X3T10/95-192 R0

ATA Software Programming Interface (ATASPI) Specification

DOS and Windows® Operating Systems

Future Domain Corporation
2801 McGaw Ave
Irvine, CA 92714
Telephone: (714) 253-0400
FAX: (714) 253-0913

Working Draft
Version 0.72
Wednesday, December 20, 1994
FDC P/N 07-00021-000-00

NOTICES

This is a working draft of a proposed specification to be submitted as an American National Standard. As such, this is not a completed standard, has not yet received any form of approval. This specification is subject to change and use of the information contained in this document is done so at your own risk. Until such time that this proposed specification is formally submitted to the X3T10 ANSI committee, Future Domain retains all exclusive rights to edit and control this specification.

Technical Editor

Kevin Calvert
Future Domain Corporation
2801 McGaw Ave.
Irvine, California 92714
USA

Telephone	(714) 253-0522
FAX	(714) 253-0913
Internet ID	KEVINC@FDC.MHS.COMPUSERVE.COM

The most recent version of this specification in Microsoft Word For Windows 2.0 format is available for downloading by logging onto the Future Domain BBS using the following settings:

Phone:	(714) 253-0432
Communication Settings:	2400 -14000,N,8,1 V.32/V.32vbis/V.42/V.42bis
Hours:	24 Hours
File Name	ATASP072.EXE

Patent/Copyright Statement

The developers of this specification have requested those holder's of patents or copyrights that may be required for the implementation of this specification, disclose such patents or copyrights to the publisher. No patent or copyright search has been undertaken at this time in order to identify any patents or copyrights that may apply.

Use Disclaimer

THIS SPECIFICATION IS MADE AVAILABLE WITHOUT CHARGE FOR USE IN DEVELOPMENT OF DEVICE DRIVERS AND APPLICATION SOFTWARE. FUTURE DOMAIN MAKES NO REPRESENTATION OR WARRANTY REGARDING THIS SPECIFICATION OR ANY ITEM BASED ON THIS SPECIFICATION, AND FUTURE DOMAIN DISCLAIMS ALL EXPRESS AND IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND FREEDOM FROM INFRINGEMENT. FUTURE DOMAIN MAKES NO WARRANTY OF ANY KIND THAT ANY ITEM DEVELOPED BASED ON THIS SPECIFICATION WILL NOT INFRINGE ANY COPYRIGHT, PATENT, TRADE SECRET OR OTHER INTELLECTUAL PROPERTY RIGHT OF ANY PERSON OR ENTITY IN ANY COUNTRY.

Table of Contents

1. Introduction 1
1.1. Document References 2
2. ATASPI for DOS Specification 3
2.1. Opening ATASPI 3
2.2. Obtaining the ATASPI Entry Point 3
2.3. Closing ATASPI 3
2.4. Calling ATASPI from DOS 4
2.5. Calling ATASPI for DOS from Windows 5
2.6. Commands Supported by ATASPI Manager 6
2.6.1. ATA Controller Inquiry 6
2.6.2. Get ATA Device Type 7
2.6.3. Execute ATA I/O 8
2.6.4. Abort ATA Request 11
2.6.5. Reset ATA Device 11
2.6.6. Set ATA Controller Parameter 12
2.6.7. Get ATA Disk Drive Information 13
3. ATASPI For Windows Specification 15
3.1. Accessing ATASPI For Windows 15
3.2. Windows ATASPI Functions 15
3.2.1. GetATASPISupportInfo 15
3.2.2. SendATASPICommand 17
3.3. Windows ATASPI Commands 17
3.3.1. ATA Controller Inquiry Command 17
3.3.2. Get ATA Device Type Command 19
3.3.3. Execute ATA I/O Command 20
3.3.4. Abort ATA Request Command 24
3.3.5. Reset ATA Device Command 26
3.4. ATASPI.H 28
4. Document History 32

List of Figures

Figure 1 - ATASPI Manager Environment 2

List of Tables

Table 1 - ATASPI Status Return Values 6
Table 2 - ATA Controller Inquiry Command 7
Table 3 - Get ATA Device Type Command 7
Table 4 - Peripheral Device Type Values for ATAPI Devices 8
Table 5 - Execute ATA I/O Command 8
Table 6 - Execute ATA I/O Request Flags 9
Table 7 - ACB Definition for Task File Structure 11
Table 8 - Abort ATA Request Command 11
Table 9 - Reset ATA Device Command 12
Table 10 - Reset ATA Device Request Flags 12
Table 11 - Set ATA Controller Parameter Command 13
Table 12 - Get ATA Disk Drive Information Command 13
Table 13 - Get ATA Disk Drive Information Drive Flags 13
Table 14 - INT 13h Info bits 14
Table 15 - GetATSPISupportInfo Status Byte return values 16
Table 16 - Valid SendATASPICommand Return Values 17
Table 17 - ATA Controller Inquiry ARB 18
Table 18 - ATA Controller Inquiry ARB Return Values 18
Table 19 - Get ATA Device Type 19
Table 20 - ATA Device Type Values 20
Table 21 - Get ATA Device Type ARB Return Values 20
Table 22 - Execute ATA I/O ARB 22
Table 23 - ARB_Flags 22
Table 24 - Execute ATA I/O ARB Return Values 24
Table 25 - Abort ATA Request Command 25
Table 26 - Abort ATA Request Command Return Values 25
Table 27 - Reset ATA Device ARB 26
Table 28 - Reset ATA Device ARB Return Values 27

1. Introduction

AT Software Programming Interface (ATASPI) specification defines an API used by developers to control IDE, eIDE and ATAPI compliant peripherals in an PC environment. ATASPI is a software oriented interface designed to simplify writing device drivers for ATA/ATAPI peripherals attached to ATA controllers. ATASPI allows all industry standard ATA controllers to be accessed in the same manner, allowing a single device driver to be written for all of the controllers.

Most often, device driver code is written to fill the gap between two interfaces. One interface is generally provided by the operating system, the other by the target hardware. In this case, the target hardware is an all industry standard ATA controller in combination with some ATA/ATAPI peripheral.

In the past, IDE hard disks were the only peripheral type attached on the ATA bus. These disks were being controlled by one host who provided the software support (e.g. system BIOS, option ROM BIOS or Windows Fast Disk ...). Due to this simple configuration, there was no need for complex ATA chips; BIOS and OS drivers designed to directly address the hardware to avoiding unnecessary programming overhead.

If all device drivers followed the same path, a new driver had to be written each time a new controller comes out with a new ATA chip. Such a new driver would take into account the differences between controllers. In well-structured code, this was laborious at best. In poorly structured code, this usually resulted in an almost completely new driver for each new controller.

Another problem often encountered by device driver developers was a lack of knowledge both about how all industry standard hardware worked, and about the lowest levels of ATA/ATAPI protocol. A programmer had to expend considerable effort programming the controller hardware instead of concentrating on programming his ATA/ATAPI device.

Finally, another problem encountered was when two device drivers were both loaded, one might try to take action while the other was already manipulating the hardware. This might cause one or both drivers to fail. The solution to all the problems stated above is a interface that:

- Can handle the low level protocol issues of an ATA controller
- Can be accessed by more than one device driver
- Can provide a consistent interface to device drivers despite the fact that the underlying hardware was completely different from one controller to another.

This solution is an ATASPI Manager that is meant to significantly simply the ATA/ATAPI device driver developer's job by handling all the low level ATA/ATAPI details and providing a consistent interface across different controllers. This lets the programmer concentrate on the important job of implementing a new driver for his new device. Since all accesses to the hardware are handled by the ATASPI Manager, multiple device drivers can use the same controller without conflicting with one another. This allows devices of different types (IDE, eIDE, ATAPI CD-ROM, ATAPI Tape ...) to share the same ATA controller. Figure 1 shows how an ATASPI Manger fits in a typical ATA/ATAPI environment:

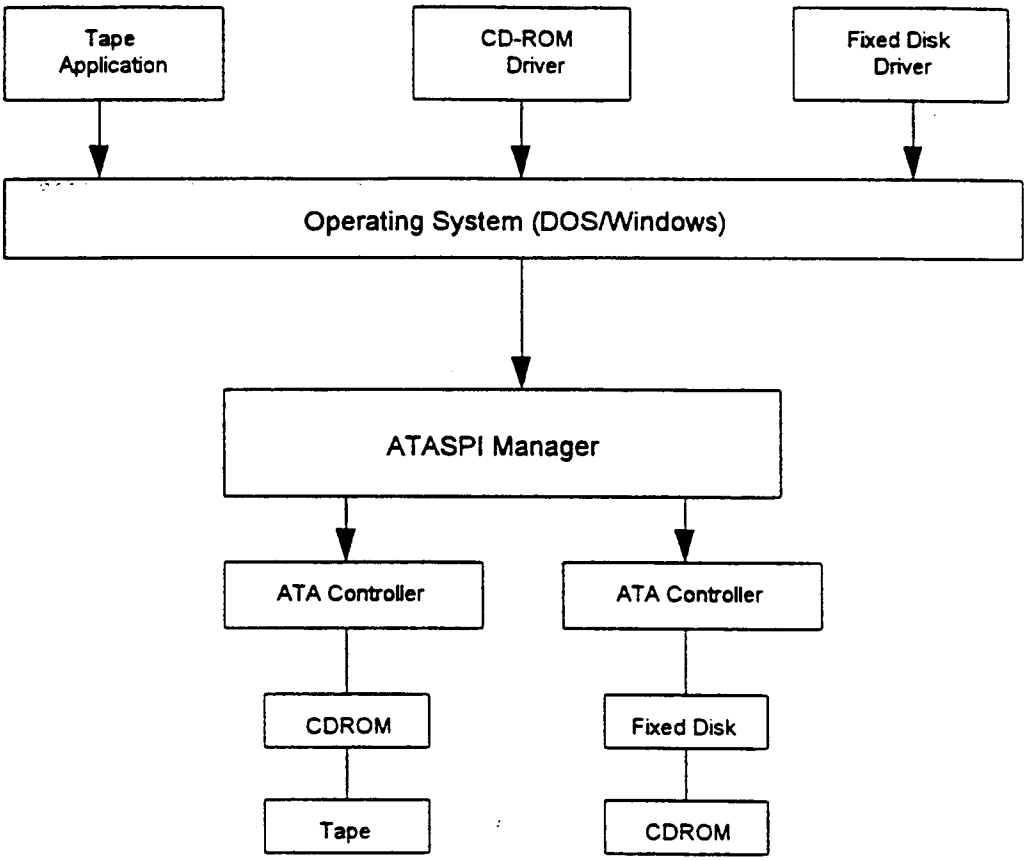

Figure 1 - ATASPI Manager Environment

1.1. Document References

- AT Attachment Interface with Extensions Revision 2d or later ("ATA-2"), ANSI X3 Committee (June 24, 1994)
- ATA Packet Interface for CD-ROM's, SFF-8020 Revision 1.2 or later ("ATAPI CD-ROM"), Small Form Factor Committee (February 24, 1994)
- ATAPI for Streaming Tape Devices Revision C or later ("ATAPI TAPE"), (May 14, 1994)

2. ATASPI for DOS Specification

2.1. Opening ATASPI

Device drivers wishing to access ATASPI must open the driver by performing a DOS INT 21h function call to OPEN A FILE. The following indicates the register condition upon entry and exit of the call to INT 21h

```
On Entry:       AX    =     3D00h
                DS:DX =     Pointer to '$ATAMGR$',0

On Return:      AX    =     File Handle if carry flag is not set
                            Error Code if carry flag is set
```

2.2. Obtaining the ATASPI Entry Point

Device drivers can obtain the entry point to ATASPI by performing the DOS INT 21h function call IOCTL READ. . The following indicates the register condition upon entry and exit of the call to INT 21h.

```
On Entry:       AX    =     4402h
                DS:DX =     Pointer to a 4 byte data buffer
                CX    =     4
                BX    =     File Handle

On Return:      AX    =     Nothing
```

The following data is returned in the buffer pointed to by DS:DX containing the ATASPI entry point.

Offset	Size in Bytes	Direction	Field Description
00h	02h	In	ATASPI entry point Offset[1]
02h	02h	In	ATASPI entry point Segment[2]

Device drivers could then use this ATASPI entry point to send a request to the ATASPI Manager.

2.3. Closing ATASPI

Device drivers wishing to close ATASPI must do it by performing a DOS INT 21h function call CLOSE A FILE. . The following indicates the register condition upon entry and exit of the call to INT 21h.

```
On Entry:       AH    =     3Eh
                BX    =     File Handle

On Return:      AX    =     Error code if carry flag is set
                            Nothing if carry flag is not set
```

1 Offset portion of a 32-bit pointer in Intel X86 format (16-bit offset, 16-bit segment)
2 Segment portion of a 32-bit pointer in Intel X86 format (16-bit offset, 16-bit segment)

2.4. Calling ATASPI from DOS

The following code segment is an example on calling ATASPI.

```
#include <stdio.h>
#include <dos.h>

#define  DOS_OPEN_FILE      0x3D00
#define  DOS_IOCTL_READ     0x4402
#define  DOS_CLOSE_FILE      0x3E

typedef  char (far * function)();

main()
{
   union       REGS        xregister;
   struct      SREGS       sregister;
   char        _far        *ATASPINamePointer;
   char        _far        *ATASPIEntryPointer;
   char                    ATASPIName[] = "$ATAMGR$";
   function                ATASPIEntryPoint;
   int                     FileHandle, i;
   char                    ATAControllerInquiry[64];


   ATASPINamePointer  = (char _far *)(&ATASPIName[0]);
   ATASPIEntryPointer = (char _far *)(&ATASPIEntryPoint);

   // Open the ATASPI Manager

   xregister.x.ax     = DOS_OPEN_FILE;
   sregister.ds       = FP_SEG(ATASPINamePointer);
   xregister.x.dx     = FP_OFF(ATASPINamePointer);
   intdosx(&xregister, &xregister, &sregister);
   if (xregister.x.cflag)
         {
           printf ("Error: ATASPI Manager not found\n");
           return;
         }

   // Get the ATASPI Entry Point

   FileHandle         = xregister.x.ax;
   xregister.x.bx     = FileHandle;
   xregister.x.ax     = DOS_IOCTL_READ;
   sregister.ds       = FP_SEG(ATASPIEntryPointer);
   xregister.x.dx     = FP_OFF(ATASPIEntryPointer);
   xregister.x.cx     = 4;
   intdosx(&xregister, &xregister, &sregister);

   // Close the ATASPI Manager

   xregister.h.ah     = DOS_CLOSE_FILE;
   xregister.x.bx     = FileHandle;
   intdosx(&xregister, &xregister, &sregister);

   // Initialize the ATASPI Controller Inquiry Command
```

```
    for (i=0; i<64; i++)
          ATAControllerInquiry[i] = 0;

    // Issue an ATASPI ATA Controller Inquiry Command

    (*ATASPIEntryPoint)(ATAControllerInquiry);

    if (ATAControllerInquiry[1]==1){
          printf ("ATA Controller Inquiry successful\n");
    }
    else{
          printf ("ATA Controller Inquiry returned with error code
                  %2x\n",ATAControllerInquiry[1]);
    }

    return;
}
```

2.5. Calling ATASPI for DOS from Windows

Microsoft Windows is a Graphical User Interface which runs under DOS as an application. However, writing a device driver or application capable of making ATASPI calls in a Windows environment is not as simple as in the strictly DOS case. The problem arises because ATASPI uses Real Mode Interface and Windows uses the DOS Protected Mode Interface (DPMI). Whereas ATASPI expects a physical Segment and Offset for the ATA Request and the entry point of ATASPI, Windows uses a "Selector" and Offset to address data and code. In order to program correctly in this environment a consortium of companies, Microsoft and Intel among them, has written a specification called the "DOS Protected Mode Interface Specification". The details of the specification are too complex to go into detail here and it is recommended a copy can be obtained from the DPMI committee for programming purposes. However, as a brief overview two steps need to be followed to comply with DPMI:

1. Allocate all ATA Requests and buffers down in Real Mode Memory. This can be accomplished using Window's "GlobalDOSAlloc" routine or using DPMI Interrupt 31h - Function 100h. This makes it possible for the ATASPI module and manager to locate the ATA Request and data buffer using segments and offsets.

2. Call the Real Mode Procedure with Far Return Frame Function (Interrupt 31h - Function 0301h). This makes it possible to call the ATASPI Manager which is a Real Mode Procedure.

Also, see ATASPI For Windows Specification section on page 14.

2.6. Commands Supported by ATASPI Manager

The ATASPI Manager supports the following ATA Request Blocks (ARB):

- ATA Controller Inquiry
- Get ATA Device Type
- Execute ATA I/O
- Abort ATA Request
- Reset ATA Device
- Set ATA Controller Parameter
- Get ATA Disk Drive Information

Upon return, the ATASPI Manager sets the Status byte to one of the following possible values. The actual values returned depend on the command called. Consult each command for valid return values.

Value	Description
00h	Request in progress
01h	Request completed without error
02h	Request aborted by host
04h	Request completed with error
80h	Invalid request
81h	Invalid ATA controller number
82h	ATA Device not installed
83h	ATA Controller/Device busy

Table 1 - ATASPI Status Return Values

Note: If ATASPI ever returns 83h as the status, it is the responsibility of the driver / applications to re-send the request at a later time.

Note: All **reserved** fields defined for each ARB must be initialized to zero.

Note: In the sections that follow, various tables use the following headings and are defined as follows:

Offset	Field starting location in data buffer.
Size in Bytes	The length of this field in bytes.
Direction	Indicates when data is valid. *In* indicates data valid after return, *Out* indicates data to be supplied when ATASPI is called.

2.6.1. ATA Controller Inquiry

ATA Controller Inquiry has two purposes. One is to get the total number of ATA controllers found by the manager and the other is to retrieve Controller Unique Parameters.

To get the total number of ATA controllers, **ATA Controller Number** must be set to 0FFh. Specific information on a controller (as specified in **ATA Controller Number**) is returned in **Controller Unique Parameters.**

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 0
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag = 0
04h	04h	--	Reserved
08h	01h	In	Total Number of ATA Controllers
09h	01h	--	Reserved
0Ah	10h	In	ATA Manager ID
1Ah	10h	In	ATA Controller ID
2Ah	10h	In	Controller Unique Parameters (Vendor Unique)

Table 2 - ATA Controller Inquiry Command

The **Status** byte returned is set to one of the following values :

01h Completed without error
81h Invalid ATA Controller number

The **ATA Manager ID** field contains a 16-byte ASCII string describing the ATA Manager.

The **ATA Controller ID** field contains a 16-byte ASCII string describing the ATA controller.

The **Controller Unique Parameters** field is vendor unique and is defined by controller vendors.

2.6.2. Get ATA Device Type

Get ATA Device Type returns the device type of the specified target. If more detail information is necessary, issue the Inquiry Command through *Execute ATA I/O*.

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 1
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag = 0
04h	04h	--	Reserved
08h	01h	Out	Device ID
09h	01h	--	Reserved
0Ah	01h	In	Peripheral Device Type

Table 3 - Get ATA Device Type Command

The **Device ID** field can be set to :

00h Device 0 (Master device)
01h Device 1 (Slave device)

The **Status** byte returned is set to one of the following values :

01h Device Present and **Peripheral Device Type** field is valid
82h Device Not Present

For ATAPI devices, the **Peripheral Device Type** field contains byte 0 of the Inquiry Command.

Value	Device Type
00h	Direct-access device (e.g. magnetic disk)
01h	Tape device (QIC-121 SCSI Architectural Model)
02h-04h	Reserved
05h	CD-ROM device
06h	Reserved
07h	Optical memory device (e.g. some optical disks)
08h-0Bh	Reserved
0Ch	Tape device (Cost Sensitive Architectural Model)
0Dh-1Eh	Reserved
1Fh	Unknown or no device type

Table 4 - Peripheral Device Type Values for ATAPI Devices

For non-ATAPI devices such as eIDE and IDE disks, the **Peripheral Device Type** field contains 80h.

2.6.3. Execute ATA I/O

The Execute ATA I/O command is used to issue ATA and ATAPI commands to IDE devices. The caller of this command may be informed of the completion of the command through either a Polling or a Posting method.

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 2
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag
04h	04h	--	Reserved
08h	01h	Out	Device ID
09h	01h	--	Reserved
0Ah	04h	Out	Data Transfer Length
0Eh	01h	Out	Sense Allocation Length (N)
0Fh	02h	Out	Data Buffer Pointer (Offset)
11h	02h	Out	Data Buffer Pointer (Segment)
13h	04h	--	Reserved
17h	01h	Out	ACB Length (M)
18h	01h	In	ATA Controller Status
19h	01h	In	Device Status
1Ah	02h	Out	Post Routine Address (Offset)
1Ch	02h	Out	Post Routine Address (Segment)
1Eh	02h	Out	Data Transfer Block Size (in bytes)
20h	20h	--	Reserved for ATASPI Workspace
40h	M	Out	ATA Command Block (ACB)
40h+M	N	In	Sense Allocation Area

Table 5 - Execute ATA I/O Command

The **Status** byte returned is set to one of the following values :

00h	Request in progress
01h	Completed without error
02h	Request aborted by host
04h	Completed with error
81h	Invalid ATA Controller Number

82h ATA Device Not Present
83h ATA Controller/Device busy

The **Request Flags** byte is defined as follows :

7	6	5	4	3	2	1	0
Rsvd	ByteXfer	DUA Bit	Direction Bits		Request Type	Rsvd	Post

Table 6 - Execute ATA I/O Request Flags

The **Post** bit indicates whether we need to call a Post routine upon completion of the request:

0 Disable Posting
1 Enable Posting

Posting is used to notify a calling routine that the command request has been completed. A posting routine is also referred to as a callback routine.

The **Request Type** bit specifies whether ARB contains an ATAPI Packet command or an AT Task File Structure :

0 ATAPI Packet command (e.g. Inquiry, Mode Select ...)
1 AT Task File Structure

The **Direction** bits specify the data transfer directions :

00 Direction determined by device
01 Data In
10 Data Out
11 No Data Transfer

If the command issued involved data transfer, setting the **Direction** bits to *Data In* or *Data Out* is required. *No Data Transfer* option can only be used with commands that have no data transfer such as Seek, Execute Device Diagnostic ... [See **Data Transfer Length** field for additional data transfer capabilities].

The **DUA Bit** (DSC Unavailable Action) for an ATAPI packet command determines what action the ATASPI manager is to take if the DSC Bit is not set in the task file status register. The ATASPI manager expects the DSC bit to be set before sending an ATAPI command block. The DUA Bit allows the caller to specify which action the ATASPI Manager to take if DSC bit not set:

0 The ATASPI manager must queue this ATAPI request and service it latter when DSC bit is set.
1 The ATASPI manager must return to the caller with status set to a BUSY condition. It is the responsibility of the caller to issue the request at a latter time.

The **ByteXfer** bit indicates the data transfer method to use and is defined as follows:

0 use word transfer mode
1 use byte transfer mode

The **Device ID** field can be set to :

00 Device 0 (Master device)
01 Device 1 (Slave device)

The **Data Transfer Length** field indicates the total number of data bytes the application wishes to send out or receive. For an ATA Task File Request, this length must be equal to the number of data bytes actually transferred on the bus. For an ATAPI Packet request, this length could be different from the number of data bytes transferred on the bus. If they are not the same, the ATASPI Manager will pad or truncate the appropriate number of bytes to complete the transaction and notify the caller with the **ATA Controller Status** field set to Data Underrun / Overrun.

The **ACB Length** field indicates the total number of valid bytes in the **ACB** field. For Task File request, ACB Length must be set to 7. For ATAPI Packet requests the value is dependent on the packet size and must match the number of bytes contained in the packet.

The **ATA Controller Status** field returns one of the following status :

00h	No error
11h	Device Not Present
12h	Data Overrun / Underrun

The **Device Status** field returns 00h (No Device status) if there is no error, otherwise it would contain the **Error Register** upon return. (Refer to the ATA-2 or ATA Packet Interface specs for detail description of the **Error Register**).

The **Post Routine Address** field is used only if the Post bit is set in the **Request Flag**.

The **Data Transfer Block Size** field is used to specify the number of data bytes to transfer per hardware interrupt. For Task File request, Data Transfer Block Size (DTBS) is used to determine the maximum number of data bytes to transfer per hardware interrupt. The default value used by the ATASPI manager (when DTBS = 0) is 512 bytes. If necessary (for Read/Write Multiple ...), this could be set to a different value. For ATAPI Packet commands, Data Transfer Block Size (DTBS) is used to tell the device how much data the host prefers to transfer per interrupt. The default value used by ATASPI (when DTBS = 0) is 930h bytes. This is only a preferred value the ATASPI Manager issues to the device but the device controls the actual number of bytes per hardware interrupt to transfer.

If the **Request Flag** indicates an ATAPI Packet command, the **ATA/ATAPI Command Block (ACB)** field contains the command bytes to be sent out to the Device. If the **Request Flag** indicates an AT Task File Structure, the **ACB** is defined as follows :

Offset	Register	Description
0	Features	Command specific features
1	Sector Count	Number of Sectors
2	Sector Number * LBA bits 0 - 7	In CHS mode, starting sector number for the command * In LBA mode, contains bits 0-7
3	Cylinder LSB * LBA bits 8-15	In CHS mode, low order 8 bits of starting cylinder address * In LBA mode, contains bits 8-15
4	Cylinder MSB * LBA bits 16-23	In CHS mode, high order bits of starting cylinder address * In LBA mode, contains bits 16-23
5	Device / Head * LBA bits 24-27	In CHS mode, refer to ATA spec for description * In LBA mode, contains bits 24-27
6	Command	Command to be sent to Device; refer to ATA spec for list of commands.

Table 7 - ACB Definition for Task File Structure

The **Sense Allocation Area** is filled with Sense Data only if a Check Condition occurred and **Sense Allocation Length** is non-zero.

2.6.4. Abort ATA Request

The *Abort ATA Request* command is used to terminate a previously issued ARB. The success of this command is never assured. This command always returns successful status.

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 3
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag = 0
04h	04h	--	Reserved
08h	02h	Out	Offset of ARB to be aborted
0Ah	02h	Out	Segment of ARB to be aborted

Table 8 - Abort ATA Request Command

The **Status** byte returned is set to the following value:

01h Completed without error

2.6.5. Reset ATA Device

The *Reset ATA Device* command is used to reset a specific device. The consequence of this command is dependent on the type of device to be reset. If the specified device does not support a packetized protocol, e.g. an IDE hard disk, the SRST bit in the Device Control Register (refer to the ATA-2 spec for more information) will be asserted. This will cause all devices in the same bus to be reset. If the device is an ATAPI (packetized protocol) device, an ATAPI Soft Reset (ATA Code 08h) will be issued and will cause only the specified device to be reset.

If the specified device is an ATAPI device, it will receive an ATAPI Soft Reset (ATA code*= 08h).

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 4
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag
04h	04h	–	Reserved
08h	01h	Out	Device ID
09h	0Fh	–	Reserved
18h	01h	In	ATA Controller Status
19h	01h	In	Device Status
1Ah	02h	Out	Post Routine Address (Offset)
1Ch	02h	Out	Post Routine Address (Segment)
1Eh	22h	–	Reserved for ATASPI Workspace

Table 9 - Reset ATA Device Command

The **Device ID** field can be set to :

0 Device 0 (Master device)
1 Device 1 (Slave device)

The **Request Flags** byte is defined as follows :

7	6	5	4	3	2	1	0
Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	Post

Table 10 - Reset ATA Device Request Flags

The **Post** bit indicates whether we need to call a Post routine upon completion of the request:

0 Disable Posting
1 Enable Posting

The **Status** byte returned is set to one of the following values:

01h Request completed without error
81h Invalid ATA Controller Number
82h Device Not Present

The **ATA Controller Status** field returns one of the following status :

00h No error
11h Device Not Present

The **Device Status** field returns 00h (No Device status) if there is no error, otherwise it contains the **Error Register** upon return. (Refer to the ATA-2 or ATA Packet Interface specs for detail description of the **Error Register**).

The **Post Routine Address** field is used only if the Post bit is set in the **Request Flag**.

2.6.6. Set ATA Controller Parameter

ATA controllers with different features, capabilities could be set using this command.

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 5
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag = 0
04h	04h	--	Reserved
08h	10h	Out	Controller Unique Parameters (Vendor Unique)

Table 11 - Set ATA Controller Parameter Command

Controller Unique Parameters field is vendor unique and to be defined by controller vendors.

2.6.7. Get ATA Disk Drive Information

The *Get ATA Disk Information* command is used to obtain the INT 13h disk drive status and information.

Offset	Size in Bytes	Direction	Field Description
00h	01h	Out	Command Code = 6
01h	01h	In	Status
02h	01h	Out	ATA Controller Number (zero-based)
03h	01h	Out	Request Flag = 0
04h	04h	--	Reserved
08h	01h	Out	Device ID
09h	01h	--	Reserved
0Ah	01h	In	Drive Flags
0Bh	01h	In	INT 13h Drive
0Ch	01h	In	Preferred Head Translation
0Dh	01h	In	Preferred Sector Translation
0Eh	0Ah	--	Reserved

Table 12 - Get ATA Disk Drive Information Command

The **Status** byte returned is set to the following value:
- 01h Completed without error

The **Device ID** field can be set to :
- 00h Device 0 (Master device)
- 01h Device 1 (Slave device)

The **Drive Flags** byte is defined as follows :

7	6	5	4	3	2	1	0
Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	INT 13h Info	

Table 13 - Get ATA Disk Drive Information Drive Flags

The **INT 13h Info** bits return information pertaining to the INT 13h Drive field. The following table defines the values returned.

Value	Description
00	The given drive (Controller Number / Device ID) is not accessible via INT 13h. If you wish to read/write to this drive, you'll need to send ATASPI read/write requests to the drive. The **INT 13h Drive** field is invalid.
01	The given drive (Controller Number / Device ID) is accessible via INT 13h. The **INT 13h Drive** field contains the drive's INT 13h drive number. This drive is under control of DOS.
10	The given drive (Controller Number / Device ID) is accessible via INT 13h. The **INT 13h Drive** field contains the drive's INT 13h drive number. This drive is NOT under control of DOS and can be used, for example, by a ATA Disk Driver.
11	Invalid

Table 14 - INT 13h Info bits

The **INT 13h Drive** field returns the INT 13h drive number for the given controller number, device ID. Valid INT 13h drive numbers range from 80-Ffh.

The **Preferred Head Translation** field indicates the given controller / disk drive's preferred head translation method.

The **Preferred Sector Translation** field indicates the given controller / disk drive's preferred sector translation method.

3. ATASPI For Windows Specification

The ATASPI for Windows Manager is a Windows v3.x Dynamic Link Library (DLL) named ATASPI.DLL. The DLL is loaded into memory by Windows when one of its exported functions is called by a Windows application. The Windows ATASPI Manager is multi-tasking and can support both polling and posting for completion of ATASPI commands.

It is beyond the scope of this specification to define the protocol between ATASPI for DOS and ATASPI For Windows. Different applications have different needs and performance may or may not be an issue. Support for DOS, Windows or both are the decision of each vendor. Hardware vendors will typically provide an ATASPI manager for their chip set or possibly a generic solution. Implementation and the complete feature set supported will depend on each vendor supplying an ATASPI manager.

3.1. Accessing ATASPI For Windows

Access to the Windows ATASPI Manager is via two library function calls. These functions are imported to the Windows application by adding the following lines in the .DEF file of the application:

```
IMPORTS
        ATASPI.GetATASPISupportInfo
        ATASPI.SendATASPICommand
```

The following briefly describes each of these functions:

GetATASPISupportInfo
This function is used to determine the status of the Windows ATASPI Manager. It is also used to find out how many controllers are supported by ATASPI.

SendATASPICommand
This function is used to issue ATASPI commands to the ATASPI Manager.

3.2. Windows ATASPI Functions

3.2.1. GetATASPISupportInfo

WORD GetATSPISupportInfo(VOID)

The GetATASPISupportInfo function returns information about the status of the Windows ATASPI Manager. This function must be called first before attempting to issue any ATASPI commands. If this function does not return a successful completion code (AS_COMP), it is not recommended to issue any ATASPI commands.

The GetATASPISupportInfo function does not pass any parameters. Upon completion, the function returns a WORD value which indicates the status of the ATASPI manager. The format of the returned WORD value is as follows:

HIBYTE: Status Byte
LOBYTE: Total Number of Controllers Supported if Status Byte indicates success or,
0 if Status Byte indicates error

Returns

The following are the valid Status Byte (HIBYTE) values returned:

Symbolic Name	Value (in Hex)	Description
AS_COMP	01	Windows ATASPI Manager is successfully initialized and is ready to accept ATASPI commands.
AS_NO_WIN_MANAGER	E1	One of the modules required by Windows ATASPI Manager is not loaded. ATASPI commands will fail.
AS_ILLEGAL_MODE	E2	Windows is not running in 386 Enhanced mode.
AS_FAILED_INIT	E4	Windows ATASPI Manager failed initialization.

Table 15 - GetATSPISupportInfo Status Byte return values

Example Code

```
WORD WINATASPIStatus;
BYTE NumControllers;
HWND hwnd;

:
:

WINATASPIStatus = GetATASPISupportInfo();
switch ( HIBYTE(WINATASPIStatus) )
{
   case AS_COMP:
         //ATASPI for Windows is properly initialized
         NumControllers = LOBYTE(WINATASPIStatus);
         break;
   case AS_ILLEGAL_MODE:
         MessageBox( hwnd,"ATASPI for Windows does not support this
                     mode!!", szAppName,MB_ICONSTOP );
         return 0;
   case AS_NO_WIN_MANAGER:
         MessageBox( hwnd,"No Windows ATASPI Manager found!!",
                     szAppName,MB_ICONSTOP );
         return 0;
   default:
         MessageBox( hwnd,"ATASPI for Windows is not initialized!!",
                     szAppName,MB_ICONSTOP );
         return 0;
}
:
:
```

3.2.2. SendATASPICommand

WORD SendATASPICommand(lpARB ARB)

The **SendATASPICommand** function is used to issue ATASPI commands to the Windows ATASPI Manager. The function is passed a far pointer (lpARB) to an ATASPI Request Block (ARB). The following are the ATASPI commands supported and their symbolic names as defined in the included ATASPI.H at the end of this chapter :

- ATA Controller Inquiry (AC_CONTROLLER_INQUIRY)
- Get ATA Device Type (AC_GET_DEVICE_TYPE)
- Execute ATA I/O (AC_EXEC_ATA_CMD)
- Abort ATA Request (AC_ABORT_ARB)
- Reset ATA Device (AC_RESET_DEV)

Detailed description of each of the ATASPI commands are described in the next sections.

Return Values

Upon completion of the **SendATASPICommand**, it returns a WORD value that indicates the outcome of the ATASPI command. The following table contains all valid return values. Actual returned values differ for each ATASPI command. Refer to the specific ATASPI command section for valid returned values.

Symbolic Name	Value(Hex)	Description
AS_PENDING	0x00	ARB being processed
AS_COMP	0x01	ARB completed without error
AS_ABORTED	0x02	ARB aborted
AS_ABORT_FAIL	0x03	Unable to abort ARB
AS_ERR	0x04	ARB completed with error
AS_INVALID_CMD	0x80	Invalid ATASPI command
AS_INVALID_CNUM	0x81	Invalid ATA controller number
AS_NO_DEVICE	0x82	ATA device not installed
AS_ATA_BUSY	0x83	ATA controller/device busy
AS_INVALID_ARB	0xE0	Invalid parameter set in ARB
AS_NO_WIN_MANAGER	0xE1	ATASPI manager doesn't support Windows
AS_ILLEGAL_MODE	0xE2	Unsupported Windows mode
AS_FAILED_INIT	0xE4	ATASPI for windows failed init
AS_ATASPI_IS_BUSY	0xE5	No resources available to execute cmd
AS_BUFFER_TOO_BIG	0xE6	Buffer size too big to handle!

Table 16 - Valid SendATASPICommand Return Values

3.3. Windows ATASPI Commands

3.3.1. ATA Controller Inquiry Command

This command is used to retrieve information about a specific controller. It can also be used to get the total number of controllers supported by the Windows ATASPI Manager.

To issue this command, the pointer passed to **SendATASPICommand** must point to an ATA Controller Inquiry ARB as defined below:

```
typedef struct {

   BYTE  ARB_Cmd;
   BYTE  ARB_Status;
   BYTE  ARB_CntlrID;
   BYTE  ARB_Flags;
   DWORD ARB_Reserved;
   BYTE  CNTLR_Count;
   BYTE  CNTLR_Reserved;
   BYTE  CNTLR_ManagerID[16];
   BYTE  CNTLR_Identifier[16];
   BYTE  CNTLR_Unique[16];

} ARB_Controller_Inquiry;
```

The following table describes each of the fields in the ATA Controller Inquiry ARB:

Symbolic Name	Offset	Direction	Description
ARB_Cmd	00h	Out	This has the ATASPI Command Code. This must be equal to 00h.
ARB_Status	01h	In	This indicates the outcome of the command. Refer to the table below for valid values.
ARB_CntlrID	02h	Out	This refers to the zero-based controller ID to interrogate. If this value is 0FFh, the field CNTLR_Count will have the total number of controllers supported.
ARB_Reserved	03h	–	Reserved. Must be zero.
CNTLR_Count	08h	In	This is the total number of controllers supported. This value is valid only if ARB_CntlrID on return is 0FFh.
CNTLR_Reserved	09h	–	Reserved. Must be zero.
CNTLR_ManagerID[..]	0Ah	In	This is a 16-byte buffer containing an ASCII string describing the ATASPI Manager
CNTLR_Identifier[..]	1Ah	In	This is a 16-byte buffer containing an ASCII string describing the controller specified by ARB_CntlrID.
CNTLR_Unique[..]	2Ah	In	This is a 16-byte buffer containing vendor unique information as defined by the controller vendor.

Table 17 - ATA Controller Inquiry ARB

Returns

Upon completion of the command, a WORD value containing the outcome of the command is returned. The following are the valid values returned:

Symbolic Name	Value (in Hex)	Description
AS_COMP	01	Command completed without error
AS_INVALID_CNUM	81	Controller number passed is invalid
AS_INVALID_ARB	E0	ARB contains invalid parameter(s).

Table 18 - ATA Controller Inquiry ARB Return Values

Example Code

```
ARB_Controller_Inquiry ATAControllerInquiryARB;
WORD ARBStatus;
   :
   :
// Make sure all reserved fields are initialized to zero
ATAControllerInquiryARB.ARB_Cmd        = 0x00;
ATAControllerInquiryARB.ARB_CntlrID    = 0;
ATAControllerInquiryARB.ARB_Flags      = 0;
ARBStatus = SendATASPICommand ((lpARB) &ATAControllerInquiryARB);
   :
   :
```

3.3.2. Get ATA Device Type Command

This command is used to obtain the device type of a specific target device. This command does not return detailed information about the device. It is recommended to issue an Execute ATA I/O Command for such purpose.

To issue this command, the pointer passed to **SendATASPICommand** must point to a Get ATA Device Type ARB defined as follows:

```
typedef struct {

   BYTE  ARB_Cmd;
   BYTE  ARB_Status;
   BYTE  ARB_CntlrID;
   BYTE  ARB_Flags;
   DWORD ARB_Reserved;
   BYTE  ARB_DeviceID;
   BYTE  ARB_Reserved1;
   BYTE  ARB_DeviceType;

} ARB_GDEVBlock;
```

The following table describes each of the fields in the Get ATA Device Type ARB:

Symbolic Name	Offset	Direction	Description
ARB_Cmd	00h	Out	This has the ATASPI Command Code. This must be equal to 01h.
ARB_Status	01h	In	This indicates the outcome of the command. Refer to the table below for valid values.
ARB_CntlrID	02h	Out	This refers to the zero-based controller ID that has the device to interrogate.
ARB_Reserved	03h	–	Reserved. Must be zero.
ARB_DeviceID	08h	Out	Device ID. Value set to 0 if Master device. Value set to 1 if Slave device.
ARB_Reserved1	09h	–	Reserved. Must be zero.
ARB_DeviceType	0Ah	In	If ARB_Status is equal to AS_COMP, this field indicates the Peripheral Device Type.

Table 19 - Get ATA Device Type

The following table include all currently defined ATA device types and their device values used in ARB_DeviceType field.

Symbolic Name	Value(Hex)	Description
DT_DirectAccess	00	Direct access deivce (e.g. magnetic disk)
DT_TapeQIC	01	Tape device (QIC-121 Model)
DT_CDROM	05	CD-ROM device
DT_Optical	07	Optical memory device
DT_TapeCS	0C	Tape device (Cost Sensitive Model)
DT_Unknown	1F	Unknown device type
DT_ATAType	80	Non-ATAPI device (IDE hard disk)

Table 20 - ATA Device Type Values

Returns

Upon completion of the command, a WORD value containing the outcome of the command is returned. The following are the valid values returned:

Symbolic Name	Value (in Hex)	Description
AS_COMP	01	Command completed without error. The field ARB_DeviceType has a valid value.
AS_INVALID_CNUM	81	Controller number passed is invalid
AS_NO_DEVICE	82	Device is not present.
AS_INVALID_ARB	E0	ARB contains invalid parameter(s).

Table 21 - Get ATA Device Type ARB Return Values

Example Code

```
ARB_GDEVBlock GetDeviceTypeARB;
WORD ARBStatus;
   :
   :
// Make sure all reserved fields are initialized to zero
GetDeviceTypeARB.ARB_Cmd         = 0x01;
GetDeviceTypeARB.ARB_CntlrID     = 0;
GetDeviceTypeARB.ARB_Flags       = 0;
GetDeviceTypeARB.ARB_DeviceID    = 0;
ARBStatus = SendATASPICommand ((lpARB) &GetDeviceTypeARB);
   :
   :
```

3.3.3. Execute ATA I/O Command

The Execute ATA I/O command is used to issue ATA and ATAPI commands to IDE devices. Multiple commands may be outstanding at any time. This multi-tasking capability is handled by the Windows ATASPI Manager. Completion of this command may be relayed to the caller either through a Polling method or a Posting method.

To issue this command, the pointer passed to **SendATASPICommand** must point to one of the Execute ATA I/O ARBs defined below:

```
// Structure for ATA Task File CDB
typedef struct {

   BYTE         ARB_Cmd;
   BYTE         ARB_Status;
   BYTE         ARB_CntlrID;
   BYTE         ARB_Flags;
   DWORD        ARB_Reserved;
   BYTE         ARB_DeviceID;
   BYTE         ARB_Reserved1;
   DWORD        ARB_BufLen;
   BYTE         ARB_SenseLen;
   BYTE  far *  ARB_BufPointer;
   DWORD        ARB_Reserved2;
   BYTE         ARB_ACBLen;
   BYTE         ARB_CntlrStat;
   BYTE         ARB_DevStat;
   FARPROC      ARB_PostProc;
   WORD         ARB_Data_BlkSize;
   BYTE         ARB_Reserved3[32];
   BYTE         ARB_ACBByte[7];
   BYTE         ARB_SenseArea[SENSE_LEN];

} ARB_ExecATATFCmd;

// Structure for ATASPI 12 Byte ATAPI CDB
typedef struct {
   BYTE         ARB_Cmd;
   BYTE         ARB_Status;
   BYTE         ARB_CntlrID;
   BYTE         ARB_Flags;
   DWORD        ARB_Reserved;
   BYTE         ARB_DeviceID;
   BYTE         ARB_Reserved1;
   DWORD        ARB_BufLen;
   BYTE         ARB_SenseLen;
   BYTE  far *  ARB_BufPointer;
   DWORD        ARB_Reserved2;
   BYTE         ARB_ACBLen;
   BYTE         ARB_CntlrStat;
   BYTE         ARB_DevStat;
   FARPROC      ARB_PostProc;
   WORD         ARB_Data_BlkSize;
   BYTE         ARB_Reserved3[32];
   BYTE         ARB_ACBByte[12];
   BYTE         ARB_SenseArea[SENSE_LEN];

} ARB_ExecATAPKT12Cmd;

// Note: For 16 byte ATAPI CDB commands, create an ARB with
// ARB_ACBByte[16].
```

The following table describes each of the fields in the Execute ATA I/O ARB:

Symbolic Name	Offset	Direction	Description
ARB_Cmd	00h	Out	This has the ATASPI Command Code. This must be equal to 02h.
ARB_Status	01h	In	This indicates the outcome of the command. Refer to the table below for valid values. See Flag definition that follows.
ARB_CntlrID	02h	Out	This refers to the zero-based controller ID of the IDE device.
ARB_Flags	03h	Out	This field tells the ATASPI Manager certain characteristics of this request.
ARB_Reserved	04h	–	Reserved. Must be zero.
ARB_DeviceID	08h	Out	Device ID. Value set to 0 if Master device. Value set to 1 if Slave device.
ARB_Reserved1	09h	–	Reserved. Must be zero.
ARB_BufLen	0Ah	Out	This refers to the data length in number of bytes. This field must be set to zero if no data transfer is expected.
ARB_SenseLen	0Eh	Out	This refers to the number (N) of sense bytes to transfer when a Check Condition occurred.
ARB_BufPointer	0Fh	Out	This is the pointer to the data buffer. The buffer area must be in locked memory.
ARB_Reserved2	13h	–	Reserved. Must be zero.
ARB_ACBLen	17h	Out	This indicates the number (M) of valid bytes in the ARB_ACBByte field.
ARB_CntlrStat	18h	In	This field indicates the status of the controller upon completion of the command.
ARB_DevStat	19h	In	This field is set to zero if no error occurred. Otherwise, this field will have the value of the Error Register on the ATA bus. Refer to the ATA-2 or ATA Packet Interface specs for a description of the Error register.
ARB_PostProc	1Ah	Out	If the Post bit is set on the ARB_Flags byte, this field contains the address to call upon completion of the command.
ARB_Data_BlkSize	1Eh	Out	The **ARB_Data_BlkSize** field is used to specify the number of data bytes to transfer per hardware interrupt. See detailed explanation that follows.
ARB_Reserved3	20h	–	Reserved. Must be zero.
ARB_ACBByte[M]	40h	Out	If Request Type is an task file request, this area contains the 7 byte Task File command. Otherwise, this area contains the command bytes of the ATAPI device.
ARB_SenseArea[N]	10h + M	In	This area will contain the sense data if a Check Condition occurred.

Table 22 - Execute ATA I/O ARB

The ARB_Flags byte is defined as follows:

7	6	5	4	3	2	1	0
Rsvd	ByteXfer	DUA Bit	Direction Bits		Request Type	Rsvd	Post

Table 23 - ARB_Flags

The **Post** bit indicates whether we need to call a Post routine upon completion of the request.

0 Disable Posting
1 Enable Posting

The **Request Type** bit specifies whether ARB contains an ATAPI Packet command or an AT Task File Structure :

0 ATAPI Packet command (e.g. Inquiry, Mode Select ...)
1 AT Task File Structure

The **Direction** bits specify the data transfer directions :

00 Direction determined by device
01 Data In
10 Data Out
11 No Data Transfer

If the command sent out involved data transfer, setting the **Direction** bits to *Data In* or *Data Out* is required. *No Data Transfer* option can only be used with commands that have no data transfer such as Seek, Execute Device Diagnostic ... [See **Data Transfer Length** field for additional data transfer capabilities].

The **DUA Bit** (DSC Unavailable Action) for an ATAPI packet command determines what action the ATASPI manager is to take if the DSC Bit is not set in the task file status register. The ATASPI manager is expecting the DSC bit to be set before sending an ATAPI command block, the caller can specify which action to take if DSC bit not set.

0 The ATASPI manager must queue this ATAPI request and service it latter when DSC bit is set.
1 The ATASPI manager must return to the caller with status set to a BUSY condition. It is the responsibility of the caller to issue the request at a latter time.

The **ByteXfer** bit indicates that byte data transfer is requested. The default mode of operation for ATASPI Manager is word transfer.

0 use word transfer mode
1 use byte transfer mode

The **ARB_Data_BlkSize** field is used to specify the number of data bytes to transfer per hardware interrupt. For Task File request, Data Transfer Block Size (DTBS) is used to determine maximum number of data bytes to transfer per interrupt. The default value used by the ATASPI manager (when DTBS = 0) is 512 bytes.

For ATAPI Packet commands, Data Transfer Block Size (DTBS) is used to tell the device how much data the host prefers to transfer per interrupt. The default value used by ATASPI (when DTBS = 0) is 930h bytes.

If neccessary (for Read/Write Multiple ...), this could be set to a different value. This is only a preferred value; for every interrupt received, the device specifies the exact amount of data to transfer

Returns

Upon completion of the command, a WORD value containing the status of the command is returned. The following are the valid values returned:

Symbolic Name	Value (in Hex)	Description
AS_PENDING	00	The command is in progress.
AS_COMP	01	Command completed without error.
AS_ABORTED	02	ATASPI command is aborted.
AS_ERR	04	Command completed with error.
AS_INVALID_CNUM	81	Controller number passed is invalid.
AS_NO_DEVICE	82	Device is not present.
AS_ATA_BUSY	83	ATA Controller or Device is busy.
AS_INVALID_ARB	E0	ARB contains invalid parameter(s).
AS_ATASPI_IS_BUSY	E5	ATASPI cannot handle the request at this time. Re-send the request later.
AS_BUFFER_TOO_BIG	E6	Transfer size too big for ATASPI Manager.

Table 24 - Execute ATA I/O ARB Return Values

Example Code

```
ARB_ExecATATFCmd ExecATAioARB;
WORD ARBStatus;
char IdentifyDriveData[512];
   :
   :
// Make sure all reserved fields are initialized to zero
ExecATAioARB.ARB_Cmd              = 0x02;
ExecATAioARB.ARB_CntlrID          = 0;
ExecATAioARB.ARB_Flags            = 0x05;
ExecATAioARB.ARB_BufLen           = 512;
ExecATAioARB.ARB_SenseLen         = SENSE_LEN;
ExecATAioARB.ARB_BufPointer       = IdentifyDriveData;
ExecATAioARB.ARB_ACBLen           = 7;
ExecATAioARB.ARB_PostProc         = lpfnPostFunction;
ExecATAioARB.ARB_ACBByte[0]       = 0;
ExecATAioARB.ARB_ACBByte[1]       = 0;
ExecATAioARB.ARB_ACBByte[2]       = 0;
ExecATAioARB.ARB_ACBByte[3]       = 0;
ExecATAioARB.ARB_ACBByte[4]       = 0;
ExecATAioARB.ARB_ACBByte[5]       = 0;
ExecATAioARB.ARB_ACBByte[6]       = 0xEC;
ARBStatus = SendATASPICommand ((lpARB) &ExecATAioARB);
   :
   :
```

3.3.4. Abort ATA Request Command

This command is used to abort a previously issued ARB. The success of this command is never assured. This command will always return success.

To issue this command, the pointer passed to SendATASPICommand must point to an Abort ATA Request ARB as defined below:

```
typedef struct {

   BYTE  ARB_Cmd;
   BYTE  ARB_Status;
   BYTE  ARB_CntlrID;
   BYTE  ARB_Flags;
   DWORD ARB_Reserved;
   lpARB ARB_ToAbort;

} ARB_Abort;
```

The following table describes each of the fields in the Abort ATA I/O ARB:

Symbolic Name	Offset	Direction	Description
ARB_Cmd	00h	Out	This has the ATASPI Command Code. This must be equal to 03h.
ARB_Status	01h	In	This indicates the outcome of the command. Refer to the table below for valid values.
ARB_CntlrID	02h	Out	This refers to the zero-based ID of the controller that owns the ARB to abort.
ARB_Reserved	03h	–	Reserved. Must be zero.
ARB_ToAbort	08h	Out	This field has the pointer to the ARB to be aborted.

Table 25 - Abort ATA Request Command

Returns

Upon completion of the command, a WORD value containing the outcome of the command is returned. The following are the valid values returned:

Symbolic Name	Value (in Hex)	Description
AS_COMP	01	Command completed without error. The field ARB_DeviceType has a valid value.
AS_INVALID_CNUM	81	Controller number passed is invalid
AS_INVALID_ARB	E0	The ARB contains invalid parameter(s).

Table 26 - Abort ATA Request Command Return Values

Example Code

```
ARB_ExecATAPKT12Cmd ARBtoAbort;
ARB_Abort AbortARB;
WORD ARBStatus;
   :
   :
// Make sure all reserved fields are initialized to zero
AbortARB.ARB_Cmd           = 0x03;
AbortARB.ARB_CntlrID       = 0;
AbortARB.ARB_Flags         = 0;
AbortARB.ARB_ToAbort       = (lpARB) &ARBtoAbort;
ARBStatus = SendATASPICommand ((lpARB) &AbortARB);
while (ResetDevARB.ARB_Status == AS_PENDING); // Poll for completion
   :
   :
```

3.3.5. Reset ATA Device Command

This command is used to reset a specific device. The consequence of this command is dependent on the type of device to be reset. If the specified device does not support a packetized protocol, e.g. an IDE hard disk, the SRST bit in the Device Control Register (refer to the ATA-2 spec for more information) will be asserted. This will cause all devices in the same bus to be reset. If the device is an ATAPI (packetized protocol) device, an ATAPI Soft Reset (ATA Code 08h) will be issued and will cause only the specified device to be reset.

To issue this command, the pointer passed to **SendATASPICommand** must point to a Reset ATA Device ARB as defined below:

```
typedef struct {

   BYTE        ARB_Cmd;
   BYTE        ARB_Status;
   BYTE        ARB_CntlrID;
   BYTE        ARB_Flags;
   DWORD       ARB_Reserved;
   BYTE        ARB_DeviceID;
   BYTE        ARB_Reserved1[15];
   BYTE        ARB_CntlrStat;
   BYTE        ARB_DevStat;
   FARPROC     ARB_PostProc;
   BYTE        ARB_Reserved2[34];

} ARB_DeviceReset;
```

The following table describes each of the fields in the Reset ATA Device ARB:

Symbolic Name	Offset	Direction	Description
ARB_Cmd	00h	Out	This has the ATASPI Command Code. This must be equal to 04h.
ARB_Status	01h	In	This indicates the outcome of the command. Refer to the table below for valid values.
ARB_CntlrID	02h	Out	This refers to the zero-based controller ID.
ARB_Flags	03h	Out	This field tells the ATASPI Manager certain characteristics of this request. See the flag definition that follows.
ARB_Reserved	04h	–	Reserved. Must be zero.
ARB_DeviceID	08h	Out	Device ID. Value set to 0 if Master device. Value set to 1 if Slave device.
ARB_Reserved1	09h	–	Reserved. Must be zero.
ARB_CntlrStat	18h	In	This field indicates the status of the controller upon completion of the command. The following are the valid values: ACSTAT_OK - No error ACSTAT_NO_DEVICE - Device not present
ARB_DevStat	19h	In	This field is set to zero if no error occurred. Otherwise, this field will have the value of the Error Register on the ATA bus. Refer to the ATA-2 or ATA Packet Interface specs for a description of the Error register.
ARB_PostProc	1Ah	Out	If the Post bit is set on the ARB_Flags byte, this field contains the address to call upon completion of the command.
ARB_Reserved2	1Eh	–	Reserved. Must be zero.

Table 27 - Reset ATA Device ARB

Returns

Upon completion of the command, a WORD value containing the outcome of the command is returned. The following are the valid values returned:

Symbolic Name	Value (in Hex)	Description
AS_COMP	01	Command completed without error. The field ARB_DeviceType has a valid value.
AS_INVALID_CNUM	81	Controller number passed is invalid
AS_INVALID_ARB	E0	ARB contains invalid parameter(s).
AS_ATASPI_IS_BUSY	E5	ATASPI cannot handle the request at this time. Re-send the request later.

Table 28 - Reset ATA Device ARB Return Values

Example Code

```
ARB_DeviceReset ResetDevARB;
WORD ARBStatus;
   :
   :
// Make sure all reserved fields are initialized to zero
ResetDevARB.ARB_Cmd         = 0x04;
ResetDevARB.ARB_CntlrID     = 0;
ResetDevARB.ARB_Flags       = 0;
ResetDevARB.DeviceID        = 0;
ARBStatus = SendATASPICommand ((lpARB) &ResetDevARB);
while (ResetDevARB.ARB_Status == AS_PENDING);
   // Poll for completion
   :
   :
```

3.3.5. Reset ATA Device Command

This command is used to reset a specific device. The consequence of this command is dependent on the type of device to be reset. If the specified device does not support a packetized protocol, e.g. an IDE hard disk, the SRST bit in the Device Control Register (refer to the ATA-2 spec for more information) will be asserted. This will cause all devices in the same bus to be reset. If the device is an ATAPI (packetized protocol) device, an ATAPI Soft Reset (ATA Code 08h) will be issued and will cause only the specified device to be reset.

To issue this command, the pointer passed to **SendATASPICommand** must point to a Reset ATA Device ARB as defined below:

```
typedef struct {

   BYTE        ARB_Cmd;
   BYTE        ARB_Status;
   BYTE        ARB_CntlrID;
   BYTE        ARB_Flags;
   DWORD       ARB_Reserved;
   BYTE        ARB_DeviceID;
   BYTE        ARB_Reserved1[15];
   BYTE        ARB_CntlrStat;
   BYTE        ARB_DevStat;
   FARPROC     ARB_PostProc;
   BYTE        ARB_Reserved2[34];

} ARB_DeviceReset;
```

The following table describes each of the fields in the Reset ATA Device ARB:

Symbolic Name	Offset	Direction	Description
ARB_Cmd	00h	Out	This has the ATASPI Command Code. This must be equal to 04h.
ARB_Status	01h	In	This indicates the outcome of the command. Refer to the table below for valid values.
ARB_CntlrID	02h	Out	This refers to the zero-based controller ID.
ARB_Flags	03h	Out	This field tells the ATASPI Manager certain characteristics of this request. See the flag definition that follows.
ARB_Reserved	04h	–	Reserved. Must be zero.
ARB_DeviceID	08h	Out	Device ID. Value set to 0 if Master device. Value set to 1 if Slave device.
ARB_Reserved1	09h	–	Reserved. Must be zero.
ARB_CntlrStat	18h	In	This field indicates the status of the controller upon completion of the command. The following are the valid values: ACSTAT_OK - No error ACSTAT_NO_DEVICE - Device not present
ARB_DevStat	19h	In	This field is set to zero if no error occurred. Otherwise, this field will have the value of the Error Register on the ATA bus. Refer to the ATA-2 or ATA Packet Interface specs for a description of the Error register.
ARB_PostProc	1Ah	Out	If the Post bit is set on the ARB_Flags byte, this field contains the address to call upon completion of the command.
ARB_Reserved2	1Eh	–	Reserved. Must be zero.

Table 27 - Reset ATA Device ARB

Returns

Upon completion of the command, a WORD value containing the outcome of the command is returned. The following are the valid values returned:

Symbolic Name	Value (in Hex)	Description
AS_COMP	01	Command completed without error. The field ARB_DeviceType has a valid value.
AS_INVALID_CNUM	81	Controller number passed is invalid
AS_INVALID_ARB	E0	ARB contains invalid parameter(s).
AS_ATASPI_IS_BUSY	E5	ATASPI cannot handle the request at this time. Re-send the request later.

Table 28 - Reset ATA Device ARB Return Values

Example Code

```
ARB_DeviceReset ResetDevARB;
WORD ARBStatus;
   :
   :
// Make sure all reserved fields are initialized to zero
ResetDevARB.ARB_Cmd         = 0x04;
ResetDevARB.ARB_CntlrID     = 0;
ResetDevARB.ARB_Flags       = 0;
ResetDevARB.DeviceID        = 0;
ARBStatus = SendATASPICommand ((lpARB) &ResetDevARB);
while (ResetDevARB.ARB_Status == AS_PENDING);
   // Poll for completion
   :
   :
```

3.4. ATASPI.H

```
//****************************************************************************
//
// Name:          ATASPI.H
//
// Description:    ATASPI for Windows definitions ('C' Language)
//
//****************************************************************************

//****************************************************************************
//               %%% GENERIC ARB (ATASPI REQUEST BLOCK) %%%
//****************************************************************************
typedef struct {

    BYTE    ARB_Cmd;              // ATASPI command code
    BYTE    ARB_Status;           // ATASPI command status byte
    BYTE    ARB_CntlrID;          // ATASPI ATA controller number
    BYTE    ARB_Flags;            // ATASPI request flags
    DWORD   ARB_Reserved;         // Reserved, MUST = 0

} ARB_Struct, FAR * ARB_Struct_Ptr;

typedef ARB_Struct far *lpARB;

WORD FAR PASCAL SendATASPICommand    (lpARB);
WORD FAR PASCAL GetATASPISupportInfo (VOID);

#define SENSE_LEN           14    // Default sense buffer length

//****************************************************************************
//                    %%% ATASPI COMMAND DEFINITIONS %%%
//****************************************************************************
#define AC_CONTROLLER_INQUIRY     0x00  // ATA Controller Inquiry
#define AC_GET_DEV_TYPE           0x01  // Get ATA Device Type
#define AC_EXEC_ATA_CMD           0x02  // Execute ATA Command
#define AC_ABORT_ARB              0x03  // Abort an ARB
#define AC_RESET_DEV              0x04  // ATA Bus Device Reset

//****************************************************************************
//                    %%% ATASPI REQUEST FLAG %%%
//****************************************************************************
#define AF_DIR_ATA                0x00  // Direction determined by ATA command
#define AF_DIR_IN                 0x08  // Transfer from ATA device to controller
#define AF_DIR_OUT                0x10  // Transfer from controller to ATA device
#define AF_POSTING                0x01  // Enable ATASPI posting
#define AF_TASK_REQ               0x04  // Task file request
#define AF_BYTE_XFR               0x40  // Byte transfers requested
#define AF_DUA                    0x20  // DSC Unavailable Action Bit

//****************************************************************************
//                         %%% ARB STATUS %%%
//****************************************************************************
#define AS_PENDING          0x00  // ARB being processed
#define AS_COMP             0x01  // ARB completed without error
#define AS_ABORTED          0x02  // ARB aborted
#define AS_ABORT_FAIL       0x03  // Unable to abort ARB
#define AS_ERR              0x04  // ARB completed with error

#define AS_INVALID_CMD      0x80  // Invalid ATASPI command
#define AS_INVALID_CNUM     0x81  // Invalid ATA controller number
#define AS_NO_DEVICE        0x82  // ATA device not installed
#define AS_ATA_BUSY         0x83  // ATA controller/device busy

#define AS_INVALID_ARB      0xE0  // Invalid parameter set in ARB
#define AS_NO_WIN_MANAGER   0xE1  // ATASPI manager doesn't support Windows
#define AS_ILLEGAL_MODE     0xE2  // Unsupported Windows mode
#define AS_FAILED_INIT      0xE4  // ATASPI for windows failed init
#define AS_ATASPI_IS_BUSY   0xE5  // No resources available to execute cmd
#define AS_BUFFER_TOO_BIG   0xE6  // Buffer size too big to handle!
```

```
//*****************************************************************************
//                    %%% ATA CONTROLLER STATUS %%%
//*****************************************************************************
#define ACSTAT_OK           0x00  // ATA controller did not detect an error
#define ACSTAT_NO_DEVICE    0x11  // Device not present
#define ACSTAT_DO_DU        0x12  // Data overrun data underrun

//*****************************************************************************
//                    %%% ATA DEVICE TYPES %%%
//*****************************************************************************
#define DT_DirectAccess           0x00  // Direct access deivce (e.g. magnetic disk)
#define DT_TapeQIC                0x01  // Tape device (QIC-121 Model)
#define DT_CDROM                  0x05  // CD-ROM device
#define DT_Optical                0x07    // Optical memory device
#define DT_TapeCS                 0x0C    // Tape device (Cost Sensitive Model)
#define DT_Unknown                0x1F    // Unknown device type
#define DT_ATAType                0x80    // Non-ATAPI device (IDE hard disk)

//*****************************************************************************
//       %%% ARB - ATA Controller INQUIRY - AC_CONTROLLER_INQUIRY %%%
//*****************************************************************************
typedef struct {

    BYTE    ARB_Cmd;                // ATASPI command code = AC_CONTROLLER_INQUIRY
    BYTE    ARB_Status;             // ATASPI command status byte
    BYTE    ARB_CntlrID;            // ATASPI ATA Controller number
    BYTE    ARB_Flags;              // ATASPI request flags
    DWORD   ARB_Reserved;           // Reserved, MUST = 0
    BYTE    CNTLR_Count;            // Number of ATA Controllers present
    BYTE    CNTLR_Reserved;         // ATA ID of ATA Controller
    BYTE    CNTLR_ManagerID[16];    // String describing the manager
    BYTE    CNTLR_Identifier[16];   // String describing the ATA Controller
    BYTE    CNTLR_Unique[16];       // ATA Controller Unique parameters

} ARB_Controller_Inquiry, FAR * ARB_Controller_Inquiry_Ptr;

//*****************************************************************************
//                    %%% ARB - GET DEVICE TYPE - AC_GET_DEV_TYPE %%%
//*****************************************************************************
typedef struct {

    BYTE    ARB_Cmd;                // ATASPI command code = AC_GET_DEV_TYPE
    BYTE    ARB_Status;             // ATASPI command status byte
    BYTE    ARB_CntlrID;            // ATASPI ATA Controller number
    BYTE    ARB_Flags;              // ATASPI request flags
    DWORD   ARB_Reserved;           // Reserved, MUST = 0
    BYTE    ARB_DeviceID;           // Device's ATA ID
    BYTE    ARB_Reserved1;          // Reserved, MUST = 0
    BYTE    ARB_DeviceType;         // Device's peripheral device type

} ARB_GDEVBlock, FAR * ARB_GDEVBlock_Ptr;

//*****************************************************************************
//            %%% ARB - EXECUTE ATA COMMAND - AC_EXEC_ATA_CMD %%%
//*****************************************************************************
typedef struct {                        // Structure for Task File Request

    BYTE    ARB_Cmd;                      // ATASPI command code = AC_EXEC_ATA_CMD
    BYTE    ARB_Status;                   // ATASPI command status byte
    BYTE    ARB_CntlrID;                  // ATASPI ATA Controller number
    BYTE    ARB_Flags;                    // ATASPI request flags
    DWORD   ARB_Reserved;                 // Reserved, MUST = 0
    BYTE    ARB_DeviceID;                 // Target's ATA ID
    BYTE    ARB_Reserved1;                // Reserved, MUST = 0
    DWORD   ARB_BufLen;                   // Data Allocation Length
    BYTE    ARB_SenseLen;                 // Sense Allocation Length
    BYTE    far *ARB_BufPointer;          // Data Buffer Pointer
    DWORD   ARB_Reserved2;                // Reserved, MUST = 0
    BYTE    ARB_ACBLen;                 // ATA CDB Length = 12
    BYTE    ARB_CntlrStat;                //      ATA Controller Status
    BYTE    ARB_DevStat;                  //      Target Status
    FARPROC ARB_PostProc;               // Post routine
    WORD  ARB_Spl_BlkSize;              // Data Block Size
    BYTE    ARB_Reserved3[32];            // Reserved, MUST = 0
    BYTE    ARB_ACBByte[7];             // ATA Task File Bytes
```

```
    BYTE    ARB_SenseArea[SENSE_LEN];      // Request Sense buffer

} ARB_ExecATAPKTCmd, FAR * ARB_ExecATAPKTCmd_Ptr;

typedef struct {                           // Structure for 12-byte ATAPI CBs

    BYTE    ARB_Cmd;                       // ATASPI command code = AC_EXEC_ATA_CMD
    BYTE    ARB_Status;                    // ATASPI command status byte
    BYTE    ARB_CntlrID;                   // ATASPI ATA Controller number
    BYTE    ARB_Flags;                     // ATASPI request flags
    DWORD   ARB_Reserved;                  // Reserved, MUST = 0
    BYTE    ARB_DeviceID;                  // Target's ATA ID
    BYTE    ARB_Reserved1;                 // Reserved, MUST = 0
    DWORD   ARB_BufLen;                    // Data Allocation Length
    BYTE    ARB_SenseLen;                  // Sense Allocation Length
    BYTE    far *ARB_BufPointer;           // Data Buffer Pointer
    DWORD   ARB_Reserved2;                 // Reserved, MUST = 0
    BYTE    ARB_ACBLen;                    // ATA CDB Length = 12
    BYTE    ARB_CntlrStat;                 // ATA Controller Status
    BYTE    ARB_DevStat;                   // Target Status
    FARPROC ARB_PostProc;                  // Post routine
    WORD    ARB_Data_BlkSize;              // Data Block Size
    BYTE    ARB_Reserved3[32];             // Reserved, MUST = 0
    BYTE    ARB_ACBByte[12];               // ATAPI Command Block Bytes
    BYTE    ARB_SenseArea[SENSE_LEN];      // Request Sense buffer

} ARB_ExecATAPKT12Cmd, FAR * ARB_ExecATAPKT12Cmd_Ptr;

typedef struct {                           // Structure for 16-byte ATAPI CBs

    BYTE    ARB_Cmd;                       // ATASPI command code = AC_EXEC_ATA_CMD
    BYTE    ARB_Status;                    // ATASPI command status byte
    BYTE    ARB_CntlrID;                   // ATASPI ATA Controller number
    BYTE    ARB_Flags;                     // ATASPI request flags
    DWORD   ARB_Reserved;                  // Reserved, MUST = 0
    BYTE    ARB_DeviceID;                  // Target's ATA ID
    BYTE    ARB_Reserved1;                 // Reserved, MUST = 0
    DWORD   ARB_BufLen;                    // Data Allocation Length
    BYTE    ARB_SenseLen;                  // Sense Allocation Length
    BYTE    far *ARB_BufPointer;           // Data Buffer Pointer
    DWORD   ARB_Reserved2;                 // Reserved, MUST = 0
    BYTE    ARB_ACBLen;                    // ATA CDB Length = 12
    BYTE    ARB_CntlrStat;                 // ATA Controller Status
    BYTE    ARB_DevStat;                   // Target Status
    FARPROC ARB_PostProc;                  // Post routine
    WORD    ARB_Data_BlkSize;              // Data Block Size
    BYTE    ARB_Reserved3[32];             // Reserved, MUST = 0
    BYTE    ARB_ACBByte[16];               // ATAPI Command Block Bytes
    BYTE    ARB_SenseArea[SENSE_LEN];      // Request Sense buffer

} ARB_ExecATAPKT16Cmd, FAR * ARB_ExecATAPKT16Cmd_Ptr;

//*****************************************************************************
//                  %%% ARB - ABORT AN ARB - AC_ABORT_ARB %%%
//*****************************************************************************
typedef struct {

    BYTE    ARB_Cmd;            // ATASPI command code = AC_ABORT_ARB
    BYTE    ARB_Status;         // ATASPI command status byte
    BYTE    ARB_CntlrID;        // ATASPI ATA Controller number
    BYTE    ARB_Flags;          // ATASPI request flags
    DWORD   ARB_Reserved;       // Reserved, MUST = 0
    lpARB   ARB_ToAbort;        // Pointer to ARB to abort

} ARB_Abort, FAR * ARB_Abort_Ptr;

//*****************************************************************************
//              %%% ARB - DEVICE RESET - AC_RESET_DEV %%%
//*****************************************************************************
typedef struct {

    BYTE    ARB_Cmd;              // ATASPI command code = AC_RESET_DEV
    BYTE    ARB_Status;           // ATASPI command status byte
    BYTE    ARB_CntlrID;          // ATASPI ATA Controller number
    BYTE    ARB_Flags;            // ATASPI request flags
```

```
    DWORD   ARB_Reserved;          // Reserved, MUST = 0
    BYTE    ARB_DeviceID;          // Device's ATA ID
    BYTE    ARB_Reserved1[15];     // Reserved, MUST = 0
    BYTE    ARB_CntlrStat;         // ATA Controller Status
    BYTE    ARB_TargStat;          // Device Status
    FARPROC ARB_PostProc;        // Post routine
    BYTE    ARB_Reserved2[34];     // Reserved, MUST = 0

} ARB_DeviceReset, FAR * ARB_DeviceReset_Ptr;
```

4. Document History

Version 0.50 (November 2, 1994) Initial draft

Version 0.60 (November 11, 1994) Correction of initial entry errors.

Version 0.70 (November 19, 1994) Added ATASPI For Windows Specification Section 4

Version 0.71 (November 23,1994) Request Flags bit 5, DCS bit changed definition to DUA bit.

Version 0.72 (December 20, 1994) Request Flags bit 5, Remove reference to Sepcial Block Size
Modify DUA bit description
General Editorial Cleanup