SATA

Driver Design Document

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Swaminathan subrathanam	Generic ATA/ATAPI Driver	Nov 2008	
0.2	Ravi B	Added support for SATA	Feb 2009	

Texas Instruments makes no implied or expressed warranties in this document and is not responsible for the products based from this document

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

This document explains the serial ATA driver software design for SATA subsystem of Freon SOC. This covers the high level architecture of ATA stack for BIOS and SATA driver design.

1.1 Purpose & Scope

This document explains high level architecture of ATA/ATAPI Driver design and AHCI Compatible Serial ATA controller driver design.

1.2 Terms & Abbreviations

Term	Description
API	Application Programming Interface
SATA	Serial ATA
ATA	Advanced Technical Attachment
AHCI	Advanced Host Controller Interface
HBA	Host Bus adopter , referred for AHCI host controller
ATAPI	ATA Packet Interface

1.3 References

#	Document	Description
1	SATA specification	SATA Specification 1.0a and 2.6
2	AHCI Specification	AHCI Specification Revision 1.1
3	ATA/ATAPI standard	ATA/ATAPI-6 standard
4	SATASS Functional spec.	C6748/OMAPL138 SOC SATA subsystem specification

2 System overview

The figure shows the Serial ATA drives are connected to AHCI compatible Serial ATA Host Bus Adopter. The AHCI compliance SATA HBA consists one or many ports, SATA hard disk drive are directly connected to AHCI port interface or through Port Multiplier support, where multiple SATA HDD can be connected to single port. The Driver or SATA driver interfaces with operating system in turn with the user application.

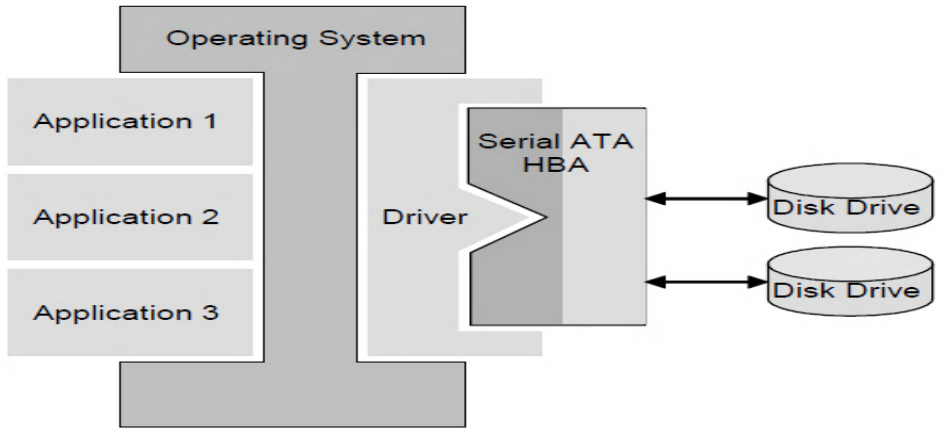

Figure 1 Serial ATA Connectivity

2.1 Freon SATA Subsystem

The AHCI compliance Freon SATA Subsystem provides the following features.

- Serial ATA 1.5Gbps and 3Gbps speeds [2]
- Integrated TI SERDES [6,7]
- Integrated Rx and Tx data buffers
- Supports all SATA power management features
- Internal DMA Engine
- Hardware-assisted Native Command Queuing (NCQ) for up to 32 entries
- Supports port multiplier with command-based switching
- Activity LED support
- Mechanical Presence Switch
- Cold Presence Detect

2.2 Serial ATA System overview

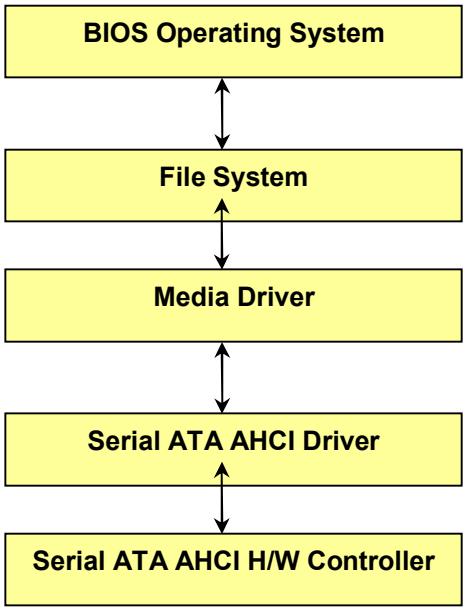

Figure 2 Serial ATA System Overview

The figure illustrate the Serial ATA systems view,

Serial ATA Driver

The Serial ATA AHCI Controller Hardware is managed by the serial ATA Driver Module. This provides an ATA/ATAPI abstraction for any type of storage device (Hard Disk, CD, DVD etc) on the SATA bus connected to the SATA AHCI Controller Hardware.

Media Driver

The "Media Driver" manages the specific type of Storage device such as Hard Disk, CD, DVD, Compact Flash etc. The "Media Driver" uses the ATA/ATAPI Driver to interact with the corresponding device. It remains transparent to the ATA/ATAPI protocol due to the abstraction provided by the ATA/ATAPI Driver. It implements the device specific functionalities such as "Read, Write, Erase, Special operations".

The Media Driver interfaces with File System above to present a device interface.

File System

File System component of the system implements the OS view of the Device Contents at the logical level while Media Driver implements the physical level.

Operating System

OS interacts with the Application and the rest of the system.

3 Design Considerations

3.1 Assumptions & Dependencies

The ATA Driver stack is re-usable component, originally implemented for other platforms and ported for the following operating system

- BIOS 5

The "ATA/ATPI Interface Driver" is implemented basically supports various IDE controllers basically of PATA devices, now is adopted to support the SATA devices.

The following assumptions are made during the Serial ATA Driver Software design:

- ATA/ATAPI-6 standard is used for the design of ATA/ATAPI Driver.
- "Serial ATA Controller Driver" design is applicable for AHCI compatible HBA.
- A Media Driver is available to manage the device and it uses the ATA/ATAPI Driver to communicate with the ATA/ATAPI device.
- The Media Driver will comply with the interfaces of the ATA/ATAPI Interface Driver.
- Implementation will be done only in ANSI C, however no assumption should be made about the compiler.

Proper documentation for ATA/ATAPI Interface will be provided for Media Driver developer's reference.

3.2 General Constraints

- Should be easily portable to Pr OS, Linux, Nucleus+, BIOS
- Must be ANSI ATA/ATAPI-6 standards compliant
- Implement ATA/ATAPI performance enhancements options such as ATA/ATAPI device queues across all supported ATA/ATAPI devices (if the same is supported by the device) as per customers requirement.

3.3 Hardware Limitations

Serial ATA AHCI compliance synopsis core for Freon subsystem support SATA-I Supports speed of 1.5Gbps and SATA-II supports 3.0Gpbs, the approximate throughput 150MB/sec and 300MB/sec respectively for SATA-I/SATA-II.

3.4 Design Goals & Guidelines

Following design goals and guidelines have been considered for designing Serial ATA/ATAPI Driver. These design goals and guidelines are applicable at a macro level and there could be necessary deviations in micro level.

- Portability across any available RTOS
- Phased ATA/ATAPI standard support based on the requirements
- Flexibility to support newer ATA/ATAPI standards/features
- Quality strategies conforming to TII process
- Hooks for easy debugging and testing
- Performance and scalability
- Predictable memory utilization (Avoid using dynamic memory allocations as much as possible) and memory optimization keeping in mind future additions.
- Reusable and modular design of software.

4 System Architecture

The following figure illustrates the system aspect of usage of the ATA/ATAPI driver in more detail.

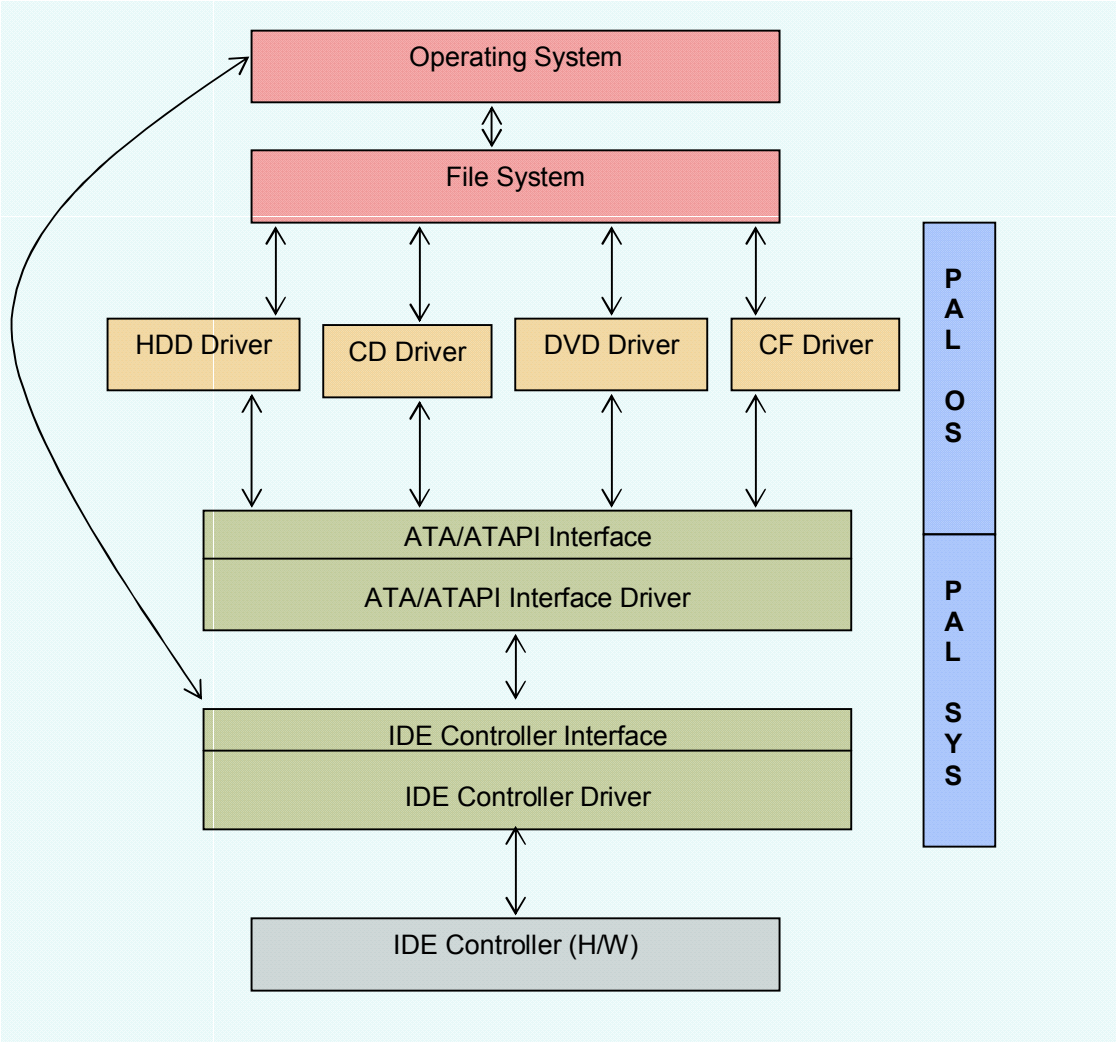

Figure 3 System Architecture

As shown in the figure above the different components that form the complete solution are

ATA/ATAPI Interface Driver

This module implements the ATA/ATAPI abstraction for any ATA/ATAPI compliant "Media Device" (Hard Disk, CD etc.). This module exports a set of ATA/ATAPI interface's to the "Media Driver". This layer contains the ATA/ATAPI standard intelligence and hence the ATA/ATAPI context of the addressed device. It is responsible for ensuring the completion of ATA/ATAPI commands on the addressed

device. It will return the status of the commands queued by the "Media Driver" back to it with or without data.

This layer will also implement the performance enhancement options such as SATA Native Command queues, overlapped operations etc. on the addressed device. Thus the "Media Driver" is agnostic of the underlying scheduling of its requests and only has to manage the sequence of requests to its addressed device.

It will schedule the requests of the addressed devices in an efficient manner so as to be fair to all the addressed devices on the same ATA bus.

SATA AHCI Controller Driver

This module will manage the interaction with the underlying AHCI Compliance SATA Host Bus Adaptor or SATA IDE controller hardware. It exposes a set of SATA IDE Controller interfaces which the ATA/ATAPI Interface Driver uses to interact with this module. By doing so we ensure an easily portable ATA/ATAPI driver across various IDE Controller hardware. The driver will translate the ATA/ATAPI Interface Driver requests to corresponding hardware requests (through register FIS) Will configure and manage the IDE Controller hardware optimally for ensuring a high level of throughput for any give system scenario.

This driver will work with the ATA/ATAPI Interface driver to ensure a high performance data path between the addressed ATA/ATAPI device and the corresponding "Media Driver".

5 Detail Design

5.1 ATA/ATAPI Interface Driver

This module provides the Media Driver with a set of ATA/ATAPI interfaces. The Media Driver uses these interfaces to communicate with the ATA/ATAPI device. The interface driver uses the underlying SATA AHCI Controller interface to queue the request of the Media driver after forming the ATA/ATAPI command for that request. The ATA/ATAPI Interface driver shall be further decomposed as follows

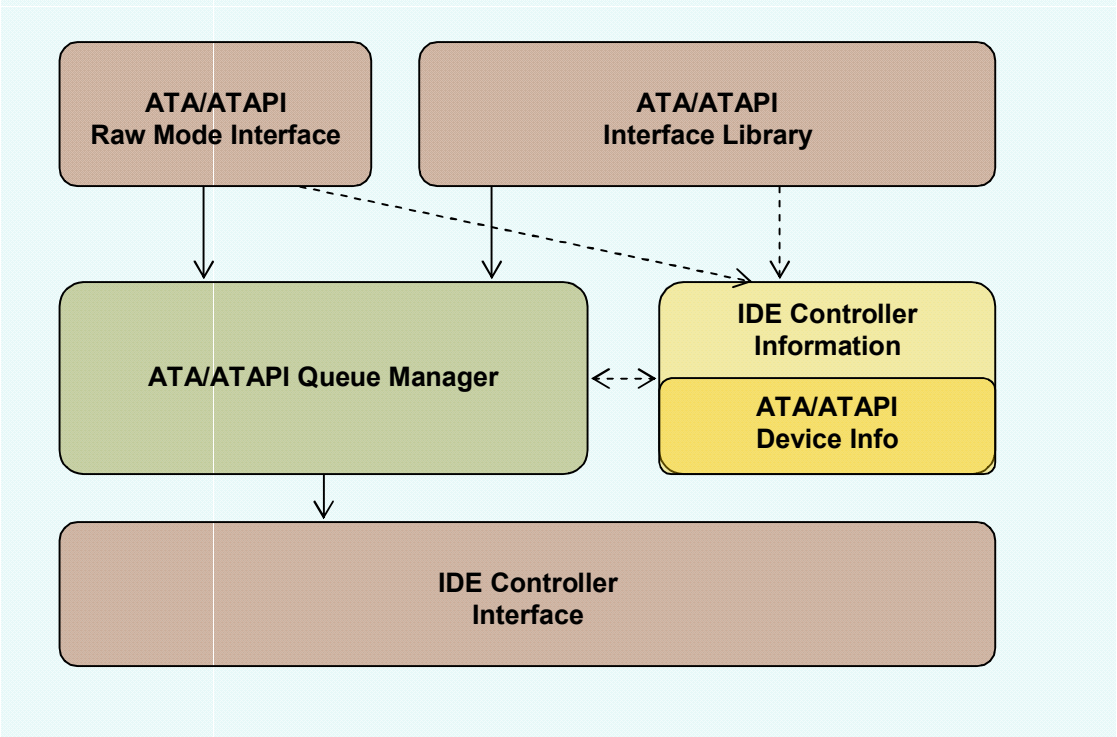

Figure 4 ATA/ATAPI Interface Driver Modules

The IDE controller used here is AHCI compliance Serial ATA Host Bus Adopter.

ATA/ATAPI Interface Driver provides the IDE Controller Driver with the following interface for registering itself with the ATA/ATAPI Interface Driver. Every IDE Controller Driver has to register with the ATA/ATAPI Interface Driver so that the devices on those Controllers can be accessed by the Media Drivers.

int ATA_IDE_Register (IDE_INT_Ops_t *pIdeOps, void *pHandle);	
To register the IDE Controller Driver with the ATA/ATAPI Interface Driver.	
pIdeOps [IN]	IDE Controller or AHCI HBA operations

pHandle [IN]	Handle to be used for future IDE transactions
Return: Int	Success/Failure

ATA/ATAPI Interface Driver provides the Media Driver with the following interface for registering itself as the driver for an ATA/ATAPI device. The device ATA/ATAPI requests are completed and returned to the registered Media Driver through the given callback.

int ATA_MED_Register (ATA_DEV_Ident_t *pDeviceIdent, **ATA_MED_Callback_t MediaCallback,** **ATA_INT_Ops_t **pAtaIntOps,** **Ptr *pHandle);**	
Identify the requested device on the ATA Bus and register as the media driver for that device with ATA Interface.	
pDeviceIdent [IN]	Device Indentification Information
pMediaCallBack [IN]	Callback routine for the registered Media Driver
pAtaIntOps [OUT]	ATA Interface API's
pHandle [OUT]	Handle to device (to be used for future transactions)
Return int	Success/Error

ATA/ATAPI Unregister provides the Media driver a means for closing the device. But this is permitted only when are no out standing requests on the device else an error is returned. If there are no out standing requests pending on this device then this device is unregistered by the ATA/ATAPI driver and the device state is set to be “Available”. Once this is done if another Media Driver wants to register for this device the same can be done.

int ATA_MED_UnRegister (Ptr Handle);	
Free the requested device on the ATA Bus. If the device has a active request we return ATA_ERR_GEN.	
pHandle [IN]	Handle to device
Return int	Success/Error

int ATA_Init (void) : Init the ATA/ATAPI Interface Driver.

int ATA_DeInit (void): De-Init the ATA/ATAPI Interface Driver.

5.1.1 ATA/ATAPI Raw Mode Interface

This provides a raw ATA/ATAPI interface. The interface driver will not form the command rather will expect a formatted ATA/ATAPI command to be given to it. In the raw mode the interface driver will just queue the request into the ATA/ATAPI device queue (without actually assembling the command) through the **ATA/ATAPI Queue Manager**. This mode can be used by the Media driver to communicate device specific ATA/ATAPI command which the generic ATA/ATAPI interface library will not provide.

5.1.2 ATA/ATAPI Interface Library

This library will assemble the requested ATA/ATAPI command depending on the request from the Media Driver. This module provides some generic interfaces and options for each interface for the Media Driver. The Media Driver can select a particular mode of an ATA command and submit the request along with supporting arguments (such as buffers).

There are other API's to set the device to operate in low power modes. Accordingly when IO request arrives the commands are appropriately handled by this library.

After the command and its associated arguments are assembled into an IO request it is queued into the specified device ATA/ATAPI device queue through the **ATA/ATAPI Queue Manager**.

5.1.3 ATA/ATAPI Queue Manager

This module will schedule the IO requests from the device/ATA Queues for submit ion to the SATA AHCI Controller. This will interact with the AHCI Controller hardware through the IDE Controller interface. If the free command slot available on SATA HBA then Queue Manager will submit the pending/current request on any device on that interface (SATA Bus). It keeps track of availability of free command slots for submitting future IO requests on the underlying Controller interfaces.

This also implements the ATA Queuing and Overlapped feature set implementations for ensuring high performance operation of the ATA devices. It communicates the result of the completion of the IO requests through "Call Back's" registered by the Media Driver's on that device.

Currently the Queue Manager is implemented in run to completion mode. In this mode the request submitted on a particular device are completed before the requests queued on the next device on the same SATA Bus are selected for execution.

5.2 AHCI compliance SATA Controller Driver

This is the AHCI compliance Serial ATA controller driver, this provides set of APIs to manage the AHCI SATA controller.

The API's to interface are –

API	Use
ahciInitSata	Initialize the AHCI SATA subsystem controller.
sataReadPio	Read bytes in PIO mode (Not supported for SATA)
sataWritePio	Write bytes in PIO Mode (Not supported for SATA)
sataSubmitReq	Submit a ATA command to the SATA device
sataCmdStatus	Return the command completion status
sataDevSetMode	Set the device operation modes and the timing registers of the IDE controller driver.
sataEndDma	End the DMA operation on the device.
sataStartDma	Start the DMA operation on the device.
sataRegAtaHandler	For registering a ATA protocol layer.

6 Data Flow

6.1 System Boot Up

When the system gets booted up the following sequence of Init is performed

- SATA AHCI Driver or IDE Driver Init
- Media Driver Init

6.1.1 ATA_Init

The ATA_Init initializes the ATA Interface Driver's internal data structures (Like creation of Device Queues, Power Mode Init etc.).

6.1.2 IDE Driver Init

During the IDE driver Init the IDE Driver initializes its own internal data structures, allocates memory etc. It further calls **ATA_IDE_Register ()** to register itself with the ATA Interface Driver. The IDE Driver passes

- IDE Interface Handle
- IDE Interface reference (API)

6.1.3 Media Driver Init

When a Media Driver say HDD Driver Init is called it initializes its internal Data Structures. It further calls the **ATA_MED_Register** (). Media Driver passes the device signature, Call back function pointer as a parameter to **ATA_MED_Register ()**. The "***Device Signature***" will be used by the ATA/ATAPI Interface Driver to

identify the device present on the ATA Bus and the Callback function will be used for returning back the status and events on the registered device.

The ATA/ATAPI Interface Driver uses the IDE Interface API "RegHndlr ()" to register a callback with the IDE Controller Driver. A ATA "IDENTIFY DEVICE" command is assembled and queued to the IDE Controller driver using "**SubmitReq**()" (of IDE Controller) interface through the **ATA/ATAPI driver Queue manager**.

The "SubmitReq()" interface of IDE Controller Driver translates the device i/o request into one or more Frame Information Structures (FIS) in memory , these FIS are similar to taskfile. The FIS are part of command table entries, the command table also includes the PRDT entries (various s/g buffers, lengths) which are used SATA controller hardware once the corresponding command slot is execution is initated. On return the ATA/ATAPI Interface driver synchronously waits to ensure a sequential init of devices. On completion of the Identify command on the requested device the ATA/ATAPI driver looks for command completion status, on error looks for "ATAPI Device Signature". If it matches then "ATAPI IDENTIFY" command is submitted and ATA/ATAPI driver synch waits for command completion. If the command results in a timeout/error then that device is marked as "NOT AVAIL".

On successful completion of ATA/ATAPI IDENTIFY command ATA/ATAPI interface starts processing the returned data. Now the ATA/ATAPI Interface Driver compares the returned device signature with the signature passed from the Media Driver. If a match is found then the ATA/ATAPI interface Driver returns the "**Device handle"**, **"DevOps Pointer"** to be used with future ATA/ATAPI interface calls from the Media Driver. The Media Driver will now register with the "File System" and the OS for a device node representing a mass storage device.

If the device identification signatures do not match then the identify process explained above is repeated on other device's on the ATA Bus until we have identified all the devices. The discovered devices are marked as being **available** while those for which identification match exists are marked as being **registered**.

The devices which are marked as being available will be marked as being registered once the Media drivers register for those drives. The process of probing will not be done in those cases (as they have already been discovered).

6.1.4 Device Operating Mode (DMA)

Based on the identification parameters of the device (collected as the part of the ATA/ATAPI Identify command) and the capabilities of the IDE Controller we can set the best/requested operating mode of the device.

The device reports in its identification data the capabilities that it possesses. Based on this we set the device in a particular operating mode (PIO/DMA). If the device support NCQ or native command Queue, then upto 32 commands can be queued to device based on the maximum number native commands supported by the device. The AHCI compliance SATA controller does not recommend to use PIO mode of operation.

6.2 Device I/O flow

On a device I/O (READ, WRITE etc.) request from the File System the Media Driver breaks it down to a sequence of device commands on the addressed device. The Media Driver uses the interfaces provided by the ATA/ATAPI Interface Driver to request for a specific ATA/ATAPI command to be sent to the device. It uses the

"**Device handle**" along with the "**DevOps**" acquired during the Boot up process (described above) to accomplish the task.

The **ATA/ATAPI Interface library** assembles the requested ATA/ATAPI Command packet and queues it into the ATA/ATAPI device queue through the **ATA/ATAPI Queue Manager**. If the bus is available then the command is directly submitted to the IDE Controller driver through the "SubmitReq()" interface.

On Command completion the AHCI Controller invokes the Callback of the ATA/ATAPI Interface Driver (registered during Boot up). In the Callback the completion status is looked at by the respective Request handler. On a successful command completion the "ATA/ATAPI request handler" calls the associated Media Driver Callback function (registered as a part of the Boot up process). If an error occurs and the error can be retried the request is again retried else the request is returned to the Media Driver (through the Callback) with appropriate error code. The ATA/ATAPI Queue Manager looks for any new request queued in the device queues for submission to the IDE Controller. If so it initiates another I/O cycle.

In the Media Driver Callback the filled buffer (with READ/WRITE data etc.) is communicated to the File System.

6.2.1 ATAPI (Packet Command) Device IO Flow

In the case of ATAPI devices that use ATA PACKET command mechanism to transfer data the requests are routed through the **ATA/ATAPI Raw Mode Interface**. In this case the ATAPI commands are assembled in the Media driver and then using the ATA/ATAPI Raw Mode API's are submitted to the ATA/ATAPI Queue Manager. In the case of ATAPI request an ATA PACKET command is assembled by the ATA/ATAPI RAW Mode API and the submitted ATAPI command is stored along with this ATA PACKET command request. Once this request is submitted to the ATA/ATAPI Queue manager an interrupt is generated to inform that the ATA PACKET command has completed. A packet request handler handles the completion of this request. Once the ATA PACKET command has completed successfully we now write the ATAPI Command (assembled by the Media Driver) to the device (which is now waiting for it). A handler is now set based on the device operating modes to handle the completion of this request. After this, the request is handled as any other normal request submitted through the ATA/ATAPI Interface Library (as described above).

6.2.2 Non-Data Command Flow

In the case of commands that do not accompany data transfers the command completion is identified through the command timeout mechanism. In this case on submission of the command a "Timeout" handler is also set for that request and queued to the ATA/ATAPI Queue Manager. When a timeout happens the "Generic" timeout handler looks for any specific timeout handler set for that request. If so calls that handler to handle the timeout scenario. In the timeout scenario the timeout handler for that request will look into the command completion state. If the command completes then we ask the "Generic" timeout not to cause a Command timeout. If the command did not complete then the timeout handler will initiate the "Generic" timeout handler to cause a Command timeout scenario as described below.

6.3 Command Timeout Flow

When an ATA/ATAPI command submitted to the device does not get completed within the specified time period the bus is considered to be in a Timeout state. Under these conditions a bus reset has to be performed to bring back the bus to a

usable state. The ATA/ATAPI Queue Manager starts a timeout timer on submitting the command to the IDE controller. If the timeout expires before the submitted command completes the command is returned with the "Command Timeout" error to the Media driver. The media driver is expected to initiate a device reset to bring back the device and the bus to fully functional mode.

6.4 Device/Bus Reset Flow

Under Command Timeout or when recovering from Device Power Sleep mode we need to do a Device/Bus/Port reset. The type of reset Device/Bus depends on the whether the device is ATAPI/ATA.

When HBA/Port reset occurs, phy communication shall be re-established with the device through a COMRESET followed by the normal out of band communication sequence defined in Serial ATA. At the end of the reset, the device, if working properly, will send a D2H register FIS, which contains the device signature. When the HBA receives this FIS, it updates PxTFD.STS and PxTFD.ERR register fields, and updates the PxSIG register with the signature.

6.4.1 Software Reset

- When issuing the software reset there should not be any command pending in the command list for execution,
  - software must check PxCMD.ST wait till port to become idle (PxCMD.CR=0), then reset PxCMD.ST=0.
  - If PxTFD.STS.BSY/DRQ is still set due failure condition then a port reset should be attempted or command list override should be used if supported.
- This is used to reset a Serial ATA device by settting the SRST bit in Device control register. Serial ATA has more robust mechanism called COMRESET, also referred to as port reset. This is the preferred mechanism for the error recovery..
- **Step for software reset**
  - Set Cmd1.H2D_FIS.SRST = 1 and CMD1.H2D_FIS.C = 0 .
  - CMD1.CommandTable.CommandHeader[R | C] = 1
    - Issue this CMD1 Register H2D for reset operation , this will reset the device, but device does not send the D2H FIS.
  - Send second Register FIS command CMD2.H2DFIS.SRST = 0, CMD2.H2DFIS.C = 0. CMD2.CommandTable.CmdHeadr[R|C]=0.

6.4.2 Port Reset

- **Steps for Port Reset**
  - wait port to goto idle state (PxCMD.CR=0, PxCMD.ST=0)
  - PxSCTL.DET = 1 (issue COMRESET)
  - Wait for 1ms time.
  - PxSCTL.DET = 0 (the HBA shall reset PxTFD.STS to 7F, upon receiving a COMINIT from the attached device, PxTFD.STS.BSY shall be set to 1.
  - Wait for communication to re-establish
  - PxSERR = #1 (set 1 for all implemented bits)

6.4.3 HBA reset

- If HBA become unusable for multiple ports and a software reset or por reset does not correct the problem, software may reset the entire HBA by setting GHC.HR = 1.

- **Steps for HBA reset**
  - GHC.HR = 1
    - HBA performs internal reset action. This bit will be cleared to 0 by HBA after the reset is complete.
  - Poll for GHC.HR till it becomes 0.
  - If GHC.HR is not cleared to 0 within 1 second the HBA is in hung or locked state.
  - HwInit registers bits in HBA register or port registers are not modified and all the port registers are reset to 0, Port specific registers PxFB,PxCLB are not affected.
  - If spin up device is supported, software should set SUD.
  - Check PxSCLT.DET field, whether device present and link is established with Serial ATA.

- Controller reset : This is the global reset which performs the complete reset of the host bus adopter. Normally this reset will be done at last during, when none of the other two reset recover the interface to operational state.
- Port Reset : This reset the specified port interface of the AHCI controller, other port interfaces will not be affected.
- Device Reset : This is also called soft reset, which performs the reset of the device connected to specified port. The SOFT RESET bit of FIS or taskfile is used to perform the device reset.

6.5 Power Management

There are three power management modes supported by AHCI controller,

- Aggressive
- Slumber
- Partial

The AHCI controller driver supports the HBA to enter to into any of the 3 power management modes, when aggressive mode is used the AHCI controller automatically enter into power down state when there are no outstanding commands are in command list.