SCSI mid_level - lower_level driver interface

Introduction

This document outlines the interface between the Linux SCSI mid level and SCSI lower level drivers. Lower level drivers (LLDs) are variously called host bus adapter (HBA) drivers and host drivers (HD). A "host" in this context is a bridge between a computer IO bus (e.g. PCI or ISA) and a single SCSI initiator port on a SCSI transport. An "initiator" port (SCSI terminology, see SAM-3 at http://www.t10.org) sends SCSI commands to "target" SCSI ports (e.g. disks). There can be many LLDs in a running system, but only one per hardware type. Most LLDs can control one or more SCSI HBAs. Some HBAs contain multiple hosts.

In some cases the SCSI transport is an external bus that already has its own subsystem in Linux (e.g. USB and ieee1394). In such cases the SCSI subsystem LLD is a software bridge to the other driver subsystem. Examples are the usb-storage driver (found in the drivers/usb/storage directory) and the ieee1394/sbp2 driver (found in the drivers/ieee1394 directory).

For example, the aic7xxx LLD controls Adaptec SCSI parallel interface (SPI) controllers based on that company's 7xxx chip series. The aic7xxx LLD can be built into the kernel or loaded as a module. There can only be one aic7xxx LLD running in a Linux system but it may be controlling many HBAs. These HBAs might be either on PCI daughter-boards or built into the motherboard (or both). Some aic7xxx based HBAs are dual controllers and thus represent two hosts. Like most modern HBAs, each aic7xxx host has its own PCI device address. [The one-to-one correspondence between a SCSI host and a PCI device is common but not required (e.g. with ISA adapters).]

The SCSI mid level isolates an LLD from other layers such as the SCSI upper layer drivers and the block layer.

This version of the document roughly matches linux kernel version 2.6.8 .

Documentation

There is a SCSI documentation directory within the kernel source tree, typically Documentation/scsi . Most documents are in plain (i.e. ASCII) text. This file is named scsi_mid_low_api.txt and can be found in that directory. A more recent copy of this document may be found at http://web.archive.org/web/20070107183357rn_1/sg.torque.net/scsi/. Many LLDs are documented there (e.g. aic7xxx.txt). The SCSI mid-level is briefly described in scsi.txt which contains a url to a document describing the SCSI subsystem in the lk 2.4 series. Two upper level drivers have documents in that directory: st.txt (SCSI tape driver) and scsi-generic.txt (for the sg driver).

Some documentation (or urls) for LLDs may be found in the C source code or in the same directory as the C source code. For example to find a url about the USB mass storage driver see the /usr/src/linux/drivers/usb/storage directory.

Driver structure

Traditionally an LLD for the SCSI subsystem has been at least two files in the drivers/scsi directory. For example, a driver called "xyz" has a header file "xyz.h" and a source file "xyz.c". [Actually there is no good reason why this couldn't all be in one file; the header file is superfluous.] Some drivers that have been ported to several operating systems have more than two files. For example the aic7xxx driver has separate files for generic and OS-specific code (e.g. FreeBSD and Linux). Such drivers tend to have their own directory under the drivers/scsi directory.

When a new LLD is being added to Linux, the following files (found in the drivers/scsi directory) will need some attention: Makefile and Kconfig . It is probably best to study how existing LLDs are organized.

As the 2.5 series development kernels evolve into the 2.6 series production series, changes are being introduced into this interface. An example of this is driver initialization code where there are now 2 models available. The older one, similar to what was found in the lk 2.4 series, is based on hosts that are detected at HBA driver load time. This will be referred to the "passive" initialization model. The newer model allows HBAs to be hot plugged (and unplugged) during the lifetime of the LLD and will be referred to as the "hotplug" initialization model. The newer model is preferred as it can handle both traditional SCSI equipment that is permanently connected as well as modern "SCSI" devices (e.g. USB or IEEE 1394 connected digital cameras) that are hotplugged. Both initialization models are discussed in the following sections.

An LLD interfaces to the SCSI subsystem several ways:

a. directly invoking functions supplied by the mid level
b. passing a set of function pointers to a registration function supplied by the mid level. The mid level will then invoke these functions at some point in the future. The LLD will supply implementations of these functions.
c. direct access to instances of well known data structures maintained by the mid level

Those functions in group a) are listed in a section entitled "Mid level supplied functions" below.

Those functions in group b) are listed in a section entitled "Interface functions" below. Their function pointers are placed in the members of "struct scsi_host_template", an instance of which is passed to scsi_host_alloc() [1]. Those interface functions that the LLD does not wish to supply should have NULL placed in the corresponding member of struct scsi_host_template. Defining an

instance of struct scsi_host_template at file scope will cause NULL to be placed in function pointer members not explicitly initialized.

Those usages in group c) should be handled with care, especially in a "hotplug" environment. LLDs should be aware of the lifetime of instances that are shared with the mid level and other layers.

All functions defined within an LLD and all data defined at file scope should be static. For example the slave_alloc() function in an LLD called "xxx" could be defined as `static int xxx_slave_alloc(struct scsi_device * sdev) { /* code */ }`

[1] the scsi_host_alloc() function is a replacement for the rather vaguely named scsi_register() function in most situations.

Hotplug initialization model

In this model an LLD controls when SCSI hosts are introduced and removed from the SCSI subsystem. Hosts can be introduced as early as driver initialization and removed as late as driver shutdown. Typically a driver will respond to a sysfs probe() callback that indicates an HBA has been detected. After confirming that the new device is one that the LLD wants to control, the LLD will initialize the HBA and then register a new host with the SCSI mid level.

During LLD initialization the driver should register itself with the appropriate IO bus on which it expects to find HBA(s) (e.g. the PCI bus). This can probably be done via sysfs. Any driver parameters (especially those that are writable after the driver is loaded) could also be registered with sysfs at this point. The SCSI mid level first becomes aware of an LLD when that LLD registers its first HBA.

At some later time, the LLD becomes aware of an HBA and what follows is a typical sequence of calls between the LLD and the mid level. This example shows the mid level scanning the newly introduced HBA for 3 scsi devices of which only the first 2 respond:

```
    HBA PROBE: assume 2 SCSI devices found in scan
LLD                  mid level                   LLD
===--------------------------=------------------===------
scsi_host_alloc()  -->
scsi_add_host()  ---->
scsi_scan_host()  -------+
                         |
                    slave_alloc()
                    slave_configure() -->  scsi_change_queue_depth()
                         |
                    slave_alloc()
                    slave_configure()
                         |
                    slave_alloc()   ***
                    slave_destroy() ***


*** For scsi devices that the mid level tries to scan but do not
    respond, a slave_alloc(), slave_destroy() pair is called.
```

If the LLD wants to adjust the default queue settings, it can invoke scsi_change_queue_depth() in its slave_configure() routine.

When an HBA is being removed it could be as part of an orderly shutdown associated with the LLD module being unloaded (e.g. with the "rmmod" command) or in response to a "hot unplug" indicated by sysfs()'s remove() callback being invoked. In either case, the sequence is the same:

```
        HBA REMOVE: assume 2 SCSI devices attached
LLD                      mid level                 LLD
===----------------------=========================---------
scsi_remove_host() ---------+
                            |
                    slave_destroy()
                    slave_destroy()
scsi_host_put()
```

It may be useful for a LLD to keep track of struct Scsi_Host instances (a pointer is returned by scsi_host_alloc()). Such instances are "owned" by the mid-level. struct Scsi_Host instances are freed from scsi_host_put() when the reference count hits zero.

Hot unplugging an HBA that controls a disk which is processing SCSI commands on a mounted file system is an interesting situation. Reference counting logic is being introduced into the mid level to cope with many of the issues involved. See the section on reference counting below.

The hotplug concept may be extended to SCSI devices. Currently, when an HBA is added, the scsi_scan_host() function causes a scan for SCSI devices attached to the HBA's SCSI transport. On newer SCSI transports the HBA may become aware of a new SCSI device _after_ the scan has completed. An LLD can use this sequence to make the mid level aware of a SCSI device:

```
                SCSI DEVICE hotplug
LLD                   mid level                   LLD
===---------------------------=------------------===------
scsi_add_device()  ------+
                         |
                    slave_alloc()
                    slave_configure()   [--> scsi_change_queue_depth()]
```

In a similar fashion, an LLD may become aware that a SCSI device has been removed (unplugged) or the connection to it has been

interrupted. Some existing SCSI transports (e.g. SPI) may not become aware that a SCSI device has been removed until a subsequent SCSI command fails which will probably cause that device to be set offline by the mid level. An LLD that detects the removal of a SCSI device can instigate its removal from upper layers with this sequence:

```
                SCSI DEVICE hot unplug
LLD                      mid level                LLD
===----------------------=============================---------
scsi_remove_device() -------+
                            |
                     slave_destroy()
```

It may be useful for an LLD to keep track of struct scsi_device instances (a pointer is passed as the parameter to slave_alloc() and slave_configure() callbacks). Such instances are "owned" by the mid-level. struct scsi_device instances are freed after slave_destroy().

Reference Counting

The Scsi_Host structure has had reference counting infrastructure added. This effectively spreads the ownership of struct Scsi_Host instances across the various SCSI layers which use them. Previously such instances were exclusively owned by the mid level. LLDs would not usually need to directly manipulate these reference counts but there may be some cases where they do.

There are 3 reference counting functions of interest associated with struct Scsi_Host:

- scsi_host_alloc():
  returns a pointer to new instance of struct Scsi_Host which has its reference count ^^ set to 1
- scsi_host_get():
  adds 1 to the reference count of the given instance
- scsi_host_put():
  decrements 1 from the reference count of the given instance. If the reference count reaches 0 then the given instance is freed

The scsi_device structure has had reference counting infrastructure added. This effectively spreads the ownership of struct scsi_device instances across the various SCSI layers which use them. Previously such instances were exclusively owned by the mid level. See the access functions declared towards the end of include/scsi/scsi_device.h . If an LLD wants to keep a copy of a pointer to a scsi_device instance it should use scsi_device_get() to bump its reference count. When it is finished with the pointer it can use scsi_device_put() to decrement its reference count (and potentially delete it).

> **Note**
>
> struct Scsi_Host actually has 2 reference counts which are manipulated in parallel by these functions.

Conventions

First, Linus Torvalds's thoughts on C coding style can be found in the Documentation/process/coding-style.rst file.

Also, most C99 enhancements are encouraged to the extent they are supported by the relevant gcc compilers. So C99 style structure and array initializers are encouraged where appropriate. Don't go too far, VLAs are not properly supported yet. An exception to this is the use of `//` style comments; `/*...*/` comments are still preferred in Linux.

Well written, tested and documented code, need not be re-formatted to comply with the above conventions. For example, the aic7xxx driver comes to Linux from FreeBSD and Adaptec's own labs. No doubt FreeBSD and Adaptec have their own coding conventions.

Mid level supplied functions

These functions are supplied by the SCSI mid level for use by LLDs. The names (i.e. entry points) of these functions are exported so an LLD that is a module can access them. The kernel will arrange for the SCSI mid level to be loaded and initialized before any LLD is initialized. The functions below are listed alphabetically and their names all start with `scsi_`.

Summary:

- scsi_add_device - creates new scsi device (lu) instance
- scsi_add_host - perform sysfs registration and set up transport class
- scsi_change_queue_depth - change the queue depth on a SCSI device
- scsi_bios_ptable - return copy of block device's partition table
- scsi_block_requests - prevent further commands being queued to given host
- scsi_host_alloc - return a new scsi_host instance whose refcount==1
- scsi_host_get - increments Scsi_Host instance's refcount
- scsi_host_put - decrements Scsi_Host instance's refcount (free if 0)

- scsi_register - create and register a scsi host adapter instance.
- scsi_remove_device - detach and remove a SCSI device
- scsi_remove_host - detach and remove all SCSI devices owned by host
- scsi_report_bus_reset - report scsi _bus_ reset observed
- scsi_scan_host - scan SCSI bus
- scsi_track_queue_full - track successive QUEUE_FULL events
- scsi_unblock_requests - allow further commands to be queued to given host
- scsi_unregister - [calls scsi_host_put()]

Details:

```
/**
* scsi_add_device - creates new scsi device (lu) instance
* @shost:   pointer to scsi host instance
* @channel: channel number (rarely other than 0)
* @id:      target id number
* @lun:     logical unit number
*
*      Returns pointer to new struct scsi_device instance or
*      ERR_PTR(-ENODEV) (or some other bent pointer) if something is
*      wrong (e.g. no lu responds at given address)
*
*      Might block: yes
*
*      Notes: This call is usually performed internally during a scsi
*      bus scan when an HBA is added (i.e. scsi_scan_host()). So it
*      should only be called if the HBA becomes aware of a new scsi
*      device (lu) after scsi_scan_host() has completed. If successful
*      this call can lead to slave_alloc() and slave_configure() callbacks
*      into the LLD.
*
*      Defined in: drivers/scsi/scsi_scan.c
**/
struct scsi_device * scsi_add_device(struct Scsi_Host *shost,
                                     unsigned int channel,
                                     unsigned int id, unsigned int lun)


/**
* scsi_add_host - perform sysfs registration and set up transport class
* @shost:   pointer to scsi host instance
* @dev:     pointer to struct device of type scsi class
*
*      Returns 0 on success, negative errno of failure (e.g. -ENOMEM)
*
*      Might block: no
*
*      Notes: Only required in "hotplug initialization model" after a
*      successful call to scsi_host_alloc().  This function does not
*   scan the bus; this can be done by calling scsi_scan_host() or
*   in some other transport-specific way.  The LLD must set up
*   the transport template before calling this function and may only
*   access the transport class data after this function has been called.
*
*      Defined in: drivers/scsi/hosts.c
**/
int scsi_add_host(struct Scsi_Host *shost, struct device * dev)


/**
* scsi_change_queue_depth - allow LLD to change queue depth on a SCSI device
* @sdev:       pointer to SCSI device to change queue depth on
* @tags        Number of tags allowed if tagged queuing enabled,
*              or number of commands the LLD can queue up
*              in non-tagged mode (as per cmd_per_lun).
*
*      Returns nothing
*
*      Might block: no
*
*      Notes: Can be invoked any time on a SCSI device controlled by this
*      LLD. [Specifically during and after slave_configure() and prior to
*      slave_destroy().] Can safely be invoked from interrupt code.
*
*      Defined in: drivers/scsi/scsi.c [see source code for more notes]
*
**/
int scsi_change_queue_depth(struct scsi_device *sdev, int tags)
```

```
/**
* scsi_bios_ptable - return copy of block device's partition table
* @dev:        pointer to block device
*
*      Returns pointer to partition table, or NULL for failure
*
*      Might block: yes
*
*      Notes: Caller owns memory returned (free with kfree() )
*
*      Defined in: drivers/scsi/scsicam.c
**/
unsigned char *scsi_bios_ptable(struct block_device *dev)


/**
* scsi_block_requests - prevent further commands being queued to given host
*
* @shost: pointer to host to block commands on
*
*      Returns nothing
*
*      Might block: no
*
*      Notes: There is no timer nor any other means by which the requests
*      get unblocked other than the LLD calling scsi_unblock_requests().
*
*      Defined in: drivers/scsi/scsi_lib.c
**/
void scsi_block_requests(struct Scsi_Host * shost)


/**
* scsi_host_alloc - create a scsi host adapter instance and perform basic
*                   initialization.
* @sht:        pointer to scsi host template
* @privsize:   extra bytes to allocate in hostdata array (which is the
*              last member of the returned Scsi_Host instance)
*
*      Returns pointer to new Scsi_Host instance or NULL on failure
*
*      Might block: yes
*
*      Notes: When this call returns to the LLD, the SCSI bus scan on
*      this host has _not_ yet been done.
*      The hostdata array (by default zero length) is a per host scratch
*      area for the LLD's exclusive use.
*      Both associated refcounting objects have their refcount set to 1.
*      Full registration (in sysfs) and a bus scan are performed later when
*      scsi_add_host() and scsi_scan_host() are called.
*
*      Defined in: drivers/scsi/hosts.c .
**/
struct Scsi_Host * scsi_host_alloc(struct scsi_host_template * sht,
                                   int privsize)


/**
* scsi_host_get - increment Scsi_Host instance refcount
* @shost:   pointer to struct Scsi_Host instance
*
*      Returns nothing
*
*      Might block: currently may block but may be changed to not block
*
*      Notes: Actually increments the counts in two sub-objects
*
*      Defined in: drivers/scsi/hosts.c
**/
void scsi_host_get(struct Scsi_Host *shost)


/**
* scsi_host_put - decrement Scsi_Host instance refcount, free if 0
* @shost:   pointer to struct Scsi_Host instance
*
*      Returns nothing
*
*      Might block: currently may block but may be changed to not block
*
```

```
*      Notes: Actually decrements the counts in two sub-objects. If the
*      latter refcount reaches 0, the Scsi_Host instance is freed.
*      The LLD need not worry exactly when the Scsi_Host instance is
*      freed, it just shouldn't access the instance after it has balanced
*      out its refcount usage.
*
*      Defined in: drivers/scsi/hosts.c
**/
void scsi_host_put(struct Scsi_Host *shost)


/**
* scsi_register - create and register a scsi host adapter instance.
* @sht:        pointer to scsi host template
* @privsize:   extra bytes to allocate in hostdata array (which is the
*              last member of the returned Scsi_Host instance)
*
*      Returns pointer to new Scsi_Host instance or NULL on failure
*
*      Might block: yes
*
*      Notes: When this call returns to the LLD, the SCSI bus scan on
*      this host has _not_ yet been done.
*      The hostdata array (by default zero length) is a per host scratch
*      area for the LLD.
*
*      Defined in: drivers/scsi/hosts.c .
**/
struct Scsi_Host * scsi_register(struct scsi_host_template * sht,
                                 int privsize)


/**
* scsi_remove_device - detach and remove a SCSI device
* @sdev:      a pointer to a scsi device instance
*
*      Returns value: 0 on success, -EINVAL if device not attached
*
*      Might block: yes
*
*      Notes: If an LLD becomes aware that a scsi device (lu) has
*      been removed but its host is still present then it can request
*      the removal of that scsi device. If successful this call will
*      lead to the slave_destroy() callback being invoked. sdev is an
*      invalid pointer after this call.
*
*      Defined in: drivers/scsi/scsi_sysfs.c .
**/
int scsi_remove_device(struct scsi_device *sdev)


/**
* scsi_remove_host - detach and remove all SCSI devices owned by host
* @shost:      a pointer to a scsi host instance
*
*      Returns value: 0 on success, 1 on failure (e.g. LLD busy ??)
*
*      Might block: yes
*
*      Notes: Should only be invoked if the "hotplug initialization
*      model" is being used. It should be called _prior_ to
*      scsi_unregister().
*
*      Defined in: drivers/scsi/hosts.c .
**/
int scsi_remove_host(struct Scsi_Host *shost)


/**
* scsi_report_bus_reset - report scsi _bus_ reset observed
* @shost: a pointer to a scsi host involved
* @channel: channel (within) host on which scsi bus reset occurred
*
*      Returns nothing
*
*      Might block: no
*
*      Notes: This only needs to be called if the reset is one which
*      originates from an unknown location.  Resets originated by the
*      mid level itself don't need to call this, but there should be
*      no harm.  The main purpose of this is to make sure that a
```

```
*      CHECK_CONDITION is properly treated.
*
*      Defined in: drivers/scsi/scsi_error.c .
**/
void scsi_report_bus_reset(struct Scsi_Host * shost, int channel)


/**
* scsi_scan_host - scan SCSI bus
* @shost: a pointer to a scsi host instance
*
*   Might block: yes
*
*   Notes: Should be called after scsi_add_host()
*
*   Defined in: drivers/scsi/scsi_scan.c
**/
void scsi_scan_host(struct Scsi_Host *shost)


/**
* scsi_track_queue_full - track successive QUEUE_FULL events on given
*                       device to determine if and when there is a need
*                       to adjust the queue depth on the device.
* @sdev:  pointer to SCSI device instance
* @depth: Current number of outstanding SCSI commands on this device,
*         not counting the one returned as QUEUE_FULL.
*
*      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
*
*      Might block: no
*
*      Notes: LLDs may call this at any time and we will do "The Right
*              Thing"; interrupt context safe.
*
*      Defined in: drivers/scsi/scsi.c .
**/
int scsi_track_queue_full(struct scsi_device *sdev, int depth)


/**
* scsi_unblock_requests - allow further commands to be queued to given host
*
* @shost: pointer to host to unblock commands on
*
*      Returns nothing
*
*      Might block: no
*
*      Defined in: drivers/scsi/scsi_lib.c .
**/
void scsi_unblock_requests(struct Scsi_Host * shost)


/**
* scsi_unregister - unregister and free memory used by host instance
* @shp:        pointer to scsi host instance to unregister.
*
*      Returns nothing
*
*      Might block: no
*
*      Notes: Should not be invoked if the "hotplug initialization
*      model" is being used. Called internally by exit_this_scsi_driver()
*      in the "passive initialization model". Hence a LLD has no need to
*      call this function directly.
*
*      Defined in: drivers/scsi/hosts.c .
**/
void scsi_unregister(struct Scsi_Host * shp)
```

Interface Functions

Interface functions are supplied (defined) by LLDs and their function pointers are placed in an instance of struct scsi_host_template which is passed to scsi_host_alloc() [or scsi_register() / init_this_scsi_driver()]. Some are mandatory. Interface functions should be declared static. The accepted convention is that driver "xyz" will declare its slave_configure() function as:

```
static int xyz_slave_configure(struct scsi_device * sdev);
```

and so forth for all interface functions listed below.

A pointer to this function should be placed in the 'slave_configure' member of a "struct scsi_host_template" instance. A pointer to such an instance should be passed to the mid level's scsi_host_alloc() [or scsi_register() / init_this_scsi_driver()].

The interface functions are also described in the include/scsi/scsi_host.h file immediately above their definition point in "struct scsi_host_template". In some cases more detail is given in scsi_host.h than below.

The interface functions are listed below in alphabetical order.

Summary:

- bios_param - fetch head, sector, cylinder info for a disk
- eh_timed_out - notify the host that a command timer expired
- eh_abort_handler - abort given command
- eh_bus_reset_handler - issue SCSI bus reset
- eh_device_reset_handler - issue SCSI device reset
- eh_host_reset_handler - reset host (host bus adapter)
- info - supply information about given host
- ioctl - driver can respond to ioctls
- proc_info - supports /proc/scsi/{driver_name}/{host_no}
- queuecommand - queue scsi command, invoke 'done' on completion
- slave_alloc - prior to any commands being sent to a new device
- slave_configure - driver fine tuning for given device after attach
- slave_destroy - given device is about to be shut down

Details:

```
/**
*      bios_param - fetch head, sector, cylinder info for a disk
*      @sdev: pointer to scsi device context (defined in
*             include/scsi/scsi_device.h)
*      @bdev: pointer to block device context (defined in fs.h)
*      @capacity:  device size (in 512 byte sectors)
*      @params: three element array to place output:
*               params[0] number of heads (max 255)
*               params[1] number of sectors (max 63)
*               params[2] number of cylinders
*
*      Return value is ignored
*
*      Locks: none
*
*      Calling context: process (sd)
*
*      Notes: an arbitrary geometry (based on READ CAPACITY) is used
*      if this function is not provided. The params array is
*      pre-initialized with made up values just in case this function
*      doesn't output anything.
*
*      Optionally defined in: LLD
**/
    int bios_param(struct scsi_device * sdev, struct block_device *bdev,
                sector_t capacity, int params[3])


/**
*      eh_timed_out - The timer for the command has just fired
*      @scp: identifies command timing out
*
*      Returns:
*
*      EH_HANDLED:             I fixed the error, please complete the command
*      EH_RESET_TIMER:         I need more time, reset the timer and
*                              begin counting again
*      EH_NOT_HANDLED          Begin normal error recovery
*
*
*      Locks: None held
*
*      Calling context: interrupt
*
*      Notes: This is to give the LLD an opportunity to do local recovery.
*      This recovery is limited to determining if the outstanding command
*      will ever complete.  You may not abort and restart the command from
*      this callback.
```

```
*
*      Optionally defined in: LLD
**/
    int eh_timed_out(struct scsi_cmnd * scp)


/**
*      eh_abort_handler - abort command associated with scp
*      @scp: identifies command to be aborted
*
*      Returns SUCCESS if command aborted else FAILED
*
*      Locks: None held
*
*      Calling context: kernel thread
*
*      Notes: If 'no_async_abort' is defined this callback
*   will be invoked from scsi_eh thread. No other commands
*   will then be queued on current host during eh.
*   Otherwise it will be called whenever scsi_times_out()
*      is called due to a command timeout.
*
*      Optionally defined in: LLD
**/
    int eh_abort_handler(struct scsi_cmnd * scp)


/**
*      eh_bus_reset_handler - issue SCSI bus reset
*      @scp: SCSI bus that contains this device should be reset
*
*      Returns SUCCESS if command aborted else FAILED
*
*      Locks: None held
*
*      Calling context: kernel thread
*
*      Notes: Invoked from scsi_eh thread. No other commands will be
*      queued on current host during eh.
*
*      Optionally defined in: LLD
**/
    int eh_bus_reset_handler(struct scsi_cmnd * scp)


/**
*      eh_device_reset_handler - issue SCSI device reset
*      @scp: identifies SCSI device to be reset
*
*      Returns SUCCESS if command aborted else FAILED
*
*      Locks: None held
*
*      Calling context: kernel thread
*
*      Notes: Invoked from scsi_eh thread. No other commands will be
*      queued on current host during eh.
*
*      Optionally defined in: LLD
**/
    int eh_device_reset_handler(struct scsi_cmnd * scp)


/**
*      eh_host_reset_handler - reset host (host bus adapter)
*      @scp: SCSI host that contains this device should be reset
*
*      Returns SUCCESS if command aborted else FAILED
*
*      Locks: None held
*
*      Calling context: kernel thread
*
*      Notes: Invoked from scsi_eh thread. No other commands will be
*      queued on current host during eh.
*      With the default eh_strategy in place, if none of the _abort_,
*      _device_reset_, _bus_reset_ or this eh handler function are
*      defined (or they all return FAILED) then the device in question
*      will be set offline whenever eh is invoked.
*
*      Optionally defined in: LLD
```

```
**/
    int eh_host_reset_handler(struct scsi_cmnd * scp)


/**
*      info - supply information about given host: driver name plus data
*             to distinguish given host
*      @shp: host to supply information about
*
*      Return ASCII null terminated string. [This driver is assumed to
*      manage the memory pointed to and maintain it, typically for the
*      lifetime of this host.]
*
*      Locks: none
*
*      Calling context: process
*
*      Notes: Often supplies PCI or ISA information such as IO addresses
*      and interrupt numbers. If not supplied struct Scsi_Host::name used
*      instead. It is assumed the returned information fits on one line
*      (i.e. does not included embedded newlines).
*      The SCSI_IOCTL_PROBE_HOST ioctl yields the string returned by this
*      function (or struct Scsi_Host::name if this function is not
*      available).
*      In a similar manner, init_this_scsi_driver() outputs to the console
*      each host's "info" (or name) for the driver it is registering.
*      Also if proc_info() is not supplied, the output of this function
*      is used instead.
*
*      Optionally defined in: LLD
**/
    const char * info(struct Scsi_Host * shp)


/**
*      ioctl - driver can respond to ioctls
*      @sdp: device that ioctl was issued for
*      @cmd: ioctl number
*      @arg: pointer to read or write data from. Since it points to
*            user space, should use appropriate kernel functions
*            (e.g. copy_from_user() ). In the Unix style this argument
*            can also be viewed as an unsigned long.
*
*      Returns negative "errno" value when there is a problem. 0 or a
*      positive value indicates success and is returned to the user space.
*
*      Locks: none
*
*      Calling context: process
*
*      Notes: The SCSI subsystem uses a "trickle down" ioctl model.
*      The user issues an ioctl() against an upper level driver
*      (e.g. /dev/sdc) and if the upper level driver doesn't recognize
*      the 'cmd' then it is passed to the SCSI mid level. If the SCSI
*      mid level does not recognize it, then the LLD that controls
*      the device receives the ioctl. According to recent Unix standards
*      unsupported ioctl() 'cmd' numbers should return -ENOTTY.
*
*      Optionally defined in: LLD
**/
    int ioctl(struct scsi_device *sdp, int cmd, void *arg)


/**
*      proc_info - supports /proc/scsi/{driver_name}/{host_no}
*      @buffer: anchor point to output to (0==writeto1_read0) or fetch from
*               (1==writeto1_read0).
*      @start: where "interesting" data is written to. Ignored when
*              1==writeto1_read0.
*      @offset: offset within buffer 0==writeto1_read0 is actually
*               interested in. Ignored when 1==writeto1_read0 .
*      @length: maximum (or actual) extent of buffer
*      @host_no: host number of interest (struct Scsi_Host::host_no)
*      @writeto1_read0: 1 -> data coming from user space towards driver
*                           (e.g. "echo some_string > /proc/scsi/xyz/2")
*                       0 -> user what data from this driver
*                           (e.g. "cat /proc/scsi/xyz/2")
*
*      Returns length when 1==writeto1_read0. Otherwise number of chars
*      output to buffer past offset.
*
```

```
*      Locks: none held
*
*      Calling context: process
*
*      Notes: Driven from scsi_proc.c which interfaces to proc_fs. proc_fs
*      support can now be configured out of the scsi subsystem.
*
*      Optionally defined in: LLD
**/
    int proc_info(char * buffer, char ** start, off_t offset,
                  int length, int host_no, int writeto1_read0)


/**
*      queuecommand - queue scsi command, invoke scp->scsi_done on completion
*      @shost: pointer to the scsi host object
*      @scp: pointer to scsi command object
*
*      Returns 0 on success.
*
*      If there's a failure, return either:
*
*      SCSI_MLQUEUE_DEVICE_BUSY if the device queue is full, or
*      SCSI_MLQUEUE_HOST_BUSY if the entire host queue is full
*
*      On both of these returns, the mid-layer will requeue the I/O
*
*      - if the return is SCSI_MLQUEUE_DEVICE_BUSY, only that particular
*      device will be paused, and it will be unpaused when a command to
*      the device returns (or after a brief delay if there are no more
*      outstanding commands to it).  Commands to other devices continue
*      to be processed normally.
*
*      - if the return is SCSI_MLQUEUE_HOST_BUSY, all I/O to the host
*      is paused and will be unpaused when any command returns from
*      the host (or after a brief delay if there are no outstanding
*      commands to the host).
*
*      For compatibility with earlier versions of queuecommand, any
*      other return value is treated the same as
*      SCSI_MLQUEUE_HOST_BUSY.
*
*      Other types of errors that are detected immediately may be
*      flagged by setting scp->result to an appropriate value,
*      invoking the scp->scsi_done callback, and then returning 0
*      from this function. If the command is not performed
*      immediately (and the LLD is starting (or will start) the given
*      command) then this function should place 0 in scp->result and
*      return 0.
*
*      Command ownership.  If the driver returns zero, it owns the
*      command and must take responsibility for ensuring the
*      scp->scsi_done callback is executed.  Note: the driver may
*      call scp->scsi_done before returning zero, but after it has
*      called scp->scsi_done, it may not return any value other than
*      zero.  If the driver makes a non-zero return, it must not
*      execute the command's scsi_done callback at any time.
*
*      Locks: up to and including 2.6.36, struct Scsi_Host::host_lock
*             held on entry (with "irqsave") and is expected to be
*             held on return. From 2.6.37 onwards, queuecommand is
*             called without any locks held.
*
*      Calling context: in interrupt (soft irq) or process context
*
*      Notes: This function should be relatively fast. Normally it
*      will not wait for IO to complete. Hence the scp->scsi_done
*      callback is invoked (often directly from an interrupt service
*      routine) some time after this function has returned. In some
*      cases (e.g. pseudo adapter drivers that manufacture the
*      response to a SCSI INQUIRY) the scp->scsi_done callback may be
*      invoked before this function returns.  If the scp->scsi_done
*      callback is not invoked within a certain period the SCSI mid
*      level will commence error processing.  If a status of CHECK
*      CONDITION is placed in "result" when the scp->scsi_done
*      callback is invoked, then the LLD driver should perform
*      autosense and fill in the struct scsi_cmnd::sense_buffer
*      array. The scsi_cmnd::sense_buffer array is zeroed prior to
*      the mid level queuing a command to an LLD.
*
*      Defined in: LLD
```

```
**/
    int queuecommand(struct Scsi_Host *shost, struct scsi_cmnd * scp)


/**
*      slave_alloc -   prior to any commands being sent to a new device
*                      (i.e. just prior to scan) this call is made
*      @sdp: pointer to new device (about to be scanned)
*
*      Returns 0 if ok. Any other return is assumed to be an error and
*      the device is ignored.
*
*      Locks: none
*
*      Calling context: process
*
*      Notes: Allows the driver to allocate any resources for a device
*      prior to its initial scan. The corresponding scsi device may not
*      exist but the mid level is just about to scan for it (i.e. send
*      and INQUIRY command plus ...). If a device is found then
*      slave_configure() will be called while if a device is not found
*      slave_destroy() is called.
*      For more details see the include/scsi/scsi_host.h file.
*
*      Optionally defined in: LLD
**/
    int slave_alloc(struct scsi_device *sdp)


/**
*      slave_configure - driver fine tuning for given device just after it
*                     has been first scanned (i.e. it responded to an
*                     INQUIRY)
*      @sdp: device that has just been attached
*
*      Returns 0 if ok. Any other return is assumed to be an error and
*      the device is taken offline. [offline devices will _not_ have
*      slave_destroy() called on them so clean up resources.]
*
*      Locks: none
*
*      Calling context: process
*
*      Notes: Allows the driver to inspect the response to the initial
*      INQUIRY done by the scanning code and take appropriate action.
*      For more details see the include/scsi/scsi_host.h file.
*
*      Optionally defined in: LLD
**/
    int slave_configure(struct scsi_device *sdp)


/**
*      slave_destroy - given device is about to be shut down. All
*                      activity has ceased on this device.
*      @sdp: device that is about to be shut down
*
*      Returns nothing
*
*      Locks: none
*
*      Calling context: process
*
*      Notes: Mid level structures for given device are still in place
*      but are about to be torn down. Any per device resources allocated
*      by this driver for given device should be freed now. No further
*      commands will be sent for this sdp instance. [However the device
*      could be re-attached in the future in which case a new instance
*      of struct scsi_device would be supplied by future slave_alloc()
*      and slave_configure() calls.]
*
*      Optionally defined in: LLD
**/
    void slave_destroy(struct scsi_device *sdp)
```

Data Structures

struct scsi_host_template

There is one "struct scsi_host_template" instance per LLD [2]. It is typically initialized as a file scope static in a driver's header file.

That way members that are not explicitly initialized will be set to 0 or NULL. Member of interest:

name
- name of driver (may contain spaces, please limit to less than 80 characters)

proc_name
- name used in "/proc/scsi/<proc_name>/<host_no>" and by sysfs in one of its "drivers" directories. Hence "proc_name" should only contain characters acceptable to a Unix file name.

`(*queuecommand)()`
- primary callback that the mid level uses to inject SCSI commands into an LLD.

The structure is defined and commented in include/scsi/scsi_host.h

[2] In extreme situations a single driver may have several instances if it controls several different classes of hardware (e.g. an LLD that handles both ISA and PCI cards and has a separate instance of struct scsi_host_template for each class).

struct Scsi_Host

There is one struct Scsi_Host instance per host (HBA) that an LLD controls. The struct Scsi_Host structure has many members in common with "struct scsi_host_template". When a new struct Scsi_Host instance is created (in scsi_host_alloc() in hosts.c) those common members are initialized from the driver's struct scsi_host_template instance. Members of interest:

host_no
- system wide unique number that is used for identifying this host. Issued in ascending order from 0.

can_queue
- must be greater than 0; do not send more than can_queue commands to the adapter.

this_id
- scsi id of host (scsi initiator) or -1 if not known

sg_tablesize
- maximum scatter gather elements allowed by host. Set this to SG_ALL or less to avoid chained SG lists. Must be at least 1.

max_sectors
- maximum number of sectors (usually 512 bytes) allowed in a single SCSI command. The default value of 0 leads to a setting of SCSI_DEFAULT_MAX_SECTORS (defined in scsi_host.h) which is currently set to 1024. So for a disk the maximum transfer size is 512 KB when max_sectors is not defined. Note that this size may not be sufficient for disk firmware uploads.

cmd_per_lun
- maximum number of commands that can be queued on devices controlled by the host. Overridden by LLD calls to scsi_change_queue_depth().

no_async_abort
- 1=>Asynchronous aborts are not supported
- 0=>Timed-out commands will be aborted asynchronously

hostt
- pointer to driver's struct scsi_host_template from which this struct Scsi_Host instance was spawned

hostt->proc_name
- name of LLD. This is the driver name that sysfs uses

transportt
- pointer to driver's struct scsi_transport_template instance (if any). FC and SPI transports currently supported.

sh_list
- a double linked list of pointers to all struct Scsi_Host instances (currently ordered by ascending host_no)

my_devices
- a double linked list of pointers to struct scsi_device instances that belong to this host.

hostdata[0]
- area reserved for LLD at end of struct Scsi_Host. Size is set by the second argument (named 'xtr_bytes') to scsi_host_alloc() or scsi_register().

vendor_id
- a unique value that identifies the vendor supplying the LLD for the Scsi_Host. Used most often in validating vendor-specific message requests. Value consists of an identifier type and a vendor-specific value. See scsi_netlink.h for a description of valid formats.

The scsi_host structure is defined in include/scsi/scsi_host.h

struct scsi_device

Generally, there is one instance of this structure for each SCSI logical unit on a host. Scsi devices connected to a host are uniquely identified by a channel number, target id and logical unit number (lun). The structure is defined in include/scsi/scsi_device.h

struct scsi_cmnd

Instances of this structure convey SCSI commands to the LLD and responses back to the mid level. The SCSI mid level will ensure that no more SCSI commands become queued against the LLD than are indicated by scsi_change_queue_depth() (or struct Scsi_Host::cmd_per_lun). There will be at least one instance of struct scsi_cmnd available for each SCSI device. Members of interest:

cmnd
- array containing SCSI command

cmnd_len
- length (in bytes) of SCSI command

sc_data_direction
- direction of data transfer in data phase. See "enum dma_data_direction" in include/linux/dma-mapping.h

request_bufflen
- number of data bytes to transfer (0 if no data phase)

use_sg
- ==0 -> no scatter gather list, hence transfer data to/from request_buffer
- >0 -> scatter gather list (actually an array) in request_buffer with use_sg elements

request_buffer
- either contains data buffer or scatter gather list depending on the setting of use_sg. Scatter gather elements are defined by 'struct scatterlist' found in include/linux/scatterlist.h .

done
- function pointer that should be invoked by LLD when the SCSI command is completed (successfully or otherwise). Should only be called by an LLD if the LLD has accepted the command (i.e. queuecommand() returned or will return 0). The LLD may invoke 'done' prior to queuecommand() finishing.

result
- should be set by LLD prior to calling 'done'. A value of 0 implies a successfully completed command (and all data (if any) has been transferred to or from the SCSI target device). 'result' is a 32 bit unsigned integer that can be viewed as 2 related bytes. The SCSI status value is in the LSB. See include/scsi/scsi.h status_byte() and host_byte() macros and related constants.

sense_buffer
- an array (maximum size: SCSI_SENSE_BUFFERSIZE bytes) that should be written when the SCSI status (LSB of 'result') is set to CHECK_CONDITION (2). When CHECK_CONDITION is set, if the top nibble of sense_buffer[0] has the value 7 then the mid level will assume the sense_buffer array contains a valid SCSI sense buffer; otherwise the mid level will issue a REQUEST_SENSE SCSI command to retrieve the sense buffer. The latter strategy is error prone in the presence of command queuing so the LLD should always "auto-sense".

device
- pointer to scsi_device object that this command is associated with.

resid
- an LLD should set this signed integer to the requested transfer length (i.e. 'request_bufflen') less the number of bytes that are actually transferred. 'resid' is preset to 0 so an LLD can ignore it if it cannot detect underruns (overruns should be rare). If possible an LLD should set 'resid' prior to invoking 'done'. The most interesting case is data transfers from a SCSI target device (e.g. READs) that underrun.

underflow
- LLD should place (DID_ERROR << 16) in 'result' if actual number of bytes transferred is less than this figure. Not many LLDs implement this check and some that do just output an error message to the log rather than report a DID_ERROR. Better for an LLD to implement 'resid'.

It is recommended that a LLD set 'resid' on data transfers from a SCSI target device (e.g. READs). It is especially important that 'resid' is set when such data transfers have sense keys of MEDIUM ERROR and HARDWARE ERROR (and possibly RECOVERED ERROR). In these cases if a LLD is in doubt how much data has been received then the safest approach is to indicate no bytes have been received. For example: to indicate that no valid data has been received a LLD might use these helpers:

```
scsi_set_resid(SCpnt, scsi_bufflen(SCpnt));
```

where 'SCpnt' is a pointer to a scsi_cmnd object. To indicate only three 512 bytes blocks has been received 'resid' could be set like this:

```
scsi_set_resid(SCpnt, scsi_bufflen(SCpnt) - (3 * 512));
```

The scsi_cmnd structure is defined in include/scsi/scsi_cmnd.h

Locks

Each struct Scsi_Host instance has a spin_lock called struct Scsi_Host::default_lock which is initialized in scsi_host_alloc() [found in hosts.c]. Within the same function the struct Scsi_Host::host_lock pointer is initialized to point at default_lock. Thereafter lock and unlock operations performed by the mid level use the struct Scsi_Host::host_lock pointer. Previously drivers could override the host_lock pointer but this is not allowed anymore.

Autosense

Autosense (or auto-sense) is defined in the SAM-2 document as "the automatic return of sense data to the application client coincident with the completion of a SCSI command" when a status of CHECK CONDITION occurs. LLDs should perform autosense. This should be done when the LLD detects a CHECK CONDITION status by either:

a. instructing the SCSI protocol (e.g. SCSI Parallel Interface (SPI)) to perform an extra data in phase on such responses
b. or, the LLD issuing a REQUEST SENSE command itself

Either way, when a status of CHECK CONDITION is detected, the mid level decides whether the LLD has performed autosense by checking struct scsi_cmnd::sense_buffer[0] . If this byte has an upper nibble of 7 (or 0xf) then autosense is assumed to have taken place. If it has another value (and this byte is initialized to 0 before each command) then the mid level will issue a REQUEST SENSE command.

In the presence of queued commands the "nexus" that maintains sense buffer data from the command that failed until a following REQUEST SENSE may get out of synchronization. This is why it is best for the LLD to perform autosense.

Changes since lk 2.4 series

io_request_lock has been replaced by several finer grained locks. The lock relevant to LLDs is struct Scsi_Host::host_lock and there is one per SCSI host.

The older error handling mechanism has been removed. This means the LLD interface functions abort() and reset() have been removed. The struct scsi_host_template::use_new_eh_code flag has been removed.

In the 2.4 series the SCSI subsystem configuration descriptions were aggregated with the configuration descriptions from all other Linux subsystems in the Documentation/Configure.help file. In the 2.6 series, the SCSI subsystem now has its own (much smaller) drivers/scsi/Kconfig file that contains both configuration and help information.

struct SHT has been renamed to struct scsi_host_template.

Addition of the "hotplug initialization model" and many extra functions to support it.

Credits

The following people have contributed to this document:

- Mike Anderson <andmike at us dot ibm dot com>
- James Bottomley <James dot Bottomley at hansenpartnership dot com>
- Patrick Mansfield <patmans at us dot ibm dot com>
- Christoph Hellwig <hch at infradead dot org>
- Doug Ledford <dledford at redhat dot com>
- Andries Brouwer <Andries dot Brouwer at cwi dot nl>
- Randy Dunlap <rdunlap at xenotime dot net>
- Alan Stern <stern at rowland dot harvard dot edu>

Douglas Gilbert dgilbert at interlog dot com

21st September 2004