

```

libata.tmpl.txt
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
    "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="libataDevGuide">
  <bookinfo>
    <title>libATA Developer's Guide</title>

    <authorgroup>
      <author>
        <firstname>Jeff</firstname>
        <surname>Garzik</surname>
      </author>
    </authorgroup>

    <copyright>
      <year>2003-2006</year>
      <holder>Jeff Garzik</holder>
    </copyright>

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

      <para>
        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.
      </para>

    </legalnotice>
  </bookinfo>

  <toc></toc>

  <chapter id="libataIntroduction">
    <title>Introduction</title>
    <para>
      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&ltamplt;-&gt;ATA translation
      for ATA devices according to the T10 SAT specification.
    </para>
  </chapter>

```

<para>

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

</para>

</chapter>

<chapter id="libataDriverApi">

<title>libata Driver API</title>

<para>

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.

</para>

<para>

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.

</para>

<sect1>

<title>struct ata_port_operations</title>

<sect2><title>Disable ATA port</title>

<programlisting>

void (*port_disable) (struct ata_port *);

</programlisting>

<para>

Called from ata_bus_probe() error path, 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.

</para>

<para>

Called from ata_bus_probe() on a failed probe.
Called from ata_scsi_release().

</para>

</sect2>

<sect2><title>Post-IDENTIFY device configuration</title>

<programlisting>

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

</programlisting>

<para>

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.

</para>

<para>

This entry may be specified as NULL in ata_port_operations.

</para>

</sect2>

<sect2><title>Set PIO/DMA mode</title>

<programlisting>

```
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);
```

</programlisting>

<para>

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.

</para>

<para>

`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.

</para>

<para>

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

</para>

<para>

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

</para>

</sect2>

<sect2><title>Taskfile read/write</title>

<programlisting>

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

</programlisting>

<para>

`->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_sff_tf_load()` and `ata_sff_tf_read()` for these hooks.

</para>

</sect2>

libata.tmpl.txt

<sect2><title>PIO data read/write</title>

<programlisting>

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

</programlisting>

<para>

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_sff_data_xfer_noirq()`, `ata_sff_data_xfer()`, or `ata_sff_data_xfer32()`.

</para>

</sect2>

<sect2><title>ATA command execute</title>

<programlisting>

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

</programlisting>

<para>

causes an ATA command, previously loaded with `->tf_load()`, to be initiated in hardware.

Most drivers for taskfile-based hardware use `ata_sff_exec_command()` for this hook.

</para>

</sect2>

<sect2><title>Per-cmd ATAPI DMA capabilities filter</title>

<programlisting>

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

</programlisting>

<para>

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.

</para>

<para>

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

</para>

</sect2>

<sect2><title>Read specific ATA shadow registers</title>

<programlisting>

```
u8 (*sff_check_status) (struct ata_port *ap);
```

```
u8 (*sff_check_altstatus) (struct ata_port *ap);
```

</programlisting>

<para>

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.

libata.tmpl.txt

Most drivers for taskfile-based hardware use
ata_sff_check_status() for this hook.
</para>

</sect2>

<sect2><title>Write specific ATA shadow register</title>
<programlisting>

```
void (*sff_set_devctl)(struct ata_port *ap, u8 ctl);
```

</programlisting>

<para>

Write the device control ATA shadow register to the hardware.
Most drivers don't need to define this.
</para>

</sect2>

<sect2><title>Select ATA device on bus</title>
<programlisting>

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

</programlisting>

<para>

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.
</para>

<para>

Most drivers for taskfile-based hardware use
ata_sff_dev_select() for this hook.
</para>

</sect2>

<sect2><title>Private tuning method</title>
<programlisting>

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

</programlisting>

<para>

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.
</para>

<warning>

<para>

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

libata.tmpl.txt

mode_filter hook instead.

</para>

</warning>

</sect2>

<sect2><title>Control PCI IDE BMDMA engine</title>

<programlisting>

```
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);
```

</programlisting>

<para>

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.

</para>

<para>

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

</para>

<para>

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.

</para>

<para>

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.

</para>

<para>

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.

</para>

<para>

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

</para>

</sect2>

<sect2><title>High-level taskfile hooks</title>

<programlisting>

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

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

</programlisting>

<para>

Higher-level hooks, these two hooks can potentially supersede several of the above taskfile/DMA engine hooks. ->qc_prep is

libata.tmpl.txt

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.

</para>

<para>

`->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`.

</para>

<para>

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

</para>

</sect2>

<sect2><title>Exception and probe handling (EH)</title>

<programlisting>

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

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

</programlisting>

<para>

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

</para>

<programlisting>

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

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

</programlisting>

<para>

`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.

</para>

<para>

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.

</para>

<para>

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.

</para>

<programlisting>

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

</programlisting>

<para>
->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:

</para>

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

</para>

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

</para>

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

</para>

<programlisting>
void (*post_internal_cmd) (struct ata_queued_cmd *qc);

</programlisting>

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

</para>

</sect2>

<sect2><title>Hardware interrupt handling</title>

<programlisting>
irqreturn_t (*irq_handler) (int, void *, struct pt_regs *);
void (*irq_clear) (struct ata_port *);

</programlisting>

<para>
->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.

</para>

<para>

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

</para>

<para>

Most legacy IDE drivers use ata_sff_interrupt() for the irq_handler hook, which scans all ports in the host_set, determines which queued command was active (if any), and calls

libata.tmpl.txt

ata_sff_host_intr(ap, qc).

</para>

<para>

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

</para>

</sect2>

<sect2><title>SATA phy read/write</title>

<programlisting>

```
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);
```

</programlisting>

<para>

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.

</para>

</sect2>

<sect2><title>Init and shutdown</title>

<programlisting>

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

</programlisting>

<para>

->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.

</para>

<para>

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.

</para>

<para>

->port_stop() is called after ->host_stop(). Its 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.

</para>

<para>

->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.

</para>

</sect2>

</sect1>

</chapter>

<chapter id="libataEH">

<title>Error handling</title>

<para>

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.

<sect1><title>Origins of commands</title>

<para>

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.

</para>

<para>

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.

</para>

</sect1>

<sect1><title>How commands are issued</title>

<variablelist>

<varlistentry><term>Internal commands</term>

<listitem>

<para>

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.

</para>

<para>

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.

</para>

<para>

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

</para>

</listitem>

</varlistentry>

<varlistentry><term>SCSI commands</term>

<listitem>

<para>

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.

</para>

<para>

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.

</para>

<para>

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().

</para>

<para>

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

</para>

</listitem>

</varlistentry>

</variablelist>

</sect1>

<sect1><title>How commands are processed</title>

<para>

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.

</para>

<para>

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

</para>

<variablelist>

<varlistentry><term>ATA NO DATA or DMA</term>

```

<listitem>
<para>
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.
</para>
</listitem>
</varlistentry>

<varlistentry><term>ATA PIO</term>
<listitem>
<para>
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.
</para>
</listitem>
</varlistentry>

<varlistentry><term>ATAPI NODATA or DMA</term>
<listitem>
<para>
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.
</para>
</listitem>
</varlistentry>

<varlistentry><term>ATAPI PIO</term>
<listitem>
<para>
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.
</para>
</listitem>
</varlistentry>
</variablelist>
</sect1>

<sect1><title>How commands are completed</title>
<para>
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.
</para>
<para>
ata_qc_complete() does the following.

```

</para>

<orderedlist>

<listitem>

<para>

DMA memory is unmapped.

</para>

</listitem>

<listitem>

<para>

ATA_QCFLAG_ACTIVE is cleared from qc->flags.

</para>

</listitem>

<listitem>

<para>

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.

</para>

</listitem>

<listitem>

<para>

__ata_qc_complete() is called, which does

<orderedlist>

<listitem>

<para>

qc->flags is cleared to zero.

</para>

</listitem>

<listitem>

<para>

ap->active_tag and qc->tag are poisoned.

</para>

</listitem>

<listitem>

<para>

qc->waiting is cleared & completed (in that order).

</para>

</listitem>

<listitem>

<para>

qc is deallocated by clearing appropriate bit in ap->qactive.

</para>

</listitem>

</orderedlist>

</para>

</listitem>

</orderedlist>

<para>

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

</para>

<para>

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.

</para>

<para>

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.

</para>

<para>

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.

</para>

</sect1>

<sect1><title>ata_scsi_error()</title>

<para>

`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()`.

</para>

<para>

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()`.

</para>

<para>

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.

</para>

<para>

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.

</para>

</sect1>

<sect1><title>Problems with the current EH</title>

<itemizedlist>

<listitem>

<para>

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.

</para>

</listitem>

<listitem>

<para>

When handling timeouts, no action is taken to make device forget about the timed out command and ready for new commands.

</para>

</listitem>

<listitem>

<para>

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.

</para>

</listitem>

<listitem>

<para>

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.

</para>

</listitem>

<listitem>

<para>

ATA errors are directly handled in the interrupt handler and

libata.tmpl.txt

PIO errors in pio_task. This is problematic for advanced error handling for the following reasons.

</para>

<para>

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

</para>

<para>

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.

</para>

<para>

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.

</para>

</listitem>

</itemizedlist>

</sect1>

</chapter>

<chapter id="libataExt">

<title>libata Library</title>

!Edrivers/ata/libata-core.c

</chapter>

<chapter id="libataInt">

<title>libata Core Internals</title>

!Idrivers/ata/libata-core.c

</chapter>

<chapter id="libataScsiInt">

<title>libata SCSI translation/emulation</title>

!Edrivers/ata/libata-scsi.c

!Idrivers/ata/libata-scsi.c

</chapter>

<chapter id="ataExceptions">

<title>ATA errors and exceptions</title>

<para>

This chapter tries to identify what error/exception conditions exist for ATA/ATAPI devices and describe how they should be handled in implementation-neutral way.

</para>

<para>

The term 'error' is used to describe conditions where either an explicit error condition is reported from device or a command has timed out.

</para>

<para>

The term 'exception' is either used to describe exceptional conditions which are not errors (say, power or hotplug events), or to describe both errors and non-error exceptional conditions. Where explicit distinction between error and exception is necessary, the term 'non-error exception' is used.

</para>

<sect1 id="excat">

<title>Exception categories</title>

<para>

Exceptions are described primarily with respect to legacy taskfile + bus master IDE interface. If a controller provides other better mechanism for error reporting, mapping those into categories described below shouldn't be difficult.

</para>

<para>

In the following sections, two recovery actions - reset and reconfiguring transport - are mentioned. These are described further in <xref linkend="exrec"/>.

</para>

<sect2 id="excathSMviolation">

<title>HSM violation</title>

<para>

This error is indicated when STATUS value doesn't match HSM requirement during issuing or execution any ATA/ATAPI command.

</para>

<itemizedlist>

<title>Examples</title>

<listitem>

<para>

trying
ATA_STATUS doesn't contain !BSY && DRDY && !DRQ while

to issue a command.

</para>

</listitem>

<listitem>

<para>

!BSY && !DRQ during PIO data transfer.

</para>

</listitem>

<listitem>

<para>

DRQ on command completion.

</para>

</listitem>

<listitem>

<para>
!BSY && ERR after CDB transfer starts but before the last byte of CDB is transferred. ATA/ATAPI standard states that "The device shall not terminate the PACKET command with an error before the last byte of the command packet has been written" in the error outputs description of PACKET command and the state diagram doesn't include such transitions.

</para>
</listitem>

</itemizedlist>

<para>
In these cases, HSM is violated and not much information regarding the error can be acquired from STATUS or ERROR register. IOW, this error can be anything - driver bug, faulty device, controller and/or cable.

</para>

<para>
As HSM is violated, reset is necessary to restore known state. Reconfiguring transport for lower speed might be helpful too as transmission errors sometimes cause this kind of errors.

</para>

</sect2>

<sect2 id="excatDevErr">

<title>ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)</title>

<para>
These are errors detected and reported by ATA/ATAPI devices indicating device problems. For this type of errors, STATUS and ERROR register values are valid and describe error condition. Note that some of ATA bus errors are detected by ATA/ATAPI devices and reported using the same mechanism as device errors. Those cases are described later in this section.

</para>

<para>
For ATA commands, this type of errors are indicated by !BSY && ERR during command execution and on completion.

</para>

<para>For ATAPI commands,</para>

<itemizedlist>

<listitem>

<para>
!BSY && ERR && ABRT right after issuing PACKET indicates that PACKET command is not supported and falls in this category.

</para>

</listitem>

```

<listitem>
<para>
!BSY && ERR(==CHK) && !ABRT after the last
byte of CDB is transferred indicates CHECK CONDITION and
doesn't fall in this category.
</para>
</listitem>

<listitem>
<para>
!BSY && ERR(==CHK) && ABRT after the last byte
of CDB is transferred *probably* indicates CHECK CONDITION and
doesn't fall in this category.
</para>
</listitem>

</itemizedlist>

<para>
Of errors detected as above, the followings are not ATA/ATAPI
device errors but ATA bus errors and should be handled
according to <xref linkend="excatATABusErr"/>.
</para>

<variablelist>

  <varlistentry>
    <term>CRC error during data transfer</term>
    <listitem>
      <para>
        This is indicated by ICRC bit in the ERROR register and
        means that corruption occurred during data transfer. Upto
        ATA/ATAPI-7, the standard specifies that this bit is only
        applicable to UDMA transfers but ATA/ATAPI-8 draft revision
        1f says that the bit may be applicable to multiword DMA and
        PIO.
      </para>
    </listitem>
  </varlistentry>

  <varlistentry>
    <term>ABRT error during data transfer or on completion</term>
    <listitem>
      <para>
        Upto ATA/ATAPI-7, the standard specifies that ABRT could be
        set on ICRC errors and on cases where a device is not able
        to complete a command. Combined with the fact that MWDMA
        and PIO transfer errors aren't allowed to use ICRC bit upto
        ATA/ATAPI-7, it seems to imply that ABRT bit alone could
        indicate tranfer errors.
      </para>
      <para>
        However, ATA/ATAPI-8 draft revision 1f removes the part
        that ICRC errors can turn on ABRT. So, this is kind of
        gray area. Some heuristics are needed here.
      </para>
    </listitem>
  </varlistentry>

```

```

</para>
</listitem>
</varlistentry>

```

```

</variablelist>

```

```

<para>
ATA/ATAPI device errors can be further categorized as follows.
</para>

```

```

<variablelist>

```

```

<varlistentry>
<term>Media errors</term>
<listitem>
<para>

```

```

This is indicated by UNC bit in the ERROR register. ATA
devices reports UNC error only after certain number of
retries cannot recover the data, so there's nothing much
else to do other than notifying upper layer.

```

```

</para>

```

```

<para>

```

```

READ and WRITE commands report CHS or LBA of the first
failed sector but ATA/ATAPI standard specifies that the
amount of transferred data on error completion is
indeterminate, so we cannot assume that sectors preceding
the failed sector have been transferred and thus cannot
complete those sectors successfully as SCSI does.

```

```

</para>

```

```

</listitem>

```

```

</varlistentry>

```

```

<varlistentry>

```

```

<term>Media changed / media change requested error</term>

```

```

<listitem>

```

```

<para>

```

```

<&lt;&lt;TODO: fill here&gt;&gt;

```

```

</para>

```

```

</listitem>

```

```

</varlistentry>

```

```

<varlistentry><term>Address error</term>

```

```

<listitem>

```

```

<para>

```

```

This is indicated by IDNF bit in the ERROR register.

```

```

Report to upper layer.

```

```

</para>

```

```

</listitem>

```

```

</varlistentry>

```

```

<varlistentry><term>Other errors</term>

```

```

<listitem>

```

```

<para>

```

```

This can be invalid command or parameter indicated by ABRT
ERROR bit or some other error condition. Note that ABRT
bit can indicate a lot of things including ICRC and Address

```

errors. Heuristics needed.

</para>

</listitem>

</varlistentry>

</variablelist>

<para>

Depending on commands, not all STATUS/ERROR bits are applicable. These non-applicable bits are marked with "na" in the output descriptions but upto ATA/ATAPI-7 no definition of "na" can be found. However, ATA/ATAPI-8 draft revision 1f describes "N/A" as follows.

</para>

<blockquote>

<variablelist>

<varlistentry><term>3.2.3.3a N/A</term>

<listitem>

<para>

A keyword the indicates a field has no defined value in this standard and should not be checked by the host or device. N/A fields should be cleared to zero.

</para>

</listitem>

</varlistentry>

</variablelist>

</blockquote>

<para>

So, it seems reasonable to assume that "na" bits are cleared to zero by devices and thus need no explicit masking.

</para>

</sect2>

<sect2 id="excatATAPIcc">

<title>ATAPI device CHECK CONDITION</title>

<para>

ATAPI device CHECK CONDITION error is indicated by set CHK bit (ERR bit) in the STATUS register after the last byte of CDB is transferred for a PACKET command. For this kind of errors, sense data should be acquired to gather information regarding the errors. REQUEST SENSE packet command should be used to acquire sense data.

</para>

<para>

Once sense data is acquired, this type of errors can be handled simily to other SCSI errors. Note that sense data may indicate ATA bus error (e.g. Sense Key 04h HARDWARE ERROR & ASC/ASCQ 47h/00h SCSI PARITY ERROR). In such cases, the error should be considered as an ATA bus error and handled according to <xref linkend="excatATABusErr"/>.

</para>

</sect2>

<sect2 id="excatNCQerr">

<title>ATA device error (NCQ)</title>

<para>

NCQ command error is indicated by cleared BSY and set ERR bit during NCQ command phase (one or more NCQ commands outstanding). Although STATUS and ERROR registers will contain valid values describing the error, READ LOG EXT is required to clear the error condition, determine which command has failed and acquire more information.

</para>

<para>

READ LOG EXT Log Page 10h reports which tag has failed and taskfile register values describing the error. With this information the failed command can be handled as a normal ATA command error as in <xref linkend="excatDevErr"/> and all other in-flight commands must be retried. Note that this retry should not be counted - it's likely that commands retried this way would have completed normally if it were not for the failed command.

</para>

<para>

Note that ATA bus errors can be reported as ATA device NCQ errors. This should be handled as described in <xref linkend="excatATABusErr"/>.

</para>

<para>

If READ LOG EXT Log Page 10h fails or reports NQ, we're thoroughly screwed. This condition should be treated according to <xref linkend="excatHSMviolation"/>.

</para>

</sect2>

<sect2 id="excatATABusErr">

<title>ATA bus error</title>

<para>

ATA bus error means that data corruption occurred during transmission over ATA bus (SATA or PATA). This type of errors can be indicated by

</para>

<itemizedlist>

<listitem>

<para>

ICRC or ABRT error as described in <xref linkend="excatDevErr"/>.

</para>

</listitem>

<listitem>

<para>

Controller-specific error completion with error information indicating transmission error.

</para>

</listitem>

<listitem>

<para>

On some controllers, command timeout. In this case, there may be a mechanism to determine that the timeout is due to transmission error.

</para>

</listitem>

<listitem>

<para>

Unknown/random errors, timeouts and all sorts of weirdities.

</para>

</listitem>

</itemizedlist>

<para>

As described above, transmission errors can cause wide variety of symptoms ranging from device ICRC error to random device lockup, and, for many cases, there is no way to tell if an error condition is due to transmission error or not; therefore, it's necessary to employ some kind of heuristic when dealing with errors and timeouts. For example, encountering repetitive ABRT errors for known supported command is likely to indicate ATA bus error.

</para>

<para>

Once it's determined that ATA bus errors have possibly occurred, lowering ATA bus transmission speed is one of actions which may alleviate the problem. See <xref linkend="exrecReconf"/> for more information.

</para>

</sect2>

<sect2 id="excatPCIbusErr">

<title>PCI bus error</title>

<para>

Data corruption or other failures during transmission over PCI (or other system bus). For standard BMDMA, this is indicated by Error bit in the BMDMA Status register. This type of errors must be logged as it indicates something is very wrong with the system. Resetting host controller is recommended.

</para>

</sect2>

<sect2 id="excatLateCompletion">
 <title>Late completion</title>

<para>

This occurs when timeout occurs and the timeout handler finds out that the timed out command has completed successfully or with error. This is usually caused by lost interrupts. This type of errors must be logged. Resetting host controller is recommended.

</para>

</sect2>

<sect2 id="excatUnknown">
 <title>Unknown error (timeout)</title>

<para>

This is when timeout occurs and the command is still processing or the host and device are in unknown state. When this occurs, HSM could be in any valid or invalid state. To bring the device to known state and make it forget about the timed out command, resetting is necessary. The timed out command may be retried.

</para>

<para>

Timeouts can also be caused by transmission errors. Refer to <xref linkend="excatATABusErr"/> for more details.

</para>

</sect2>

<sect2 id="excatHoplugPM">
 <title>Hotplug and power management exceptions</title>

<para>

<<TODO: fill here>>

</para>

</sect2>

</sect1>

<sect1 id="exrec">
 <title>EH recovery actions</title>

<para>

This section discusses several important recovery actions.

</para>

<sect2 id="exrecClr">
 <title>Clearing error condition</title>

<para>

libata.tmpl.txt

Many controllers require its error registers to be cleared by error handler. Different controllers may have different requirements.

</para>

<para>

For SATA, it's strongly recommended to clear at least SError register during error handling.

</para>

</sect2>

<sect2 id="exrecRst">

<title>Reset</title>

<para>

During EH, resetting is necessary in the following cases.

</para>

<itemizedlist>

<listitem>

<para>

HSM is in unknown or invalid state

</para>

</listitem>

<listitem>

<para>

HBA is in unknown or invalid state

</para>

</listitem>

<listitem>

<para>

EH needs to make HBA/device forget about in-flight commands

</para>

</listitem>

<listitem>

<para>

HBA/device behaves weirdly

</para>

</listitem>

</itemizedlist>

<para>

Resetting during EH might be a good idea regardless of error condition to improve EH robustness. Whether to reset both or either one of HBA and device depends on situation but the following scheme is recommended.

</para>

<itemizedlist>

<listitem>

libata.tmpl.txt

<para>

When it's known that HBA is in ready state but ATA/ATAPI device is in unknown state, reset only device.

</para>

</listitem>

<listitem>

<para>

If HBA is in unknown state, reset both HBA and device.

</para>

</listitem>

</itemizedlist>

<para>

HBA resetting is implementation specific. For a controller complying to taskfile/BMDMA PCI IDE, stopping active DMA transaction may be sufficient iff BMDMA state is the only HBA context. But even mostly taskfile/BMDMA PCI IDE complying controllers may have implementation specific requirements and mechanism to reset themselves. This must be addressed by specific drivers.

</para>

<para>

OTOH, ATA/ATAPI standard describes in detail ways to reset ATA/ATAPI devices.

</para>

<variablelist>

<varlistentry><term>PATA hardware reset</term>

<listitem>

<para>

This is hardware initiated device reset signalled with asserted PATA RESET- signal. There is no standard way to initiate hardware reset from software although some hardware provides registers that allow driver to directly tweak the RESET- signal.

</para>

</listitem>

</varlistentry>

<varlistentry><term>Software reset</term>

<listitem>

<para>

This is achieved by turning CONTROL SRST bit on for at least 5us. Both PATA and SATA support it but, in case of SATA, this may require controller-specific support as the second Register FIS to clear SRST should be transmitted while BSY bit is still set. Note that on PATA, this resets both master and slave devices on a channel.

</para>

</listitem>

</varlistentry>

libata.tmpl.txt

<varlistentry><term>EXECUTE DEVICE DIAGNOSTIC command</term>

<listitem>

<para>

Although ATA/ATAPI standard doesn't describe exactly, EDD implies some level of resetting, possibly similar level with software reset. Host-side EDD protocol can be handled with normal command processing and most SATA controllers should be able to handle EDD's just like other commands. As in software reset, EDD affects both devices on a PATA bus.

</para>

<para>

Although EDD does reset devices, this doesn't suit error handling as EDD cannot be issued while BSY is set and it's unclear how it will act when device is in unknown/weird state.

</para>

</listitem>

</varlistentry>

<varlistentry><term>ATAPI DEVICE RESET command</term>

<listitem>

<para>

This is very similar to software reset except that reset can be restricted to the selected device without affecting the other device sharing the cable.

</para>

</listitem>

</varlistentry>

<varlistentry><term>SATA phy reset</term>

<listitem>

<para>

This is the preferred way of resetting a SATA device. In effect, it's identical to PATA hardware reset. Note that this can be done with the standard SCR Control register. As such, it's usually easier to implement than software reset.

</para>

</listitem>

</varlistentry>

</variablelist>

<para>

One more thing to consider when resetting devices is that resetting clears certain configuration parameters and they need to be set to their previous or newly adjusted values after reset.

</para>

<para>

Parameters affected are.

</para>

<itemizedlist>

<listitem>

<para>

CHS set up with INITIALIZE DEVICE PARAMETERS (seldomly used)

</para>

</listitem>

<listitem>

<para>

Parameters set with SET FEATURES including transfer mode setting

</para>

</listitem>

<listitem>

<para>

Block count set with SET MULTIPLE MODE

</para>

</listitem>

<listitem>

<para>

Other parameters (SET MAX, MEDIA LOCK...)

</para>

</listitem>

</itemizedlist>

<para>

ATA/ATAPI standard specifies that some parameters must be maintained across hardware or software reset, but doesn't strictly specify all of them. Always reconfiguring needed parameters after reset is required for robustness. Note that this also applies when resuming from deep sleep (power-off).

</para>

<para>

Also, ATA/ATAPI standard requires that IDENTIFY DEVICE / IDENTIFY PACKET DEVICE is issued after any configuration parameter is updated or a hardware reset and the result used for further operation. OS driver is required to implement revalidation mechanism to support this.

</para>

</sect2>

<sect2 id="exrecReconf">

<title>Reconfigure transport</title>

<para>

For both PATA and SATA, a lot of corners are cut for cheap connectors, cables or controllers and it's quite common to see high transmission error rate. This can be mitigated by lowering transmission speed.

</para>

<para>

The following is a possible scheme Jeff Garzik suggested.

</para>

<blockquote>

<para>

If more than \$N (3?) transmission errors happen in 15 minutes,

</para>

<itemizedlist>

<listitem>

<para>

if SATA, decrease SATA PHY speed. if speed cannot be decreased,

</para>

</listitem>

<listitem>

<para>

decrease UDMA xfer speed. if at UDMA0, switch to PIO4,

</para>

</listitem>

<listitem>

<para>

decrease PIO xfer speed. if at PIO3, complain, but continue

</para>

</listitem>

</itemizedlist>

</blockquote>

</sect2>

</sect1>

</chapter>

<chapter id="PiixInt">

<title>ata_piix Internals</title>

!Idrivers/ata/ata_piix.c

</chapter>

<chapter id="SILInt">

<title>sata_sil Internals</title>

!Idrivers/ata/sata_sil.c

</chapter>

<chapter id="libataThanks">

<title>Thanks</title>

<para>

The bulk of the ATA knowledge comes thanks to long conversations with Andre Hedrick (www.linux-ide.org), and long hours pondering the ATA and SCSI specifications.

</para>

<para>

Thanks to Alan Cox for pointing out similarities between SATA and SCSI, and in general for motivation to hack on libata.

</para>

<para>

libata's device detection

libata.tmpl.txt

method, ata_pio_devchk, and in general all the early probing was based on extensive study of Hale Landis's probe/reset code in his ATADRV driver (www.ata-atapi.com).

</para>

</chapter>

</book>