libATA Developer's Guide

Jeff Garzik

Table of Contents

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 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 contollers) 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

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 hostt->queuecommand callback. scmds 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 clared 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 claread & 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 transportt->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

Name

ata_link_next — link iteration helper

Synopsis