AHCI

From OSDev Wiki

Contents

Introduction

AHCI (Advance Host Controller Interface) is developed by Intel to facilitate handling SATA devices. The AHCI specification emphasizes that an AHCI controller (referred to as host bus adapter, or HBA) is designed to be a data movement engine between system memory and SATA devices. It encapsulates SATA devices and provides a standard PCI interface to the host. System designers can easily access SATA drives using system memory and memory mapped registers, without the need for manipulating the annoying task files as IDE do.

An AHCI controller may support up to 32 ports which can attach different SATA devices such as disk drives, port multipliers, or an enclosure management bridge. AHCI supports all native SATA features such as command queueing, hot plugging, power management, etc. To a software developer, an AHCI controller is just a PCI device with bus master capability.

AHCI is a new standard compared to IDE, which has been around for twenty years. There exists little documentation about its programming tips and tricks. Possibly the only available resource is the Intel AHCI specification (see External Links) and some open source operating systems such as Linux. This article shows the minimal steps an OS (not BIOS) should do to put AHCI controller into a workable state, how to identify drives attached, and how to read physical sectors from a SATA disk. To keep concise, many technical details and deep explanations of some data structures have been omitted.

It should be noted that IDE also supports SATA devices and there are still debates about which one, IDE or AHCI, is better. Some tests even show that a SATA disk acts better in IDE mode than AHCI mode. But the common idea is that AHCI performs better and will be the standard PC to SATA interface, though some driver software should be enhanced to fully cultivate AHCI capability.

All the diagrams in this article are copied from the Intel AHCI specification 1.3.

SATA basic

There are at least two SATA standards maintained respectively by T13 (http://www.t13.org) and SATA-IO (http://www.sata-io.org) . The SATA-IO focuses on serial ATA and T13 encompasses traditional parallel ATA specifications as well.

To a software developer, the biggest difference between SATA and parallel ATA is that SATA uses FIS (Frame Information Structure) packet to transport data between host and device, though at hardware level they differ much. An FIS can be viewed as a data set of traditional task files, or an encapsulation of ATA commands. SATA uses the same command set as parallel ATA.

1) FIS types

Following code defines different kinds of FIS specified in Serial ATA Revision 3.0.

```
typedef enum
{
        FIS_TYPE_REG_H2D        = 0x27, // Register FIS - host to device
        FIS_TYPE_REG_D2H        = 0x34, // Register FIS - device to host
        FIS_TYPE_DMA_ACT        = 0x39, // DMA activate FIS - device to host
        FIS_TYPE_DMA_SETUP      = 0x41, // DMA setup FIS - bidirectional
        FIS_TYPE_DATA           = 0x46, // Data FIS - bidirectional
        FIS_TYPE_BIST           = 0x58, // BIST activate FIS - bidirectional
```

```
        FIS_TYPE_PIO_SETUP      = 0x5F, // PIO setup FIS - device to host
        FIS_TYPE_DEV_BITS       = 0xA1, // Set device bits FIS - device to host
} FIS_TYPE;
```

2) Register FIS – Host to Device

A host to device register FIS is used by the host to send command or control to a device. As illustrated in the following data structure, it contains the IDE registers such as command, LBA, device, feature, count and control. An ATA command is constructed in this structure and issued to the device. All reserved fields in an FIS should be cleared to zero.

```
typedef struct tagFIS_REG_H2D
{
        // DWORD 0
        BYTE    fis_type;       // FIS_TYPE_REG_H2D

        BYTE    pmport:4;       // Port multiplier
        BYTE    rsv0:3;         // Reserved
        BYTE    c:1;            // 1: Command, 0: Control

        BYTE    command;        // Command register
        BYTE    featurel;       // Feature register, 7:0

        // DWORD 1
        BYTE    lba0;           // LBA low register, 7:0
        BYTE    lba1;           // LBA mid register, 15:8
        BYTE    lba2;           // LBA high register, 23:16
        BYTE    device;         // Device register

        // DWORD 2
        BYTE    lba3;           // LBA register, 31:24
        BYTE    lba4;           // LBA register, 39:32
        BYTE    lba5;           // LBA register, 47:40
        BYTE    featureh;       // Feature register, 15:8

        // DWORD 3
        BYTE    countl;         // Count register, 7:0
        BYTE    counth;         // Count register, 15:8
        BYTE    icc;            // Isochronous command completion
        BYTE    control;        // Control register

        // DWORD 4
        BYTE    rsv1[4];        // Reserved
} FIS_REG_H2D;
```

3) Register FIS – Device to Host

A device to host register FIS is used by the device to notify the host that some ATA register has changed. It contains the updated task files such as status, error and other registers.

```
typedef struct tagFIS_REG_D2H
{
        // DWORD 0
        BYTE    fis_type;    // FIS_TYPE_REG_D2H

        BYTE    pmport:4;    // Port multiplier
        BYTE    rsv0:2;      // Reserved
        BYTE    i:1;         // Interrupt bit
        BYTE    rsv1:1;      // Reserved

        BYTE    status;      // Status register
        BYTE    error;       // Error register

        // DWORD 1
        BYTE    lba0;        // LBA low register, 7:0
```

```
        BYTE    lba1;        // LBA mid register, 15:8
        BYTE    lba2;        // LBA high register, 23:16
        BYTE    device;      // Device register

        // DWORD 2
        BYTE    lba3;        // LBA register, 31:24
        BYTE    lba4;        // LBA register, 39:32
        BYTE    lba5;        // LBA register, 47:40
        BYTE    rsv2;        // Reserved

        // DWORD 3
        BYTE    countl;      // Count register, 7:0
        BYTE    counth;      // Count register, 15:8
        BYTE    rsv3[2];     // Reserved

        // DWORD 4
        BYTE    rsv4[4];     // Reserved
} FIS_REG_D2H;
```

4) Data FIS – Bidirectional

This FIS is used by the host or device to send data payload. The data size can be varied.

```
typedef struct tagFIS_DATA
{
        // DWORD 0
        BYTE    fis_type;       // FIS_TYPE_DATA

        BYTE    pmport:4;       // Port multiplier
        BYTE    rsv0:4;         // Reserved

        BYTE    rsv1[2];        // Reserved

        // DWORD 1 ~ N
        DWORD   data[1];        // Payload
} FIS_DATA;
```

5) PIO Setup – Device to Host

This FIS is used by the device to tell the host that it's about to send or ready to receive a PIO data payload.

```
typedef struct tagFIS_PIO_SETUP
{
        // DWORD 0
        BYTE    fis_type;       // FIS_TYPE_PIO_SETUP

        BYTE    pmport:4;       // Port multiplier
        BYTE    rsv0:1;         // Reserved
        BYTE    d:1;            // Data transfer direction, 1 - device to host
        BYTE    i:1;            // Interrupt bit
        BYTE    rsv1:1;

        BYTE    status;         // Status register
        BYTE    error;          // Error register

        // DWORD 1
        BYTE    lba0;           // LBA low register, 7:0
        BYTE    lba1;           // LBA mid register, 15:8
        BYTE    lba2;           // LBA high register, 23:16
        BYTE    device;         // Device register

        // DWORD 2
        BYTE    lba3;           // LBA register, 31:24
        BYTE    lba4;           // LBA register, 39:32
```

```
        BYTE    lba5;           // LBA register, 47:40
        BYTE    rsv2;           // Reserved

        // DWORD 3
        BYTE    countl;         // Count register, 7:0
        BYTE    counth;         // Count register, 15:8
        BYTE    rsv3;           // Reserved
        BYTE    e_status;       // New value of status register

        // DWORD 4
        WORD    tc;             // Transfer count
        BYTE    rsv4[2];        // Reserved
} FIS_PIO_SETUP;
```

5) DMA Setup – Device to Host

```
typedef struct tagFIS_DMA_SETUP
{
        // DWORD 0
        BYTE    fis_type;       // FIS_TYPE_DMA_SETUP

        BYTE    pmport:4;       // Port multiplier
        BYTE    rsv0:1;         // Reserved
        BYTE    d:1;            // Data transfer direction, 1 - device to host
        BYTE    i:1;            // Interrupt bit
        BYTE    a:1;            // Auto-activate. Specifies if DMA Activate FIS is needed

        BYTE    rsved[2];       // Reserved

        //DWORD 1&2

        QWORD   DMAbufferID;    // DMA Buffer Identifier. Used to Identify DMA buffer in host me

        //DWORD 3
        DWORD   rsvd;           //More reserved

        //DWORD 4
        DWORD   DMAbufOffset;   //Byte offset into buffer. First 2 bits must be 0

        //DWORD 5
        DWORD   TransferCount;  //Number of bytes to transfer. Bit 0 must be 0

        //DWORD 6
        DWORD   resvd;          //Reserved

} FIS_DMA_SETUP;
```

7) Example

This example illustrates the steps to read the Identify data from a device. Error detection and recovery is ignored.

To issue an ATA Identify command to the device, the FIS is constructed at follows.

```
FIS_REG_H2D fis;
memset(&fis, 0, sizeof(FIS_REG_H2D));
fis->fis_type = FIS_TYPE_REG_H2D;
fis->command = ATA_CMD_IDENTIFY;        // 0xEC
fis->device = 0;                        // Master device
fis->c = 1;                             // Write command register
```

After the device receives this FIS and successfully read the 256 words data into its internal buffer, it sends a PIO Setup FIS – Device to Host to tell the host that it's ready to transfer data and the data size (FIS_PIO_SETUP.tc).

After the PIO Setup FIS – Device to Host has been sent correctly, the device sends a DATA FIS to the host which contains the received data payload (FIS_DATA.data).

This scenario is described in SATA revision 3.0 as a PIO data-in command protocol. But an AHCI controller will do the latter two steps for the host. The host software needs only setup and issue the command FIS, and tells the AHCI controller the memory address and size to store the received data. After everything is done, the AHCI controller will issue an interrupt (if enabled) to notify the host to check the data.

Find an AHCI controller

An AHCI controller can be found by enumerating the PCI bus. It has a class id 0x01 (mass storage device) and normally a subclass id 0x06 (serial ATA). The vendor id and device id should also be checked to ensure it's really an AHCI controller.

Determining what mode the controller is in

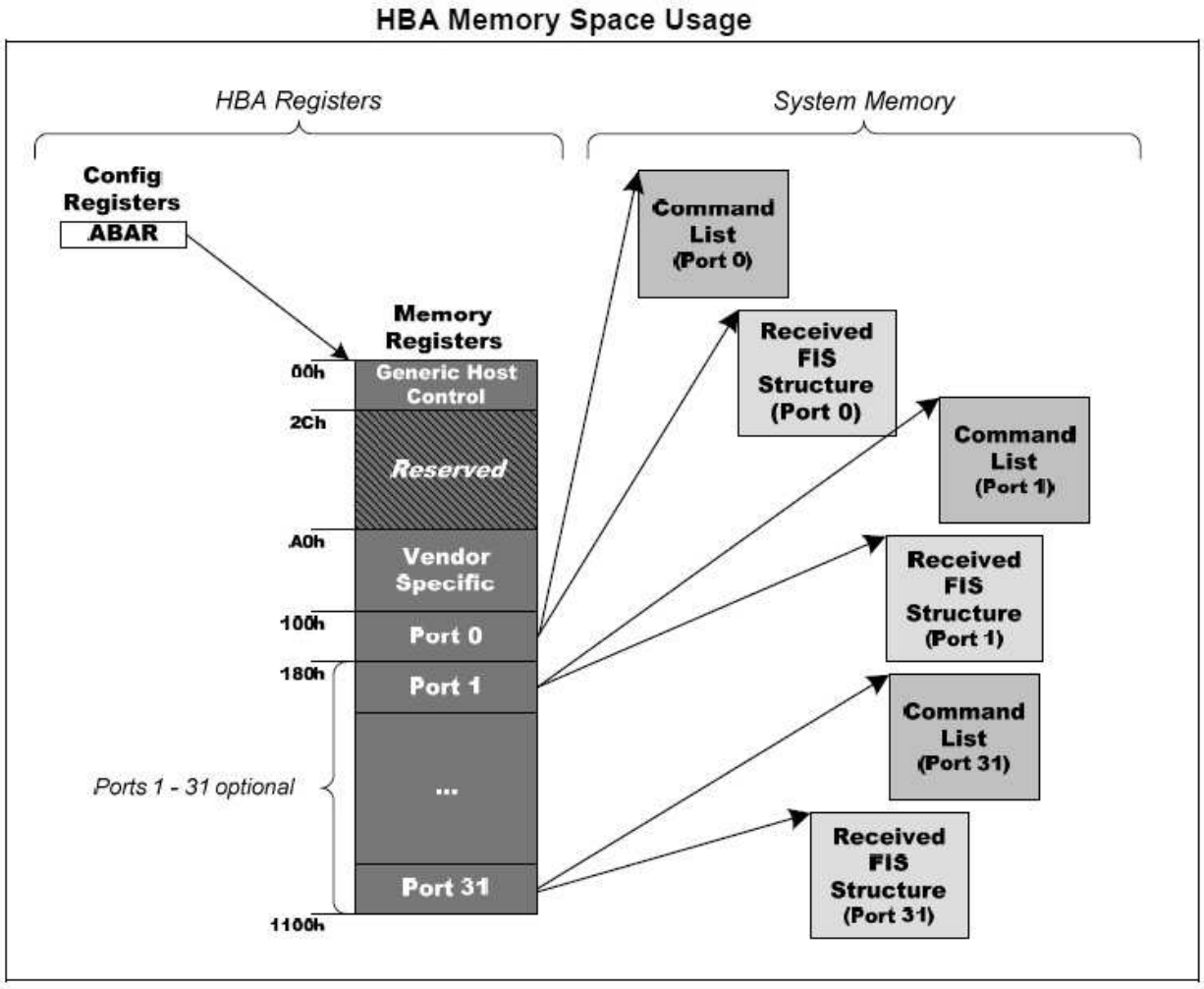
As you may be aware, a SATA controller can either be in IDE emulation mode or in AHCI mode. The problem that enters here is simple:
How to find what mode the controller is in. The documentation is really obscure on this. Perhaps the best way is to initialize a SATA controller as both IDE and AHCI. In this way, as long as you are careful about non-existent ports, you cannot go wrong.

AHCI Registers and Memory Structures

As mentioned above, host communicates with the AHCI controller through system memory and memory mapped registers. The last PCI base address register (BAR[5], header offset 0x24) points to the AHCI base memory, it's called ABAR (AHCI Base Memory Register). All AHCI registers and memories can be located through ABAR. The other PCI base address registers act same as a traditional IDE controller. Some AHCI controller can be configured to simulate a legacy IDE one.

1) HBA memory registers

An AHCI controller can support up to 32 ports. HBA memory registers can be divided into two parts: Generic Host Control registers and Port Control registers. Generic Host Control registers controls the behavior of the whole controller, while each port owns its own set of Port Control registers. The actual ports an AHCI controller supported and implemented can be calculated from the Capacity register (HBA_MEM.cap) and the Port Implemented register (HBA_MEM.pi).



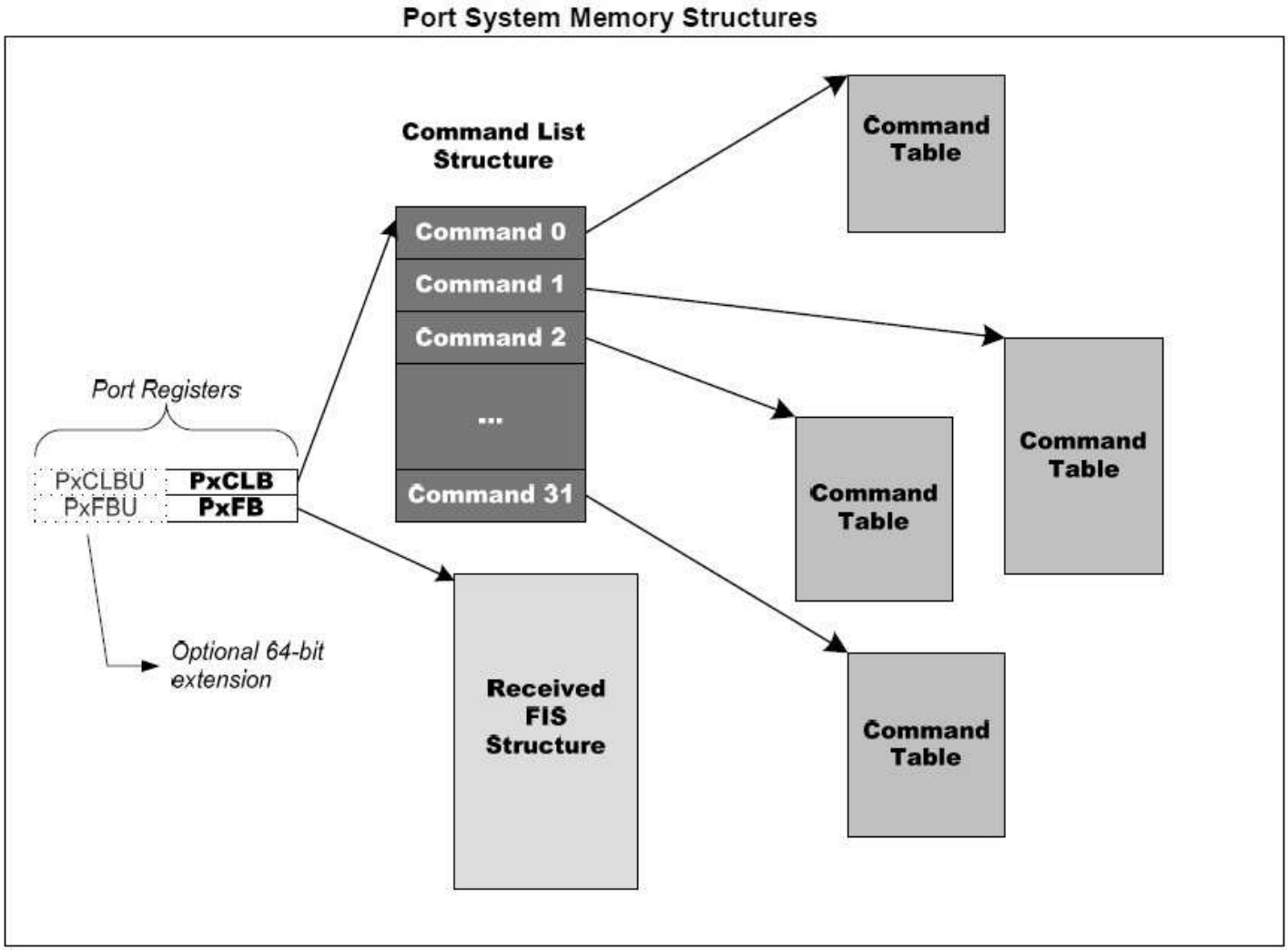

```
typedef volatile struct tagHBA_MEM
{
        // 0x00 - 0x2B, Generic Host Control
        DWORD   cap;            // 0x00, Host capability
        DWORD   ghc;            // 0x04, Global host control
        DWORD   is;             // 0x08, Interrupt status
        DWORD   pi;             // 0x0C, Port implemented
        DWORD   vs;             // 0x10, Version
        DWORD   ccc_ctl;        // 0x14, Command completion coalescing control
        DWORD   ccc_pts;        // 0x18, Command completion coalescing ports
        DWORD   em_loc;         // 0x1C, Enclosure management location
        DWORD   em_ctl;         // 0x20, Enclosure management control
        DWORD   cap2;           // 0x24, Host capabilities extended
        DWORD   bohc;           // 0x28, BIOS/OS handoff control and status

        // 0x2C - 0x9F, Reserved
        BYTE    rsv[0xA0-0x2C];

        // 0xA0 - 0xFF, Vendor specific registers
        BYTE    vendor[0x100-0xA0];

        // 0x100 - 0x10FF, Port control registers
        HBA_PORT        ports[1];       // 1 ~ 32
} HBA_MEM;

typedef volatile struct tagHBA_PORT
{
        DWORD   clb;            // 0x00, command list base address, 1K-byte aligned
        DWORD   clbu;           // 0x04, command list base address upper 32 bits
        DWORD   fb;             // 0x08, FIS base address, 256-byte aligned
        DWORD   fbu;            // 0x0C, FIS base address upper 32 bits
        DWORD   is;             // 0x10, interrupt status
```

```
        DWORD   ie;             // 0x14, interrupt enable
        DWORD   cmd;            // 0x18, command and status
        DWORD   rsv0;           // 0x1C, Reserved
        DWORD   tfd;            // 0x20, task file data
        DWORD   sig;            // 0x24, signature
        DWORD   ssts;           // 0x28, SATA status (SCR0:SStatus)
        DWORD   sctl;           // 0x2C, SATA control (SCR2:SControl)
        DWORD   serr;           // 0x30, SATA error (SCR1:SError)
        DWORD   sact;           // 0x34, SATA active (SCR3:SActive)
        DWORD   ci;             // 0x38, command issue
        DWORD   sntf;           // 0x3C, SATA notification (SCR4:SNotification)
        DWORD   fbs;            // 0x40, FIS-based switch control
        DWORD   rsv1[11];       // 0x44 ~ 0x6F, Reserved
        DWORD   vendor[4];      // 0x70 ~ 0x7F, vendor specific
} HBA_PORT;
```

This memory area should be configured as uncacheable as they are memory mapped hardware registers, not normal prefetchable RAM. For the same reason, the data structures are declared as "volatile" to prevent the compiler from over optimizing the code.

2) Port Received FIS and Command List Memory

Each port can attach a single SATA device. Host sends commands to the device using Command List and device delivers information to the host using Received FIS structure. They are located at HBA_PORT.clb/clbu, and HBA_PORT.fb/fbu. The most important part of AHCI initialization is to set correctly these two pointers and the data structures they point to.



3) Received FIS

There are four kinds of FIS which may be sent to the host by the device as indicated in the following structure declaration. When an FIS has been copied into the host specified memory, an according bit will be set in the Port Interrupt Status register (HBA_PORT.is).

Data FIS – Device to Host is not copied to this structure. Data payload is sent and received through PRDT (Physical Region Descriptor Table) in Command List, as will be introduced later.

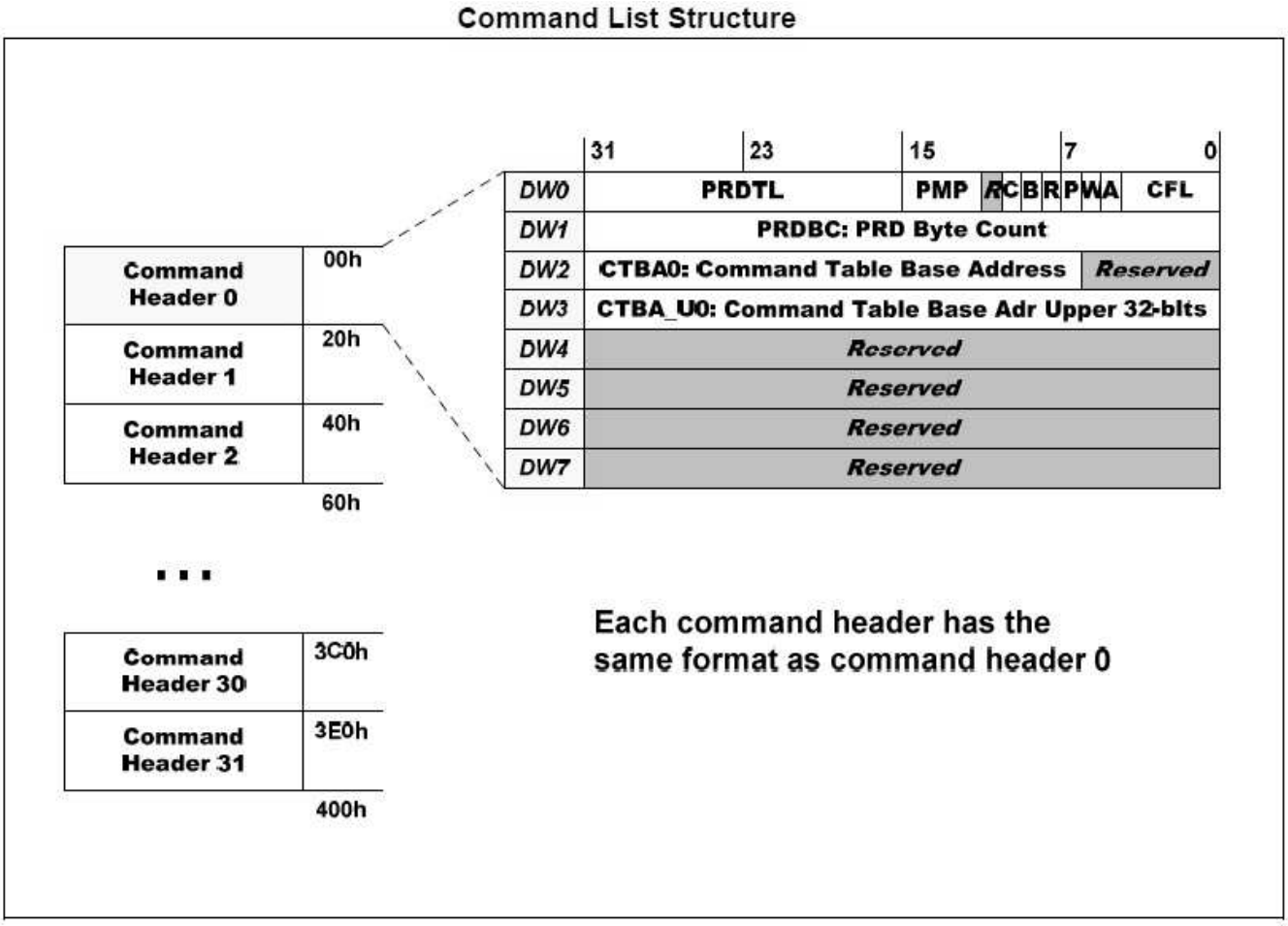
```
typedef volatile struct tagHBA_FIS
{
        // 0x00
        FIS_DMA_SETUP   dsfis;          // DMA Setup FIS
        BYTE            pad0[4];

        // 0x20
        FIS_PIO_SETUP   psfis;          // PIO Setup FIS
        BYTE            pad1[12];

        // 0x40
        FIS_REG_D2H     rfis;           // Register – Device to Host FIS
        BYTE            pad2[4];

        // 0x58
        FIS_DEV_BITS    sdbfis;         // Set Device Bit FIS

        // 0x60
        BYTE            ufis[64];

        // 0xA0
        BYTE            rsv[0x100-0xA0];
} HBA_FIS;
```

4) Command List

Host sends commands to the device through Command List. Command List consists of 1 to 32 command headers, each one is called a slot. Each command header describes an ATA or ATAPI command, including a Command FIS, an ATAPI command buffer and a bunch of Physical Region Descriptor Tables specifying the data payload address and size.

To send a command, the host constructs a command header, and set the according bit in the Port Command Issue register (HBA_PORT.ci). The AHCI controller will automatically send the command to the device and wait for response. If there are some errors, error bits in the Port Interrupt register (HBA_PORT.is) will be set and additional information can be retrieved from the Port Task File register (HBA_PORT.tfd), SStatus register (HBA_PORT.ssts) and SError register (HBA_PORT.serr). If it succeeds, the Command Issue register bit will be cleared and the received data payload, if any, will be copied from the device to the host memory by the AHCI controller.

How many slots a Command List holds can be got from the Host capability register (HBA_MEM.cap). It must be within 1 and 32. SATA supports queued commands to increase throughput. Unlike traditional parallel ATA drive; a SATA drive can process a new command when an old one is still running. With AHCI, a host can send up to 32 commands to device simultaneously.



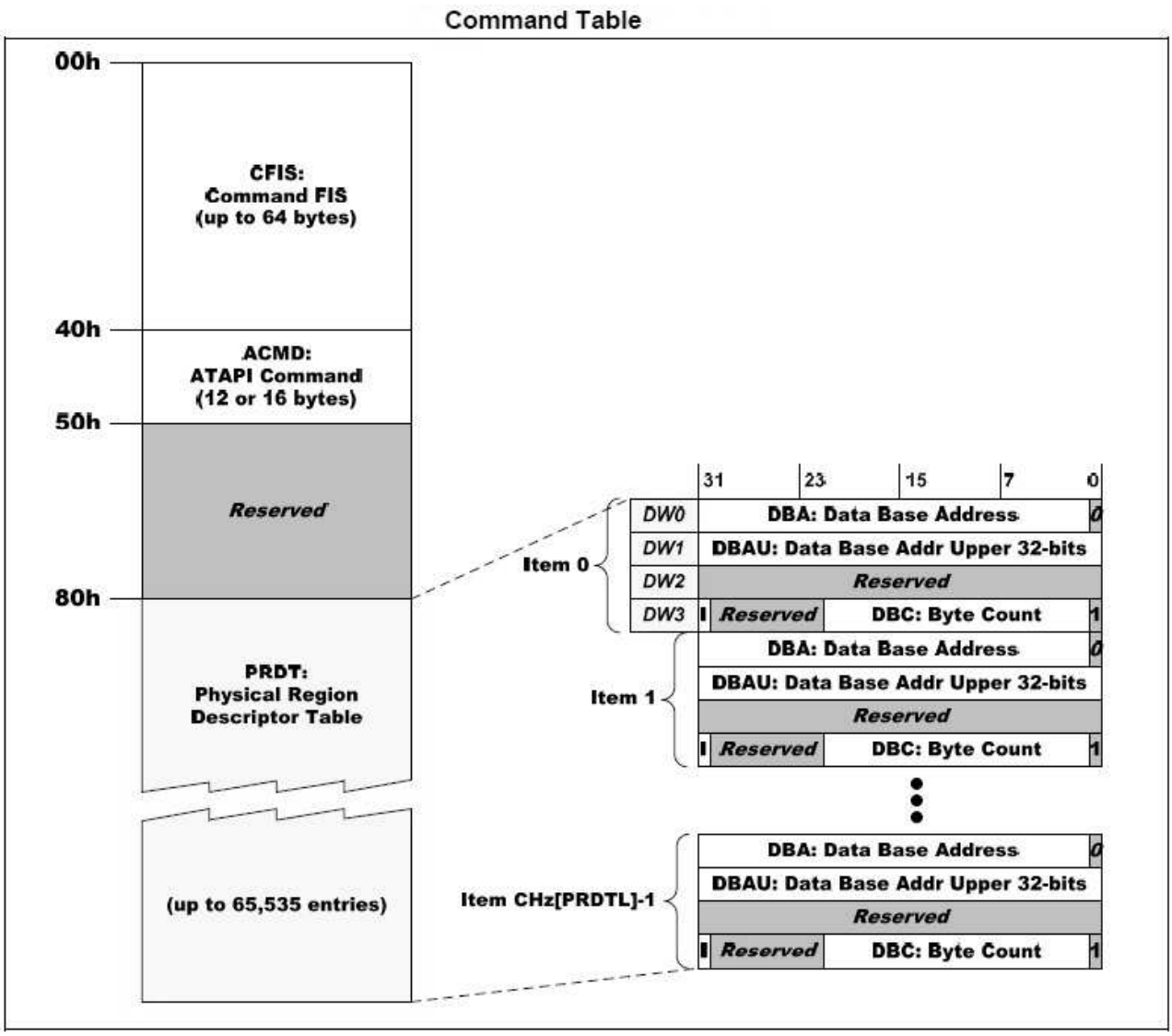

```
typedef struct tagHBA_CMD_HEADER
{
	// DW0
	BYTE	cfl:5;		// Command FIS length in DWORDS, 2 ~ 16
	BYTE	a:1;		// ATAPI
	BYTE	w:1;		// Write, 1: H2D, 0: D2H
	BYTE	p:1;		// Prefetchable

	BYTE	r:1;		// Reset
	BYTE	b:1;		// BIST
	BYTE	c:1;		// Clear busy upon R_OK
	BYTE	rsv0:1;		// Reserved
	BYTE	pmp:4;		// Port multiplier port

	WORD	prdtl;		// Physical region descriptor table length in entries

	// DW1
	volatile
	DWORD	prdbc;		// Physical region descriptor byte count transferred

	// DW2, 3
	DWORD	ctba;		// Command table descriptor base address
	DWORD	ctbau;		// Command table descriptor base address upper 32 bits

	// DW4 - 7
	DWORD	rsv1[4];	// Reserved
} HBA_CMD_HEADER;
```

5) Command Table and Physical Region Descriptor Table

As described above, a command table contains an ATA command FIS, an ATAPI command buffer and a bunch of PRDT (Physical Region Descriptor Table) specifying the data payload address and size.

A command table may have 0 to 65535 PRDT entries. The actual PRDT entries count is set in the command header (HBA_CMD_HEADER.prdtl). As an example, if a host wants to read 100K bytes continuously from a disk, the first half to memory address A1, and the second half to address A2. It must set two PRDT entries, the first PRDT.DBA = A1, and the second PRDT.DBA = A2.

An AHCI controller acts as a PCI bus master to transfer data payload directly between device and system memory.




```
typedef struct tagHBA_CMD_TBL
{
	// 0x00
	BYTE	cfis[64];	// Command FIS

	// 0x40
	BYTE	acmd[16];	// ATAPI command, 12 or 16 bytes

	// 0x50
	BYTE	rsv[48];	// Reserved

	// 0x80
	HBA_PRDT_ENTRY	prdt_entry[1];	// Physical region descriptor table entries, 0 ~ 65535
} HBA_CMD_TBL;

typedef struct tagHBA_PRDT_ENTRY
{
	DWORD	dba;		// Data base address
	DWORD	dbau;		// Data base address upper 32 bits
	DWORD	rsv0;		// Reserved

	// DW3
```

```
        DWORD   dbc:22;         // Byte count, 4M max
        DWORD   rsv1:9;         // Reserved
        DWORD   i:1;            // Interrupt on completion
} HBA_PRDT_ENTRY;
```

Detect attached SATA devices

1) Which port is device attached

As specified in the AHCI specification, firmware (BIOS) should initialize the AHCI controller into a minimal workable state. OS usually needn't reinitialize it from the bottom. Much information is already there when the OS boots.

The Port Implemented register (HBA_MEM.pi) is a 32 bit value and each bit represents a port. If the bit is set, the according port has a device attached, otherwise the port is free.

2) What kind of device is attached

There are four kinds of SATA devices, and their signatures are defined as below. The Port Signature register (HBA_PORT.sig) contains the device signature, just read this register to find which kind of device is attached at the port. Some buggy AHCI controllers may not set the Signature register correctly. The most reliable way is to judge from the Identify data read back from the device.

```
#define SATA_SIG_ATA    0x00000101      // SATA drive
#define SATA_SIG_ATAPI  0xEB140101      // SATAPI drive
#define SATA_SIG_SEMB   0xC33C0101      // Enclosure management bridge
#define SATA_SIG_PM     0x96690101      // Port multiplier

void probe_port(HBA_MEM *abar)
{
        // Search disk in impelemented ports
        DWORD pi = abar->pi;
        int i = 0;
        while (i<32)
        {
                if (pi & 1)
                {
                        int dt = check_type(&abar->ports[i]);
                        if (dt == AHCI_DEV_SATA)
                        {
                                trace_ahci("SATA drive found at port %d\n", i);
                        }
                        else if (dt == AHCI_DEV_SATAPI)
                        {
                                trace_ahci("SATAPI drive found at port %d\n", i);
                        }
                        else if (dt == AHCI_DEV_SEMB)
                        {
                                trace_ahci("SEMB drive found at port %d\n", i);
                        }
                        else if (dt == AHCI_DEV_PM)
                        {
                                trace_ahci("PM drive found at port %d\n", i);
                        }
                        else
                        {
                                trace_ahci("No drive found at port %d\n", i);
                        }
                }

                pi >>= 1;
                i ++;
        }
}

// Check device type
static int check_type(HBA_PORT *port)
```

```
{
        DWORD ssts = port->ssts;

        BYTE ipm = (ssts >> 8) & 0x0F;
        BYTE det = ssts & 0x0F;

        if (det != HBA_PORT_DET_PRESENT)        // Check drive status
                return AHCI_DEV_NULL;
        if (ipm != HBA_PORT_IPM_ACTIVE)
                return AHCI_DEV_NULL;

        switch (port->sig)
        {
        case SATA_SIG_ATAPI:
                return AHCI_DEV_SATAPI;
        case SATA_SIG_SEMB:
                return AHCI_DEV_SEMB;
        case SATA_SIG_PM:
                return AHCI_DEV_PM;
        default:
                return AHCI_DEV_SATA;
        }
}
```

AHCI port memory space initialization

BIOS may have already configured all the necessary AHCI memory spaces. But the OS usually needs to reconfigure them to make them fit its requirements. It should be noted that Command List must be located at 1K aligned memory address and Received FIS be 256 bytes aligned.

Before rebasing Port memory space, OS must wait for current pending commands to finish and tell HBA to stop receiving FIS from the port. Otherwise an accidently incoming FIS may be written into a partially configured memory area. This is done by checking and setting corresponding bits at the Port Command And Status register (HBA_PORT.cmd). The example subroutines stop_cmd() and start_cmd() do the job.

The following example assumes that the HBA has 32 ports implemented and each port contains 32 command slots, and will allocate 8 PRDTs for each command slot.

```
#define AHCI_BASE       0x400000        // 4M

void port_rebase(HBA_PORT *port, int portno)
{
        stop_cmd(port); // Stop command engine

        // Command list offset: 1K*portno
        // Command list entry size = 32
        // Command list entry maxim count = 32
        // Command list maxim size = 32*32 = 1K per port
        port->clb = AHCI_BASE + (portno<<10);
        port->clbu = 0;
        memset((void*)(port->clb), 0, 1024);

        // FIS offset: 32K+256*portno
        // FIS entry size = 256 bytes per port
        port->fb = AHCI_BASE + (32<<10) + (portno<<8);
        port->fbu = 0;
        memset((void*)(port->fb), 0, 256);

        // Command table offset: 40K + 8K*portno
        // Command table size = 256*32 = 8K per port
        HBA_CMD_HEADER *cmdheader = (HBA_CMD_HEADER*)(port->clb);
        for (int i=0; i<32; i++)
        {
                cmdheader[i].prdtl = 8; // 8 prdt entries per command table
                                        // 256 bytes per command table, 64+16+48+16*8
```

```
                // Command table offset: 40K + 8K*portno + cmdheader_index*256
                cmdheader[i].ctba = AHCI_BASE + (40<<10) + (portno<<13) + (i<<8);
                cmdheader[i].ctbau = 0;
                memset((void*)cmdheader[i].ctba, 0, 256);
        }

        start_cmd(port);        // Start command engine
}

// Start command engine
void start_cmd(HBA_PORT *port)
{
        // Wait until CR (bit15) is cleared
        while (port->cmd & HBA_PxCMD_CR);

        // Set FRE (bit4) and ST (bit0)
        port->cmd |= HBA_PxCMD_FRE;
        port->cmd |= HBA_PxCMD_ST;
}

// Stop command engine
void stop_cmd(HBA_PORT *port)
{
        // Clear ST (bit0)
        port->cmd &= ~HBA_PxCMD_ST;

        // Wait until FR (bit14), CR (bit15) are cleared
        while(1)
        {
                if (port->cmd & HBA_PxCMD_FR)
                        continue;
                if (port->cmd & HBA_PxCMD_CR)
                        continue;
                break;
        }

        // Clear FRE (bit4)
        port->cmd &= ~HBA_PxCMD_FRE;
}
```

AHCI & ATAPI

The documentation regarding using the AHCI interface to access an ATAPI device (most likely an optical drive) is rather poorly explained in the specification. However, once you understand that the HBA does most of the work for you it is rather simple. The AHCI/ATAPI method works by issuing the ATA PACKET command (0xA0) instead of the READ (READ is shown in the example below) and populating the ACMD field of the HBA_CMD_TBL with the 12/16 byte ATAPI command and setting the 'a' field to 1 in the HBA_CMD_HEADER which tells the HBA to perform the multi-step process (all done automatically) of transmitting the PACKET command, then sending the ATAPI device the ACMD.

Example - Read hard disk sectors

The code example shows how to read "count" sectors from sector offset "starth:startl" to "buf" with LBA48 mode from HBA "port". Every PRDT entry contains 8K bytes data payload at most.

```
#define ATA_DEV_BUSY 0x80
#define ATA_DEV_DRQ 0x08

BOOL read(HBA_PORT *port, DWORD startl, DWORD starth, DWORD count, WORD *buf)
{
        port->is = (DWORD)-1;           // Clear pending interrupt bits
        int spin = 0; // Spin lock timeout counter
        int slot = find_cmdslot(port);
        if (slot == -1)
                return FALSE;
```

```
	HBA_CMD_HEADER *cmdheader = (HBA_CMD_HEADER*)port->clb;
	cmdheader += slot;
	cmdheader->cfl = sizeof(FIS_REG_H2D)/sizeof(DWORD);	// Command FIS size
	cmdheader->w = 0;		// Read from device
	cmdheader->prdtl = (WORD)((count-1)>>4) + 1;	// PRDT entries count

	HBA_CMD_TBL *cmdtbl = (HBA_CMD_TBL*)(cmdheader->ctba);
	memset(cmdtbl, 0, sizeof(HBA_CMD_TBL) +
		(cmdheader->prdtl-1)*sizeof(HBA_PRDT_ENTRY));

	// 8K bytes (16 sectors) per PRDT
	for (int i=0; i<cmdheader->prdtl-1; i++)
	{
		cmdtbl->prdt_entry[i].dba = (DWORD)buf;
		cmdtbl->prdt_entry[i].dbc = 8*1024;	// 8K bytes
		cmdtbl->prdt_entry[i].i = 1;
		buf += 4*1024;	// 4K words
		count -= 16;	// 16 sectors
	}
	// Last entry
	cmdtbl->prdt_entry[i].dba = (DWORD)buf;
	cmdtbl->prdt_entry[i].dbc = count<<9;	// 512 bytes per sector
	cmdtbl->prdt_entry[i].i = 1;

	// Setup command
	FIS_REG_H2D *cmdfis = (FIS_REG_H2D*)(&cmdtbl->cfis);

	cmdfis->fis_type = FIS_TYPE_REG_H2D;
	cmdfis->c = 1;	// Command
	cmdfis->command = ATA_CMD_READ_DMA_EX;

	cmdfis->lba0 = (BYTE)startl;
	cmdfis->lba1 = (BYTE)(startl>>8);
	cmdfis->lba2 = (BYTE)(startl>>16);
	cmdfis->device = 1<<6;	// LBA mode

	cmdfis->lba3 = (BYTE)(startl>>24);
	cmdfis->lba4 = (BYTE)starth;
	cmdfis->lba5 = (BYTE)(starth>>8);

	cmdfis->countl = LOBYTE(count);
	cmdfis->counth = HIBYTE(count);

	// The below loop waits until the port is no longer busy before issuing a new command
	while ((port->tfd & (ATA_DEV_BUSY | ATA_DEV_DRQ)) && spin < 1000000)
	{
		spin++;
	}
	if (spin == 1000000)
	{
		trace_ahci("Port is hung\n");
		return FALSE;
	}

	port->ci = 1<<slot;	// Issue command

	// Wait for completion
	while (1)
	{
		// In some longer duration reads, it may be helpful to spin on the DPS bit
		// in the PxIS port field as well (1 << 5)
		if ((port->ci & (1<<slot)) == 0)
			break;
		if (port->is & HBA_PxIS_TFES)	// Task file error
		{
			trace_ahci("Read disk error\n");
```

```
                        return FALSE;
                }
        }

        // Check again
        if (port->is & HBA_PxIS_TFES)
        {
                trace_ahci("Read disk error\n");
                return FALSE;
        }

        return TRUE;
}

// Find a free command list slot
int find_cmdslot(HBA_PORT *port)
{
        // If not set in SACT and CI, the slot is free
        DWORD slots = (m_port->sact | m_port->ci);
        for (int i=0; i<cmdslots; i++)
        {
                if ((slots&1) == 0)
                        return i;
                slots >>= 1;
        }
        trace_ahci("Cannot find free command list entry\n");
        return -1;
}
```

External Links

- Serial ATA Advance Host Controller Interface (AHCI) 1.3 (http://www.intel.com/technology/serialata/ahci.htm)
- Serial ATA Revision 3.0 (http://www.sata-io.org)
- ATA8-ACS, ATA8-AAM (http://www.t13.org)
- Haiku's AHCI implementation (https://github.com/haiku/haiku/tree/master/src/add-ons/kernel/busses/scsi/ahci)