libATA Developer's Guide

Jeff Garzik

libATA Developer's Guide
by 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

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 *ap);
```

Hooks called prior to the issue of SET FEATURES - XFER MODE command. dev->pio_mode is guaranteed to be valid when ->set_piomode() is called, and dev->dma_mode is guaranteed to be valid when ->set_dmamode() is called. ->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.

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

Reads the Status/AltStatus/Error 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().

Reset ATA bus

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

The very first step in the probe phase. Actions vary depending on the bus type, typically. After waking up the device and probing for device presence (PATA and SATA), typically a soft reset (SRST) will be performed. Drivers typically use the helper functions ata_bus_reset() or sata_phy_reset() for this hook. Many SATA drivers use sata_phy_reset() or call it from within their own phy_reset() functions.

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.

Timeout (error) handling

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

This is a high level error handling function, called from the error handling thread, when a command times out. Most newer hardware will implement its own error handling code here. IDE BMDMA drivers may use the helper function ata_eng_timeout().

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

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

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 pre-allocated 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 hostt->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 mul-

tiple 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

ata_tf_load

Name

`ata_tf_load` — send taskfile registers to host controller

Synopsis

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

Arguments

`ap`

Port to which output is sent

`tf`

ATA taskfile register set

Description

Outputs ATA taskfile to standard ATA host controller using MMIO or PIO as indicated by the ATA_FLAG_MMIO flag. Writes the control, feature, nsect, lbal, lbam, and lbah registers. Optionally (ATA_TFLAG_LBA48) writes hob_feature, hob_nsect, hob_lbal, hob_lbam, and hob_lbah.

This function waits for idle (!BUSY and !DRQ) after writing registers. If the control register has a new value, this function also waits for idle after writing control and before writing the remaining registers.

May be used as the `tf_load` entry in ata_port_operations.

LOCKING

Inherited from caller.

ata_exec_command

Name

`ata_exec_command` — issue ATA command to host controller

Synopsis

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

Arguments

ap

port to which command is being issued

tf

ATA taskfile register set

Description

Issues PIO/MMIO write to ATA command register, with proper synchronization with interrupt handler / other threads.

LOCKING

spin_lock_irqsave(host_set lock)

ata_tf_read

Name

`ata_tf_read` — input device's ATA taskfile shadow registers

Synopsis

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

Arguments

ap

Port from which input is read

tf

ATA taskfile register set for storing input

Description

Reads ATA taskfile registers for currently-selected device into *tf*.

Reads nsect, lbal, lbam, lbah, and device. If ATA_TFLAG_LBA48 is set, also reads the hob registers.

May be used as the `tf_read` entry in ata_port_operations.

LOCKING

Inherited from caller.

ata_check_status

Name

`ata_check_status` — Read device status reg & clear interrupt

Synopsis

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

Arguments

ap

port where the device is

Description

Reads ATA taskfile status register for currently-selected device and return its value. This also clears pending interrupts from this device

May be used as the `check_status` entry in ata_port_operations.

LOCKING

Inherited from caller.

ata_altstatus

Name

`ata_altstatus` — Read device alternate status reg

Synopsis

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

Arguments

ap

port where the device is

Description

Reads ATA taskfile alternate status register for currently-selected device and return its value.

Note

may NOT be used as the `check_altstatus` entry in ata_port_operations.

LOCKING

Inherited from caller.

ata_chk_err

Name

`ata_chk_err` — Read device error reg

Synopsis

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

Arguments

ap

port where the device is

Description

Reads ATA taskfile error register for currently-selected device and return its value.

Note

may NOT be used as the `check_err` entry in ata_port_operations.

LOCKING

Inherited from caller.

ata_tf_to_fis

Name

`ata_tf_to_fis` — Convert ATA taskfile to SATA FIS structure

Synopsis

```
void ata_tf_to_fis (struct ata_taskfile * tf, u8 * fis, u8 pmp);
```

Arguments

`tf`
: Taskfile to convert

`fis`
: Buffer into which data will output

`pmp`
: Port multiplier port

Description

Converts a standard ATA taskfile to a Serial ATA FIS structure (Register - Host to Device).

LOCKING

Inherited from caller.

ata_tf_from_fis

Name

`ata_tf_from_fis` — Convert SATA FIS to ATA taskfile

Synopsis

```
void ata_tf_from_fis (u8 * fis, struct ata_taskfile * tf);
```

Arguments

`fis`
Buffer from which data will be input

`tf`
Taskfile to output

Description

Converts a standard ATA taskfile to a Serial ATA FIS structure (Register - Host to Device).

LOCKING

Inherited from caller.

ata_dev_classify

Name

`ata_dev_classify` — determine device type based on ATA-spec signature

Synopsis

```
unsigned int ata_dev_classify (struct ata_taskfile * tf);
```

Arguments

tf

ATA taskfile register set for device to be identified

Description

Determine from taskfile register contents whether a device is ATA or ATAPI, as per "Signature and persistence" section of ATA/PI spec (volume 1, sect 5.14).

LOCKING

None.

RETURNS

Device type, `ATA_DEV_ATA`, `ATA_DEV_ATAPI`, or `ATA_DEV_UNKNOWN` the event of failure.

ata_dev_id_string

Name

`ata_dev_id_string` — Convert IDENTIFY DEVICE page into string

Synopsis

```
void ata_dev_id_string (u16 * id, unsigned char * s, unsigned int ofs,
unsigned int len);
```

Arguments

id

IDENTIFY DEVICE results we will examine

s

string into which data is output

ofs

offset into identify device page

len

length of string to return. must be an even number.

Description

The strings in the IDENTIFY DEVICE page are broken up into 16-bit chunks. Run through the string, and output each 8-bit chunk linearly, regardless of platform.

LOCKING

caller.

ata_noop_dev_select

Name

`ata_noop_dev_select` — Select device 0/1 on ATA bus

Synopsis

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

Arguments

`ap`
ATA channel to manipulate

`device`
ATA device (numbered from zero) to select

Description

This function performs no actual function.

May be used as the `dev_select` entry in ata_port_operations.

LOCKING

caller.

ata_std_dev_select

Name

`ata_std_dev_select` — Select device 0/1 on ATA bus

Synopsis

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

Arguments

ap

ATA channel to manipulate

device

ATA device (numbered from zero) to select

Description

Use the method defined in the ATA specification to make either device 0, or device 1, active on the ATA channel. Works with both PIO and MMIO.

May be used as the `dev_select` entry in ata_port_operations.

LOCKING

caller.

ata_dev_config

Name

`ata_dev_config` — Run device specific handlers and check for

Synopsis

```
void ata_dev_config (struct ata_port * ap, unsigned int i);
```

Arguments

ap

Bus

i

Device

Description

SATA->PATA bridges

ata_port_probe

Name

`ata_port_probe` — Mark port as enabled

Synopsis

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

Arguments

ap

Port for which we indicate enablement

Description

Modify *ap* data structure such that the system thinks that the entire port is enabled.

LOCKING

host_set lock, or some other form of serialization.

__sata_phy_reset

Name

`__sata_phy_reset` — Wake/reset a low-level SATA PHY

Synopsis

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

Arguments

ap

SATA port associated with target SATA PHY.

Description

This function issues commands to standard SATA Sxxx PHY registers, to wake up the phy (and device), and clear any reset condition.

LOCKING

PCI/etc. bus probe sem.

sata_phy_reset

Name

`sata_phy_reset` — Reset SATA bus.

Synopsis

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

Arguments

ap

SATA port associated with target SATA PHY.

Description

This function resets the SATA bus, and then probes the bus for devices.

LOCKING

PCI/etc. bus probe sem.

ata_port_disable

Name

`ata_port_disable` — Disable port.

Synopsis

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

Arguments

ap

Port to be disabled.

Description

Modify *ap* data structure such that the system thinks that the entire port is disabled, and should never attempt to probe or communicate with devices on this port.

LOCKING

host_set lock, or some other form of serialization.

ata_bus_reset

Name

`ata_bus_reset` — reset host port and associated ATA channel

Synopsis

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

Arguments

ap

port to reset

Description

This is typically the first time we actually start issuing commands to the ATA channel. We wait for BSY to clear, then issue EXECUTE DEVICE DIAGNOSTIC command, polling for its result. Determine what devices, if any, are on the channel by looking at the device 0/1 error register. Look at the signature stored in each device's taskfile registers, to determine if the device is ATA or ATAPI.

LOCKING

PCI/etc. bus probe sem. Obtains host_set lock.

SIDE EFFECTS

Sets ATA_FLAG_PORT_DISABLED if bus reset fails.

ata_qc_prep

Name

`ata_qc_prep` — Prepare taskfile for submission

Synopsis

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

Arguments

qc

Metadata associated with taskfile to be prepared

Description

Prepare ATA taskfile for submission.

LOCKING

spin_lock_irqsave(host_set lock)

ata_sg_init_one

Name

ata_sg_init_one — Associate command with memory buffer

Synopsis

```
void ata_sg_init_one (struct ata_queued_cmd * qc, void * buf, unsigned
int buflen);
```

Arguments

qc

Command to be associated

buf

Memory buffer

buflen

Length of memory buffer, in bytes.

Description

Initialize the data-related elements of queued_cmd *qc* to point to a single memory buffer, *buf* of byte length *buflen*.

LOCKING

spin_lock_irqsave(host_set lock)

ata_sg_init

Name

ata_sg_init — Associate command with scatter-gather table.

Synopsis

```
void ata_sg_init (struct ata_queued_cmd * qc, struct scatterlist *
sg, unsigned int n_elem);
```

Arguments

qc

Command to be associated

sg

Scatter-gather table.

n_elem

Number of elements in s/g table.

Description

Initialize the data-related elements of queued_cmd *qc* to point to a scatter-gather table *sg*, containing *n_elem* elements.

LOCKING

spin_lock_irqsave(host_set lock)

ata_eng_timeout

Name

`ata_eng_timeout` — Handle timeout of queued command

Synopsis

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

Arguments

ap

Port on which timed-out command is active

Description

Some part of the kernel (currently, only the SCSI layer) has noticed that the active command on port *ap* has not completed after a specified length of time. Handle this condition by disabling DMA (if necessary) and completing transactions, with error if necessary.

This also handles the case of the "lost interrupt", where for some reason (possibly hardware bug, possibly driver bug) an interrupt was not delivered to the driver, even though the transaction completed successfully.

LOCKING

Inherited from SCSI layer (none, can sleep)

ata_qc_complete

Name

`ata_qc_complete` — Complete an active ATA command

Synopsis

```
void ata_qc_complete (struct ata_queued_cmd * qc, u8 drv_stat);
```

Arguments

`qc`

Command to complete

`drv_stat`

ATA Status register contents

Description

Indicate to the mid and upper layers that an ATA command has completed, with either an ok or not-ok status.

LOCKING

spin_lock_irqsave(host_set lock)

ata_qc_issue_prot

Name

`ata_qc_issue_prot` — issue taskfile to device in proto-dependent manner

Synopsis

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

Arguments

qc

command to issue to device

Description

Using various libata functions and hooks, this function starts an ATA command. ATA commands are grouped into classes called "protocols", and issuing each type of protocol is slightly different.

May be used as the `qc_issue` entry in ata_port_operations.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, negative on error.

ata_bmdma_start

Name

`ata_bmdma_start` — Start a PCI IDE BMDMA transaction

Synopsis

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

Arguments

qc

Info associated with this ATA transaction.

Description

Writes the ATA_DMA_START flag to the DMA command register.

May be used as the `bmdma_start` entry in ata_port_operations.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_setup

Name

`ata_bmdma_setup` — Set up PCI IDE BMDMA transaction

Synopsis

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

Arguments

qc

Info associated with this ATA transaction.

Description

Writes address of PRD table to device's PRD Table Address register, sets the DMA control register, and calls ops->`exec_command` to start the transfer.

May be used as the `bmdma_setup` entry in ata_port_operations.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_irq_clear

Name

`ata_bmdma_irq_clear` — Clear PCI IDE BMDMA interrupt.

Synopsis

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

Arguments

ap

Port associated with this ATA transaction.

Description

Clear interrupt and error flags in DMA status register.
May be used as the `irq_clear` entry in ata_port_operations.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_status

Name

`ata_bmdma_status` — Read PCI IDE BMDMA status

Synopsis

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

Arguments

ap

Port associated with this ATA transaction.

Description

Read and return BMDMA status register.
May be used as the `bmdma_status` entry in ata_port_operations.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_stop

Name

`ata_bmdma_stop` — Stop PCI IDE BMDMA transfer

Synopsis

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

Arguments

qc

Command we are ending DMA for

Description

Clears the ATA_DMA_START flag in the dma control register
May be used as the `bmdma_stop` entry in ata_port_operations.

LOCKING

spin_lock_irqsave(host_set lock)

ata_host_intr

Name

`ata_host_intr` — Handle host interrupt for given (port, task)

Synopsis

```
unsigned int ata_host_intr (struct ata_port * ap, struct
ata_queued_cmd * qc);
```

Arguments

ap

Port on which interrupt arrived (possibly...)

qc

Taskfile currently active in engine

Description

Handle host interrupt for given queued command. Currently, only DMA interrupts are handled. All other commands are handled via polling with interrupts disabled (nIEN bit).

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

One if interrupt was handled, zero if not (shared irq).

ata_interrupt

Name

`ata_interrupt` — Default ATA host interrupt handler

Synopsis

```
irqreturn_t ata_interrupt (int irq, void * dev_instance, struct
pt_regs * regs);
```

Arguments

irq

irq line (unused)

dev_instance

pointer to our ata_host_set information structure

regs

unused

Description

Default interrupt handler for PCI IDE devices. Calls `ata_host_intr` for each port that is not disabled.

LOCKING

Obtains host_set lock during operation.

RETURNS

IRQ_NONE or IRQ_HANDLED.

ata_port_start

Name

`ata_port_start` — Set port up for dma.

Synopsis

```
int ata_port_start (struct ata_port * ap);
```

Arguments

ap

Port to initialize

Description

Called just after data structures for each port are initialized. Allocates space for PRD table.

May be used as the `port_start` entry in ata_port_operations.

ata_port_stop

Name

`ata_port_stop` — Undo `ata_port_start`

Synopsis

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

Arguments

ap

Port to shut down

Description

Frees the PRD table.

May be used as the `port_stop` entry in ata_port_operations.

ata_device_add

Name

`ata_device_add` — Register hardware device with ATA and SCSI layers

Synopsis

```
int ata_device_add (struct ata_probe_ent * ent);
```

Arguments

ent

Probe information describing hardware device to be registered

Description

This function processes the information provided in the probe information struct *`ent`*, allocates the necessary ATA and SCSI host information structures, initializes them, and registers everything with requisite kernel subsystems.

This function requests irqs, probes the ATA bus, and probes the SCSI bus.

LOCKING

PCI/etc. bus probe sem.

RETURNS

Number of ports registered. Zero on error (no ports registered).

ata_host_set_remove

Name

`ata_host_set_remove` — PCI layer callback for device removal

Synopsis

```
void ata_host_set_remove (struct ata_host_set * host_set);
```

Arguments

`host_set`
ATA host set that was removed

Description

Unregister all objects associated with this host set. Free those objects.

LOCKING

Inherited from calling layer (may sleep).

ata_scsi_release

Name

`ata_scsi_release` — SCSI layer callback hook for host unload

Synopsis

```
int ata_scsi_release (struct Scsi_Host * host);
```

Arguments

host

libata host to be unloaded

Description

Performs all duties necessary to shut down a libata port... Kill port kthread, disable port, and release resources.

LOCKING

Inherited from SCSI layer.

RETURNS

One.

ata_std_ports

Name

`ata_std_ports` — initialize ioaddr with standard port offsets.

Synopsis

```
void ata_std_ports (struct ata_ioports * ioaddr);
```

Arguments

ioaddr

IO address structure to be initialized

Description

Utility function which initializes data_addr, error_addr, feature_addr, nsect_addr, lbal_addr, lbam_addr, lbah_addr, device_addr, status_addr, and command_addr to standard offsets relative to cmd_addr.

Does not set ctl_addr, altstatus_addr, bmdma_addr, or scr_addr.

ata_pci_init_native_mode

Name

ata_pci_init_native_mode — Initialize native-mode driver

Synopsis

```
struct ata_probe_ent * ata_pci_init_native_mode (struct pci_dev * pdev,
struct ata_port_info ** port, int ports);
```

Arguments

pdev

pci device to be initialized

port

array[2] of pointers to port info structures.

ports

bitmap of ports present

Description

Utility function which allocates and initializes an ata_probe_ent structure for a standard dual-port PIO-based IDE controller. The returned ata_probe_ent structure can be passed to ata_device_add. The returned ata_probe_ent structure should then be freed with kfree.

The caller need only pass the address of the primary port, the secondary will be deduced automatically. If the device has non standard secondary port mappings this function can be called twice, once for each interface.

ata_pci_init_one

Name

ata_pci_init_one — Initialize/register PCI IDE host controller

Synopsis

```
int ata_pci_init_one (struct pci_dev * pdev, struct ata_port_info **
port_info, unsigned int n_ports);
```

Arguments

pdev

Controller to be initialized

port_info

Information from low-level host driver

n_ports

Number of ports attached to host controller

Description

This is a helper function which can be called from a driver's xxx_init_one probe function if the hardware uses traditional IDE taskfile registers.

This function calls pci_enable_device, reserves its register regions, sets the dma mask, enables bus master mode, and calls ata_device_add

LOCKING

Inherited from PCI layer (may sleep).

RETURNS

Zero on success, negative on errno-based value on error.

ata_pci_remove_one

Name

ata_pci_remove_one — PCI layer callback for device removal

Synopsis

```
void ata_pci_remove_one (struct pci_dev * pdev);
```

Arguments

pdev

PCI device that was removed

Description

PCI layer indicates to libata via this hook that hot-unplug or module unload event has occured. Handle this by unregistering all objects associated with this PCI device. Free those objects. Then finally release PCI resources and disable device.

LOCKING

Inherited from PCI layer (may sleep).

Chapter 5. libata Core Internals

ata_tf_load_pio

Name

`ata_tf_load_pio` — send taskfile registers to host controller

Synopsis

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

Arguments

ap
: Port to which output is sent

tf
: ATA taskfile register set

Description

Outputs ATA taskfile to standard ATA host controller.

LOCKING

Inherited from caller.

ata_tf_load_mmio

Name

`ata_tf_load_mmio` — send taskfile registers to host controller

Synopsis

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

Arguments

ap

Port to which output is sent

tf

ATA taskfile register set

Description

Outputs ATA taskfile to standard ATA host controller using MMIO.

LOCKING

Inherited from caller.

ata_exec_command_pio

Name

`ata_exec_command_pio` — issue ATA command to host controller

Synopsis

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

Arguments

ap

port to which command is being issued

tf

ATA taskfile register set

Description

Issues PIO write to ATA command register, with proper synchronization with interrupt handler / other threads.

LOCKING

spin_lock_irqsave(host_set lock)

ata_exec_command_mmio

Name

`ata_exec_command_mmio` — issue ATA command to host controller

Synopsis

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

Arguments

ap

port to which command is being issued

tf

ATA taskfile register set

Description

Issues MMIO write to ATA command register, with proper synchronization with interrupt handler / other threads.

LOCKING

spin_lock_irqsave(host_set lock)

ata_exec

Name

`ata_exec` — issue ATA command to host controller

Synopsis

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

Arguments

ap

port to which command is being issued

tf

ATA taskfile register set

Description

Issues PIO/MMIO write to ATA command register, with proper synchronization with interrupt handler / other threads.

LOCKING

Obtains host_set lock.

ata_tf_to_host

Name

`ata_tf_to_host` — issue ATA taskfile to host controller

Synopsis

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

Arguments

ap

port to which command is being issued

tf

ATA taskfile register set

Description

Issues ATA taskfile register set to ATA host controller, with proper synchronization with interrupt handler and other threads.

LOCKING

Obtains host_set lock.

ata_tf_to_host_nolock

Name

`ata_tf_to_host_nolock` — issue ATA taskfile to host controller

Synopsis

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

Arguments

ap

port to which command is being issued

tf

ATA taskfile register set

Description

Issues ATA taskfile register set to ATA host controller, with proper synchronization with interrupt handler and other threads.

LOCKING

spin_lock_irqsave(host_set lock)

ata_tf_read_pio

Name

`ata_tf_read_pio` — input device's ATA taskfile shadow registers

Synopsis

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

Arguments

ap
Port from which input is read

tf
ATA taskfile register set for storing input

Description

Reads ATA taskfile registers for currently-selected device into *tf*.

LOCKING

Inherited from caller.

ata_tf_read_mmio

Name

ata_tf_read_mmio — input device's ATA taskfile shadow registers

Synopsis

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

Arguments

ap
Port from which input is read

tf
ATA taskfile register set for storing input

Description

Reads ATA taskfile registers for currently-selected device into *tf* via MMIO.

LOCKING

Inherited from caller.

ata_check_status_pio

Name

`ata_check_status_pio` — Read device status reg & clear interrupt

Synopsis

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

Arguments

ap

port where the device is

Description

Reads ATA taskfile status register for currently-selected device and return its value. This also clears pending interrupts from this device

LOCKING

Inherited from caller.

ata_check_status_mmio

Name

`ata_check_status_mmio` — Read device status reg & clear interrupt

Synopsis

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

Arguments

ap

port where the device is

Description

Reads ATA taskfile status register for currently-selected device via MMIO and return its value. This also clears pending interrupts from this device

LOCKING

Inherited from caller.

ata_prot_to_cmd

Name

`ata_prot_to_cmd` — determine which read/write opcodes to use

Synopsis

```
int ata_prot_to_cmd (int protocol, int lba48);
```

Arguments

protocol

ATA_PROT_xxx taskfile protocol

lba48

true is lba48 is present

Description

Given necessary input, determine which read/write commands to use to transfer data.

LOCKING

None.

ata_dev_set_protocol

Name

`ata_dev_set_protocol` — set taskfile protocol and r/w commands

Synopsis

```
void ata_dev_set_protocol (struct ata_device * dev);
```

Arguments

dev

device to examine and configure

Description

Examine the device configuration, after we have read the identify-device page and configured the data transfer mode. Set internal state related to the ATA taskfile protocol (pio, pio mult, dma, etc.) and calculate the proper read/write commands to use.

LOCKING

caller.

ata_mode_string

Name

`ata_mode_string` — convert UDMA bit offset to string

Synopsis

```
const char * ata_mode_string (unsigned int mask);
```

Arguments

mask

mask of bits supported; only highest bit counts.

Description

Determine string which represents the highest speed (highest bit in *udma_mask*).

LOCKING

None.

RETURNS

Constant C string representing highest speed listed in *udma_mask*, or the constant C string "<n/a>".

ata_pio_devchk

Name

ata_pio_devchk — PATA device presence detection

Synopsis

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

Arguments

ap

ATA channel to examine

device

Device to examine (starting at zero)

Description

This technique was originally described in Hale Landis's ATADRVR (www.ata-atapi.com), and later found its way into the ATA/ATAPI spec.

Write a pattern to the ATA shadow registers, and if a device is present, it will respond by correctly storing and echoing back the ATA shadow register contents.

LOCKING

caller.

ata_mmio_devchk

Name

`ata_mmio_devchk` — PATA device presence detection

Synopsis

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

Arguments

`ap`

ATA channel to examine

`device`

Device to examine (starting at zero)

Description

This technique was originally described in Hale Landis's ATADRVR (www.ata-atapi.com), and later found its way into the ATA/ATAPI spec.

Write a pattern to the ATA shadow registers, and if a device is present, it will respond by correctly storing and echoing back the ATA shadow register contents.

LOCKING

caller.

ata_devchk

Name

`ata_devchk` — PATA device presence detection

Synopsis

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

Arguments

ap

ATA channel to examine

device

Device to examine (starting at zero)

Description

Dispatch ATA device presence detection, depending on whether we are using PIO or MMIO to talk to the ATA shadow registers.

LOCKING

caller.

ata_dev_try_classify

Name

`ata_dev_try_classify` — Parse returned ATA device signature

Synopsis

```
u8 ata_dev_try_classify (struct ata_port * ap, unsigned int device);
```

Arguments

ap

ATA channel to examine

device

Device to examine (starting at zero)

Description

After an event -- SRST, E.D.D., or SATA COMRESET -- occurs, an ATA/ATAPI-defined set of values is placed in the ATA shadow registers, indicating the results of device detection and diagnostics.

Select the ATA device, and read the values from the ATA shadow registers. Then parse according to the Error register value, and the spec-defined values examined by `ata_dev_classify`.

LOCKING

caller.

ata_dev_select

Name

`ata_dev_select` — Select device 0/1 on ATA bus

Synopsis

```
void ata_dev_select (struct ata_port * ap, unsigned int device,
unsigned int wait, unsigned int can_sleep);
```

Arguments

`ap`

ATA channel to manipulate

`device`

ATA device (numbered from zero) to select

`wait`

non-zero to wait for Status register BSY bit to clear

`can_sleep`

non-zero if context allows sleeping

Description

Use the method defined in the ATA specification to make either device 0, or device 1, active on the ATA channel.

This is a high-level version of `ata_std_dev_select`, which additionally provides the services of inserting the proper pauses and status polling, where needed.

LOCKING

caller.

ata_dump_id

Name

`ata_dump_id` — IDENTIFY DEVICE info debugging output

Synopsis

```
void ata_dump_id (struct ata_device * dev);
```

Arguments

dev

Device whose IDENTIFY DEVICE page we will dump

Description

Dump selected 16-bit words from a detected device's IDENTIFY PAGE page.

LOCKING

caller.

ata_dev_identify

Name

`ata_dev_identify` — obtain IDENTIFY x DEVICE page

Synopsis

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

Arguments

ap

port on which device we wish to probe resides

device

device bus address, starting at zero

Description

Following bus reset, we issue the IDENTIFY [PACKET] DEVICE command, and read back the 512-byte device information page. The device information page is fed to us via the standard PIO-IN protocol, but we hand-code it here. (TODO: investigate using standard PIO-IN paths)

After reading the device information page, we use several bits of information from it to initialize data structures that will be used during the lifetime of the ata_device. Other data from the info page is used to disqualify certain older ATA devices we do not wish to support.

LOCKING

Inherited from caller. Some functions called by this function obtain the host_set lock.

ata_bus_probe

Name

`ata_bus_probe` — Reset and probe ATA bus

Synopsis

```
int ata_bus_probe (struct ata_port * ap);
```

Arguments

ap

Bus to probe

Description

Master ATA bus probing function. Initiates a hardware-dependent bus reset, then attempts to identify any devices found on the bus.

LOCKING

PCI/etc. bus probe sem.

RETURNS

Zero on success, non-zero on error.

ata_set_mode

Name

`ata_set_mode` — Program timings and issue SET FEATURES - XFER

Synopsis

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

Arguments

ap
: port on which timings will be programmed

Description

Set ATA device disk transfer mode (PIO3, UDMA6, etc.).

LOCKING

PCI/etc. bus probe sem.

ata_busy_sleep

Name

`ata_busy_sleep` — sleep until BSY clears, or timeout

Synopsis

```
unsigned int ata_busy_sleep (struct ata_port * ap, unsigned long
tmout_pat, unsigned long tmout);
```

Arguments

ap

port containing status register to be polled

tmout_pat

impatience timeout

tmout

overall timeout

Description

Sleep until ATA Status register bit BSY clears, or a timeout occurs.

LOCKING

None.

ata_bus_edd

Name

ata_bus_edd — Issue EXECUTE DEVICE DIAGNOSTIC command.

Synopsis

```
unsigned int ata_bus_edd (struct ata_port * ap);
```

Arguments

ap

Port to reset and probe

Description

Use the EXECUTE DEVICE DIAGNOSTIC command to reset and probe the bus. Not often used these days.

LOCKING

PCI/etc. bus probe sem.

ata_choose_xfer_mode

Name

`ata_choose_xfer_mode` — attempt to find best transfer mode

Synopsis

```
int ata_choose_xfer_mode (struct ata_port * ap, u8 * xfer_mode_out,
unsigned int * xfer_shift_out);
```

Arguments

`ap`

Port for which an xfer mode will be selected

`xfer_mode_out`

(output) SET FEATURES - XFER MODE code

`xfer_shift_out`

(output) bit shift that selects this mode

Description

Based on host and device capabilities, determine the maximum transfer mode that is amenable to all.

LOCKING

PCI/etc. bus probe sem.

RETURNS

Zero on success, negative on error.

ata_dev_set_xfermode

Name

`ata_dev_set_xfermode` — Issue SET FEATURES - XFER MODE command

Synopsis

```
void ata_dev_set_xfermode (struct ata_port * ap, struct ata_device *
dev);
```

Arguments

ap

Port associated with device *dev*

dev

Device to which command will be sent

Description

Issue SET FEATURES - XFER MODE command to device *dev* on port *ap*.

LOCKING

PCI/etc. bus probe sem.

ata_dev_init_params

Name

`ata_dev_init_params` — Issue INIT DEV PARAMS command

Synopsis

```
void ata_dev_init_params (struct ata_port * ap, struct ata_device *
dev);
```

Arguments

ap

Port associated with device *dev*

dev

Device to which command will be sent

ata_sg_clean

Name

ata_sg_clean — Unmap DMA memory associated with command

Synopsis

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

Arguments

qc

Command containing DMA memory to be released

Description

Unmap all mapped DMA memory associated with this command.

LOCKING

spin_lock_irqsave(host_set lock)

ata_fill_sg

Name

`ata_fill_sg` — Fill PCI IDE PRD table

Synopsis

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

Arguments

qc

Metadata associated with taskfile to be transferred

Description

Fill PCI IDE PRD (scatter-gather) table with segments associated with the current disk command.

LOCKING

spin_lock_irqsave(host_set lock)

ata_check_atapi_dma

Name

`ata_check_atapi_dma` — Check whether ATAPI DMA can be supported

Synopsis

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

Arguments

qc

Metadata associated with taskfile to check

Description

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.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

0 when ATAPI DMA can be used nonzero otherwise

ata_sg_setup_one

Name

`ata_sg_setup_one` — DMA-map the memory buffer associated with a command.

Synopsis

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

Arguments

qc

Command with memory buffer to be mapped.

Description

DMA-map the memory buffer associated with queued_cmd *qc*.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, negative on error.

ata_sg_setup

Name

`ata_sg_setup` — DMA-map the scatter-gather table associated with a command.

Synopsis

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

Arguments

qc

Command with scatter-gather table to be mapped.

Description

DMA-map the scatter-gather table associated with queued_cmd *qc*.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, negative on error.

ata_poll_qc_complete

Name

`ata_poll_qc_complete` — turn irq back on and finish qc

Synopsis

```
void ata_poll_qc_complete (struct ata_queued_cmd * qc, u8 drv_stat);
```

Arguments

qc
Command to complete

drv_stat
ATA status register content

LOCKING

None. (grabs host lock)

ata_pio_poll

Name

ata_pio_poll —

Synopsis

```
unsigned long ata_pio_poll (struct ata_port * ap);
```

Arguments

ap

LOCKING

None. (executing in kernel thread context)

RETURNS

ata_pio_complete

Name

ata_pio_complete —

Synopsis

```
int ata_pio_complete (struct ata_port * ap);
```

Arguments

ap

LOCKING

None. (executing in kernel thread context)

RETURNS

Non-zero if qc completed, zero otherwise.

swap_buf_le16

Name

swap_buf_le16 —

Synopsis

```
void swap_buf_le16 (u16 * buf, unsigned int buf_words);
```

Arguments

buf

Buffer to swap

buf_words

Number of 16-bit words in buffer.

Description

Swap halves of 16-bit words if needed to convert from little-endian byte order to native cpu byte order, or vice-versa.

ata_mmio_data_xfer

Name

`ata_mmio_data_xfer` — Transfer data by MMIO

Synopsis

```
void ata_mmio_data_xfer (struct ata_port * ap, unsigned char * buf,
unsigned int buflen, int write_data);
```

Arguments

`ap`
port to read/write

`buf`
data buffer

`buflen`
buffer length

`write_data`
read/write

Description

Transfer data from/to the device data register by MMIO.

LOCKING

Inherited from caller.

ata_pio_data_xfer

Name

`ata_pio_data_xfer` — Transfer data by PIO

Synopsis

```
void ata_pio_data_xfer (struct ata_port * ap, unsigned char * buf,
unsigned int buflen, int write_data);
```

Arguments

ap
: port to read/write

buf
: data buffer

buflen
: buffer length

write_data
: read/write

Description

Transfer data from/to the device data register by PIO.

LOCKING

Inherited from caller.

ata_data_xfer

Name

`ata_data_xfer` — Transfer data from/to the data register.

Synopsis

```
void ata_data_xfer (struct ata_port * ap, unsigned char * buf,
unsigned int buflen, int do_write);
```

Arguments

ap
: port to read/write

buf
: data buffer

buflen
: buffer length

do_write
: read/write

Description

Transfer data from/to the device data register.

LOCKING

Inherited from caller.

ata_pio_sector

Name

`ata_pio_sector` — Transfer ATA_SECT_SIZE (512 bytes) of data.

Synopsis

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

Arguments

qc
: Command on going

Description

Transfer ATA_SECT_SIZE of data from/to the ATA device.

LOCKING

Inherited from caller.

__atapi_pio_bytes

Name

__atapi_pio_bytes — Transfer data from/to the ATAPI device.

Synopsis

```
void __atapi_pio_bytes (struct ata_queued_cmd * qc, unsigned int
bytes);
```

Arguments

qc

Command on going

bytes

number of bytes

Description

Transfer Transfer data from/to the ATAPI device.

LOCKING

Inherited from caller.

atapi_pio_bytes

Name

atapi_pio_bytes — Transfer data from/to the ATAPI device.

Synopsis

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

Arguments

qc
Command on going

Description

Transfer Transfer data from/to the ATAPI device.

LOCKING

Inherited from caller.

ata_pio_block

Name

`ata_pio_block` —

Synopsis

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

Arguments

ap

LOCKING

None. (executing in kernel thread context)

ata_qc_timeout

Name

`ata_qc_timeout` — Handle timeout of queued command

Synopsis

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

Arguments

qc

Command that timed out

Description

Some part of the kernel (currently, only the SCSI layer) has noticed that the active command on port *ap* has not completed after a specified length of time. Handle this condition by disabling DMA (if necessary) and completing transactions, with error if necessary.

This also handles the case of the "lost interrupt", where for some reason (possibly hardware bug, possibly driver bug) an interrupt was not delivered to the driver, even though the transaction completed successfully.

LOCKING

Inherited from SCSI layer (none, can sleep)

ata_qc_new

Name

ata_qc_new — Request an available ATA command, for queueing

Synopsis

```
struct ata_queued_cmd * ata_qc_new (struct ata_port * ap);
```

Arguments

ap

Port associated with device *dev*

LOCKING

None.

ata_qc_new_init

Name

`ata_qc_new_init` — Request an available ATA command, and initialize it

Synopsis

```
struct ata_queued_cmd * ata_qc_new_init (struct ata_port * ap, struct
ata_device * dev);
```

Arguments

ap

Port associated with device *dev*

dev

Device from whom we request an available command structure

LOCKING

None.

ata_qc_free

Name

`ata_qc_free` — free unused ata_queued_cmd

Synopsis

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

Arguments

qc

Command to complete

Description

Designed to free unused ata_queued_cmd object in case something prevents using it.

LOCKING

spin_lock_irqsave(host_set lock)

ata_qc_issue

Name

`ata_qc_issue` — issue taskfile to device

Synopsis

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

Arguments

qc

command to issue to device

Description

Prepare an ATA command to submission to device. This includes mapping the data into a DMA-able area, filling in the S/G table, and finally writing the taskfile to hardware, starting the command.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, negative on error.

ata_bmdma_setup_mmio

Name

`ata_bmdma_setup_mmio` — Set up PCI IDE BMDMA transaction

Synopsis

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

Arguments

qc

Info associated with this ATA transaction.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_start_mmio

Name

`ata_bmdma_start_mmio` — Start a PCI IDE BMDMA transaction

Synopsis

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

Arguments

qc

Info associated with this ATA transaction.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_setup_pio

Name

`ata_bmdma_setup_pio` — Set up PCI IDE BMDMA transaction (PIO)

Synopsis

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

Arguments

qc

Info associated with this ATA transaction.

LOCKING

spin_lock_irqsave(host_set lock)

ata_bmdma_start_pio

Name

`ata_bmdma_start_pio` — Start a PCI IDE BMDMA transaction (PIO)

Synopsis

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

Arguments

qc

Info associated with this ATA transaction.

LOCKING

spin_lock_irqsave(host_set lock)

atapi_packet_task

Name

`atapi_packet_task` — Write CDB bytes to hardware

Synopsis

```
void atapi_packet_task (void * _data);
```

Arguments

`_data`

Port to which ATAPI device is attached.

Description

When device has indicated its readiness to accept a CDB, this function is called. Send the CDB. If DMA is to be performed, exit immediately. Otherwise, we are in polling mode, so poll status under operation succeeds or fails.

LOCKING

Kernel thread context (may sleep)

ata_host_remove

Name

`ata_host_remove` — Unregister SCSI host structure with upper layers

Synopsis

```
void ata_host_remove (struct ata_port * ap, unsigned int
do_unregister);
```

Arguments

ap

Port to unregister

do_unregister

1 if we fully unregister, 0 to just stop the port

ata_host_init

Name

ata_host_init — Initialize an ata_port structure

Synopsis

```
void ata_host_init (struct ata_port * ap, struct Scsi_Host * host,
struct ata_host_set * host_set, struct ata_probe_ent * ent, unsigned
int port_no);
```

Arguments

ap

Structure to initialize

host

associated SCSI mid-layer structure

host_set

Collection of hosts to which *ap* belongs

ent

Probe information provided by low-level driver

port_no

Port number associated with this ata_port

Description

Initialize a new ata_port structure, and its associated scsi_host.

LOCKING

Inherited from caller.

ata_host_add

Name

`ata_host_add` — Attach low-level ATA driver to system

Synopsis

```
struct ata_port * ata_host_add (struct ata_probe_ent * ent, struct
ata_host_set * host_set, unsigned int port_no);
```

Arguments

`ent`

Information provided by low-level driver

`host_set`

Collections of ports to which we add

`port_no`

Port number associated with this host

Description

Attach low-level ATA driver to system.

LOCKING

PCI/etc. bus probe sem.

RETURNS

New ata_port on success, for NULL on error.

Chapter 6. libata SCSI translation/emulation

ata_std_bios_param

Name

`ata_std_bios_param` — generic bios head/sector/cylinder calculator used by sd.

Synopsis

```
int ata_std_bios_param (struct scsi_device * sdev, struct block_device
* bdev, sector_t capacity, int geom[]);
```

Arguments

`sdev`

SCSI device for which BIOS geometry is to be determined

`bdev`

block device associated with *`sdev`*

`capacity`

capacity of SCSI device

`geom[]`

location to which geometry will be output

Description

Generic bios head/sector/cylinder calculator used by sd. Most BIOSes nowadays expect a XXX/255/16 (CHS) mapping. Some situations may arise where the disk is not bootable if this is not used.

LOCKING

Defined by the SCSI layer. We don't really care.

RETURNS

Zero.

ata_scsi_slave_config

Name

`ata_scsi_slave_config` — Set SCSI device attributes

Synopsis

```
int ata_scsi_slave_config (struct scsi_device * sdev);
```

Arguments

`sdev`
SCSI device to examine

Description

This is called before we actually start reading and writing to the device, to configure certain SCSI mid-layer behaviors.

LOCKING

Defined by SCSI layer. We don't really care.

ata_scsi_error

Name

`ata_scsi_error` — SCSI layer error handler callback

Synopsis

```
int ata_scsi_error (struct Scsi_Host * host);
```

Arguments

`host`
SCSI host on which error occurred

Description

Handles SCSI-layer-thrown error events.

LOCKING

Inherited from SCSI layer (none, can sleep)

RETURNS

Zero.

ata_scsi_queuecmd

Name

`ata_scsi_queuecmd` — Issue SCSI cdb to libata-managed device

Synopsis

```
int ata_scsi_queuecmd (struct scsi_cmnd * cmd, void (*done) (struct
scsi_cmnd *));
```

Arguments

cmd

SCSI command to be sent

done

Completion function, called when command is complete

Description

In some cases, this function translates SCSI commands into ATA taskfiles, and queues the taskfiles to be sent to hardware. In other cases, this function simulates a SCSI device by evaluating and responding to certain SCSI commands. This creates the overall effect of ATA and ATAPI devices appearing as SCSI devices.

LOCKING

Releases scsi-layer-held lock, and obtains host_set lock.

RETURNS

Zero.

ata_scsi_simulate

Name

`ata_scsi_simulate` — simulate SCSI command on ATA device

Synopsis

```
void ata_scsi_simulate (u16 * id, struct scsi_cmnd * cmd, void (*done)
(struct scsi_cmnd *));
```

Arguments

id

current IDENTIFY data for target device.

cmd

SCSI command being sent to device.

done

SCSI command completion function.

Description

Interprets and directly executes a select list of SCSI commands that can be handled internally.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsi_qc_new

Name

`ata_scsi_qc_new` — acquire new ata_queued_cmd reference

Synopsis

```
struct ata_queued_cmd * ata_scsi_qc_new (struct ata_port * ap, struct
ata_device * dev, struct scsi_cmnd * cmd, void (*done) (struct
scsi_cmnd *));
```

Arguments

ap

ATA port to which the new command is attached

dev

ATA device to which the new command is attached

cmd

SCSI command that originated this ATA command

done

SCSI command completion function

Description

Obtain a reference to an unused ata_queued_cmd structure, which is the basic libata structure representing a single ATA command sent to the hardware.

If a command was available, fill in the SCSI-specific portions of the structure with information on the current command.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Command allocated, or `NULL` if none available.

ata_to_sense_error

Name

`ata_to_sense_error` — convert ATA error to SCSI error

Synopsis

```
void ata_to_sense_error (struct ata_queued_cmd * qc, u8 drv_stat);
```

Arguments

qc

Command that we are erroring out

drv_stat

value contained in ATA status register

Description

Converts an ATA error into a SCSI error. While we are at it we decode and dump the ATA error for the user so that they have some idea what really happened at the non make-believe layer.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsi_start_stop_xlat

Name

`ata_scsi_start_stop_xlat` — Translate SCSI START STOP UNIT command

Synopsis

```
unsigned int ata_scsi_start_stop_xlat (struct ata_queued_cmd * qc, u8 *
scsicmd);
```

Arguments

qc

Storage for translated ATA taskfile

scsicmd

SCSI command to translate

Description

Sets up an ATA taskfile to issue STANDBY (to stop) or READ VERIFY (to start). Perhaps these commands should be preceded by CHECK POWER MODE to see what power mode the device is already in. [See SAT revision 5 at www.t10.org]

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, non-zero on error.

ata_scsi_flush_xlat

Name

`ata_scsi_flush_xlat` — Translate SCSI SYNCHRONIZE CACHE command

Synopsis

```
unsigned int ata_scsi_flush_xlat (struct ata_queued_cmd * qc, u8 *
scsicmd);
```

Arguments

qc

Storage for translated ATA taskfile

scsicmd

SCSI command to translate (ignored)

Description

Sets up an ATA taskfile to issue FLUSH CACHE or FLUSH CACHE EXT.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, non-zero on error.

scsi_6_lba_len

Name

`scsi_6_lba_len` — Get LBA and transfer length

Synopsis

```
void scsi_6_lba_len (u8 * scsicmd, u64 * plba, u32 * plen);
```

Arguments

`scsicmd`
SCSI command to translate

`plba`
the LBA

`plen`
the transfer length

Description

Calculate LBA and transfer length for 6-byte commands.

scsi_10_lba_len

Name

`scsi_10_lba_len` — Get LBA and transfer length

Synopsis

```
void scsi_10_lba_len (u8 * scsicmd, u64 * plba, u32 * plen);
```

Arguments

scsicmd
SCSI command to translate

plba
the LBA

plen
the transfer length

Description

Calculate LBA and transfer length for 10-byte commands.

scsi_16_lba_len

Name

scsi_16_lba_len — Get LBA and transfer length

Synopsis

```
void scsi_16_lba_len (u8 * scsicmd, u64 * plba, u32 * plen);
```

Arguments

scsicmd
SCSI command to translate

plba
the LBA

plen
the transfer length

Description

Calculate LBA and transfer length for 16-byte commands.

ata_scsi_verify_xlat

Name

`ata_scsi_verify_xlat` — Translate SCSI VERIFY command into an ATA one

Synopsis

```
unsigned int ata_scsi_verify_xlat (struct ata_queued_cmd * qc, u8 *
scsicmd);
```

Arguments

qc

Storage for translated ATA taskfile

scsicmd

SCSI command to translate

Description

Converts SCSI VERIFY command to an ATA READ VERIFY command.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, non-zero on error.

ata_scsi_rw_xlat

Name

`ata_scsi_rw_xlat` — Translate SCSI r/w command into an ATA one

Synopsis

```
unsigned int ata_scsi_rw_xlat (struct ata_queued_cmd * qc, u8 *
scsicmd);
```

Arguments

qc

Storage for translated ATA taskfile

scsicmd

SCSI command to translate

Description

Converts any of six SCSI read/write commands into the ATA counterpart, including starting sector (LBA), sector count, and taking into account the device's LBA48 support.

Commands `READ_6`, `READ_10`, `READ_16`, `WRITE_6`, `WRITE_10`, and `WRITE_16` are currently supported.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, non-zero on error.

ata_scsi_translate

Name

`ata_scsi_translate` — Translate then issue SCSI command to ATA device

Synopsis

```
void ata_scsi_translate (struct ata_port * ap, struct ata_device *
dev, struct scsi_cmnd * cmd, void (*done) (struct scsi_cmnd *),
ata_xlat_func_t xlat_func);
```

Arguments

ap

ATA port to which the command is addressed

dev

ATA device to which the command is addressed

cmd

SCSI command to execute

done

SCSI command completion function

xlat_func

Actor which translates *cmd* to an ATA taskfile

Description

Our ->queuecommand function has decided that the SCSI command issued can be directly translated into an ATA command, rather than handled internally.

This function sets up an ata_queued_cmd structure for the SCSI command, and sends that ata_queued_cmd to the hardware.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsi_rbuf_get

Name

ata_scsi_rbuf_get — Map response buffer.

Synopsis

```
unsigned int ata_scsi_rbuf_get (struct scsi_cmnd * cmd, u8 **
buf_out);
```

Arguments

cmd

SCSI command containing buffer to be mapped.

buf_out

Pointer to mapped area.

Description

Maps buffer contained within SCSI command *cmd*.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Length of response buffer.

ata_scsi_rbuf_put

Name

`ata_scsi_rbuf_put` — Unmap response buffer.

Synopsis

```
void ata_scsi_rbuf_put (struct scsi_cmnd * cmd, u8 * buf);
```

Arguments

`cmd`
SCSI command containing buffer to be unmapped.

`buf`
buffer to unmap

Description

Unmaps response buffer contained within *`cmd`*.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsi_rbuf_fill

Name

`ata_scsi_rbuf_fill` — wrapper for SCSI command simulators

Synopsis

```
void ata_scsi_rbuf_fill (struct ata_scsi_args * args, unsigned int
(*actor) (struct ata_scsi_args *args, u8 *rbuf, unsigned int
buflen));
```

Arguments

args

device IDENTIFY data / SCSI command of interest.

actor

Callback hook for desired SCSI command simulator

Description

Takes care of the hard work of simulating a SCSI command... Mapping the response buffer, calling the command's handler, and handling the handler's return value. This return value indicates whether the handler wishes the SCSI command to be completed successfully, or not.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_inq_std

Name

`ata_scsiop_inq_std` — Simulate INQUIRY command

Synopsis

```
unsigned int ata_scsiop_inq_std (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

args

device IDENTIFY data / SCSI command of interest.

rbuf

Response buffer, to which simulated SCSI cmd output is sent.

buflen

Response buffer length.

Description

Returns standard device identification data associated with non-EVPD INQUIRY command output.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_inq_00

Name

`ata_scsiop_inq_00` — Simulate INQUIRY EVPD page 0, list of pages

Synopsis

```
unsigned int ata_scsiop_inq_00 (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

args

device IDENTIFY data / SCSI command of interest.

rbuf

Response buffer, to which simulated SCSI cmd output is sent.

buflen

Response buffer length.

Description

Returns list of inquiry EVPD pages available.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_inq_80

Name

`ata_scsiop_inq_80` — Simulate INQUIRY EVPD page 80, device serial number

Synopsis

```
unsigned int ata_scsiop_inq_80 (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

`args`
device IDENTIFY data / SCSI command of interest.

`rbuf`
Response buffer, to which simulated SCSI cmd output is sent.

`buflen`
Response buffer length.

Description

Returns ATA device serial number.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_inq_83

Name

`ata_scsiop_inq_83` — Simulate INQUIRY EVPD page 83, device identity

Synopsis

```
unsigned int ata_scsiop_inq_83 (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

args

device IDENTIFY data / SCSI command of interest.

rbuf

Response buffer, to which simulated SCSI cmd output is sent.

buflen

Response buffer length.

Description

Returns device identification. Currently hardcoded to return “Linux ATA-SCSI simulator”.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_noop

Name

`ata_scsiop_noop` — Command handler that simply returns success.

Synopsis

```
unsigned int ata_scsiop_noop (struct ata_scsi_args * args, u8 * rbuf,
unsigned int buflen);
```

Arguments

args

device IDENTIFY data / SCSI command of interest.

rbuf

Response buffer, to which simulated SCSI cmd output is sent.

buflen

Response buffer length.

Description

No operation. Simply returns success to caller, to indicate that the caller should successfully complete this SCSI command.

LOCKING

spin_lock_irqsave(host_set lock)

ata_msense_push

Name

`ata_msense_push` — Push data onto MODE SENSE data output buffer

Synopsis

```
void ata_msense_push (u8 ** ptr_io, const u8 * last, const u8 * buf,
unsigned int buflen);
```

Arguments

ptr_io

(input/output) Location to store more output data

last

End of output data buffer

buf

Pointer to BLOB being added to output buffer

buflen

Length of BLOB

Description

Store MODE SENSE data on an output buffer.

LOCKING

None.

ata_msense_caching

Name

`ata_msense_caching` — Simulate MODE SENSE caching info page

Synopsis

```
unsigned int ata_msense_caching (u16 * id, u8 ** ptr_io, const u8 *
last);
```

Arguments

`id`

device IDENTIFY data

`ptr_io`

(input/output) Location to store more output data

`last`

End of output data buffer

Description

Generate a caching info page, which conditionally indicates write caching to the SCSI layer, depending on device capabilities.

LOCKING

None.

ata_msense_ctl_mode

Name

`ata_msense_ctl_mode` — Simulate MODE SENSE control mode page

Synopsis

```
unsigned int ata_msense_ctl_mode (u8 ** ptr_io, const u8 * last);
```

Arguments

ptr_io
(input/output) Location to store more output data

last
End of output data buffer

Description

Generate a generic MODE SENSE control mode page.

LOCKING

None.

ata_msense_rw_recovery

Name

`ata_msense_rw_recovery` — Simulate MODE SENSE r/w error recovery page

Synopsis

```
unsigned int ata_msense_rw_recovery (u8 ** ptr_io, const u8 * last);
```

Arguments

ptr_io
(input/output) Location to store more output data

last
End of output data buffer

Description

Generate a generic MODE SENSE r/w error recovery page.

LOCKING

None.

ata_scsiop_mode_sense

Name

`ata_scsiop_mode_sense` — Simulate MODE SENSE 6, 10 commands

Synopsis

```
unsigned int ata_scsiop_mode_sense (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

`args`
device IDENTIFY data / SCSI command of interest.

`rbuf`
Response buffer, to which simulated SCSI cmd output is sent.

`buflen`
Response buffer length.

Description

Simulate MODE SENSE commands.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_read_cap

Name

`ata_scsiop_read_cap` — Simulate READ CAPACITY[16] commands

Synopsis

```
unsigned int ata_scsiop_read_cap (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

`args`
device IDENTIFY data / SCSI command of interest.

`rbuf`
Response buffer, to which simulated SCSI cmd output is sent.

`buflen`
Response buffer length.

Description

Simulate READ CAPACITY commands.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsiop_report_luns

Name

`ata_scsiop_report_luns` — Simulate REPORT LUNS command

Synopsis

```
unsigned int ata_scsiop_report_luns (struct ata_scsi_args * args, u8 *
rbuf, unsigned int buflen);
```

Arguments

args
: device IDENTIFY data / SCSI command of interest.

rbuf
: Response buffer, to which simulated SCSI cmd output is sent.

buflen
: Response buffer length.

Description

Simulate REPORT LUNS command.

LOCKING

spin_lock_irqsave(host_set lock)

ata_scsi_badcmd

Name

`ata_scsi_badcmd` — End a SCSI request with an error

Synopsis

```
void ata_scsi_badcmd (struct scsi_cmnd * cmd, void (*done) (struct
scsi_cmnd *), u8 asc, u8 ascq);
```

Arguments

cmd
: SCSI request to be handled

done
: SCSI command completion function

asc
: SCSI-defined additional sense code

ascq
: SCSI-defined additional sense code qualifier

Description

Helper function that completes a SCSI command with `SAM_STAT_CHECK_CONDITION`, with a sense key `ILLEGAL_REQUEST` and the specified additional sense codes.

LOCKING

spin_lock_irqsave(host_set lock)

atapi_xlat

Name

`atapi_xlat` — Initialize PACKET taskfile

Synopsis

```
unsigned int atapi_xlat (struct ata_queued_cmd * qc, u8 * scsicmd);
```

Arguments

`qc`
command structure to be initialized

`scsicmd`
SCSI CDB associated with this PACKET command

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Zero on success, non-zero on failure.

ata_scsi_find_dev

Name

`ata_scsi_find_dev` — lookup ata_device from scsi_cmnd

Synopsis

```
struct ata_device * ata_scsi_find_dev (struct ata_port * ap, struct
scsi_device * scsidev);
```

Arguments

ap

ATA port to which the device is attached

scsidev

SCSI device from which we derive the ATA device

Description

Given various information provided in struct scsi_cmnd, map that onto an ATA bus, and using that mapping determine which ata_device is associated with the SCSI command to be sent.

LOCKING

spin_lock_irqsave(host_set lock)

RETURNS

Associated ATA device, or `NULL` if not found.

ata_get_xlat_func

Name

`ata_get_xlat_func` — check if SCSI to ATA translation is possible

Synopsis

```
ata_xlat_func_t ata_get_xlat_func (struct ata_device * dev, u8 cmd);
```

Arguments

dev

ATA device

cmd

SCSI command opcode to consider

Description

Look up the SCSI command given, and determine whether the SCSI command is to be translated or simulated.

RETURNS

Pointer to translation function if possible, `NULL` if not.

ata_scsi_dump_cdb

Name

`ata_scsi_dump_cdb` — dump SCSI command contents to dmesg

Synopsis

```
void ata_scsi_dump_cdb (struct ata_port * ap, struct scsi_cmnd * cmd);
```

Arguments

ap

ATA port to which the command was being sent

cmd

SCSI command to dump

Description

Prints the contents of a SCSI command via `printk`.

Chapter 7. ATA errors & exceptions

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.

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

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.

Exception categories

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.

In the following sections, two recovery actions - reset and reconfiguring transport - are mentioned. These are described further in the Section called *EH recovery actions*.

HSM violation

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

Examples

- ATA_STATUS doesn't contain !BSY && DRDY && !DRQ while trying to issue a command.
- !BSY && !DRQ during PIO data transfer.
- DRQ on command completion.
- !BSY && ERR after CDB tranfer 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.

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.

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.

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

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.

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

For ATAPI commands,

- !BSY && ERR && ABRT right after issuing PACKET indicates that PACKET command is not supported and falls in this category.
- !BSY && ERR(==CHK) && !ABRT after the last byte of CDB is transferred indicates CHECK CONDITION and doesn't fall in this category.
- !BSY && ERR(==CHK) && ABRT after the last byte of CDB is transferred *probably* indicates CHECK CONDITION and doesn't fall in this category.

Of errors detected as above, the followings are not ATA/ATAPI device errors but ATA bus errors and should be handled according to the Section called *ATA bus error*.

CRC error during data transfer

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.

ABRT error during data transfer or on completion

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.

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.

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

Media errors

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.

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.

Media changed / media change requested error

<<TODO: fill here>>

Address error

This is indicated by IDNF bit in the ERROR register. Report to upper layer.

Other errors

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.

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.

3.2.3.3a N/A

> 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.

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

ATAPI device CHECK CONDITION

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.

Once sense data is acquired, this type of errors can be handled similary 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 the Section called *ATA bus error*.

ATA device error (NCQ)

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.

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 the Section called *ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)* 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.

Note that ATA bus errors can be reported as ATA device NCQ errors. This should be handled as described in the Section called *ATA bus error*.

If READ LOG EXT Log Page 10h fails or reports NQ, we're thoroughly screwed. This condition should be treated according to the Section called *HSM violation*.

ATA bus error

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

- ICRC or ABRT error as described in the Section called *ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)*.
- Controller-specific error completion with error information indicating transmission error.
- On some controllers, command timeout. In this case, there may be a mechanism to determine that the timeout is due to transmission error.
- Unknown/random errors, timeouts and all sorts of weirdities.

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.

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 the Section called *Reconfigure transport* for more information.

PCI bus error

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.

Late completion

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.

Unknown error (timeout)

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.

Timeouts can also be caused by transmission errors. Refer to the Section called *ATA bus error* for more details.

Hotplug and power management exceptions

<<TODO: fill here>>

EH recovery actions

This section discusses several important recovery actions.

Clearing error condition

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

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

Reset

During EH, resetting is necessary in the following cases.

- HSM is in unknown or invalid state
- HBA is in unknown or invalid state
- EH needs to make HBA/device forget about in-flight commands
- HBA/device behaves weirdly

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.

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

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.

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

PATA hardware reset

> 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.

Software reset

> 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.

EXECUTE DEVICE DIAGNOSTIC command

> 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.
>
> 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.

ATAPI DEVICE RESET command

> 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.

SATA phy reset

> 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.

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.

Parameters affected are.

- CHS set up with INITIALIZE DEVICE PARAMETERS (seldomly used)
- Parameters set with SET FEATURES including transfer mode setting
- Block count set with SET MULTIPLE MODE
- Other parameters (SET MAX, MEDIA LOCK...)

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).

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.

Reconfigure transport

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.

The following is a possible scheme Jeff Garzik suggested.

> If more than $N (3?) transmission errors happen in 15 minutes,
>
> - if SATA, decrease SATA PHY speed. if speed cannot be decreased,
> - decrease UDMA xfer speed. if at UDMA0, switch to PIO4,
> - decrease PIO xfer speed. if at PIO3, complain, but continue

Chapter 8. ata_piix Internals

piix_pata_cbl_detect

Name

`piix_pata_cbl_detect` — Probe host controller cable detect info

Synopsis

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

Arguments

ap

Port for which cable detect info is desired

Description

Read 80c cable indicator from ATA PCI device's PCI config register. This register is normally set by firmware (BIOS).

LOCKING

None (inherited from caller).

piix_pata_phy_reset

Name

`piix_pata_phy_reset` — Probe specified port on PATA host controller

Synopsis

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

Arguments

ap

Port to probe

Description

Probe PATA phy.

LOCKING

None (inherited from caller).

piix_sata_probe

Name

`piix_sata_probe` — Probe PCI device for present SATA devices

Synopsis

```
int piix_sata_probe (struct ata_port * ap);
```

Arguments

ap

Port associated with the PCI device we wish to probe

Description

Reads SATA PCI device's PCI config register Port Configuration and Status (PCS) to determine port and device availability.

LOCKING

None (inherited from caller).

RETURNS

Non-zero if port is enabled, it may or may not have a device attached in that case (PRESENT bit would only be set if BIOS probe was done). Zero is returned if port is disabled.

piix_sata_phy_reset

Name

`piix_sata_phy_reset` — Probe specified port on SATA host controller

Synopsis

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

Arguments

ap

Port to probe

Description

Probe SATA phy.

LOCKING

None (inherited from caller).

piix_set_piomode

Name

`piix_set_piomode` — Initialize host controller PATA PIO timings

Synopsis

```
void piix_set_piomode (struct ata_port * ap, struct ata_device *
adev);
```

Arguments

ap

Port whose timings we are configuring

adev

um

Description

Set PIO mode for device, in host controller PCI config space.

LOCKING

None (inherited from caller).

piix_set_dmamode

Name

`piix_set_dmamode` — Initialize host controller PATA PIO timings

Synopsis

```
void piix_set_dmamode (struct ata_port * ap, struct ata_device *
adev);
```

Arguments

ap

Port whose timings we are configuring

adev

um

Description

Set UDMA mode for device, in host controller PCI config space.

LOCKING

None (inherited from caller).

piix_init_one

Name

`piix_init_one` — Register PIIX ATA PCI device with kernel services

Synopsis

```
int piix_init_one (struct pci_dev * pdev, const struct pci_device_id *
ent);
```

Arguments

pdev

PCI device to register

ent

Entry in piix_pci_tbl matching with *pdev*

Description

Called from kernel PCI layer. We probe for combined mode (sigh), and then hand over control to libata, for it to do the rest.

LOCKING

Inherited from PCI layer (may sleep).

RETURNS

Zero on success, or -ERRNO value.

Chapter 9. sata_sil Internals

sil_dev_config

Name

`sil_dev_config` — Apply device/host-specific errata fixups

Synopsis

```
void sil_dev_config (struct ata_port * ap, struct ata_device * dev);
```

Arguments

ap

Port containing device to be examined

dev

Device to be examined

Description

After the IDENTIFY [PACKET] DEVICE step is complete, and a device is known to be present, this function is called. We apply two errata fixups which are specific to Silicon Image, a Seagate and a Maxtor fixup.

For certain Seagate devices, we must limit the maximum sectors to under 8K.

For certain Maxtor devices, we must not program the drive beyond udma5.

Both fixups are unfairly pessimistic. As soon as I get more information on these errata, I will create a more exhaustive list, and apply the fixups to only the specific devices/hosts/firmwares that need it.

20040111 - Seagate drives affected by the Mod15Write bug are blacklisted The Maxtor quirk is in the blacklist, but I'm keeping the original pessimistic fix for the following reasons... - There seems to be less info on it, only one device gleaned off the Windows driver, maybe only one is affected. More info would be greatly appreciated. - But then again UDMA5 is hardly anything to complain about

Chapter 10. Thanks

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.

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

libata's device detection 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 ATADRVR driver (www.ata-atapi.com).