

SCSI Subsystem Interfaces

Douglas Gilbert

dgilbert@interlog.com

SCSI Subsystem Interfaces

by Douglas Gilbert

Published 2003-08-11

Copyright © 2002, 2003 Douglas Gilbert

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

Table of Contents

1. Introduction.....	1
2. Driver structure.....	3
3. Interface Functions	5
scsi_activate_tcq	5
scsi_unblock_requests.....	6
bios_param	7
queuecommand	8
4. Locks	11
5. Changes since lk 2.4 series.....	13
6. Credits	15

Chapter 1. Introduction

This document outlines the interface between the Linux scsi mid level and lower level drivers. Lower level drivers are variously called HBA (host bus adapter) drivers, host drivers (HD) or pseudo adapter drivers. The latter alludes to the fact that a lower level driver may be a bridge to another IO subsystem (and the "ide-scsi" driver is an example of this). There can be many lower level drivers active in a running system, but only one per hardware type. For example, the `aic7xxx` driver controls adaptec controllers based on the 7xxx chip series. Most lower level drivers can control one or more scsi hosts (a.k.a. scsi initiators).

This document can be found in an ASCII text file in the linux kernel source: `Documentation/scsi/scsi_mid_low_api.txt`. It currently hold a little more information than this document. The `drivers/scsi/hosts.h` and `drivers/scsi/scsi.h` headers contain descriptions of members of important structures for the scsi subsystem.

Chapter 2. Driver structure

Traditionally a lower level driver 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.] Some drivers that have been ported to several operating systems (e.g. aic7xxx which has separate files for generic and OS-specific code) have more than two files. Such drivers tend to have their own directory under the drivers/scsi directory.

scsi_module.c is normally included at the end of a lower level driver. For it to work a declaration like this is needed before it is included:

```
static Scsi_Host_Template driver_template = DRIVER_TEMPLATE;
/* DRIVER_TEMPLATE should contain pointers to supported interface
   functions. Scsi_Host_Template is defined hosts.h */
#include "scsi_module.c"
```

The scsi_module.c assumes the name "driver_template" is appropriately defined. It contains 2 functions:

1. init_this_scsi_driver() called during builtin and module driver initialization:
invokes mid level's scsi_register_host()
2. exit_this_scsi_driver() called during closedown: invokes mid level's
scsi_unregister_host()

When a new, lower level driver is being added to Linux, the following files (all found in the drivers/scsi directory) will need some attention: Makefile, Config.help and Config.in . It is probably best to look at what an existing lower level driver does in this regard.

Chapter 3. Interface Functions

scsi_activate_tcq

Name

`scsi_activate_tcq` — turn on tag command queueing (“ordered” task attribute)

Synopsis

```
void scsi_activate_tcq (struct scsi_device * sdev, int depth);
```

Arguments

sdev

device to turn on TCQ for

depth

queue depth

Description

Returns nothing

Might block

no

Notes

Eventually, it is hoped depth would be the maximum depth the device could cope with and the real queue depth would be adjustable from 0 to depth.

Defined (inline) in: `include/scsi/scsi_tcq.h`

scsi_unblock_requests

Name

`scsi_unblock_requests` — allow further commands to be queued to given host

Synopsis

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

Arguments

shost

pointer to host to unblock commands on

Description

Returns nothing

Description

Returns nothing

Might block

no

Defined in

drivers/scsi/scsi_lib.c .

bios_param

Name

`bios_param` — fetch head, sector, cylinder info for a disk

Synopsis

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

Arguments

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[3]`

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

Description

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

queuecommand

Name

`queuecommand` — queue scsi command, invoke 'done' on completion

Synopsis

```
int queuecommand (struct scsi_cmnd * scp, void (*done) (struct
scsi_cmnd *));
```

Arguments

scp

pointer to scsi command object

done

function pointer to be invoked on completion

Description

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 '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 'done' callback is executed. Note: the driver may call done before returning zero, but after it has called 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 done callback at any time.

Locks

struct Scsi_Host::host_lock held on entry (with "irqsave") and is expected to be held on return.

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 '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 'done' callback may be invoked before this function returns. If the '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 '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

Chapter 4. Locks

Each `Scsi_Host` instance has a `spin_lock` called `Scsi_Host::default_lock` which is initialized in `scsi_register()` [found in `hosts.c`]. Within the same function the `Scsi_Host::host_lock` pointer is initialized to point at `default_lock` with the `scsi_assign_lock()` function. Thereafter lock and unlock operations performed by the mid level use the `Scsi_Host::host_lock` pointer.

Lower level drivers can override the use of `Scsi_Host::default_lock` by using `scsi_assign_lock()`. The earliest opportunity to do this would be in the `detect()` function after it has invoked `scsi_register()`. It could be replaced by a coarser grain lock (e.g. per driver) or a lock of equal granularity (i.e. per host). Using finer grain locks (e.g. per scsi device) may be possible by juggling locks in `queuecommand()`.

Chapter 5. Changes since lk 2.4 series

`io_request_lock` has been replaced by several finer grained locks. The lock relevant to lower level drivers is `Scsi_Host::host_lock` and there is one per scsi host.

The older error handling mechanism has been removed. This means the lower level interface functions `abort()` and `reset()` have 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 2.5 series, the scsi subsystem now has its own (much smaller) `drivers/scsi/Config.help` file.

Chapter 6. Credits

The following people have contributed to this document:

1. Mike Anderson <andmike@us.ibm.com>
2. James Bottomley <James.Bottomley@steeleye.com>
3. Patrick Mansfield <patmans@us.ibm.com>

