

libATA Developer's Guide

Jeff Garzik

libATA Developer's Guide

by Jeff Garzik

Copyright © 2003-2006 Jeff Garzik

The contents of this file are subject to the Open Software License version 1.1 that can be found at <http://fedoraproject.org/wiki/Licensing:OSL1.1> and is included herein by reference.

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

Table of Contents

| | |
|---|----|
| 1. Introduction | 1 |
| 2. libata Driver API | 2 |
| struct ata_port_operations | 2 |
| Disable ATA port | 2 |
| Post-IDENTIFY device configuration | 2 |
| Set PIO/DMA mode | 2 |
| Taskfile read/write | 3 |
| PIO data read/write | 3 |
| ATA command execute | 3 |
| Per-cmd ATAPI DMA capabilities filter | 3 |
| Read specific ATA shadow registers | 3 |
| Write specific ATA shadow register | 4 |
| Select ATA device on bus | 4 |
| Private tuning method | 4 |
| Control PCI IDE BMDMA engine | 4 |
| High-level taskfile hooks | 5 |
| Exception and probe handling (EH) | 5 |
| Hardware interrupt handling | 6 |
| SATA phy read/write | 6 |
| Init and shutdown | 7 |
| 3. Error handling | 8 |
| Origins of commands | 8 |
| How commands are issued | 8 |
| How commands are processed | 9 |
| How commands are completed | 9 |
| ata_scsi_error() | 10 |
| Problems with the current EH | 10 |
| 4. libata Library | 12 |
| ata_link_next | 13 |
| ata_dev_next | 14 |
| atapi_cmd_type | 15 |
| ata_tf_to_fis | 16 |
| ata_tf_from_fis | 17 |
| ata_pack_xfermask | 18 |
| ata_unpack_xfermask | 19 |
| ata_xfer_mask2mode | 20 |
| ata_xfer_mode2mask | 21 |
| ata_xfer_mode2shift | 22 |
| ata_mode_string | 23 |
| ata_dev_classify | 24 |
| ata_id_string | 25 |
| ata_id_c_string | 26 |
| ata_id_xfermask | 27 |
| ata_pio_need_iordy | 28 |
| ata_do_dev_read_id | 29 |
| ata_cable_40wire | 30 |
| ata_cable_80wire | 31 |
| ata_cable_unknown | 32 |
| ata_cable_ignore | 33 |
| ata_cable_sata | 34 |
| ata_dev_pair | 35 |

| | |
|-----------------------------------|----|
| sata_set_spd | 36 |
| ata_timing_cycle2mode | 37 |
| ata_do_set_mode | 38 |
| ata_wait_after_reset | 39 |
| sata_link_debounce | 40 |
| sata_link_resume | 41 |
| sata_link_scr_lpm | 42 |
| ata_std_prereset | 43 |
| sata_link_hardreset | 44 |
| sata_std_hardreset | 45 |
| ata_std_postreset | 46 |
| ata_dev_set_feature | 47 |
| ata_std_qc_defer | 48 |
| ata_sg_init | 49 |
| ata_qc_complete | 50 |
| ata_qc_complete_multiple | 51 |
| sata_scr_valid | 52 |
| sata_scr_read | 53 |
| sata_scr_write | 54 |
| sata_scr_write_flush | 55 |
| ata_link_online | 56 |
| ata_link_offline | 57 |
| ata_host_suspend | 58 |
| ata_host_resume | 59 |
| ata_host_alloc | 60 |
| ata_host_alloc_pinfo | 61 |
| ata_slave_link_init | 62 |
| ata_host_start | 63 |
| ata_host_init | 64 |
| ata_host_register | 65 |
| ata_host_activate | 66 |
| ata_host_detach | 67 |
| ata_pci_remove_one | 68 |
| ata_platform_remove_one | 69 |
| ata_msleep | 70 |
| ata_wait_register | 71 |
| sata_lpm_ignore_phy_events | 72 |
| 5. libata Core Internals | 73 |
| ata_dev_phys_link | 74 |
| ata_force_cbl | 75 |
| ata_force_link_limits | 76 |
| ata_force_xfermask | 77 |
| ata_force_horkage | 78 |
| ata_rwcmd_protocol | 79 |
| ata_tf_read_block | 80 |
| ata_build_rw_tf | 81 |
| ata_read_native_max_address | 82 |
| ata_set_max_sectors | 83 |
| ata_hpa_resize | 84 |
| ata_dump_id | 85 |
| ata_exec_internal_sg | 86 |
| ata_exec_internal | 87 |
| ata_pio_mask_no_iordy | 88 |
| ata_dev_read_id | 89 |

| | |
|--|-----|
| ata_dev_configure | 90 |
| ata_bus_probe | 91 |
| sata_print_link_status | 92 |
| sata_down_spd_limit | 93 |
| sata_set_spd_needed | 94 |
| ata_down_xfermask_limit | 95 |
| ata_wait_ready | 96 |
| ata_dev_same_device | 97 |
| ata_dev_reread_id | 98 |
| ata_dev_revalidate | 99 |
| ata_is_40wire | 100 |
| cable_is_40wire | 101 |
| ata_dev_xfermask | 102 |
| ata_dev_set_xfermode | 103 |
| ata_dev_init_params | 104 |
| ata_sg_clean | 105 |
| atapi_check_dma | 106 |
| ata_sg_setup | 107 |
| swap_buf_le16 | 108 |
| ata_qc_new_init | 109 |
| ata_qc_free | 110 |
| ata_qc_issue | 111 |
| ata_phys_link_online | 112 |
| ata_phys_link_offline | 113 |
| ata_dev_init | 114 |
| ata_link_init | 115 |
| sata_link_init_spd | 116 |
| ata_port_alloc | 117 |
| ata_finalize_port_ops | 118 |
| ata_port_detach | 119 |
| 6. libata SCSI translation/emulation | 120 |
| ata_std_bios_param | 121 |
| ata_scsi_unlock_native_capacity | 122 |
| ata_scsi_slave_config | 123 |
| ata_scsi_slave_destroy | 124 |
| __ata_change_queue_depth | 125 |
| ata_scsi_change_queue_depth | 126 |
| ata_scsi_queuecmd | 127 |
| ata_scsi_simulate | 128 |
| ata_sas_port_alloc | 129 |
| ata_sas_port_start | 130 |
| ata_sas_port_stop | 131 |
| ata_sas_async_probe | 132 |
| ata_sas_port_init | 133 |
| ata_sas_port_destroy | 134 |
| ata_sas_slave_configure | 135 |
| ata_sas_queuecmd | 136 |
| ata_get_identity | 137 |
| ata_cmd_ioctl | 138 |
| ata_task_ioctl | 139 |
| ata_scsi_qc_new | 140 |
| ata_dump_status | 141 |
| ata_to_sense_error | 142 |
| ata_gen_ata_sense | 143 |

| | |
|--|-----|
| ataapi_drain_needed | 144 |
| ata_scsi_start_stop_xlat | 145 |
| ata_scsi_flush_xlat | 146 |
| scsi_6_lba_len | 147 |
| scsi_10_lba_len | 148 |
| scsi_16_lba_len | 149 |
| ata_scsi_verify_xlat | 150 |
| ata_scsi_rw_xlat | 151 |
| ata_scsi_translate | 152 |
| ata_scsi_rbuf_get | 153 |
| ata_scsi_rbuf_put | 154 |
| ata_scsi_rbuf_fill | 155 |
| ata_scsiop_inq_std | 156 |
| ata_scsiop_inq_00 | 157 |
| ata_scsiop_inq_80 | 158 |
| ata_scsiop_inq_83 | 159 |
| ata_scsiop_inq_89 | 160 |
| ata_scsiop_noop | 161 |
| modecpy | 162 |
| ata_msense_caching | 163 |
| ata_msense_ctl_mode | 164 |
| ata_msense_rw_recovery | 165 |
| ata_scsiop_mode_sense | 166 |
| ata_scsiop_read_cap | 167 |
| ata_scsiop_report_luns | 168 |
| ataapi_xlat | 169 |
| ata_scsi_find_dev | 170 |
| ata_scsi_pass_thru | 171 |
| ata_scsi_report_zones_complete | 172 |
| ata_mselect_caching | 173 |
| ata_mselect_control | 174 |
| ata_scsi_mode_select_xlat | 175 |
| ata_get_xlat_func | 176 |
| ata_scsi_dump_cdb | 177 |
| ata_scsi_offline_dev | 178 |
| ata_scsi_remove_dev | 179 |
| ata_scsi_media_change_notify | 180 |
| ata_scsi_hotplug | 181 |
| ata_scsi_user_scan | 182 |
| ata_scsi_dev_rescan | 183 |
| 7. ATA errors and exceptions | 184 |
| Exception categories | 184 |
| HSM violation | 184 |
| ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION) | 184 |
| ATAPI device CHECK CONDITION | 186 |
| ATA device error (NCQ) | 186 |
| ATA bus error | 186 |
| PCI bus error | 187 |
| Late completion | 187 |
| Unknown error (timeout) | 187 |
| Hotplug and power management exceptions | 187 |
| EH recovery actions | 187 |
| Clearing error condition | 187 |
| Reset | 187 |

| | |
|-------------------------------|-----|
| Reconfigure transport | 189 |
| 8. ata_piix Internals | 190 |
| ich_pata_cable_detect | 191 |
| piix_pata_prereset | 192 |
| piix_set_piomode | 193 |
| do_pata_set_dmamode | 194 |
| piix_set_dmamode | 195 |
| ich_set_dmamode | 196 |
| piix_check_450nx_errata | 197 |
| piix_init_one | 198 |
| 9. sata_sil Internals | 199 |
| sil_set_mode | 200 |
| sil_dev_config | 201 |
| 10. Thanks | 202 |

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

Called from ata_bus_probe() on a failed probe. 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.

This entry may be specified as NULL in ata_port_operations.

Set PIO/DMA mode

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

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

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

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

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

Taskfile read/write

```
void (*sff_tf_load) (struct ata_port *ap, struct ata_taskfile *tf);
void (*sff_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_sff_tf_load() and ata_sff_tf_read() for these hooks.

PIO data read/write

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

All bmdma-style drivers must implement this hook. This is the low-level operation that actually copies the data bytes during a PIO data transfer. Typically the driver will choose one of ata_sff_data_xfer_noirq(), ata_sff_data_xfer(), or ata_sff_data_xfer32().

ATA command execute

```
void (*sff_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_sff_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 (*sff_check_status)(struct ata_port *ap);
u8 (*sff_check_altstatus)(struct ata_port *ap);
```

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

Write specific ATA shadow register

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

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

Select ATA device on bus

```
void (*sff_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_sff_dev_select()` for this hook.

Private tuning method

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

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

Warning

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

Control PCI IDE BMDMA engine

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

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

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

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

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

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

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

High-level taskfile hooks

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

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

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

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

Exception and probe handling (EH)

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

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

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

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

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

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

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

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

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

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

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

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

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

Hardware interrupt handling

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

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

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

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

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

SATA phy read/write

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

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

Init and shutdown

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

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

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

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

->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 preallocated during port initialization and repetitively used for command executions. Currently only one `qc` is allocated per port but yet-to-be-merged NCQ branch allocates one for each tag and maps each `qc` to NCQ tag 1-to-1.

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

How commands are issued

Internal commands

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

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

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

SCSI commands

All libata drivers use `ata_scsi_queuecmd()` as `host->queuecommand` callback. `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 `ata_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 `host->queuecommand` while holding `host_set` lock, so all above occur while holding `host_set` lock.

How commands are processed

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

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

| | |
|---------------------|---|
| ATA NO DATA or DMA | ATA_PROT_NODATA and ATA_PROT_DMA fall into this category. These types of commands don't require any software intervention once issued. Device will raise interrupt on completion. |
| ATA PIO | ATA_PROT_PIO is in this category. libata currently implements PIO with polling. ATA_NIEN bit is set to turn off interrupt and pio_task on ata_wq performs polling and IO. |
| ATAPI NODATA or DMA | ATA_PROT_ATAPI_NODATA and ATA_PROT_ATAPI_DMA are in this category. packet_task is used to poll BSY bit after issuing PACKET command. Once BSY is turned off by the device, packet_task transfers CDB and hands off processing to interrupt handler. |
| ATAPI PIO | ATA_PROT_ATAPI is in this category. ATA_NIEN bit is set and, as in ATAPI NODATA or DMA, packet_task submits cdb. However, after submitting cdb, further processing (data transfer) is handed off to pio_task. |

How commands are completed

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

ata_qc_complete() does the following.

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

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

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

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

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

ata_scsi_error()

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

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

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

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

Problems with the current EH

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

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

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

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

Chapter 4. libata Library

Name

`ata_link_next` — link iteration helper

Synopsis

```
struct ata_link * ata_link_next (struct ata_link * link, struct ata_port  
* ap, enum ata_link_iter_mode mode);
```

Arguments

link the previous link, NULL to start

ap ATA port containing links to iterate

mode iteration mode, one of ATA_LITER_*

LOCKING

Host lock or EH context.

RETURNS

Pointer to the next link.

Name

`ata_dev_next` — device iteration helper

Synopsis

```
struct ata_device * ata_dev_next (struct ata_device * dev, struct
ata_link * link, enum ata_dev_iter_mode mode);
```

Arguments

dev the previous device, NULL to start

link ATA link containing devices to iterate

mode iteration mode, one of `ATA_DITER_*`

LOCKING

Host lock or EH context.

RETURNS

Pointer to the next device.

Name

`atapi_cmd_type` — Determine ATAPI command type from SCSI opcode

Synopsis

```
int atapi_cmd_type (u8 opcode);
```

Arguments

opcode SCSI opcode

Description

Determine ATAPI command type from *opcode*.

LOCKING

None.

RETURNS

`ATAPI_{READ|WRITE|READ_CD|PASS_THRU|MISC}`

Name

`ata_tf_to_fis` — Convert ATA taskfile to SATA FIS structure

Synopsis

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

Arguments

| | |
|---------------|------------------------------------|
| <i>tf</i> | Taskfile to convert |
| <i>pmp</i> | Port multiplier port |
| <i>is_cmd</i> | This FIS is for command |
| <i>fis</i> | Buffer into which data will output |

Description

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

LOCKING

Inherited from caller.

Name

`ata_tf_from_fis` — Convert SATA FIS to ATA taskfile

Synopsis

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

Arguments

fis Buffer from which data will be input

tf Taskfile to output

Description

Converts a serial ATA FIS structure to a standard ATA taskfile.

LOCKING

Inherited from caller.

Name

`ata_pack_xfermask` — Pack pio, mwdma and udma masks into xfer_mask

Synopsis

```
unsigned long ata_pack_xfermask (unsigned long pio_mask, unsigned long  
mwdma_mask, unsigned long udma_mask);
```

Arguments

pio_mask pio_mask

mwdma_mask mwdma_mask

udma_mask udma_mask

Description

Pack *pio_mask*, *mwdma_mask* and *udma_mask* into a single unsigned int xfer_mask.

LOCKING

None.

RETURNS

Packed xfer_mask.

Name

`ata_unpack_xfermask` — Unpack `xfer_mask` into pio, mwdma and udma masks

Synopsis

```
void ata_unpack_xfermask (unsigned long xfer_mask, unsigned long * pio_mask, unsigned long * mwdma_mask, unsigned long * udma_mask);
```

Arguments

| | |
|-------------------|-----------------------------------|
| <i>xfer_mask</i> | <code>xfer_mask</code> to unpack |
| <i>pio_mask</i> | resulting <code>pio_mask</code> |
| <i>mwdma_mask</i> | resulting <code>mwdma_mask</code> |
| <i>udma_mask</i> | resulting <code>udma_mask</code> |

Description

Unpack *xfer_mask* into *pio_mask*, *mwdma_mask* and *udma_mask*. Any NULL destination masks will be ignored.

Name

`ata_xfer_mask2mode` — Find matching XFER_* for the given `xfer_mask`

Synopsis

```
u8 ata_xfer_mask2mode (unsigned long xfer_mask);
```

Arguments

xfer_mask `xfer_mask` of interest

Description

Return matching XFER_* value for *xfer_mask*. Only the highest bit of *xfer_mask* is considered.

LOCKING

None.

RETURNS

Matching XFER_* value, 0xff if no match found.

Name

`ata_xfer_mode2mask` — Find matching `xfer_mask` for `XFER_*`

Synopsis

```
unsigned long ata_xfer_mode2mask (u8 xfer_mode);
```

Arguments

xfer_mode `XFER_*` of interest

Description

Return matching `xfer_mask` for *xfer_mode*.

LOCKING

None.

RETURNS

Matching `xfer_mask`, 0 if no match found.

Name

`ata_xfer_mode2shift` — Find matching `xfer_shift` for `XFER_*`

Synopsis

```
int ata_xfer_mode2shift (unsigned long xfer_mode);
```

Arguments

xfer_mode `XFER_*` of interest

Description

Return matching `xfer_shift` for *xfer_mode*.

LOCKING

None.

RETURNS

Matching `xfer_shift`, -1 if no match found.

Name

`ata_mode_string` — convert `xfer_mask` to string

Synopsis

```
const char * ata_mode_string (unsigned long xfer_mask);
```

Arguments

xfer_mask mask of bits supported; only highest bit counts.

Description

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

LOCKING

None.

RETURNS

Constant C string representing highest speed listed in *mode_mask*, or the constant C string “<n/a>”.

Name

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

Synopsis

```
unsigned int ata_dev_classify (const 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`, `ATA_DEV_PMP`, `ATA_DEV_ZAC`, or `ATA_DEV_UNKNOWN` the event of failure.

Name

`ata_id_string` — Convert IDENTIFY DEVICE page into string

Synopsis

```
void ata_id_string (const ul6 * 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.

Name

`ata_id_c_string` — Convert IDENTIFY DEVICE page into C string

Synopsis

```
void ata_id_c_string (const 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 odd number.

Description

This function is identical to `ata_id_string` except that it trims trailing spaces and terminates the resulting string with null. *len* must be actual maximum length (even number) + 1.

LOCKING

caller.

Name

`ata_id_xfermask` — Compute xfermask from the given IDENTIFY data

Synopsis

```
unsigned long ata_id_xfermask (const u16 * id);
```

Arguments

id IDENTIFY data to compute xfer mask from

Description

Compute the xfermask for this device. This is not as trivial as it seems if we must consider early devices correctly.

FIXME

pre IDE drive timing (do we care ?).

LOCKING

None.

RETURNS

Computed xfermask

Name

`ata_pio_need_iordy` — check if iordy needed

Synopsis

```
unsigned int ata_pio_need_iordy (const struct ata_device * adev);
```

Arguments

adev ATA device

Description

Check if the current speed of the device requires IORDY. Used by various controllers for chip configuration.

Name

`ata_do_dev_read_id` — default ID read method

Synopsis

```
unsigned int ata_do_dev_read_id (struct ata_device * dev, struct
ata_taskfile * tf, u16 * id);
```

Arguments

dev device

tf proposed taskfile

id data buffer

Description

Issue the identify taskfile and hand back the buffer containing identify data. For some RAID controllers and for pre ATA devices this function is wrapped or replaced by the driver

Name

ata_cable_40wire — return 40 wire cable type

Synopsis

```
int ata_cable_40wire (struct ata_port * ap);
```

Arguments

ap port

Description

Helper method for drivers which want to hardwire 40 wire cable detection.

Name

`ata_cable_80wire` — return 80 wire cable type

Synopsis

```
int ata_cable_80wire (struct ata_port * ap);
```

Arguments

ap port

Description

Helper method for drivers which want to hardwire 80 wire cable detection.

Name

`ata_cable_unknown` — return unknown PATA cable.

Synopsis

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

Arguments

ap port

Description

Helper method for drivers which have no PATA cable detection.

Name

`ata_cable_ignore` — return ignored PATA cable.

Synopsis

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

Arguments

ap port

Description

Helper method for drivers which don't use cable type to limit transfer mode.

Name

`ata_cable_sata` — return SATA cable type

Synopsis

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

Arguments

ap port

Description

Helper method for drivers which have SATA cables

Name

`ata_dev_pair` — return other device on cable

Synopsis

```
struct ata_device * ata_dev_pair (struct ata_device * adev);
```

Arguments

adev device

Description

Obtain the other device on the same cable, or if none is present NULL is returned

Name

`sata_set_spd` — set SATA spd according to spd limit

Synopsis

```
int sata_set_spd (struct ata_link * link);
```

Arguments

link Link to set SATA spd for

Description

Set SATA spd of *link* according to `sata_spd_limit`.

LOCKING

Inherited from caller.

RETURNS

0 if spd doesn't need to be changed, 1 if spd has been changed. Negative errno if SCR registers are inaccessible.

Name

`ata_timing_cycle2mode` — find xfer mode for the specified cycle duration

Synopsis

```
u8 ata_timing_cycle2mode (unsigned int xfer_shift, int cycle);
```

Arguments

xfer_shift ATA_SHIFT_* value for transfer type to examine.

cycle cycle duration in ns

Description

Return matching xfer mode for *cycle*. The returned mode is of the transfer type specified by *xfer_shift*. If *cycle* is too slow for *xfer_shift*, 0xff is returned. If *cycle* is faster than the fastest known mode, the fastest mode is returned.

LOCKING

None.

RETURNS

Matching xfer_mode, 0xff if no match found.

Name

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

Synopsis

```
int ata_do_set_mode (struct ata_link * link, struct ata_device **  
r_failed_dev);
```

Arguments

link link on which timings will be programmed

r_failed_dev out parameter for failed device

Description

Standard implementation of the function used to tune and set ATA device disk transfer mode (PIO3, UDMA6, etc.). If `ata_dev_set_mode` fails, pointer to the failing device is returned in *r_failed_dev*.

LOCKING

PCI/etc. bus probe sem.

RETURNS

0 on success, negative errno otherwise

Name

`ata_wait_after_reset` — wait for link to become ready after reset

Synopsis

```
int ata_wait_after_reset (struct ata_link * link, unsigned long deadline, int (*check_ready) (struct ata_link *link));
```

Arguments

| | |
|--------------------|------------------------------------|
| <i>link</i> | link to be waited on |
| <i>deadline</i> | deadline jiffies for the operation |
| <i>check_ready</i> | callback to check link readiness |

Description

Wait for *link* to become ready after reset.

LOCKING

EH context.

RETURNS

0 if *link* is ready before *deadline*; otherwise, -errno.

Name

sata_link_debounce — debounce SATA phy status

Synopsis

```
int sata_link_debounce (struct ata_link * link, const unsigned long *  
params, unsigned long deadline);
```

Arguments

link ATA link to debounce SATA phy status for

params timing parameters { interval, duration, timeout } in msec

deadline deadline jiffies for the operation

Description

Make sure SStatus of *link* reaches stable state, determined by holding the same value where DET is not 1 for *duration* polled every *interval*, before *timeout*. Timeout constraints the beginning of the stable state. Because DET gets stuck at 1 on some controllers after hot unplugging, this functions waits until timeout then returns 0 if DET is stable at 1.

timeout is further limited by *deadline*. The sooner of the two is used.

LOCKING

Kernel thread context (may sleep)

RETURNS

0 on success, -errno on failure.

Name

sata_link_resume — resume SATA link

Synopsis

```
int sata_link_resume (struct ata_link * link, const unsigned long *  
params, unsigned long deadline);
```

Arguments

link ATA link to resume SATA

params timing parameters { interval, duration, timeout } in msec

deadline deadline jiffies for the operation

Description

Resume SATA phy *link* and debounce it.

LOCKING

Kernel thread context (may sleep)

RETURNS

0 on success, -errno on failure.

Name

`sata_link_scr_lpm` — manipulate SControl IPM and SPM fields

Synopsis

```
int sata_link_scr_lpm (struct ata_link * link, enum ata_lpm_policy policy, bool spm_wakeup);
```

Arguments

link ATA link to manipulate SControl for

policy LPM policy to configure

spm_wakeup initiate LPM transition to active state

Description

Manipulate the IPM field of the SControl register of *link* according to *policy*. If *policy* is `ATA_LPM_MAX_POWER` and *spm_wakeup* is `true`, the SPM field is manipulated to wake up the link. This function also clears `PHYRDY_CHG` before returning.

LOCKING

EH context.

RETURNS

0 on success, -errno otherwise.

Name

`ata_std_prereset` — prepare for reset

Synopsis

```
int ata_std_prereset (struct ata_link * link, unsigned long deadline);
```

Arguments

link ATA link to be reset

deadline deadline jiffies for the operation

Description

link is about to be reset. Initialize it. Failure from prereset makes libata abort whole reset sequence and give up that port, so prereset should be best-effort. It does its best to prepare for reset sequence but if things go wrong, it should just whine, not fail.

LOCKING

Kernel thread context (may sleep)

RETURNS

0 on success, -errno otherwise.

Name

`sata_link_hardreset` — reset link via SATA phy reset

Synopsis

```
int sata_link_hardreset (struct ata_link * link, const unsigned long
* timing, unsigned long deadline, bool * online, int (*check_ready)
(struct ata_link *));
```

Arguments

| | |
|--------------------|--|
| <i>link</i> | link to reset |
| <i>timing</i> | timing parameters { interval, duratinon, timeout } in msec |
| <i>deadline</i> | deadline jiffies for the operation |
| <i>online</i> | optional out parameter indicating link onlineness |
| <i>check_ready</i> | optional callback to check link readiness |

Description

SATA phy-reset *link* using DET bits of SControl register. After hardreset, link readiness is waited upon using `ata_wait_ready` if *check_ready* is specified. LLDs are allowed to not specify *check_ready* and wait itself after this function returns. Device classification is LLD's responsibility.

**online* is set to one iff reset succeeded and *link* is online after reset.

LOCKING

Kernel thread context (may sleep)

RETURNS

0 on success, -errno otherwise.

Name

sata_std_hardreset — COMRESET w/o waiting or classification

Synopsis

```
int sata_std_hardreset (struct ata_link * link, unsigned int * class,  
unsigned long deadline);
```

Arguments

| | |
|-----------------|------------------------------------|
| <i>link</i> | link to reset |
| <i>class</i> | resulting class of attached device |
| <i>deadline</i> | deadline jiffies for the operation |

Description

Standard SATA COMRESET w/o waiting or classification.

LOCKING

Kernel thread context (may sleep)

RETURNS

0 if link offline, -EAGAIN if link online, -errno on errors.

Name

`ata_std_postreset` — standard postreset callback

Synopsis

```
void ata_std_postreset (struct ata_link * link, unsigned int * classes);
```

Arguments

link the target `ata_link`

classes classes of attached devices

Description

This function is invoked after a successful reset. Note that the device might have been reset more than once using different reset methods before `postreset` is invoked.

LOCKING

Kernel thread context (may sleep)

Name

`ata_dev_set_feature` — Issue SET FEATURES - SATA FEATURES

Synopsis

```
unsigned int ata_dev_set_feature (struct ata_device * dev, u8 enable,  
u8 feature);
```

Arguments

dev Device to which command will be sent

enable Whether to enable or disable the feature

feature The sector count represents the feature to set

Description

Issue SET FEATURES - SATA FEATURES command to device *dev* on port *ap* with sector count

LOCKING

PCI/etc. bus probe sem.

RETURNS

0 on success, AC_ERR_* mask otherwise.

Name

`ata_std_qc_defer` — Check whether a qc needs to be deferred

Synopsis

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

Arguments

qc ATA command in question

Description

Non-NCQ commands cannot run with any other command, NCQ or not. As upper layer only knows the queue depth, we are responsible for maintaining exclusion. This function checks whether a new command *qc* can be issued.

LOCKING

`spin_lock_irqsave(host lock)`

RETURNS

`ATA_DEFER_*` if deferring is needed, 0 otherwise.

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 lock)`

Name

`ata_qc_complete` — Complete an active ATA command

Synopsis

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

Arguments

qc Command to complete

Description

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

Refrain from calling this function multiple times when successfully completing multiple NCQ commands. `ata_qc_complete_multiple` should be used instead, which will properly update IRQ expect state.

LOCKING

`spin_lock_irqsave(host lock)`

Name

`ata_qc_complete_multiple` — Complete multiple qcs successfully

Synopsis

```
int ata_qc_complete_multiple (struct ata_port * ap, u32 qc_active);
```

Arguments

ap port in question

qc_active new qc_active mask

Description

Complete in-flight commands. This functions is meant to be called from low-level driver's interrupt routine to complete requests normally. `ap->qc_active` and *qc_active* is compared and commands are completed accordingly.

Always use this function when completing multiple NCQ commands from IRQ handlers instead of calling `ata_qc_complete` multiple times to keep IRQ expect status properly in sync.

LOCKING

`spin_lock_irqsave(host lock)`

RETURNS

Number of completed commands on success, -errno otherwise.

Name

`sata_scr_valid` — test whether SCRs are accessible

Synopsis

```
int sata_scr_valid (struct ata_link * link);
```

Arguments

link ATA link to test SCR accessibility for

Description

Test whether SCRs are accessible for *link*.

LOCKING

None.

RETURNS

1 if SCRs are accessible, 0 otherwise.

Name

`sata_scr_read` — read SCR register of the specified port

Synopsis

```
int sata_scr_read (struct ata_link * link, int reg, u32 * val);
```

Arguments

link ATA link to read SCR for

reg SCR to read

val Place to store read value

Description

Read SCR register *reg* of *link* into **val*. This function is guaranteed to succeed if *link* is `ap->link`, the cable type of the port is SATA and the port implements `->scr_read`.

LOCKING

None if *link* is `ap->link`. Kernel thread context otherwise.

RETURNS

0 on success, negative `errno` on failure.

Name

sata_scr_write — write SCR register of the specified port

Synopsis

```
int sata_scr_write (struct ata_link * link, int reg, u32 val);
```

Arguments

link ATA link to write SCR for

reg SCR to write

val value to write

Description

Write *val* to SCR register *reg* of *link*. This function is guaranteed to succeed if *link* is ap->link, the cable type of the port is SATA and the port implements ->scr_read.

LOCKING

None if *link* is ap->link. Kernel thread context otherwise.

RETURNS

0 on success, negative errno on failure.

Name

`sata_scr_write_flush` — write SCR register of the specified port and flush

Synopsis

```
int sata_scr_write_flush (struct ata_link * link, int reg, u32 val);
```

Arguments

link ATA link to write SCR for

reg SCR to write

val value to write

Description

This function is identical to `sata_scr_write` except that this function performs flush after writing to the register.

LOCKING

None if *link* is `ap->link`. Kernel thread context otherwise.

RETURNS

0 on success, negative errno on failure.

Name

`ata_link_online` — test whether the given link is online

Synopsis

```
bool ata_link_online (struct ata_link * link);
```

Arguments

link ATA link to test

Description

Test whether *link* is online. This is identical to `ata_phys_link_online` when there's no slave link. When there's a slave link, this function should only be called on the master link and will return true if any of M/S links is online.

LOCKING

None.

RETURNS

True if the port online status is available and online.

Name

`ata_link_offline` — test whether the given link is offline

Synopsis

```
bool ata_link_offline (struct ata_link * link);
```

Arguments

link ATA link to test

Description

Test whether *link* is offline. This is identical to `ata_phys_link_offline` when there's no slave link. When there's a slave link, this function should only be called on the master link and will return true if both M/S links are offline.

LOCKING

None.

RETURNS

True if the port offline status is available and offline.

Name

`ata_host_suspend` — suspend host

Synopsis

```
int ata_host_suspend (struct ata_host * host, pm_message_t mesg);
```

Arguments

host host to suspend

mesg PM message

Description

Suspend *host*. Actual operation is performed by port suspend.

Name

`ata_host_resume` — resume host

Synopsis

```
void ata_host_resume (struct ata_host * host);
```

Arguments

host host to resume

Description

Resume *host*. Actual operation is performed by port resume.

Name

`ata_host_alloc` — allocate and init basic ATA host resources

Synopsis

```
struct ata_host * ata_host_alloc (struct device * dev, int max_ports);
```

Arguments

dev generic device this host is associated with

max_ports maximum number of ATA ports associated with this host

Description

Allocate and initialize basic ATA host resources. LLD calls this function to allocate a host, initializes it fully and attaches it using `ata_host_register`.

max_ports ports are allocated and `host->n_ports` is initialized to *max_ports*. The caller is allowed to decrease `host->n_ports` before calling `ata_host_register`. The unused ports will be automatically freed on registration.

RETURNS

Allocate ATA host on success, NULL on failure.

LOCKING

Inherited from calling layer (may sleep).

Name

`ata_host_alloc_pinfo` — alloc host and init with `port_info` array

Synopsis

```
struct ata_host * ata_host_alloc_pinfo (struct device * dev, const
struct ata_port_info *const * ppi, int n_ports);
```

Arguments

dev generic device this host is associated with

ppi array of ATA `port_info` to initialize host with

n_ports number of ATA ports attached to this host

Description

Allocate ATA host and initialize with info from *ppi*. If NULL terminated, *ppi* may contain fewer entries than *n_ports*. The last entry will be used for the remaining ports.

RETURNS

Allocate ATA host on success, NULL on failure.

LOCKING

Inherited from calling layer (may sleep).

Name

`ata_slave_link_init` — initialize slave link

Synopsis

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

Arguments

ap port to initialize slave link for

Description

Create and initialize slave link for *ap*. This enables slave link handling on the port.

In libata, a port contains links and a link contains devices. There is single host link but if a PMP is attached to it, there can be multiple fan-out links. On SATA, there's usually a single device connected to a link but PATA and SATA controllers emulating TF based interface can have two - master and slave.

However, there are a few controllers which don't fit into this abstraction too well - SATA controllers which emulate TF interface with both master and slave devices but also have separate SCR register sets for each device. These controllers need separate links for physical link handling (e.g. onlineness, link speed) but should be treated like a traditional M/S controller for everything else (e.g. command issue, softreset).

`slave_link` is libata's way of handling this class of controllers without impacting core layer too much. For anything other than physical link handling, the default host link is used for both master and slave. For physical link handling, separate *ap->slave_link* is used. All dirty details are implemented inside libata core layer. From LLD's POV, the only difference is that `prereset`, `hardreset` and `postreset` are called once more for the slave link, so the reset sequence looks like the following.

`prereset(M) -> prereset(S) -> hardreset(M) -> hardreset(S) -> softreset(M) -> postreset(M) -> postreset(S)`

Note that `softreset` is called only for the master. `Softreset` resets both M/S by definition, so `SRST` on master should handle both (the standard method will work just fine).

LOCKING

Should be called before host is registered.

RETURNS

0 on success, `-errno` on failure.

Name

`ata_host_start` — start and freeze ports of an ATA host

Synopsis

```
int ata_host_start (struct ata_host * host);
```

Arguments

host ATA host to start ports for

Description

Start and then freeze ports of *host*. Started status is recorded in `host->flags`, so this function can be called multiple times. Ports are guaranteed to get started only once. If `host->ops` isn't initialized yet, its set to the first non-dummy port ops.

LOCKING

Inherited from calling layer (may sleep).

RETURNS

0 if all ports are started successfully, `-errno` otherwise.

Name

`ata_host_init` — Initialize a host struct for sas (ipr, libsas)

Synopsis

```
void ata_host_init (struct ata_host * host, struct device * dev, struct  
ata_port_operations * ops);
```

Arguments

host host to initialize

dev device host is attached to

ops port_ops

Name

`ata_host_register` — register initialized ATA host

Synopsis

```
int ata_host_register (struct ata_host * host, struct scsi_host_template  
* sht);
```

Arguments

host ATA host to register

sht template for SCSI host

Description

Register initialized ATA host. *host* is allocated using `ata_host_alloc` and fully initialized by LLD. This function starts ports, registers *host* with ATA and SCSI layers and probe registered devices.

LOCKING

Inherited from calling layer (may sleep).

RETURNS

0 on success, -errno otherwise.

Name

`ata_host_activate` — start host, request IRQ and register it

Synopsis

```
int ata_host_activate (struct ata_host * host, int irq, irq_handler_t
irq_handler, unsigned long irq_flags, struct scsi_host_template * sht);
```

Arguments

| | |
|--------------------|--|
| <i>host</i> | target ATA host |
| <i>irq</i> | IRQ to request |
| <i>irq_handler</i> | <code>irq_handler</code> used when requesting IRQ |
| <i>irq_flags</i> | <code>irq_flags</code> used when requesting IRQ |
| <i>sht</i> | <code>scsi_host_template</code> to use when registering the host |

Description

After allocating an ATA host and initializing it, most libata LLDs perform three steps to activate the host - start host, request IRQ and register it. This helper takes necessary arguments and performs the three steps in one go.

An invalid IRQ skips the IRQ registration and expects the host to have set polling mode on the port. In this case, *irq_handler* should be NULL.

LOCKING

Inherited from calling layer (may sleep).

RETURNS

0 on success, -errno otherwise.

Name

`ata_host_detach` — Detach all ports of an ATA host

Synopsis

```
void ata_host_detach (struct ata_host * host);
```

Arguments

host Host to detach

Description

Detach all ports of *host*.

LOCKING

Kernel thread context (may sleep).

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 occurred. Detach all ports. Resource release is handled via devres.

LOCKING

Inherited from PCI layer (may sleep).

Name

`ata_platform_remove_one` — Platform layer callback for device removal

Synopsis

```
int ata_platform_remove_one (struct platform_device * pdev);
```

Arguments

pdev Platform device that was removed

Description

Platform layer indicates to libata via this hook that hot-unplug or module unload event has occurred. Detach all ports. Resource release is handled via devres.

LOCKING

Inherited from platform layer (may sleep).

Name

`ata_msleep` — ATA EH owner aware msleep

Synopsis

```
void ata_msleep (struct ata_port * ap, unsigned int msecs);
```

Arguments

ap ATA port to attribute the sleep to

msecs duration to sleep in milliseconds

Description

Sleeps *msecs*. If the current task is owner of *ap*'s EH, the ownership is released before going to sleep and reacquired after the sleep is complete. IOW, other ports sharing the *ap*->host will be allowed to own the EH while this task is sleeping.

LOCKING

Might sleep.

Name

`ata_wait_register` — wait until register value changes

Synopsis

```
u32 ata_wait_register (struct ata_port * ap, void __iomem * reg, u32
mask, u32 val, unsigned long interval, unsigned long timeout);
```

Arguments

| | |
|-----------------|--|
| <i>ap</i> | ATA port to wait register for, can be NULL |
| <i>reg</i> | IO-mapped register |
| <i>mask</i> | Mask to apply to read register value |
| <i>val</i> | Wait condition |
| <i>interval</i> | polling interval in milliseconds |
| <i>timeout</i> | timeout in milliseconds |

Description

Waiting for some bits of register to change is a common operation for ATA controllers. This function reads 32bit LE IO-mapped register *reg* and tests for the following condition.

$(*reg \& \text{mask}) \neq \text{val}$

If the condition is met, it returns; otherwise, the process is repeated after *interval_msec* until timeout.

LOCKING

Kernel thread context (may sleep)

RETURNS

The final register value.

Name

`sata_lpm_ignore_phy_events` — test if PHY event should be ignored

Synopsis

```
bool sata_lpm_ignore_phy_events (struct ata_link * link);
```

Arguments

link Link receiving the event

Description

Test whether the received PHY event has to be ignored or not.

RETURNS

True if the event has to be ignored.

Chapter 5. libata Core Internals

Name

`ata_dev_phys_link` — find physical link for a device

Synopsis

```
struct ata_link * ata_dev_phys_link (struct ata_device * dev);
```

Arguments

dev ATA device to look up physical link for

Description

Look up physical link which *dev* is attached to. Note that this is different from *dev->link* only when *dev* is on slave link. For all other cases, it's the same as *dev->link*.

LOCKING

Don't care.

RETURNS

Pointer to the found physical link.

Name

`ata_force_cbl` — force cable type according to `libata.force`

Synopsis

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

Arguments

ap ATA port of interest

Description

Force cable type according to `libata.force` and whine about it. The last entry which has matching port number is used, so it can be specified as part of device force parameters. For example, both “a:40c,1.00:udma4” and “1.00:40c,udma4” have the same effect.

LOCKING

EH context.

Name

`ata_force_link_limits` — force link limits according to `libata.force`

Synopsis

```
void ata_force_link_limits (struct ata_link * link);
```

Arguments

link ATA link of interest

Description

Force link flags and SATA spd limit according to `libata.force` and whine about it. When only the port part is specified (e.g. 1:), the limit applies to all links connected to both the host link and all fan-out ports connected via PMP. If the device part is specified as 0 (e.g. 1.00:), it specifies the first fan-out link not the host link. Device number 15 always points to the host link whether PMP is attached or not. If the controller has slave link, device number 16 points to it.

LOCKING

EH context.

Name

`ata_force_xfermask` — force xfermask according to `libata.force`

Synopsis

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

Arguments

dev ATA device of interest

Description

Force `xfer_mask` according to `libata.force` and whine about it. For consistency with link selection, device number 15 selects the first device connected to the host link.

LOCKING

EH context.

Name

`ata_force_horkage` — force horkage according to `libata.force`

Synopsis

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

Arguments

dev ATA device of interest

Description

Force horkage according to `libata.force` and whine about it. For consistency with link selection, device number 15 selects the first device connected to the host link.

LOCKING

EH context.

Name

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

Synopsis

```
int ata_rwcmd_protocol (struct ata_taskfile * tf, struct ata_device *  
dev);
```

Arguments

tf command to examine and configure

dev device *tf* belongs to

Description

Examine the device configuration and *tf->flags* to calculate the proper read/write commands and protocol to use.

LOCKING

caller.

Name

`ata_tf_read_block` — Read block address from ATA taskfile

Synopsis

```
u64 ata_tf_read_block (const struct ata_taskfile * tf, struct ata_device  
* dev);
```

Arguments

tf ATA taskfile of interest

dev ATA device *tf* belongs to

LOCKING

None.

Read block address from *tf*. This function can handle all three address formats - LBA, LBA48 and CHS. *tf*->protocol and flags select the address format to use.

RETURNS

Block address read from *tf*.

Name

`ata_build_rw_tf` — Build ATA taskfile for given read/write request

Synopsis

```
int ata_build_rw_tf (struct ata_taskfile * tf, struct ata_device * dev,
u64 block, u32 n_block, unsigned int tf_flags, unsigned int tag);
```

Arguments

| | |
|-----------------|---------------------------------|
| <i>tf</i> | Target ATA taskfile |
| <i>dev</i> | ATA device <i>tf</i> belongs to |
| <i>block</i> | Block address |
| <i>n_block</i> | Number of blocks |
| <i>tf_flags</i> | RW/FUA etc... |
| <i>tag</i> | tag |

LOCKING

None.

Build ATA taskfile *tf* for read/write request described by *block*, *n_block*, *tf_flags* and *tag* on *dev*.

RETURNS

0 on success, -ERANGE if the request is too large for *dev*, -EINVAL if the request is invalid.

Name

`ata_read_native_max_address` — Read native max address

Synopsis

```
int ata_read_native_max_address (struct ata_device * dev, u64 * max_sectors);
```

Arguments

dev target device

max_sectors out parameter for the result native max address

Description

Perform an LBA48 or LBA28 native size query upon the device in question.

RETURNS

0 on success, -EACCES if command is aborted by the drive. -EIO on other errors.

Name

`ata_set_max_sectors` — Set max sectors

Synopsis

```
int ata_set_max_sectors (struct ata_device * dev, u64 new_sectors);
```

Arguments

dev target device

new_sectors new max sectors value to set for the device

Description

Set max sectors of *dev* to *new_sectors*.

RETURNS

0 on success, -EACCES if command is aborted or denied (due to previous non-volatile SET_MAX) by the drive, -EIO on other errors.

Name

`ata_hpa_resize` — Resize a device with an HPA set

Synopsis

```
int ata_hpa_resize (struct ata_device * dev);
```

Arguments

dev Device to resize

Description

Read the size of an LBA28 or LBA48 disk with HPA features and resize it if required to the full size of the media. The caller must check the drive has the HPA feature set enabled.

RETURNS

0 on success, -errno on failure.

Name

`ata_dump_id` — IDENTIFY DEVICE info debugging output

Synopsis

```
void ata_dump_id (const u16 * id);
```

Arguments

id IDENTIFY DEVICE page to dump

Description

Dump selected 16-bit words from the given IDENTIFY DEVICE page.

LOCKING

caller.

Name

`ata_exec_internal_sg` — execute libata internal command

Synopsis

```
unsigned ata_exec_internal_sg (struct ata_device * dev, struct ata_task-  
file * tf, const u8 * cdb, int dma_dir, struct scatterlist * sgl, un-  
signed int n_elem, unsigned long timeout);
```

Arguments

| | |
|----------------|---|
| <i>dev</i> | Device to which the command is sent |
| <i>tf</i> | Taskfile registers for the command and the result |
| <i>cdb</i> | CDB for packet command |
| <i>dma_dir</i> | Data transfer direction of the command |
| <i>sgl</i> | sg list for the data buffer of the command |
| <i>n_elem</i> | Number of sg entries |
| <i>timeout</i> | Timeout in msecs (0 for default) |

Description

Executes libata internal command with timeout. *tf* contains command on entry and result on return. Timeout and error conditions are reported via return value. No recovery action is taken after a command times out. It's caller's duty to clean up after timeout.

LOCKING

None. Should be called with kernel context, might sleep.

RETURNS

Zero on success, `AC_ERR_*` mask on failure

Name

`ata_exec_internal` — execute libata internal command

Synopsis

```
unsigned ata_exec_internal (struct ata_device * dev, struct ata_taskfile  
* tf, const u8 * cdb, int dma_dir, void * buf, unsigned int buflen,  
unsigned long timeout);
```

Arguments

| | |
|----------------|---|
| <i>dev</i> | Device to which the command is sent |
| <i>tf</i> | Taskfile registers for the command and the result |
| <i>cdb</i> | CDB for packet command |
| <i>dma_dir</i> | Data transfer direction of the command |
| <i>buf</i> | Data buffer of the command |
| <i>buflen</i> | Length of data buffer |
| <i>timeout</i> | Timeout in msecs (0 for default) |

Description

Wrapper around `ata_exec_internal_sg` which takes simple buffer instead of sg list.

LOCKING

None. Should be called with kernel context, might sleep.

RETURNS

Zero on success, `AC_ERR_*` mask on failure

Name

`ata_pio_mask_no_iordy` — Return the non IORDY mask

Synopsis

```
u32 ata_pio_mask_no_iordy (const struct ata_device * adev);
```

Arguments

adev ATA device

Description

Compute the highest mode possible if we are not using iordy. Return -1 if no iordy mode is available.

Name

`ata_dev_read_id` — Read ID data from the specified device

Synopsis

```
int ata_dev_read_id (struct ata_device * dev, unsigned int * p_class,
unsigned int flags, u16 * id);
```

Arguments

| | |
|----------------|--|
| <i>dev</i> | target device |
| <i>p_class</i> | pointer to class of the target device (may be changed) |
| <i>flags</i> | ATA_READID_* flags |
| <i>id</i> | buffer to read IDENTIFY data into |

Description

Read ID data from the specified device. `ATA_CMD_ID_ATA` is performed on ATA devices and `ATA_CMD_ID_ATAPI` on ATAPI devices. This function also issues `ATA_CMD_INIT_DEV_PARAMS` for pre-ATA4 drives.

FIXME

`ATA_CMD_ID_ATA` is optional for early drives and right now we abort if we hit that case.

LOCKING

Kernel thread context (may sleep)

RETURNS

0 on success, -errno otherwise.

Chapter 6. libata SCSI translation/ emulation

Name

`ata_sas_port_destroy` — Destroy a SATA port allocated by `ata_sas_port_alloc`

Synopsis

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

Arguments

ap SATA port to destroy

Name

`ata_scsi_dev_rescan` — initiate `scsi_rescan_device`

Synopsis

```
void ata_scsi_dev_rescan (struct work_struct * work);
```

Arguments

work Pointer to ATA port to perform `scsi_rescan_device`

Description

After ATA pass thru (SAT) commands are executed successfully, libata need to propagate the changes to SCSI layer.

LOCKING

Kernel thread context (may sleep).

- 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 is 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 | <p>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.</p> <p>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.</p> |
| 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 (seldom 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

Name

`ich_pata_cable_detect` — Probe host controller cable detect info

Synopsis

```
int ich_pata_cable_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).

Name

`piix_pata_prereset` — prereset for PATA host controller

Synopsis

```
int piix_pata_prereset (struct ata_link * link, unsigned long deadline);
```

Arguments

link Target link

deadline deadline jiffies for the operation

LOCKING

None (inherited from caller).

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 Drive in question

Description

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

LOCKING

None (inherited from caller).

Name

`do_pata_set_dmamode` — Initialize host controller PATA PIO timings

Synopsis

```
void do_pata_set_dmamode (struct ata_port * ap, struct ata_device *  
adev, int isich);
```

Arguments

ap Port whose timings we are configuring

adev Drive in question

isich set if the chip is an ICH device

Description

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

LOCKING

None (inherited from caller).

Name

`piix_set_dmamode` — Initialize host controller PATA DMA 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 MW/UDMA mode for device, in host controller PCI config space.

LOCKING

None (inherited from caller).

Name

`ich_set_dmamode` — Initialize host controller PATA DMA timings

Synopsis

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

Arguments

ap Port whose timings we are configuring

adev um

Description

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

LOCKING

None (inherited from caller).

Name

`piix_check_450nx_errata` — Check for problem 450NX setup

Synopsis

```
int piix_check_450nx_errata (struct pci_dev * ata_dev);
```

Arguments

ata_dev the PCI device to check

Description

Check for the present of 450NX errata #19 and errata #25. If they are found return an error code so we can turn off DMA

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

Name

`sil_set_mode` — wrap `set_mode` functions

Synopsis

```
int sil_set_mode (struct ata_link * link, struct ata_device ** r_failed);
```

Arguments

link link to set up

r_failed returned device when we fail

Description

Wrap the libata method for device setup as after the setup we need to inspect the results and do some configuration work

Name

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

Synopsis

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

Arguments

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).