

libATA Developer's Guide

Jeff Garzik

Copyright © 2003-2006 Jeff Garzik

The contents of this file are subject to the Open Software License version 1.1 that can be found at <http://www.opensource.org/licenses/osl-1.1.txt> and is included herein by reference.

Alternatively, the contents of this file may be used under the terms of the GNU General Public License version 2 (the "GPL") as distributed in the kernel source COPYING file, in which case the provisions of the GPL are applicable instead of the above. If you wish to allow the use of your version of this file only under the terms of the GPL and not to allow others to use your version of this file under the OSL, indicate your decision by deleting the provisions above and replace them with the notice and other provisions required by the GPL. If you do not delete the provisions above, a recipient may use your version of this file under either the OSL or the GPL.

Table of Contents

[1. Introduction](#)

[2. libata Driver API](#)

[struct ata_port_operations](#)

[Disable ATA port](#)

[Post-IDENTIFY device configuration](#)

[Set PIO/DMA mode](#)

[Taskfile read/write](#)

[PIO data read/write](#)

[ATA command execute](#)

[Per-cmd ATAPI DMA capabilities filter](#)

[Read specific ATA shadow registers](#)

[Select ATA device on bus](#)

[Private tuning method](#)

[Control PCI IDE BMDMA engine](#)

[High-level taskfile hooks](#)

[Exception and probe handling \(EH\)](#)

[Hardware interrupt handling](#)

[SATA phy read/write](#)

[Init and shutdown](#)

[3. Error handling](#)

[Origins of commands](#)

[How commands are issued](#)

[How commands are processed](#)

[How commands are completed](#)

[ata_scsi_error\(\)](#)

Problems with the current EH

4. libata Library

[ata link next](#) — link iteration helper
[ata dev next](#) — device iteration helper
[atapi cmd type](#) — Determine ATAPI command type from SCSI opcode
[ata tf to fis](#) — Convert ATA taskfile to SATA FIS structure
[ata tf from fis](#) — Convert SATA FIS to ATA taskfile
[ata pack xfermask](#) — Pack pio, mwdma and udma masks into xfer_mask
[ata unpack xfermask](#) — Unpack xfer_mask into pio, mwdma and udma masks
[ata xfer_mask2mode](#) — Find matching XFER_* for the given xfer_mask
[ata xfer mode2mask](#) — Find matching xfer_mask for XFER_*
[ata xfer mode2shift](#) — Find matching xfer_shift for XFER_*
[ata mode string](#) — convert xfer_mask to string
[ata dev classify](#) — determine device type based on ATA-spec signature
[ata id string](#) — Convert IDENTIFY DEVICE page into string
[ata id c string](#) — Convert IDENTIFY DEVICE page into C string
[ata id xfermask](#) — Compute xfermask from the given IDENTIFY data
[ata pio queue task](#) — Queue port_task
[ata pio need iordy](#) — check if iordy needed
[ata do dev read id](#) — default ID read method
[ata cable 40wire](#) — return 40 wire cable type
[ata cable 80wire](#) — return 80 wire cable type
[ata cable unknown](#) — return unknown PATA cable.
[ata cable ignore](#) — return ignored PATA cable.
[ata cable sata](#) — return SATA cable type
[ata port probe](#) — Mark port as enabled
[ata dev pair](#) — return other device on cable
[ata port disable](#) — Disable port.
[sata set spd](#) — set SATA spd according to spd limit
[ata timing cycle2mode](#) — find xfer mode for the specified cycle duration
[ata do set mode](#) — Program timings and issue SET FEATURES - XFER
[ata wait after reset](#) — wait for link to become ready after reset
[sata link debounce](#) — debounce SATA phy status
[sata link resume](#) — resume SATA link
[ata std prereset](#) — prepare for reset
[sata link hardreset](#) — reset link via SATA phy reset
[sata std hardreset](#) — COMRESET w/o waiting or classification
[ata std postreset](#) — standard postreset callback
[ata std qc defer](#) — Check whether a qc needs to be deferred
[ata sg init](#) — Associate command with scatter-gather table.
[ata qc complete](#) — Complete an active ATA command
[ata qc complete multiple](#) — Complete multiple qcs successfully
[sata scr valid](#) — test whether SCRs are accessible
[sata scr read](#) — read SCR register of the specified port
[sata scr write](#) — write SCR register of the specified port
[sata scr write flush](#) — write SCR register of the specified port and flush
[ata link online](#) — test whether the given link is online
[ata link offline](#) — test whether the given link is offline
[ata host suspend](#) — suspend host

[ata_host_resume](#) — resume host
[ata_port_start](#) — Set port up for dma.
[ata_host_alloc](#) — allocate and init basic ATA host resources
[ata_host_alloc_pininfo](#) — alloc host and init with port_info array
[ata_slave_link_init](#) — initialize slave link
[ata_host_start](#) — start and freeze ports of an ATA host
[ata_host_init](#) — Initialize a host struct
[ata_host_register](#) — register initialized ATA host
[ata_host_activate](#) — start host, request IRQ and register it
[ata_host_detach](#) — Detach all ports of an ATA host
[ata_pci_remove_one](#) — PCI layer callback for device removal
[ata_wait_register](#) — wait until register value changes

5. libata Core Internals

[ata_dev_phys_link](#) — find physical link for a device
[ata_force_cbl](#) — force cable type according to libata.force
[ata_force_link_limits](#) — force link limits according to libata.force
[ata_force_xfermask](#) — force xfermask according to libata.force
[ata_force_horkage](#) — force horkage according to libata.force
[ata_rwcmd_protocol](#) — set taskfile r/w commands and protocol
[ata_tf_read_block](#) — Read block address from ATA taskfile
[ata_build_rw_tf](#) — Build ATA taskfile for given read/write request
[ata_dev_enable_pm](#) — enable SATA interface power management
[ata_dev_disable_pm](#) — disable SATA interface power management
[ata_read_native_max_address](#) — Read native max address
[ata_set_max_sectors](#) — Set max sectors
[ata_hpa_resize](#) — Resize a device with an HPA set
[ata_dump_id](#) — IDENTIFY DEVICE info debugging output
[ata_port_flush_task](#) — Flush port_task
[ata_exec_internal_sg](#) — execute libata internal command
[ata_exec_internal](#) — execute libata internal command
[ata_do_simple_cmd](#) — execute simple internal command
[ata_pio_mask_no_iordy](#) — Return the non IORDY mask
[ata_dev_read_id](#) — Read ID data from the specified device
[ata_dev_configure](#) — Configure the specified ATA/ATAPI device
[ata_bus_probe](#) — Reset and probe ATA bus
[sata_print_link_status](#) — Print SATA link status
[sata_down_spd_limit](#) — adjust SATA spd limit downward
[sata_set_spd_needed](#) — is SATA spd configuration needed
[ata_down_xfermask_limit](#) — adjust dev xfer masks downward
[ata_wait_ready](#) — wait for link to become ready
[ata_dev_same_device](#) — Determine whether new ID matches configured device
[ata_dev_reread_id](#) — Re-read IDENTIFY data
[ata_dev_revalidate](#) — Revalidate ATA device
[ata_is_40wire](#) — check drive side detection
[cable_is_40wire](#) — 40/80/SATA decider
[ata_dev_xfermask](#) — Compute supported xfermask of the given device
[ata_dev_set_xfermode](#) — Issue SET FEATURES - XFER MODE command
[ata_dev_set_feature](#) — Issue SET FEATURES - SATA FEATURES
[ata_dev_init_params](#) — Issue INIT DEV PARAMS command

[ata_sg_clean](#) — Unmap DMA memory associated with command
[atapi_check_dma](#) — Check whether ATAPI DMA can be supported
[ata_sg_setup](#) — DMA-map the scatter-gather table associated with a command.
[swap_buf_le16](#) — swap halves of 16-bit words in place
[ata_qc_new](#) — Request an available ATA command, for queueing
[ata_qc_new_init](#) — Request an available ATA command, and initialize it
[ata_qc_free](#) — free unused `ata_queued_cmd`
[ata_qc_issue](#) — issue taskfile to device
[ata_phys_link_online](#) — test whether the given link is online
[ata_phys_link_offline](#) — test whether the given link is offline
[ata_dev_init](#) — Initialize an `ata_device` structure
[ata_link_init](#) — Initialize an `ata_link` structure
[sata_link_init_spd](#) — Initialize link->sata_spd_limit
[ata_port_alloc](#) — allocate and initialize basic ATA port resources
[ata_finalize_port_ops](#) — finalize `ata_port_operations`
[ata_port_detach](#) — Detach ATA port in preparation of device removal

6. libata SCSI translation/emulation

[ata_std_bios_param](#) — generic bios head/sector/cylinder calculator used by sd.
[ata_scsi_slave_config](#) — Set SCSI device attributes
[ata_scsi_slave_destroy](#) — SCSI device is about to be destroyed
[ata_scsi_change_queue_depth](#) — SCSI callback for queue depth config
[ata_scsi_queuecmd](#) — Issue SCSI cdb to libata-managed device
[ata_scsi_simulate](#) — simulate SCSI command on ATA device
[ata_sas_port_alloc](#) — Allocate port for a SAS attached SATA device
[ata_sas_port_start](#) — Set port up for dma.
[ata_sas_port_stop](#) — Undo `ata_sas_port_start`
[ata_sas_port_init](#) — Initialize a SATA device
[ata_sas_port_destroy](#) — Destroy a SATA port allocated by `ata_sas_port_alloc`
[ata_sas_slave_configure](#) — Default slave_config routine for libata devices
[ata_sas_queuecmd](#) — Issue SCSI cdb to libata-managed device
[ata_get_identity](#) — Handler for HDIO_GET_IDENTITY ioctl
[ata_cmd_ioctl](#) — Handler for HDIO_DRIVE_CMD ioctl
[ata_task_ioctl](#) — Handler for HDIO_DRIVE_TASK ioctl
[ata_scsi_qc_new](#) — acquire new `ata_queued_cmd` reference
[ata_dump_status](#) — user friendly display of error info
[ata_to_sense_error](#) — convert ATA error to SCSI error
[ata_gen_ata_sense](#) — generate a SCSI fixed sense block
[atapi_drain_needed](#) — Check whether data transfer may overflow
[ata_scsi_start_stop_xlat](#) — Translate SCSI START STOP UNIT command
[ata_scsi_flush_xlat](#) — Translate SCSI SYNCHRONIZE CACHE command
[scsi_6_lba_len](#) — Get LBA and transfer length
[scsi_10_lba_len](#) — Get LBA and transfer length
[scsi_16_lba_len](#) — Get LBA and transfer length
[ata_scsi_verify_xlat](#) — Translate SCSI VERIFY command into an ATA one
[ata_scsi_rw_xlat](#) — Translate SCSI r/w command into an ATA one
[ata_scsi_translate](#) — Translate then issue SCSI command to ATA device
[ata_scsi_rbuf_get](#) — Map response buffer.
[ata_scsi_rbuf_put](#) — Unmap response buffer.
[ata_scsi_rbuf_fill](#) — wrapper for SCSI command simulators

[ata scsiop inq std](#) — Simulate INQUIRY command
[ata scsiop inq 00](#) — Simulate INQUIRY VPD page 0, list of pages
[ata scsiop inq 80](#) — Simulate INQUIRY VPD page 80, device serial number
[ata scsiop inq 83](#) — Simulate INQUIRY VPD page 83, device identity
[ata scsiop inq 89](#) — Simulate INQUIRY VPD page 89, ATA info
[ata scsiop noop](#) — Command handler that simply returns success.
[ata msense caching](#) — Simulate MODE SENSE caching info page
[ata msense ctl mode](#) — Simulate MODE SENSE control mode page
[ata msense rw recovery](#) — Simulate MODE SENSE r/w error recovery page
[ata scsiop mode sense](#) — Simulate MODE SENSE 6, 10 commands
[ata scsiop read cap](#) — Simulate READ CAPACITY[16] commands
[ata scsiop report luns](#) — Simulate REPORT LUNS command
[atapi xlat](#) — Initialize PACKET taskfile
[ata scsi find dev](#) — lookup ata_device from scsi_cmnd
[ata scsi pass thru](#) — convert ATA pass-thru CDB to taskfile
[ata get xlat func](#) — check if SCSI to ATA translation is possible
[ata scsi dump cdb](#) — dump SCSI command contents to dmesg
[ata scsi offline dev](#) — offline attached SCSI device
[ata scsi remove dev](#) — remove attached SCSI device
[ata scsi media change notify](#) — send media change event
[ata scsi hotplug](#) — SCSI part of hotplug
[ata scsi user scan](#) — indication for user-initiated bus scan
[ata scsi dev rescan](#) — initiate scsi_rescan_device

7. ATA errors and exceptions

Exception categories

[HSM violation](#)
[ATA/ATAPI device error \(non-NCQ / non-CHECK CONDITION\)](#)
[ATAPI device CHECK CONDITION](#)
[ATA device error \(NCQ\)](#)
[ATA bus error](#)
[PCI bus error](#)
[Late completion](#)
[Unknown error \(timeout\)](#)
[Hotplug and power management exceptions](#)

EH recovery actions

[Clearing error condition](#)
[Reset](#)
[Reconfigure transport](#)

8. ata_piix Internals

[ich_pata cable detect](#) — Probe host controller cable detect info
[piix_pata prereset](#) — prereset for PATA host controller
[piix_set piomode](#) — Initialize host controller PATA PIO timings
[do_pata_set dmamode](#) — Initialize host controller PATA PIO timings
[piix_set dmamode](#) — Initialize host controller PATA DMA timings

[ich_set_dmamode](#) — Initialize host controller PATA DMA timings
[piix_check_450nx_errata](#) — Check for problem 450NX setup
[piix_init_one](#) — Register PIIX ATA PCI device with kernel services

[9. sata_sil Internals](#)

[sil_set_mode](#) — wrap set_mode functions
[sil_dev_config](#) — Apply device/host-specific errata fixups

[10. Thanks](#)

Chapter 1. Introduction

libATA is a library used inside the Linux kernel to support ATA host controllers and devices. libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI<->ATA translation for ATA devices according to the T10 SAT specification.

This Guide documents the libATA driver API, library functions, library internals, and a couple sample ATA low-level drivers.

Chapter 2. libata Driver API

Table of Contents

[struct ata_port_operations](#)

[Disable ATA port](#)
[Post-IDENTIFY device configuration](#)
[Set PIO/DMA mode](#)
[Taskfile read/write](#)
[PIO data read/write](#)
[ATA command execute](#)
[Per-cmd ATAPI DMA capabilities filter](#)
[Read specific ATA shadow registers](#)
[Select ATA device on bus](#)
[Private tuning method](#)
[Control PCI IDE BMDMA engine](#)
[High-level taskfile hooks](#)
[Exception and probe handling \(EH\)](#)
[Hardware interrupt handling](#)
[SATA phy read/write](#)
[Init and shutdown](#)

struct ata_port_operations is defined for every low-level libata hardware driver, and it controls how the low-level driver interfaces with the ATA and SCSI layers.

FIS-based drivers will hook into the system with ->qc_prep() and ->qc_issue() high-level hooks. Hardware which behaves in a manner similar to PCI IDE hardware may utilize several generic helpers, defining at a bare minimum the bus I/O addresses of the ATA shadow register blocks.

struct ata_port_operations

Disable ATA port

```
void (*port_disable) (struct ata_port *);
```

Called from ata_bus_probe() and ata_bus_reset() error paths, as well as when unregistering from the SCSI module (rmmod, hot unplug). This function should do whatever needs to be done to take the port out of use. In most cases, ata_port_disable() can be used as this hook.

Called from ata_bus_probe() on a failed probe. Called from ata_bus_reset() on a failed bus reset. Called from ata_scsi_release().

Post-IDENTIFY device configuration

```
void (*dev_config) (struct ata_port *, struct ata_device *);
```

Called after IDENTIFY [PACKET] DEVICE is issued to each device found. Typically used to apply device-specific fixups prior to issue of SET FEATURES - XFER MODE, and prior to operation.

Called by ata_device_add() after ata_dev_identify() determines a device is present.

This entry may be specified as NULL in ata_port_operations.

Set PIO/DMA mode

```
void (*set_piomode) (struct ata_port *, struct ata_device *);
void (*set_dmamode) (struct ata_port *, struct ata_device *);
void (*post_set_mode) (struct ata_port *);
unsigned int (*mode_filter) (struct ata_port *, struct ata_device *, unsigned int);
```

Hooks called prior to the issue of SET FEATURES - XFER MODE command. The optional ->mode_filter() hook is called when libata has built a mask of the possible modes. This is passed to the ->mode_filter() function which should return a mask of valid modes after filtering those unsuitable due to hardware limits. It is not valid to use this interface to add modes.

dev->pio_mode and dev->dma_mode are guaranteed to be valid when ->set_piomode() and when ->set_dmamode() is called. The timings for any other drive sharing the cable will also be valid at this point. That is the library records the decisions for the modes of each drive on a channel before it attempts to set any of them.

->post_set_mode() is called unconditionally, after the SET FEATURES - XFER MODE command completes successfully.

->set_piomode() is always called (if present), but ->set_dma_mode() is only called if DMA is possible.

Taskfile read/write

```
void (*tf_load) (struct ata_port *ap, struct ata_taskfile *tf);
```



```
void (*tf_read) (struct ata_port *ap, struct ata_taskfile *tf);
```

->tf_load() is called to load the given taskfile into hardware registers / DMA buffers. ->tf_read() is called to read the hardware registers / DMA buffers, to obtain the current set of taskfile register values. Most drivers for taskfile-based hardware (PIO or MMIO) use ata_tf_load() and ata_tf_read() for these hooks.

PIO data read/write

```
void (*data_xfer) (struct ata_device *, unsigned char *, unsigned int, int);
```

All bmdma-style drivers must implement this hook. This is the low-level operation that actually copies the data bytes during a PIO data transfer. Typically the driver will choose one of ata_pio_data_xfer_noirq(), ata_pio_data_xfer(), or ata_mmio_data_xfer().

ATA command execute

```
void (*exec_command)(struct ata_port *ap, struct ata_taskfile *tf);
```

causes an ATA command, previously loaded with ->tf_load(), to be initiated in hardware. Most drivers for taskfile-based hardware use ata_exec_command() for this hook.

Per-cmd ATAPI DMA capabilities filter

```
int (*check_atapi_dma) (struct ata_queued_cmd *qc);
```

Allow low-level driver to filter ATA PACKET commands, returning a status indicating whether or not it is OK to use DMA for the supplied PACKET command.

This hook may be specified as NULL, in which case libata will assume that atapi dma can be supported.

Read specific ATA shadow registers

```
u8 (*check_status)(struct ata_port *ap);
u8 (*check_altstatus)(struct ata_port *ap);
```

Reads the Status/AltStatus ATA shadow register from hardware. On some hardware, reading the Status register has the side effect of clearing the interrupt condition. Most drivers for taskfile-based hardware use ata_check_status() for this hook.

Note that because this is called from ata_device_add(), at least a dummy function that clears device interrupts must be provided for all drivers, even if the controller doesn't actually have a taskfile status register.

Select ATA device on bus

```
void (*dev_select)(struct ata_port *ap, unsigned int device);
```


Issues the low-level hardware command(s) that causes one of N hardware devices to be considered 'selected' (active and available for use) on the ATA bus. This generally has no meaning on FIS-based devices.

Most drivers for taskfile-based hardware use `ata_std_dev_select()` for this hook. Controllers which do not support second drives on a port (such as SATA controllers) will use `ata_noop_dev_select()`.

Private tuning method

```
void (*set_mode) (struct ata_port *ap);
```

By default libata performs drive and controller tuning in accordance with the ATA timing rules and also applies blacklists and cable limits. Some controllers need special handling and have custom tuning rules, typically raid controllers that use ATA commands but do not actually do drive timing.

Warning

This hook should not be used to replace the standard controller tuning logic when a controller has quirks. Replacing the default tuning logic in that case would bypass handling for drive and bridge quirks that may be important to data reliability. If a controller needs to filter the mode selection it should use the `mode_filter` hook instead.

Control PCI IDE BMDMA engine

```
void (*bmdma_setup) (struct ata_queued_cmd *qc);  
void (*bmdma_start) (struct ata_queued_cmd *qc);  
void (*bmdma_stop) (struct ata_port *ap);  
u8 (*bmdma_status) (struct ata_port *ap);
```

When setting up an IDE BMDMA transaction, these hooks arm (`->bmdma_setup`), fire (`->bmdma_start`), and halt (`->bmdma_stop`) the hardware's DMA engine. `->bmdma_status` is used to read the standard PCI IDE DMA Status register.

These hooks are typically either no-ops, or simply not implemented, in FIS-based drivers.

Most legacy IDE drivers use `ata_bmdma_setup()` for the `bmdma_setup()` hook. `ata_bmdma_setup()` will write the pointer to the PRD table to the IDE PRD Table Address register, enable DMA in the DMA Command register, and call `exec_command()` to begin the transfer.

Most legacy IDE drivers use `ata_bmdma_start()` for the `bmdma_start()` hook. `ata_bmdma_start()` will write the `ATA_DMA_START` flag to the DMA Command register.

Many legacy IDE drivers use `ata_bmdma_stop()` for the `bmdma_stop()` hook. `ata_bmdma_stop()` clears the `ATA_DMA_START` flag in the DMA command register.

Many legacy IDE drivers use `ata_bmdma_status()` as the `bmdma_status()` hook.

High-level taskfile hooks

```
void (*qc_prep) (struct ata_queued_cmd *qc);
int (*qc_issue) (struct ata_queued_cmd *qc);
```

Higher-level hooks, these two hooks can potentially supercede several of the above taskfile/DMA engine hooks. `->qc_prep` is called after the buffers have been DMA-mapped, and is typically used to populate the hardware's DMA scatter-gather table. Most drivers use the standard `ata_qc_prep()` helper function, but more advanced drivers roll their own.

`->qc_issue` is used to make a command active, once the hardware and S/G tables have been prepared. IDE BMDMA drivers use the helper function `ata_qc_issue_prot()` for taskfile protocol-based dispatch. More advanced drivers implement their own `->qc_issue`.

`ata_qc_issue_prot()` calls `->tf_load()`, `->bmdma_setup()`, and `->bmdma_start()` as necessary to initiate a transfer.

Exception and probe handling (EH)

```
void (*eng_timeout) (struct ata_port *ap);
void (*phy_reset) (struct ata_port *ap);
```

Deprecated. Use `->error_handler()` instead.

```
void (*freeze) (struct ata_port *ap);
void (*thaw) (struct ata_port *ap);
```

`ata_port_freeze()` is called when HSM violations or some other condition disrupts normal operation of the port. A frozen port is not allowed to perform any operation until the port is thawed, which usually follows a successful reset.

The optional `->freeze()` callback can be used for freezing the port hardware-wise (e.g. mask interrupt and stop DMA engine). If a port cannot be frozen hardware-wise, the interrupt handler must ack and clear interrupts unconditionally while the port is frozen.

The optional `->thaw()` callback is called to perform the opposite of `->freeze()`: prepare the port for normal operation once again. Unmask interrupts, start DMA engine, etc.

```
void (*error_handler) (struct ata_port *ap);
```

`->error_handler()` is a driver's hook into probe, hotplug, and recovery and other exceptional conditions. The primary responsibility of an implementation is to call `ata_do_eh()` or `ata_bmdma_drive_eh()` with a set of EH hooks as arguments:

'prereset' hook (may be NULL) is called during an EH reset, before any other actions are taken.

'postreset' hook (may be NULL) is called after the EH reset is performed. Based on existing conditions, severity of the problem, and hardware capabilities,

Either 'softreset' (may be NULL) or 'hardreset' (may be NULL) will be called to perform the low-level EH reset.

```
void (*post_internal_cmd) (struct ata_queued_cmd *qc);
```

Perform any hardware-specific actions necessary to finish processing after executing a probe-time or EH-time command via `ata_exec_internal()`.

Hardware interrupt handling

```
irqreturn_t (*irq_handler)(int, void *, struct pt_regs *);
void (*irq_clear) (struct ata_port *);
```

->`irq_handler` is the interrupt handling routine registered with the system, by libata. ->`irq_clear` is called during probe just before the interrupt handler is registered, to be sure hardware is quiet.

The second argument, `dev_instance`, should be cast to a pointer to `struct ata_host_set`.

Most legacy IDE drivers use `ata_interrupt()` for the `irq_handler` hook, which scans all ports in the `host_set`, determines which queued command was active (if any), and calls `ata_host_intr(ap,qc)`.

Most legacy IDE drivers use `ata_bmdma_irq_clear()` for the `irq_clear()` hook, which simply clears the interrupt and error flags in the DMA status register.

SATA phy read/write

```
int (*scr_read) (struct ata_port *ap, unsigned int sc_reg,
                u32 *val);
int (*scr_write) (struct ata_port *ap, unsigned int sc_reg,
                 u32 val);
```

Read and write standard SATA phy registers. Currently only used if ->`phy_reset` hook called the `sata_phy_reset()` helper function. `sc_reg` is one of `SCR_STATUS`, `SCR_CONTROL`, `SCR_ERROR`, or `SCR_ACTIVE`.

Init and shutdown

```
int (*port_start) (struct ata_port *ap);
void (*port_stop) (struct ata_port *ap);
void (*host_stop) (struct ata_host_set *host_set);
```

->`port_start()` is called just after the data structures for each port are initialized. Typically this is used to alloc per-port DMA buffers / tables / rings, enable DMA engines, and similar tasks. Some drivers also use this entry point as a chance to allocate driver-private memory for `ap->private_data`.

Many drivers use `ata_port_start()` as this hook or call it from their own `port_start()` hooks. `ata_port_start()` allocates space for a legacy IDE PRD table and returns.

->`port_stop()` is called after ->`host_stop()`. It's sole function is to release DMA/memory resources, now that they are no longer actively being used. Many drivers also free driver-private data from port at this time.

Many drivers use `ata_port_stop()` as this hook, which frees the PRD table.

`->host_stop()` is called after all `->port_stop()` calls have completed. The hook must finalize hardware shutdown, release DMA and other resources, etc. This hook may be specified as `NULL`, in which case it is not called.

Chapter 3. Error handling

Table of Contents

[Origins of commands](#)

[How commands are issued](#)

[How commands are processed](#)

[How commands are completed](#)

[ata_scsi_error\(\)](#)

[Problems with the current EH](#)

This chapter describes how errors are handled under libata. Readers are advised to read SCSI EH (Documentation/scsi/scsi_eh.txt) and ATA exceptions doc first.

Origins of commands

In libata, a command is represented with struct `ata_queued_cmd` or `qc`. `qc`'s are preallocated during port initialization and repetitively used for command executions. Currently only one `qc` is allocated per port but yet-to-be-merged NCQ branch allocates one for each tag and maps each `qc` to NCQ tag 1-to-1.

libata commands can originate from two sources - libata itself and SCSI midlayer. libata internal commands are used for initialization and error handling. All normal blk requests and commands for SCSI emulation are passed as SCSI commands through `queuecommand` callback of SCSI host template.

How commands are issued

Internal commands

First, `qc` is allocated and initialized using `ata_qc_new_init()`. Although `ata_qc_new_init()` doesn't implement any wait or retry mechanism when `qc` is not available, internal commands are currently issued only during initialization and error recovery, so no other command is active and allocation is guaranteed to succeed.

Once allocated `qc`'s taskfile is initialized for the command to be executed. `qc` currently has two mechanisms to notify completion. One is via `qc->complete_fn()` callback and the other is completion `qc->waiting`. `qc->complete_fn()` callback is the asynchronous path used by normal SCSI translated commands and `qc->waiting` is the synchronous (issuer sleeps in process context) path used by internal commands.

Once initialization is complete, `host_set` lock is acquired and the `qc` is issued.

SCSI commands

All libata drivers use `ata_scsi_queuecmd()` as `host->queuecommand` callback. `cmds` can either be

simulated or translated. No qc is involved in processing a simulated scmd. The result is computed right away and the scmd is completed.

For a translated scmd, `ata_qc_new_init()` is invoked to allocate a qc and the scmd is translated into the qc. SCSI midlayer's completion notification function pointer is stored into `qc->scsidone`.

`qc->complete_fn()` callback is used for completion notification. ATA commands use `ata_scsi_qc_complete()` while ATAPI commands use `atapi_qc_complete()`. Both functions end up calling `qc->scsidone` to notify upper layer when the qc is finished. After translation is completed, the qc is issued with `ata_qc_issue()`.

Note that SCSI midlayer invokes `hostt->queuecommand` while holding `host_set` lock, so all above occur while holding `host_set` lock.

How commands are processed

Depending on which protocol and which controller are used, commands are processed differently. For the purpose of discussion, a controller which uses taskfile interface and all standard callbacks is assumed.

Currently 6 ATA command protocols are used. They can be sorted into the following four categories according to how they are processed.

ATA NO DATA or DMA

`ATA_PROT_NODATA` and `ATA_PROT_DMA` fall into this category. These types of commands don't require any software intervention once issued. Device will raise interrupt on completion.

ATA PIO

`ATA_PROT_PIO` is in this category. libata currently implements PIO with polling. `ATA_NIEN` bit is set to turn off interrupt and `pio_task` on `ata_wq` performs polling and IO.

ATAPI NODATA or DMA

`ATA_PROT_ATAPI_NODATA` and `ATA_PROT_ATAPI_DMA` are in this category. `packet_task` is used to poll BSY bit after issuing `PACKET` command. Once BSY is turned off by the device, `packet_task` transfers CDB and hands off processing to interrupt handler.

ATAPI PIO

`ATA_PROT_ATAPI` is in this category. `ATA_NIEN` bit is set and, as in ATAPI NODATA or DMA, `packet_task` submits cdb. However, after submitting cdb, further processing (data transfer) is handed off to `pio_task`.

How commands are completed

Once issued, all qc's are either completed with `ata_qc_complete()` or time out. For commands which are handled by interrupts, `ata_host_intr()` invokes `ata_qc_complete()`, and, for PIO tasks, `pio_task` invokes `ata_qc_complete()`. In error cases, `packet_task` may also complete commands.

`ata_qc_complete()` does the following.

1. DMA memory is unmapped.
2. `ATA_QCFLAG_ACTIVE` is cleared from `qc->flags`.
3. `qc->complete_fn()` callback is invoked. If the return value of the callback is not zero. Completion is short circuited and `ata_qc_complete()` returns.
4. `__ata_qc_complete()` is called, which does
 - a. `qc->flags` is cleared to zero.
 - b. `ap->active_tag` and `qc->tag` are poisoned.
 - c. `qc->waiting` is cleared & completed (in that order).
 - d. `qc` is deallocated by clearing appropriate bit in `ap->qactive`.

So, it basically notifies upper layer and deallocates `qc`. One exception is short-circuit path in #3 which is used by `atapi_qc_complete()`.

For all non-ATAPI commands, whether it fails or not, almost the same code path is taken and very little error handling takes place. A `qc` is completed with success status if it succeeded, with failed status otherwise.

However, failed ATAPI commands require more handling as REQUEST SENSE is needed to acquire sense data. If an ATAPI command fails, `ata_qc_complete()` is invoked with error status, which in turn invokes `atapi_qc_complete()` via `qc->complete_fn()` callback.

This makes `atapi_qc_complete()` set `scmd->result` to `SAM_STAT_CHECK_CONDITION`, complete the `scmd` and return 1. As the sense data is empty but `scmd->result` is `CHECK_CONDITION`, SCSI midlayer will invoke EH for the `scmd`, and returning 1 makes `ata_qc_complete()` to return without deallocating the `qc`. This leads us to `ata_scsi_error()` with partially completed `qc`.

`ata_scsi_error()`

`ata_scsi_error()` is the current `transport->eh_strategy_handler()` for libata. As discussed above, this will be entered in two cases - timeout and ATAPI error completion. This function calls low level libata driver's `eng_timeout()` callback, the standard callback for which is `ata_eng_timeout()`. It checks if a `qc` is active and calls `ata_qc_timeout()` on the `qc` if so. Actual error handling occurs in `ata_qc_timeout()`.

If EH is invoked for timeout, `ata_qc_timeout()` stops BMDMA and completes the `qc`. Note that as we're currently in EH, we cannot call `scsi_done`. As described in SCSI EH doc, a recovered `scmd` should be either retried with `scsi_queue_insert()` or finished with `scsi_finish_command()`. Here, we override `qc->scsidone` with `scsi_finish_command()` and calls `ata_qc_complete()`.

If EH is invoked due to a failed ATAPI `qc`, the `qc` here is completed but not deallocated. The purpose of this half-completion is to use the `qc` as place holder to make EH code reach this place. This is a bit hackish, but it works.

Once control reaches here, the `qc` is deallocated by invoking `__ata_qc_complete()` explicitly. Then,

internal qc for REQUEST SENSE is issued. Once sense data is acquired, scmd is finished by directly invoking `scsi_finish_command()` on the scmd. Note that as we already have completed and deallocated the qc which was associated with the scmd, we don't need to/cannot call `ata_qc_complete()` again.

Problems with the current EH

- Error representation is too crude. Currently any and all error conditions are represented with ATA STATUS and ERROR registers. Errors which aren't ATA device errors are treated as ATA device errors by setting ATA_ERR bit. Better error descriptor which can properly represent ATA and other errors/exceptions is needed.
- When handling timeouts, no action is taken to make device forget about the timed out command and ready for new commands.
- EH handling via `ata_scsi_error()` is not properly protected from usual command processing. On EH entrance, the device is not in quiescent state. Timed out commands may succeed or fail any time. `pio_task` and `atapi_task` may still be running.
- Too weak error recovery. Devices / controllers causing HSM mismatch errors and other errors quite often require reset to return to known state. Also, advanced error handling is necessary to support features like NCQ and hotplug.
- ATA errors are directly handled in the interrupt handler and PIO errors in `pio_task`. This is problematic for advanced error handling for the following reasons.

First, advanced error handling often requires context and internal qc execution.

Second, even a simple failure (say, CRC error) needs information gathering and could trigger complex error handling (say, resetting & reconfiguring). Having multiple code paths to gather information, enter EH and trigger actions makes life painful.

Third, scattered EH code makes implementing low level drivers difficult. Low level drivers override libata callbacks. If EH is scattered over several places, each affected callbacks should perform its part of error handling. This can be error prone and painful.

Chapter 4. libata Library

Table of Contents

[ata_link_next](#) — link iteration helper
[ata_dev_next](#) — device iteration helper
[atapi_cmd_type](#) — Determine ATAPI command type from SCSI opcode
[ata_tf_to_fis](#) — Convert ATA taskfile to SATA FIS structure
[ata_tf_from_fis](#) — Convert SATA FIS to ATA taskfile
[ata_pack_xfermask](#) — Pack pio, mwdma and udma masks into `xfer_mask`
[ata_unpack_xfermask](#) — Unpack `xfer_mask` into pio, mwdma and udma masks
[ata_xfer_mask2mode](#) — Find matching XFER_* for the given `xfer_mask`
[ata_xfer_mode2mask](#) — Find matching `xfer_mask` for XFER_*
[ata_xfer_mode2shift](#) — Find matching `xfer_shift` for XFER_*
[ata_mode_string](#) — convert `xfer_mask` to string

[ata_dev_classify](#) — determine device type based on ATA-spec signature
[ata_id_string](#) — Convert IDENTIFY DEVICE page into string
[ata_id_c_string](#) — Convert IDENTIFY DEVICE page into C string
[ata_id_xfermask](#) — Compute xfermask from the given IDENTIFY data
[ata_pio_queue_task](#) — Queue port_task
[ata_pio_need_iordy](#) — check if iordy needed
[ata_do_dev_read_id](#) — default ID read method
[ata_cable_40wire](#) — return 40 wire cable type
[ata_cable_80wire](#) — return 80 wire cable type
[ata_cable_unknown](#) — return unknown PATA cable.
[ata_cable_ignore](#) — return ignored PATA cable.
[ata_cable_sata](#) — return SATA cable type
[ata_port_probe](#) — Mark port as enabled
[ata_dev_pair](#) — return other device on cable
[ata_port_disable](#) — Disable port.
[sata_set_spd](#) — set SATA spd according to spd limit
[ata_timing_cycle2mode](#) — find xfer mode for the specified cycle duration
[ata_do_set_mode](#) — Program timings and issue SET FEATURES - XFER
[ata_wait_after_reset](#) — wait for link to become ready after reset
[sata_link_debounce](#) — debounce SATA phy status
[sata_link_resume](#) — resume SATA link
[ata_std_prereset](#) — prepare for reset
[sata_link_hardreset](#) — reset link via SATA phy reset
[sata_std_hardreset](#) — COMRESET w/o waiting or classification
[ata_std_postreset](#) — standard postreset callback
[ata_std_qc_defer](#) — Check whether a qc needs to be deferred
[ata_sg_init](#) — Associate command with scatter-gather table.
[ata_qc_complete](#) — Complete an active ATA command
[ata_qc_complete_multiple](#) — Complete multiple qcs successfully
[sata_scr_valid](#) — test whether SCRs are accessible
[sata_scr_read](#) — read SCR register of the specified port
[sata_scr_write](#) — write SCR register of the specified port
[sata_scr_write_flush](#) — write SCR register of the specified port and flush
[ata_link_online](#) — test whether the given link is online
[ata_link_offline](#) — test whether the given link is offline
[ata_host_suspend](#) — suspend host
[ata_host_resume](#) — resume host
[ata_port_start](#) — Set port up for dma.
[ata_host_alloc](#) — allocate and init basic ATA host resources
[ata_host_alloc_pinfo](#) — alloc host and init with port_info array
[ata_slave_link_init](#) — initialize slave link
[ata_host_start](#) — start and freeze ports of an ATA host
[ata_host_init](#) — Initialize a host struct
[ata_host_register](#) — register initialized ATA host
[ata_host_activate](#) — start host, request IRQ and register it
[ata_host_detach](#) — Detach all ports of an ATA host
[ata_pci_remove_one](#) — PCI layer callback for device removal
[ata_wait_register](#) — wait until register value changes

Name

ata_link_next — link iteration helper

Synopsis