

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.

Chapter 3. SCSI mid layer

SCSI midlayer implementation

`include/scsi/scsi_device.h`

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

<i>shost</i>	host the event occurred on
<i>event_number</i>	fc event number obtained from <code>get_fc_event_number</code>
<i>event_code</i>	fc_host event being posted
<i>event_data</i>	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

<i>shost</i>	host the event occurred on
<i>event_number</i>	fc event number obtained from <code>get_fc_event_number</code>
<i>data_len</i>	amount, in bytes, of vendor unique data
<i>data_buf</i>	pointer to vendor unique data
<i>vendor_id</i>	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, its 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 its 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 remains.

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 to 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 * cmd);
```

Arguments

cmd 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

<i>shost</i>	pointer to host data
<i>index</i>	index of flashnode to add in sysfs
<i>transport</i>	pointer to transport data
<i>dd_size</i>	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

<i>shost</i>	pointer to host data
<i>fnode_sess</i>	pointer to the parent flashnode session entry
<i>transport</i>	pointer to transport data
<i>dd_size</i>	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 * fn-  
ode_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

<i>shost</i>	scsi host
<i>transport</i>	iscsi transport
<i>dd_size</i>	private driver data size
<i>target_id</i>	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 approximated 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

Setting `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

Setting `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...