 is assumed to already have been scanned) to some maximum lun until a LUN is found with no device attached. Use the bflags to figure out any oddities.

Modifies sdevscan->lun.

Name

scsi_report_lun_scan — Scan using SCSI REPORT LUN results

Synopsis

```
int scsi_report_lun_scan (struct scsi_target * starget, int bflags, enum
scsi_scan_mode rescan);
```

Arguments

starget which target

bflags Zero or a mix of BLIST_NOLUN, BLIST_REPORTLUN2, or BLIST_NOREPORTLUN

rescan nonzero if we can skip code only needed on first scan

Description

Fast scanning for modern (SCSI-3) devices by sending a REPORT LUN command. Scan the resulting list of LUNs by calling scsi_probe_and_add_lun.

If BLINK_REPORTLUN2 is set, scan a target that supports more than 8 LUNs even if it's older than SCSI-3. If BLIST_NOREPORTLUN is set, return 1 always. If BLIST_NOLUN is set, return 0 always. If starget->no_report_luns is set, return 1 always.

0

scan completed (or no memory, so further scanning is futile)

1

could not scan with REPORT LUN

Name

scsi_prep_async_scan — prepare for an async scan

Synopsis

```
struct async_scan_data * scsi_prep_async_scan (struct Scsi_Host *
shost);
```

Arguments

shost the host which will be scanned

Returns

a cookie to be passed to `scsi_finish_async_scan`

Tells the midlayer this host is going to do an asynchronous scan. It reserves the host's position in the scanning list and ensures that other asynchronous scans started after this one won't affect the ordering of the discovered devices.

Name

scsi_finish_async_scan — asynchronous scan has finished

Synopsis

```
void scsi_finish_async_scan (struct async_scan_data * data);
```

Arguments

data cookie returned from earlier call to `scsi_prep_async_scan`

Description

All the devices currently attached to this host have been found. This function announces all the devices it has found to the rest of the system.

drivers/scsi/scsi_sysctl.c

Set up the sysctl entry: "/dev/scsi/logging_level" (DEV_SCSI_LOGGING_LEVEL) which sets/returns scsi_logging_level.

drivers/scsi/scsi_sysfs.c

SCSI sysfs interface routines.

Name

scsi_remove_device — unregister a device from the scsi bus

Synopsis

```
void scsi_remove_device (struct scsi_device * sdev);
```

Arguments

sdev scsi_device to unregister

Name

scsi_remove_target — try to remove a target and all its devices

Synopsis

```
void scsi_remove_target (struct device * dev);
```

Arguments

dev generic starget or parent of generic stargets to be removed

Note

This is slightly racy. It is possible that if the user requests the addition of another device then the target won't be removed.

drivers/scsi/hosts.c

mid to lowlevel SCSI driver interface

Name

scsi_host_set_state — Take the given host through the host state model.

Synopsis

```
int scsi_host_set_state (struct Scsi_Host * shost, enum scsi_host_state
state);
```

Arguments

shost scsi host to change the state of.

state state to change to.

Description

Returns zero if unsuccessful or an error if the requested transition is illegal.

Name

scsi_remove_host — remove a scsi host

Synopsis

```
void scsi_remove_host (struct Scsi_Host * shost);
```

Arguments

shost a pointer to a scsi host to remove

Name

scsi_add_host_with_dma — add a scsi host with dma device

Synopsis

```
int scsi_add_host_with_dma (struct Scsi_Host * shost, struct device *
dev, struct device * dma_dev);
```

Arguments

shost scsi host pointer to add

dev a struct device of type scsi class

dma_dev dma device for the host

Note

You rarely need to worry about this unless you're in a virtualised host environments, so use the simpler `scsi_add_host` function instead.

Return value

0 on success / != 0 for error

Name

scsi_host_alloc — register a scsi host adapter instance.

Synopsis

```
struct Scsi_Host * scsi_host_alloc (struct scsi_host_template * sht,
int privsize);
```

Arguments

sht pointer to scsi host template

privsize extra bytes to allocate for driver

Note

Allocate a new Scsi_Host and perform basic initialization. The host is not published to the scsi midlayer until scsi_add_host is called.

Return value

Pointer to a new Scsi_Host

Name

scsi_host_lookup — get a reference to a Scsi_Host by host no

Synopsis

```
struct Scsi_Host * scsi_host_lookup (unsigned short hostnum);
```

Arguments

hostnum host number to locate

Return value

A pointer to located Scsi_Host or NULL.

The caller must do a `scsi_host_put` to drop the reference that `scsi_host_get` took. The `put_device` below dropped the reference from `class_find_device`.

Name

scsi_host_get — inc a Scsi_Host ref count

Synopsis

```
struct Scsi_Host * scsi_host_get (struct Scsi_Host * shost);
```

Arguments

shost Pointer to Scsi_Host to inc.

Name

scsi_host_put — dec a Scsi_Host ref count

Synopsis

```
void scsi_host_put (struct Scsi_Host * shost);
```

Arguments

shost Pointer to Scsi_Host to dec.

Name

scsi_queue_work — Queue work to the Scsi_Host workqueue.

Synopsis

```
int scsi_queue_work (struct Scsi_Host * shost, struct work_struct *
work);
```

Arguments

shost Pointer to Scsi_Host.

work Work to queue for execution.

Return value

1 - work queued for execution 0 - work is already queued -EINVAL - work queue doesn't exist

Name

scsi_flush_work — Flush a Scsi_Host's workqueue.

Synopsis

```
void scsi_flush_work (struct Scsi_Host * shost);
```

Arguments

shost Pointer to Scsi_Host.

drivers/scsi/constants.c

mid to lowlevel SCSI driver interface

Name

drivers/scsi/constants.c — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `drivers/scsi/constants.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Transport classes

Transport classes are service libraries for drivers in the SCSI lower layer, which expose transport attributes in sysfs.

Fibre Channel transport

The file drivers/scsi/scsi_transport_fc.c defines transport attributes for Fibre Channel.

Name

fc_get_event_number — Obtain the next sequential FC event number

Synopsis

```
u32 fc_get_event_number ( void);
```

Arguments

void no arguments

Notes

We could have inlined this, but it would have required fc_event_seq to be exposed. For now, live with the subroutine call. Atomic used to avoid lock/unlock...

Name

fc_host_post_event — called to post an even on an fc_host.

Synopsis

```
void fc_host_post_event (struct Scsi_Host * shost, u32 event_number,
enum fc_host_event_code event_code, u32 event_data);
```

Arguments

shost	host the event occurred on
event_number	fc event number obtained from `get_fc_event_number`
event_code	fc_host event being posted
event_data	32bits of data for the event being posted

Notes

This routine assumes no locks are held on entry.

Name

fc_host_post_vendor_event — called to post a vendor unique event on an fc_host

Synopsis

```
void fc_host_post_vendor_event (struct Scsi_Host * shost, u32 event_number, u32 data_len, char * data_buf, u64 vendor_id);
```

Arguments

shost	host the event occurred on
event_number	fc event number obtained from `get_fc_event_number`
data_len	amount, in bytes, of vendor unique data
data_buf	pointer to vendor unique data
vendor_id	Vendor id

Notes

This routine assumes no locks are held on entry.

Name

fc_remove_host — called to terminate any fc_transport-related elements for a scsi host.

Synopsis

```
void fc_remove_host (struct Scsi_Host * shost);
```

Arguments

shost Which Scsi_Host

Description

This routine is expected to be called immediately preceding the a driver's call to `scsi_remove_host`.

WARNING

A driver utilizing the fc_transport, which fails to call this routine prior to `scsi_remove_host`, will leave dangling objects in /sys/class/fc_remote_ports. Access to any of these objects can result in a system crash !!!

Notes

This routine assumes no locks are held on entry.

Name

fc_remote_port_add — notify fc transport of the existence of a remote FC port.

Synopsis

```
struct fc_rport * fc_remote_port_add (struct Scsi_Host * shost, int channel, struct fc_rport_identifiers * ids);
```

Arguments

shost scsi host the remote port is connected to.

channel Channel on shost port connected to.

ids The world wide names, fc address, and FC4 port roles for the remote port.

Description

The LLDD calls this routine to notify the transport of the existence of a remote port. The LLDD provides the unique identifiers (wwpn,wwn) of the port, it's FC address (port_id), and the FC4 roles that are active for the port.

For ports that are FCP targets (aka scsi targets), the FC transport maintains consistent target id bindings on behalf of the LLDD. A consistent target id binding is an assignment of a target id to a remote port identifier, which persists while the scsi host is attached. The remote port can disappear, then later reappear, and it's target id assignment remains the same. This allows for shifts in FC addressing (if binding by wwpn or wwnn) with no apparent changes to the scsi subsystem which is based on scsi host number and target id values. Bindings are only valid during the attachment of the scsi host. If the host detaches, then later re-attaches, target id bindings may change.

This routine is responsible for returning a remote port structure. The routine will search the list of remote ports it maintains internally on behalf of consistent target id mappings. If found, the remote port structure will be reused. Otherwise, a new remote port structure will be allocated.

Whenever a remote port is allocated, a new fc_remote_port class device is created.

Should not be called from interrupt context.

Notes

This routine assumes no locks are held on entry.

Name

fc_remote_port_delete — notifies the fc transport that a remote port is no longer in existence.

Synopsis

```
void fc_remote_port_delete (struct fc_rport * rport);
```

Arguments

rport The remote port that no longer exists

Description

The LLDD calls this routine to notify the transport that a remote port is no longer part of the topology. Note: Although a port may no longer be part of the topology, it may persist in the remote ports displayed by the fc_host. We do this under 2 conditions: 1) If the port was a scsi target, we delay its deletion by “blocking” it. This allows the port to temporarily disappear, then reappear without disrupting the SCSI device tree attached to it. During the “blocked” period the port will still exist. 2) If the port was a scsi target and disappears for longer than we expect, we'll delete the port and the tear down the SCSI device tree attached to it. However, we want to semi-persist the target id assigned to that port if it eventually does exist. The port structure will remain (although with minimal information) so that the target id bindings remails.

If the remote port is not an FCP Target, it will be fully torn down and deallocated, including the fc_remote_port class device.

If the remote port is an FCP Target, the port will be placed in a temporary blocked state. From the LLDD's perspective, the rport no longer exists. From the SCSI midlayer's perspective, the SCSI target exists, but all sdevs on it are blocked from further I/O. The following is then expected.

If the remote port does not return (signaled by a LLDD call to `fc_remote_port_add`) within the dev_loss_tmo timeout, then the scsi target is removed - killing all outstanding i/o and removing the scsi devices attached ot it. The port structure will be marked Not Present and be partially cleared, leaving only enough information to recognize the remote port relative to the scsi target id binding if it later appears. The port will remain as long as there is a valid binding (e.g. until the user changes the binding type or unloads the scsi host with the binding).

If the remote port returns within the dev_loss_tmo value (and matches according to the target id binding type), the port structure will be reused. If it is no longer a SCSI target, the target will be torn down. If it continues to be a SCSI target, then the target will be unblocked (allowing i/o to be resumed), and a scan will be activated to ensure that all luns are detected.

Called from normal process context only - cannot be called from interrupt.

Notes

This routine assumes no locks are held on entry.

Name

fc_remote_port_rolechg — notifies the fc transport that the roles on a remote may have changed.

Synopsis

```
void fc_remote_port_rolechg (struct fc_rport * rport, u32 roles);
```

Arguments

rport The remote port that changed.

roles New roles for this port.

Description

The LLDD calls this routine to notify the transport that the roles on a remote port may have changed. The largest effect of this is if a port now becomes a FCP Target, it must be allocated a scsi target id. If the port is no longer a FCP target, any scsi target id value assigned to it will persist in case the role changes back to include FCP Target. No changes in the scsi midlayer will be invoked if the role changes (in the expectation that the role will be resumed. If it doesn't normal error processing will take place).

Should not be called from interrupt context.

Notes

This routine assumes no locks are held on entry.

Name

fc_block_scsi_eh — Block SCSI eh thread for blocked fc_rport

Synopsis

```
int fc_block_scsi_eh (struct scsi_cmnd * cmnd);
```

Arguments

cmnd SCSI command that scsi_eh is trying to recover

Description

This routine can be called from a FC LLD scsi_eh callback. It blocks the scsi_eh thread until the fc_rport leaves the FC_PORTSTATE_BLOCKED, or the fast_io_fail_tmo fires. This is necessary to avoid the scsi_eh failing recovery actions for blocked rports which would lead to offlined SCSI devices.

Returns

0 if the fc_rport left the state FC_PORTSTATE_BLOCKED. FAST_IO_FAIL if the fast_io_fail_tmo fired, this should be passed back to scsi_eh.

Name

fc_vport_create — Admin App or LLDD requests creation of a vport

Synopsis

```
struct fc_vport * fc_vport_create (struct Scsi_Host * shost, int channel,
struct fc_vport_identifiers * ids);
```

Arguments

shost scsi host the virtual port is connected to.

channel channel on shost port connected to.

ids The world wide names, FC4 port roles, etc for the virtual port.

Notes

This routine assumes no locks are held on entry.

Name

fc_vport_terminate — Admin App or LLDD requests termination of a vport

Synopsis

```
int fc_vport_terminate (struct fc_vport * vport);
```

Arguments

vport fc_vport to be terminated

Description

Calls the LLDD `vport_delete` function, then deallocates and removes the vport from the shost and object tree.

Notes

This routine assumes no locks are held on entry.

iSCSI transport class

The file drivers/scsi/scsi_transport_iscsi.c defines transport attributes for the iSCSI class, which sends SCSI packets over TCP/IP connections.

Name

iscsi_create_flashnode_sess — Add flashnode session entry in sysfs

Synopsis

```
struct iscsi_bus_flash_session * iscsi_create_flashnode_sess (struct
Scsi_Host * shost, int index, struct iscsi_transport * transport, int
dd_size);
```

Arguments

shost	pointer to host data
index	index of flashnode to add in sysfs
transport	pointer to transport data
dd_size	total size to allocate

Description

Adds a sysfs entry for the flashnode session attributes

Returns

pointer to allocated flashnode sess on success NULL on failure

Name

iscsi_create_flashnode_conn — Add flashnode conn entry in sysfs

Synopsis

```
struct iscsi_bus_flash_conn * iscsi_create_flashnode_conn (struct
Scsi_Host * shost, struct iscsi_bus_flash_session * fnode_sess, struct
iscsi_transport * transport, int dd_size);
```

Arguments

shost	pointer to host data
fnode_sess	pointer to the parent flashnode session entry
transport	pointer to transport data
dd_size	total size to allocate

Description

Adds a sysfs entry for the flashnode connection attributes

Returns

pointer to allocated flashnode conn on success NULL on failure

Name

iscsi_is_flashnode_conn_dev — verify passed device is to be flashnode conn

Synopsis

```
int iscsi_is_flashnode_conn_dev (struct device * dev, void * data);
```

Arguments

dev device to verify

data pointer to data containing value to use for verification

Description

Verifies if the passed device is flashnode conn device

Returns

1 on success 0 on failure

Name

iscsi_find_flashnode_sess — finds flashnode session entry

Synopsis

```
struct device * iscsi_find_flashnode_sess (struct Scsi_Host * shost,
void * data, int (*fn) (struct device *dev, void *data));
```

Arguments

shost pointer to host data

data pointer to data containing value to use for comparison

fn function pointer that does actual comparison

Description

Finds the flashnode session object comparing the data passed using logic defined in passed function pointer

Returns

pointer to found flashnode session device object on success NULL on failure

Name

iscsi_find_flashnode_conn — finds flashnode connection entry

Synopsis

```
struct device * iscsi_find_flashnode_conn (struct
iscsi_bus_flash_session * fnode_sess);
```

Arguments

fnode_sess pointer to parent flashnode session entry

Description

Finds the flashnode connection object comparing the data passed using logic defined in passed function pointer

Returns

pointer to found flashnode connection device object on success NULL on failure

Name

iscsi_destroy_flashnode_sess — destroy flashnode session entry

Synopsis

```
void iscsi_destroy_flashnode_sess (struct iscsi_bus_flash_session *
fnode_sess);
```

Arguments

fnode_sess pointer to flashnode session entry to be destroyed

Description

Deletes the flashnode session entry and all children flashnode connection entries from sysfs

Name

iscsi_destroy_all_flashnode — destroy all flashnode session entries

Synopsis

```
void iscsi_destroy_all_flashnode (struct Scsi_Host * shost);
```

Arguments

shost pointer to host data

Description

Destroys all the flashnode session entries and all corresponding children flashnode connection entries from sysfs

Name

iscsi_scan_finished — helper to report when running scans are done

Synopsis

```
int iscsi_scan_finished (struct Scsi_Host * shost, unsigned long time);
```

Arguments

shost scsi host

time scan run time

Description

This function can be used by drives like qla4xxx to report to the scsi layer when the scans it kicked off at module load time are done.

Name

iscsi_block_scsi_eh — block scsi eh until session state has transistioned

Synopsis

```
int iscsi_block_scsi_eh (struct scsi_cmnd * cmd);
```

Arguments

cmd scsi cmd passed to scsi eh handler

Description

If the session is down this function will wait for the recovery timer to fire or for the session to be logged back in. If the recovery timer fires then FAST_IO_FAIL is returned. The caller should pass this error value to the scsi eh.

Name

iscsi_unblock_session — set a session as logged in and start IO.

Synopsis

```
void iscsi_unblock_session (struct iscsi_cls_session * session);
```

Arguments

session iscsi session

Description

Mark a session as ready to accept IO.

Name

iscsi_create_session — create iscsi class session

Synopsis

```
struct iscsi_cls_session * iscsi_create_session (struct Scsi_Host *
shost, struct iscsi_transport * transport, int dd_size, unsigned int
target_id);
```

Arguments

shost	scsi host
transport	iscsi transport
dd_size	private driver data size
target_id	which target

Description

This can be called from a LLD or iscsi_transport.

Name

iscsi_destroy_session — destroy iscsi session

Synopsis

```
int iscsi_destroy_session (struct iscsi_cls_session * session);
```

Arguments

session iscsi_session

Description

Can be called by a LLD or iscsi_transport. There must not be any running connections.

Name

iscsi_create_conn — create iscsi class connection

Synopsis

```
struct iscsi_cls_conn * iscsi_create_conn (struct iscsi_cls_session *
session, int dd_size, uint32_t cid);
```

Arguments

session iscsi cls session

dd_size private driver data size

cid connection id

Description

This can be called from a LLD or iscsi_transport. The connection is child of the session so cid must be unique for all connections on the session.

Since we do not support MCS, cid will normally be zero. In some cases for software iscsi we could be trying to preallocate a connection struct in which case there could be two connection structs and cid would be non-zero.

Name

iscsi_destroy_conn — destroy iscsi class connection

Synopsis

```
int iscsi_destroy_conn (struct iscsi_cls_conn * conn);
```

Arguments

conn iscsi cls session

Description

This can be called from a LLD or iscsi_transport.

Name

iscsi_session_event — send session destr. completion event

Synopsis

```
int iscsi_session_event (struct iscsi_cls_session * session, enum iscsi_uevent_e event);
```

Arguments

session iscsi class session

event type of event

Serial Attached SCSI (SAS) transport class

The file drivers/scsi/scsi_transport_sas.c defines transport attributes for Serial Attached SCSI, a variant of SATA aimed at large high-end systems.

The SAS transport class contains common code to deal with SAS HBAs, an aproximated representation of SAS topologies in the driver model, and various sysfs attributes to expose these topologies and management interfaces to userspace.

In addition to the basic SCSI core objects this transport class introduces two additional intermediate objects: The SAS PHY as represented by struct sas_phy defines an "outgoing" PHY on a SAS HBA or Expander, and the SAS remote PHY represented by struct sas_rphy defines an "incoming" PHY on a SAS Expander or end device. Note that this is purely a software concept, the underlying hardware for a PHY and a remote PHY is the exactly the same.

There is no concept of a SAS port in this code, users can see what PHYs form a wide port based on the port_identifier attribute, which is the same for all PHYs in a port.

Name

is_sas_attached — check if device is SAS attached

Synopsis

```
int is_sas_attached (struct scsi_device * sdev);
```

Arguments

sdev scsi device to check

Description

returns true if the device is SAS attached

Name

sas_remove_children — tear down a devices SAS data structures

Synopsis

```
void sas_remove_children (struct device * dev);
```

Arguments

dev device belonging to the sas object

Description

Removes all SAS PHYs and remote PHYs for a given object

Name

sas_remove_host — tear down a Scsi_Host's SAS data structures

Synopsis

```
void sas_remove_host (struct Scsi_Host * shost);
```

Arguments

shost Scsi Host that is torn down

Description

Removes all SAS PHYs and remote PHYs for a given Scsi_Host. Must be called just before scsi_remove_host for SAS HBAs.

Name

sas_get_address — return the SAS address of the device

Synopsis

```
u64 sas_get_address (struct scsi_device * sdev);
```

Arguments

sdev scsi device

Description

Returns the SAS address of the scsi device

Name

sas_tlr_supported — checking TLR bit in vpd 0x90

Synopsis

```
unsigned int sas_tlr_supported (struct scsi_device * sdev);
```

Arguments

sdev scsi device struct

Description

Check Transport Layer Retries are supported or not. If vpd page 0x90 is present, TRL is supported.

Name

sas_disable_tlr — setting TLR flags

Synopsis

```
void sas_disable_tlr (struct scsi_device * sdev);
```

Arguments

sdev scsi device struct

Description

Seting tlr_enabled flag to 0.

Name

sas_enable_tlr — setting TLR flags

Synopsis

```
void sas_enable_tlr (struct scsi_device * sdev);
```

Arguments

sdev scsi device struct

Description

Seting tlr_enabled flag 1.

Name

sas_phy_alloc — allocates and initialize a SAS PHY structure

Synopsis

```
struct sas_phy * sas_phy_alloc (struct device * parent, int number);
```

Arguments

parent Parent device

number Phy index

Description

Allocates an SAS PHY structure. It will be added in the device tree below the device specified by *parent*, which has to be either a Scsi_Host or sas_rphy.

Returns

SAS PHY allocated or NULL if the allocation failed.

Name

sas_phy_add — add a SAS PHY to the device hierarchy

Synopsis

```
int sas_phy_add (struct sas_phy * phy);
```

Arguments

phy The PHY to be added

Description

Publishes a SAS PHY to the rest of the system.

Name

sas_phy_free — free a SAS PHY

Synopsis

```
void sas_phy_free (struct sas_phy * phy);
```

Arguments

phy SAS PHY to free

Description

Frees the specified SAS PHY.

Note

This function must only be called on a PHY that has not successfully been added using `sas_phy_add`.

Name

sas_phy_delete — remove SAS PHY

Synopsis

```
void sas_phy_delete (struct sas_phy * phy);
```

Arguments

phy SAS PHY to remove

Description

Removes the specified SAS PHY. If the SAS PHY has an associated remote PHY it is removed before.

Name

scsi_is_sas_phy — check if a struct device represents a SAS PHY

Synopsis

```
int scsi_is_sas_phy (const struct device * dev);
```

Arguments

dev device to check

Returns

`1` if the device represents a SAS PHY, `0` else

Name

sas_port_add — add a SAS port to the device hierarchy

Synopsis

```
int sas_port_add (struct sas_port * port);
```

Arguments

port port to be added

Description

publishes a port to the rest of the system

Name

sas_port_free — free a SAS PORT

Synopsis

```
void sas_port_free (struct sas_port * port);
```

Arguments

port SAS PORT to free

Description

Frees the specified SAS PORT.

Note

This function must only be called on a PORT that has not successfully been added using `sas_port_add`.

Name

sas_port_delete — remove SAS PORT

Synopsis

```
void sas_port_delete (struct sas_port * port);
```

Arguments

port SAS PORT to remove

Description

Removes the specified SAS PORT. If the SAS PORT has an associated phys, unlink them from the port as well.

Name

scsi_is_sas_port — check if a struct device represents a SAS port

Synopsis

```
int scsi_is_sas_port (const struct device * dev);
```

Arguments

dev device to check

Returns

`1` if the device represents a SAS Port, `0` else

Name

sas_port_get_phy — try to take a reference on a port member

Synopsis

```
struct sas_phy * sas_port_get_phy (struct sas_port * port);
```

Arguments

port port to check

Name

sas_port_add_phy — add another phy to a port to form a wide port

Synopsis

```
void sas_port_add_phy (struct sas_port * port, struct sas_phy * phy);
```

Arguments

port port to add the phy to

phy phy to add

Description

When a port is initially created, it is empty (has no phys). All ports must have at least one phy to operated, and all wide ports must have at least two. The current code makes no difference between ports and wide ports, but the only object that can be connected to a remote device is a port, so ports must be formed on all devices with phys if they're connected to anything.

Name

sas_port_delete_phy — remove a phy from a port or wide port

Synopsis

```
void sas_port_delete_phy (struct sas_port * port, struct sas_phy * phy);
```

Arguments

port port to remove the phy from

phy phy to remove

Description

This operation is used for tearing down ports again. It must be done to every port or wide port before calling sas_port_delete.

Name

sas_end_device_alloc — allocate an rphy for an end device

Synopsis

```
struct sas_rphy * sas_end_device_alloc (struct sas_port * parent);
```

Arguments

parent which port

Description

Allocates an SAS remote PHY structure, connected to *parent*.

Returns

SAS PHY allocated or NULL if the allocation failed.

Name

sas_expander_alloc — allocate an rphy for an end device

Synopsis

```
struct sas_rphy * sas_expander_alloc (struct sas_port * parent, enum
sas_device_type type);
```

Arguments

parent which port

type SAS_EDGE_EXPANDER_DEVICE or SAS_FANOUT_EXPANDER_DEVICE

Description

Allocates an SAS remote PHY structure, connected to *parent*.

Returns

SAS PHY allocated or NULL if the allocation failed.

Name

sas_rphy_add — add a SAS remote PHY to the device hierarchy

Synopsis

```
int sas_rphy_add (struct sas_rphy * rphy);
```

Arguments

rphy The remote PHY to be added

Description

Publishes a SAS remote PHY to the rest of the system.

Name

sas_rphy_free — free a SAS remote PHY

Synopsis

```
void sas_rphy_free (struct sas_rphy * rphy);
```

Arguments

rphy SAS remote PHY to free

Description

Frees the specified SAS remote PHY.

Note

This function must only be called on a remote PHY that has not successfully been added using `sas_rphy_add` (or has been `sas_rphy_remove`'d)

Name

sas_rphy_delete — remove and free SAS remote PHY

Synopsis

```
void sas_rphy_delete (struct sas_rphy * rphy);
```

Arguments

rphy SAS remote PHY to remove and free

Description

Removes the specified SAS remote PHY and frees it.

Name

sas_rphy_unlink — unlink SAS remote PHY

Synopsis

```
void sas_rphy_unlink (struct sas_rphy * rphy);
```

Arguments

rphy SAS remote phy to unlink from its parent port

Description

Removes port reference to an rphy

Name

sas_rphy_remove — remove SAS remote PHY

Synopsis

```
void sas_rphy_remove (struct sas_rphy * rphy);
```

Arguments

rphy SAS remote phy to remove

Description

Removes the specified SAS remote PHY.

Name

scsi_is_sas_rphy — check if a struct device represents a SAS remote PHY

Synopsis

```
int scsi_is_sas_rphy (const struct device * dev);
```

Arguments

dev device to check

Returns

`1` if the device represents a SAS remote PHY, `0` else

Name

sas_attach_transport — instantiate SAS transport template

Synopsis

```
struct scsi_transport_template * sas_attach_transport (struct
sas_function_template * ft);
```

Arguments

ft SAS transport class function template

Name

sas_release_transport — release SAS transport template instance

Synopsis

```
void sas_release_transport (struct scsi_transport_template * t);
```

Arguments

t transport template instance

SATA transport class

The SATA transport is handled by libata, which has its own book of documentation in this directory.

Parallel SCSI (SPI) transport class

The file drivers/scsi/scsi_transport_spi.c defines transport attributes for traditional (fast/wide/ultra) SCSI busses.

Name

spi_schedule_dv_device — schedule domain validation to occur on the device

Synopsis

```
void spi_schedule_dv_device (struct scsi_device * sdev);
```

Arguments

sdev The device to validate

Description

Identical to `spi_dv_device` above, except that the DV will be scheduled to occur in a workqueue later. All memory allocations are atomic, so may be called from any context including those holding SCSI locks.

Name

spi_display_xfer_agreement — Print the current target transfer agreement

Synopsis

```
void spi_display_xfer_agreement (struct scsi_target * starget);
```

Arguments

starget The target for which to display the agreement

Description

Each SPI port is required to maintain a transfer agreement for each other port on the bus. This function prints a one-line summary of the current agreement; more detailed information is available in sysfs.

Name

spi_populate_tag_msg — place a tag message in a buffer

Synopsis

```
int spi_populate_tag_msg (unsigned char * msg, struct scsi_cmnd * cmd);
```

Arguments

msg pointer to the area to place the tag

cmd pointer to the scsi command for the tag

Notes

designed to create the correct type of tag message for the particular request. Returns the size of the tag message. May return 0 if TCQ is disabled for this device.

SCSI RDMA (SRP) transport class

The file drivers/scsi/scsi_transport_srp.c defines transport attributes for SCSI over Remote Direct Memory Access.

Name

srp_tmo_valid — check timeout combination validity

Synopsis

```
int srp_tmo_valid (int reconnect_delay, int fast_io_fail_tmo, int dev_loss_tmo);
```

Arguments

reconnect_delay	Reconnect delay in seconds.
fast_io_fail_tmo	Fast I/O fail timeout in seconds.
dev_loss_tmo	Device loss timeout in seconds.

Description

The combination of the timeout parameters must be such that SCSI commands are finished in a reasonable time. Hence do not allow the fast I/O fail timeout to exceed SCSI_DEVICE_BLOCK_MAX_TIMEOUT nor allow dev_loss_tmo to exceed that limit if failing I/O fast has been disabled. Furthermore, these parameters must be such that multipath can detect failed paths timely. Hence do not allow all three parameters to be disabled simultaneously.

Name

srp_start_tl_fail_timers — start the transport layer failure timers

Synopsis

```
void srp_start_tl_fail_timers (struct srp_rport * rport);
```

Arguments

rport SRP target port.

Description

Start the transport layer fast I/O failure and device loss timers. Do not modify a timer that was already started.

Name

srp_reconnect_rport — reconnect to an SRP target port

Synopsis

```
int srp_reconnect_rport (struct srp_rport * rport);
```

Arguments

rport SRP target port.

Description

Blocks SCSI command queueing before invoking `reconnect` such that `queuecommand` won't be invoked concurrently with `reconnect` from outside the SCSI EH. This is important since a `reconnect` implementation may reallocate resources needed by `queuecommand`.

Notes

- This function neither waits until outstanding requests have finished nor tries to abort these. It is the responsibility of the `reconnect` function to finish outstanding commands before reconnecting to the target port. - It is the responsibility of the caller to ensure that the resources reallocated by the `reconnect` function won't be used while this function is in progress. One possible strategy is to invoke this function from the context of the SCSI EH thread only. Another possible strategy is to lock the rport mutex inside each SCSI LLD callback that can be invoked by the SCSI EH (the scsi_host_template.eh_*() functions and also the scsi_host_template.`queuecommand` function).

Name

srp_rport_get — increment rport reference count

Synopsis

```
void srp_rport_get (struct srp_rport * rport);
```

Arguments

rport SRP target port.

Name

srp_rport_put — decrement rport reference count

Synopsis

```
void srp_rport_put (struct srp_rport * rport);
```

Arguments

rport SRP target port.

Name

srp_rport_add — add a SRP remote port to the device hierarchy

Synopsis

```
struct srp_rport * srp_rport_add (struct Scsi_Host * shost, struct
srp_rport_identifiers * ids);
```

Arguments

shost scsi host the remote port is connected to.

ids The port id for the remote port.

Description

Publishes a port to the rest of the system.

Name

srp_rport_del — remove a SRP remote port

Synopsis

```
void srp_rport_del (struct srp_rport * rport);
```

Arguments

rport SRP remote port to remove

Description

Removes the specified SRP remote port.

Name

srp_remove_host — tear down a Scsi_Host's SRP data structures

Synopsis

```
void srp_remove_host (struct Scsi_Host * shost);
```

Arguments

shost Scsi Host that is torn down

Description

Removes all SRP remote ports for a given Scsi_Host. Must be called just before scsi_remove_host for SRP HBAs.

Name

srp_stop_rport_timers — stop the transport layer recovery timers

Synopsis

```
void srp_stop_rport_timers (struct srp_rport * rport);
```

Arguments

rport SRP remote port for which to stop the timers.

Description

Must be called after `srp_remove_host` and `scsi_remove_host`. The caller must hold a reference on the rport (rport->dev) and on the SCSI host (rport->dev.parent).

Name

srp_attach_transport — instantiate SRP transport template

Synopsis

```
struct scsi_transport_template * srp_attach_transport (struct
srp_function_template * ft);
```

Arguments

ft SRP transport class function template

Name

srp_release_transport — release SRP transport template instance

Synopsis

```
void srp_release_transport (struct scsi_transport_template * t);
```

Arguments

t transport template instance

Chapter 4. SCSI lower layer

Host Bus Adapter transport types

Many modern device controllers use the SCSI command set as a protocol to communicate with their devices through many different types of physical connections.

In SCSI language a bus capable of carrying SCSI commands is called a "transport", and a controller connecting to such a bus is called a "host bus adapter" (HBA).

Debug transport

The file drivers/scsi/scsi_debug.c simulates a host adapter with a variable number of disks (or disk like devices) attached, sharing a common amount of RAM. Does a lot of checking to make sure that we are not getting blocks mixed up, and panics the kernel if anything out of the ordinary is seen.

To be more realistic, the simulated devices have the transport attributes of SAS disks.

For documentation see http://sg.danny.cz/sg/sdebug26.html

todo

Parallel (fast/wide/ultra) SCSI, USB, SATA, SAS, Fibre Channel, FireWire, ATAPI devices, Infiniband, I20, iSCSI, Parallel ports, netlink...