Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification

Draft for Review

Version 0.9

August 9, 2004

Intel order number xxxxxx-001

Revision History

Revision	Revision History	Date
0.9	First public release.	8/9/04

Contents

Figures

Tables

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the IDE Controller Initialization Protocol of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This protocol is used by an IDE bus driver to program an IDE controller and to obtain IDE device timing information. This protocol abstracts the nonstandard parts of an IDE controller. This protocol is not tied to any specific bus.

This specification does the following:

- Describes the basic components of the IDE Controller Initialization Protocol
- Provides code definitions for the IDE Controller Initialization Protocol and other IDE-controller-related type definitions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

Data Structure Descriptions

Intel® processors based on 32-bit Intel® architecture (IA-32) are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Intel® Itanium® processor family may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

STRUCTURE NAME:	The formal name of the data structure.
Summary:	A brief description of the data structure.
Prototype:	A “C-style” type declaration for the data structure.
Parameters:	A brief description of each field in the data structure prototype.
Description:	A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this data structure.

Protocol Descriptions

The protocols described in this document generally have the following format:

Protocol Name:	The formal name of the protocol interface.
Summary:	A brief description of the protocol interface.
GUID:	The 128-bit Globally Unique Identifier (GUID) for the protocol interface.
Protocol Interface Structure:	A "C-style" data structure definition containing the procedures and data fields produced by this protocol interface.
Parameters:	A brief description of each field in the protocol interface structure.
Description:	A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used in the protocol interface structure or any of its procedures.

Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName():	The formal name of the procedure.
Summary:	A brief description of the procedure.
Prototype:	A "C-style" procedure header defining the calling sequence.
Parameters:	A brief description of each field in the procedure prototype.
Description:	A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this procedure.
Status Codes Returned:	A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *Extensible Firmware Interface Specification*.

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace`	Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`Bold Monospace`	In the online help version of this specification, words in a **`Bold Monospace`** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **`Bold Monospace`** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

text text text	In the PDF of this specification, text that is highlighted in yellow indicates that a change was made to that text since the previous revision of the PDF. The highlighting indicates only that a change was made since the previous version; it does not specify what changed. If text was deleted and thus cannot be highlighted, a note in red and highlighted in yellow (that looks like *(Note: text text text.)*) appears where the deletion occurred.

See the master Framework glossary in the Framework Interoperability and Component Specifications help system for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the master Framework references in the Interoperability and Component Specifications help system for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

The Framework Interoperability and Component Specifications help system is available at the following URL:

http://www.intel.com/technology/framework/spec.htm

2 Design Discussion

IDE Controller Initialization Protocol Overview

This chapter discusses the IDE Controller Initialization Protocol. This protocol is used by an IDE bus driver to program an IDE controller and to obtain IDE device timing information. This protocol abstracts the nonstandard parts of IDE controller. This protocol is mandatory on platforms with IDE controllers that are managed by an IDE bus driver.

See IDE Controller Initialization Protocol in Code Definitions for the definition of **`EFI_IDE_CONTROLLER_INIT_PROTOCOL`**.

IDE Controller Terms

The following terms are used throughout this document.

AHCI

Advanced Host Controller Interface.

enumeration group

The set of IDE devices that must be enumerated as a group. In other words, if device A and device B belong to an enumeration group and device A needs to be configured, device B must be configured at the same time and vice versa. There are two possible enumeration groupings for an IDE controller:

- All the devices on a channel. In this case, the number of enumeration groups is equal to the number of channels.
- All the devices on all the channels behind an IDE controller. This enumeration grouping may arise because multiple channels share some hardware registers or have some other dependencies. In this case, the number of enumeration groups is 1.

The IDE controller indicates the type of enumeration group that is applicable. In case 2, the IDE bus driver must enumerate all the devices on all the channels if there is a request to configure a single device. In case 1, the IDE bus driver must enumerate all the devices on the same channel if there is a request to configure a single device. Case 1 will lead to faster boot.

IDE controller

The hardware device that produces one or more IDE buses (channels). Each channel can host one or more IDE devices.

PATA

Parallel ATA.

PATA controller

An IDE controller that supports PATA devices. Traditionally, a PATA controller supports up to two channels: primary and secondary. Each channel traditionally supports up to two devices: master and slave.

SATA

Serial ATA.

SATA controller

An IDE controller that supports the SATA driver. SATA controllers can emulate PATA behavior. The behavior of command and control block registers, PIO and DMA data transfers, resets, and interrupts are all emulated. In addition, SATA controllers can implement a more modern register interface, namely AHCI. AHCI allows the host software to overcome the limitations that are imposed by PATA emulation and to use advanced SATA features.

Some chipsets contain both PATA and SATA controllers and support a combined mode. In combined mode, the two controllers are logically merged into one controller. The PATA drives can appear behind the SATA controller to the host software. In such a mode, all the PATA rules in terms of IDE timing configuration apply to SATA controllers.

IDE Controller Initialization Protocol References

The following sources of information are referenced in this specification or may be useful to you. See References in the master Framework help system for additional references.

- *ATA Host Adapter Standards*, Working Draft Version 0f: http://www.t13.org/*
- *Information Technology - AT Attachment with Packet Interface - 6* (ATA/ATAPI-6): http://www.t13.org/*
- *Serial ATA Advanced Host Controller Interface (AHCI) Specification*, version 1.0: http://developer.intel.com/technology/serialata/ahci.htm
- *Serial ATA: High Speed Serialized AT Attachment*, revision 1.0a (may also be referred to as *Serial ATA Specification 1.0a*): http://www.serialata.org/*
- *Serial ATA II: Port Multiplier Specification*, revision 1.1: http://www.serialata.org/*

Background

IDE Requirements

The IDE Controller Initialization Protocol is designed to work for both Parallel ATA (PATA) and Serial ATA (SATA) IDE controllers.

This protocol is designed with the following requirements in mind:

1. The timing registers in a PATA IDE controller are vendor specific. (See *ATA Host Adapter Standards,* Working Draft Version 0f, for more information.) The programming of these registers needs to be abstracted from the IDE bus driver.
2. The IDE Controller Initialization Protocol should also support a case where a specific channel is disabled and/or it should not be scanned. This protocol also needs a mechanism to address individual devices in various SATA and PATA configurations. This protocol needs to support the following:
   - A variable number of channels per controller
   - A variable number of devices per channel

   PATA controllers support up to two channels and each channel can have a maximum of two devices.

   SATA controllers can support standard ATA emulation. As described in the *Serial ATA Specification 1.0a*, ATA emulation can either be master-only emulation or master-slave emulation. In either case, the SATA controller appears to have one or two channels. In master-only emulation, a maximum of one drive appears on a channel. In master-slave emulation, one or two drives can show up behind a channel. When an SATA controller is operating in Advanced Host Controller Interface (AHCI) mode, it can support up to 32 ports. The SATA port that is generated by an SATA controller can host an SATA port multiplier. There can be up to 16 SATA devices on the other side of the SATA port multiplier. In this geometry, each SATA port that is generated by the SATA controller is treated as a channel, and this channel can have up to 16 devices. This is done so that PATA drives as well as SATA drives can be represented using a (`Channel, Device`) address pair. Note that the SATA channels work very differently from PATA channels in the sense that the SATA channels do not have the concept of master/slave or daisy chaining.

   See Figure 2-1 and Figure 2-2 below for explanations how the devices are addressed.
3. **Bus Neutral:** It should be possible to use the same abstractions to support an IDE controller on the PCI bus or some other bus. The IDE controller driver will know which controller devices it can support. Because the majority of IDE controllers that exist today are located on the PCI bus, all the examples will refer to PCI IDE controllers, but the protocol is not tied to the PCI bus.
4. PCI IDE controllers can operate in native PCI mode or compatibility mode. The IDE Controller Initialization Protocol should permit both modes.

5. The design should use the EFI Driver Model to support the quick boot feature. The smallest unit of initialization is one channel. By default, the IDE bus driver initializes only the channel on which the user-requested drive resides. The IDE Controller Initialization Protocol should support the case where various channels share the same hardware bits and cannot be independently enumerated. The controller driver can specify that all the channels should be enumerated as one unit.
6. The IDE Controller Initialization Protocol must support SATA controllers that may or may not implement AHCI register interface.

The two figures below show how the devices are addressed in various SATA and PATA configurations. Figure 2-1 below shows a PATA controller with a maximum of two channels (thick lines) and two devices per channel.

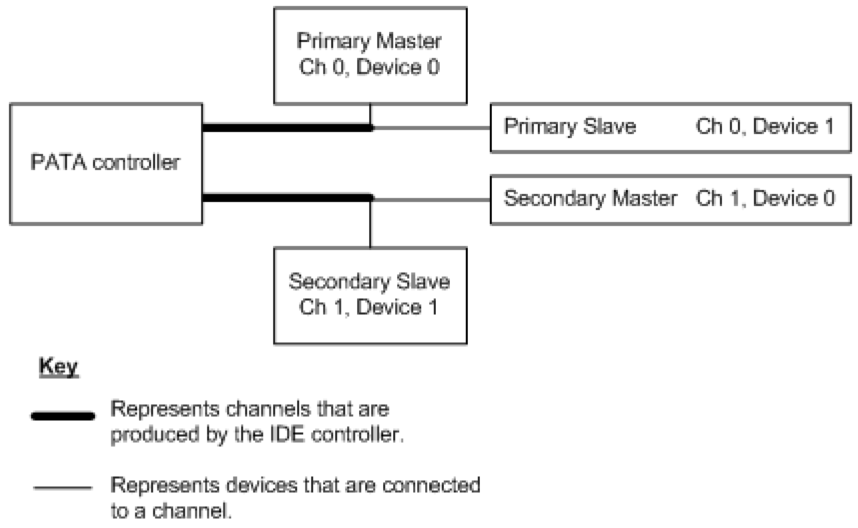
Primary Master Ch 0, Device 0
PATA controller
Primary Slave Ch 0, Device 1
Secondary Master Ch 1, Device 0
Secondary Slave Ch 1, Device 1
Key
Represents channels that are produced by the IDE controller.
Represents devices that are connected to a channel.

Figure 2-1. PATA Controller

Figure 2-2 below shows an SATA AHCI controller. The first port that is generated by the SATA controller is connected to a port multiplier. There are 16 drives connected to the port multipliers. These drives are addressed as drive 0–15 on channel 0. All other ports that are generated by the SATA controller can support a port multiplier, but they are directly connected to an SATA drive. All these devices are addressed as device 0 on the respective channel.

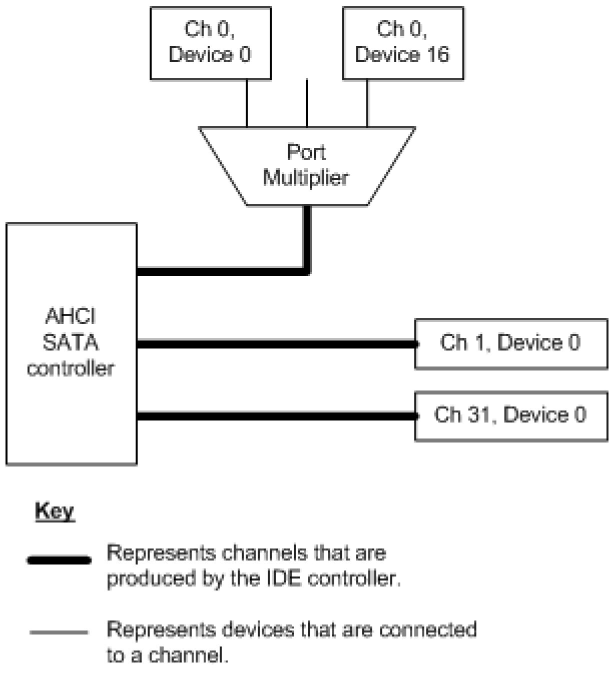
Ch 0, Device 0
Ch 0, Device 16
Port Multiplier
AHCI SATA controller
Ch 1, Device 0
Ch 31, Device 0
Key
Represents channels that are produced by the IDE controller.
Represents devices that are connected to a channel.

Figure 2-2. AHCI SATA Controller

Simplifying the Design of IDE Drivers

The IDE bus is not a general-purpose bus. The standard ATA and ATAPI command sets support only a storage class of devices. The following design decisions can be made to simplify the IDE Controller Initialization Protocol and the design of IDE drivers:

- The IDE bus driver is the only driver that will send commands to the ATA devices. No device-specific drivers are needed for IDE devices because all the devices belong to the same class (i.e., storage) and the IDE bus driver can have inherent knowledge of these commands. IDE bus equivalents of **`EFI_PCI_IO_PROTOCOL`** and **`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`** for accessing IDE devices are not required. It is possible to further simplify the design of the IDE bus driver if it does not have to deal with the ATAPI devices. It can enumerate the ATA and ATAPI devices and install the **`EFI_SCSI_PASSTHRU_PROTOCOL`** on ATAPI device handles. Either way, IDE-bus-specific I/O protocols are not needed. See the *EFI 1.10 Specification* for the definitions of the EFI PCI I/O Protocol, PCI Root Bridge I/O Protocol, and the SCSI Pass Thru Protocol.
- IDE devices are accessed and configured through a set of standard registers in the IDE controller. The ATA committee is standardizing the layout of these registers. (See *ATA Host Adapter Standards,* Working Draft Version 0f, for more information.) For Serial ATA (SATA) controllers, the *Serial ATA Advanced Host Controller Interface (AHCI) Specification* defines a standard register interface. Although the layout is dependent on the bus on which the controller is located, the layout for a particular bus is fixed. As a result, the IDE bus driver can be required to know about the register layout for buses that it chooses to support. For example, for a PCI IDE controller, the IDE driver can access the base of the command block register for channel 0 using the following steps:

1. Check bit 0 of register 0x9 (Programming Interface Code) in the PCI configuration space of the controller to determine whether it is operating in compatibility mode or native PCI mode. For this example, we will assume that the controller is operating in native mode.
2. Read register 0x10 (Base Address Register [BAR] 0) of the controller. Clear bit 0 of the value that was read to get the command block base

Configuring Devices on the IDE Bus

The table below lists the various drivers that may participate in configuring the devices on the IDE bus.

Table 2-1. Drivers Involved in Configuring IDE Devices

Driver	Follows the EFI Driver Model?	Description
IDE controller driver	Yes	Produces the **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. Consumes the bus-specific I/O protocol. **EFI_IDE_CONTROLLER_INIT_PROTOCOL** abstracts the chipset-specific IDE controller registers and is responsible for early initialization of the IDE controller. Note that **EFI_IDE_CONTROLLER_INIT_PROTOCOL** is not tied to a specific bus although most IDE controllers today are on the PCI or ISA bus.
IDE bus driver	Yes	Consumes the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** and the bus-specific I/O protocol. It enumerates the IDE buses. This driver will check for the presence of the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** on the controller handle before enumerating the child devices. This driver uses the presence of the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** to determine whether a controller is an IDE controller or not. This driver will use bus-specific methods to access the standard ATA registers (such as the control block, command block, and bus master DMA registers) for a particular device. The driver not only knows the address of a specific register block, but it also knows the layout of that register block. This driver may produce the **EFI_SCSI_PASSTHRU_PROTOCOL** for ATAPI devices or it may directly manage the ATAPI devices by producing the **EFI_BLOCK_IO_PROTOCOL**. This driver produces the **EFI_BLOCK_IO_PROTOCOL** for ATA devices.
Generic SCSI or ATAPI storage driver	Yes	This optional driver manages the ATAPI device using the **EFI_SCSI_PASSTHRU_PROTOCOL** and produces the **EFI_BLOCK_IO_PROTOCOL** if requested.

continued

Table 2-1. Drivers Involved in Configuring IDE Devices (continued)

Driver	Follows the EFI Driver Model?	Description
IDE bus driver and IDE controller driver combined as one driver	Yes	It is also possible to combine the IDE bus driver and the IDE controller driver into one driver. In this case, **`EFI_IDE_CONTROLLER_INIT_PROTOCOL`** is not installed on the IDE controller handle. The monolithic driver is responsible for initializing the IDE controller as well as the IDE devices behind that controller. **`EFI_IDE_CONTROLLER_INIT_PROTOCOL`** is mandatory if the IDE devices behind the controller are to be enumerated by the generic IDE bus driver.

See the *EFI 1.10 Specification* for the definitions of the Block I/O Protocol and the SCSI Pass Thru Protocol. The IDE Controller Initialization Protocol is defined in Code Definitions in this specification.

Sample Implementation for a Simple PCI IDE Controller

This topic provides a sample implementation only. The sequencing of various notifications cannot be changed. The steps below apply if **`EFI_IDE_CONTROLLER_INIT_PROTOCOL`**`.`*`EnumAll`* **`= FALSE`**.

See the *EFI 1.10 Specification* for definitions of the Driver Binding Protocol, EFI PCI I/O Protocol, Device Path Protocol, and Block I/O Protocol. See Code Definitions in this specification for the definition of the IDE Controller Initialization Protocol.

1. The IDE controller driver as well as the IDE bus driver follow the EFI Driver Model. They are loaded and both install (at least) one instance of the **`EFI_DRIVER_BINDING_PROTOCOL`** on their image handle. An ATA hard drive behind a PCI IDE controller is one of the boot devices.
2. The PCI bus driver enumerates the PCI bus, finds the PCI IDE controller, creates a handle for it, and installs an instance of **`EFI_PCI_IO_PROTOCOL`** and **`EFI_DEVICE_PATH_PROTOCOL`** on that handle.
3. The Boot Device Selection (BDS) phase searches for an appropriate driver to own the IDE controller device and finds the IDE controller driver. It then connects the IDE controller device and the IDE controller driver. The IDE controller driver opens the **`EFI_PCI_IO_PROTOCOL BY_DRIVER`**. It may perform some other preprogramming at this point.
4. BDS searches for a driver to own the IDE device and finds the IDE bus driver. The IDE bus driver's **`Supported()`** function checks for the presence of **`EFI_IDE_CONTROLLER_INIT_PROTOCOL`** on the parent of the IDE device (i.e., the IDE controller).
5. The EFI Boot Services function **`ConnectController()`** calls the **`Start()`** function of the IDE bus driver, which starts the IDE bus enumeration. The following steps are performed by the **`Start()`** function.
   a. The IDE bus driver locates the **`EFI_IDE_CONTROLLER_INIT_PROTOCOL`**. It opens the **`EFI_IDE_CONTROLLER_INIT_PROTOCOL BY_DRIVER`**. If it needs to open

EFI_PCI_IO_PROTOCOL, it may open it by **GET_PROTOCOL**. The IDE bus driver reads the *EnumAll* and *ChannelCount* fields in **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. In this case, *EnumAll* is **FALSE**. The IDE bus driver also obtains the channel number from **Start().***RemainingDevicePath*.

b. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase (This, EfiIdeBeforeChannelEnumeration, Channel)**.

c. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo (This, Channel, *Enabled, *MaxDevices)** to find out the number of devices on this channel. If **Enabled* **= FALSE**, it exits with an error code. If the device number of the device to be connected is too large, it exits with an error code.

d. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase (This, EfiIdeBeforeChannelReset, Channel)**.

e. The IDE bus driver resets the channel.

f. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase (This, EfiIdeAfterChannelReset, Channel)**.

g. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase (This, EfiIdeBeforeDevicePresenceDetection, Channel)**. The IDE controller driver may insert a predelay here or may ensure that various IDE bus signals are at desired levels.

h. The IDE bus driver attempts to detect devices on the channel. Note than there can be no more than *MaxDevices* on the channel.

i. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase (This, EfiIdeAfterDevicePresenceDetection, Channel)**.

j. The IDE bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase (This, EfiIdeResetMode, Channel)**. The IDE controller sets up the controller with the default timings.

k. For all the devices on this channel:

   1. The IDE bus driver gathers **EFI_IDENTIFY_DATA** for the device and submits it to the IDE controller driver using **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()**. Submit **NULL** data for devices that do not exist.

   2. The IDE bus driver may call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()** to disqualify modes that it does not support.

6. For all the detected devices on this channel:
   a. Call **`EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()`** to get the optimum mode settings. The IDE controller driver uses controller-specific algorithms and platform information to calculate the best modes.
      `EFI_PLATFORM_IDE_INIT_PROTOCOL` provides platform information such as the following during this calculation:
      - User policies (for example, setup options for manual mode selection)
      - Platform-implementation details (for example, the platform does not support UDMA mode 4 or the cable is not an 80 pin cable)

      `EFI_PLATFORM_IDE_INIT_PROTOCOL` is defined in the *Intel® Platform Innovation Framework for EFI Platform IDE Initialization Protocol Specification*.
   b. The IDE bus driver enables the appropriate modes by sending an ATA **`SET_FEATURES`** command to the device. It the device returns an error, it disqualifies that mode for that device and goes back to step 6. This time step 6a will not consider the failed mode. The implementation then returns here to step 6b with new (less optimum) modes.
7. For all the detected devices on this channel, call **`EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming()`** to program the timings. Note that we reset the mode settings in step 5j, so the settings for nonexistent devices will remain at their default levels.
8. The IDE bus driver calls **`EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase ( This, EfiIdeAfterChannelEnumeration, Channel)`**.
9. Install **`EFI_BLOCK_IO_PROTOCOL`** on that device handle.

3 Code Definitions

Introduction

This section contains the basic definitions of the IDE Controller Initialization Protocol. The following protocol is defined in this section:

- EFI_IDE_CONTROLLER_INIT_PROTOCOL

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

- EFI_IDE_CONTROLLER_ENUM_PHASE
- EFI_IDENTIFY_DATA
- EFI_ATA_IDENTIFY_DATA
- EFI_ATAPI_IDENTIFY_DATA
- EFI_ATA_COLLECTIVE_MODE
- EFI_ATA_MODE
- EFI_ATA_EXTENDED_MODE
- EFI_ATA_EXT_TRANSFER_PROTOCOL

IDE Controller Initialization Protocol

EFI_IDE_CONTROLLER_INIT_PROTOCOL

Summary

Provides the basic interfaces to abstract an IDE controller.

GUID

```
#define EFI_IDE_CONTROLLER_INIT_PROTOCOL_GUID \
{ 0xa1e37052, 0x80d9, 0x4e65, 0xa3, 0x17, 0x3e, 0x9a, 0x55, 0xc4,
0x3e, 0xc9 }
```

Protocol Interface Structure

```
typedef struct _EFI_IDE_CONTROLLER_INIT_PROTOCOL {
  EFI_IDE_CONTROLLER_GET_CHANNEL_INFO    GetChannelInfo;
  EFI_IDE_CONTROLLER_NOTIFY_PHASE        NotifyPhase;
  EFI_IDE_CONTROLLER_SUBMIT_DATA         SubmitData;
  EFI_IDE_CONTROLLER_DISQUALIFY_MODE     DisqualifyMode;
  EFI_IDE_CONTROLLER_CALCULATE_MODE      CalculateMode;
  EFI_IDE_CONTROLLER_SET_TIMING          SetTiming;
  BOOLEAN                                EnumAll;
  UINT8                                  ChannelCount;
} EFI_IDE_CONTROLLER_INIT_PROTOCOL;
```

Parameters

GetChannelInfo

Returns the information about a specific channel. See the **GetChannelInfo()** function description.

NotifyPhase

The notification that the IDE bus driver is about to enter the specified phase during the enumeration process. See the **NotifyPhase()** function description.

SubmitData

Submits the Drive Identify data that was returned by the device. See the **SubmitData()** function description.

DisqualifyMode

Submits information about modes that should be disqualified. The specified IDE device does not support these modes and these modes should not be returned by *CalculateMode* . See the **DisqualifyMode()** function description.

CalculateMode

Calculates and returns the optimum mode for a particular IDE device. See the **CalculateMode()** function description.

SetTiming

Programs the IDE controller hardware to the default timing or per the modes that were returned by the last call to **CalculateMode()**. See the **SetTiming()** function description.

EnumAll

Set to **TRUE** if the enumeration group includes all the channels that are produced by this controller. **FALSE** if an enumeration group consists of only one channel.

ChannelCount

The number of channels that are produced by this controller. Parallel ATA (PATA) controllers can support up to two channels. Advanced Host Controller Interface (AHCI) Serial ATA (SATA) controllers can support up to 32 channels, each of which can have up to one device.

Description

The **EFI_IDE_CONTROLLER_INIT_PROTOCOL** provides the chipset-specific information to the IDE bus driver. This protocol is mandatory for IDE controllers if the IDE devices behind the controller are to be enumerated by an IDE bus driver.

There can only be one instance of **EFI_IDE_CONTROLLER_INIT_PROTOCOL** for each IDE controller in a system. It is installed on the handle that corresponds to the IDE controller. An IDE bus driver that wishes to manage an IDE bus and possibly IDE devices in a system will have to retrieve the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance that is associated with the controller to be managed.

A device handle for an IDE controller must contain an **EFI_DEVICE_PATH_PROTOCOL**.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo()

Summary

Returns the information about the specified IDE channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_GET_CHANNEL_INFO) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN UINT8                              Channel,
  OUT BOOLEAN                           *Enabled,
  OUT UINT8                             *MaxDevices
  );
```

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Enabled

TRUE if this channel is enabled. Disabled channels are not scanned to see if any devices are present.

MaxDevices

The maximum number of IDE devices that the bus driver can expect on this channel. For the ATA/ATAPI specification, version 6, this number will either be 1 or 2. For Serial ATA (SATA) configurations with a port multiplier, this number can be as large as 16.

Description

This function can be used to obtain information about a particular IDE channel. The IDE bus driver uses this information during the enumeration process.

If `Enabled` is set to **`FALSE`**, the IDE bus driver will not scan the channel. Note that it will not prevent an operating system driver from scanning the channel.

For most of today's controllers, `MaxDevices` will either be 1 or 2. For SATA controllers, this value will always be 1. SATA configurations can contain SATA port multipliers. SATA port multipliers behave like SATA bridges and can support up to 16 devices on the other side. If an SATA port out of the IDE controller is connected to a port multiplier, `MaxDevices` will be set to the number of SATA devices that the port multiplier supports. Because today's port multipliers support up to 16 SATA devices, this number can be as large as 16. The IDE bus driver is required to scan for the presence of port multipliers behind an SATA controller and enumerate up to `MaxDevices` number of devices behind the port multiplier.

In this context, the devices behind a port multiplier constitute a channel.

Status Codes Returned

EFI_SUCCESS	Information was returned without any errors.
EFI_INVALID_PARAMETER	`Channel` is invalid (`Channel >= ChannelCount`).

EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()

Summary

The notifications from the IDE bus driver that it is about to enter a certain phase of the IDE channel enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_NOTIFY_PHASE) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN EFI_IDE_CONTROLLER_ENUM_PHASE      Phase,
  IN UINT8                              Channel
  );
```

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Phase

The phase during enumeration. Type **EFI_IDE_CONTROLLER_ENUM_PHASE** is defined in "Related Definitions" below.

Channel

Zero-based channel number.

Description

This function can be used to notify the IDE controller driver to perform specific actions, including any chipset-specific initialization, so that the chipset is ready to enter the next phase. Seven notification points are defined at this time. See "Related Definitions" below for the definition of various notification points and Sample Implementation for a Simple PCI IDE Controller in the Design Discussion chapter for usage.

More synchronization points may be added as required in the future.

Related Definitions

```
//*******************************************************
// EFI_IDE_CONTROLLER_ENUM_PHASE
//*******************************************************
typedef enum {
  EfiIdeBeforeChannelEnumeration,
  EfiIdeAfterChannelEnumeration,
  EfiIdeBeforeChannelReset,
  EfiIdeAfterChannelReset,
  EfiIdeBusBeforeDevicePresenceDetection,
  EfiIdeBusAfterDevicePresenceDetection,
  EfiIdeResetMode,
  EfiIdeBusPhaseMaximum
} EFI_IDE_CONTROLLER_ENUM_PHASE;
```

Following is a description of the fields in the above definition.

EfiIdeBeforeChannelEnumeration	The IDE bus driver is about to begin enumerating the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeAfterChannelEnumeration	The IDE bus driver has completed enumerating the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeBeforeChannelReset	The IDE bus driver is about to reset the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeAfterChannelReset	The IDE bus driver has completed resetting the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeBusBeforeDevicePresenceDetection	The IDE bus driver is about to detect the presence of devices behind the specified channel. This notification can be used to set up the bus signals to default levels or for implementing predelays.
EfiIdeBusAfterDevicePresenceDetection	The IDE bus driver is done with detecting the presence of devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeResetMode	The IDE bus is requesting the IDE controller driver to reprogram the IDE controller hardware and thereby reset all the mode and timing settings to default settings.

Status Codes Returned

EFI_SUCCESS	The notification was accepted without any errors.
EFI_NOT_SUPPORTED	*Phase* is not supported.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_NOT_READY	This phase cannot be entered at this time; for example, an attempt was made to enter a *Phase* without having entered one or more previous *Phase*.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()

Summary

Submits the device information to the IDE controller driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_SUBMIT_DATA) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN UINT8                              Channel,
  IN UINT8                              Device,
  IN EFI_IDENTIFY_DATA                  *IdentifyData
  );
```

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

IdentifyData

The device's response to the ATA **IDENTIFY_DEVICE** command. Type **EFI_IDENTIFY_DATA** is defined in "Related Definitions" below.

Description

This function is used by the IDE bus driver to pass detailed information about a particular device to the IDE controller driver. The IDE bus driver obtains this information by issuing an ATA or ATAPI **IDENTIFY_DEVICE** command. *IdentifyData* is the pointer to the response data buffer. The *IdentifyData* buffer is owned by the IDE bus driver, and the IDE controller driver must make a local copy of the entire buffer or parts of the buffer as needed. The original *IdentifyData* buffer pointer may not be valid when **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** or **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()** is called at a later point.

The IDE controller driver may consult various fields of **EFI_IDENTIFY_DATA** to compute the optimum mode for the device. These fields are not limited to the timing information. For example, an implementation of the IDE controller driver may examine the vendor and type/mode field to match known bad drives.

The IDE bus driver may submit drive information in any order, as long as it submits information for all the devices belonging to the enumeration group before **CalculateMode()** is called for any device in that enumeration group. If a device is absent, **SubmitData()** should be called with

IdentifyData set to **NULL**. The IDE controller driver may not have any other mechanism to know whether a device is present or not. Therefore, setting *IdentifyData* to **NULL** does not constitute an error condition. **SubmitData()** can be called only once for a given (*Channel*, *Device*) pair.

Related Definitions

```
//*******************************************************
// EFI_IDENTIFY_DATA
//*******************************************************
typedef union {
  EFI_ATA_IDENTIFY_DATA        AtaData;
  EFI_ATAPI_IDENTIFY_DATA      AtapiData;
} EFI_IDENTIFY_DATA;

#define   EFI_ATAPI_DEVICE_IDENTIFY_DATA   0x8000
```

AtaData

The data that is returned by an ATA device upon successful completion of the ATA **IDENTIFY_DEVICE** command. The **IDENTIFY_DEVICE** command is defined in the ATA/ATAPI specification. Type **EFI_ATA_IDENTIFY_DATA** is defined below.

AtapiData

The data that is returned by an ATAPI device upon successful completion of the ATA **IDENTIFY_PACKET_DEVICE** command. The **IDENTIFY_PACKET_DEVICE** command is defined in the ATA/ATAPI specification. Type **EFI_ATAPI_IDENTIFY_DATA** is defined below.

Following is a description of the field in the above definition.

EFI_ATAPI_DEVICE_IDENTIFY_DATA	This flag indicates whether the IDENTIFY data is a response from an ATA device (**EFI_ATA_IDENTIFY_DATA**) or response from an ATAPI device (**EFI_ATAPI_IDENTIFY_DATA**). According to the ATA/ATAPI specification, **EFI_IDENTIFY_DATA** is for an ATA device if bit 15 of the Config field is zero. The Config field is common to both **EFI_ATA_IDENTIFY_DATA** and **EFI_ATAPI_IDENTIFY_DATA**.

```
//*******************************************************
// EFI_ATA_IDENTIFY_DATA
//*******************************************************
//
// This structure definition is not part of the protocol
// definition because the ATA/ATAPI Specification controls
// the definition of all the fields. The ATA/ATAPI
// Specification can obsolete old fields or redefine existing
// fields. This definition is provided here for reference only.
//

#pragma pack(1)
typedef struct {
  UINT16  config;             // General Configuration
  UINT16  cylinders;          // Number of Cylinders
  UINT16  reserved_2;
  UINT16  heads;              //Number of logical heads
  UINT16  vendor_data1;
  UINT16  vendor_data2;
  UINT16  sectors_per_track;
  UINT16  vendor_specific_7_9[3];
  CHAR8   SerialNo[20];       // ASCII
  UINT16  vendor_specific_20_21[2];
  UINT16  ecc_bytes_available;
  CHAR8   FirmwareVer[8];     // ASCII
  CHAR8   ModelName[40];      // ASCII
  UINT16  multi_sector_cmd_max_sct_cnt;
  UINT16  reserved_48;
  UINT16  capabilities;
  UINT16  reserved_50;
  UINT16  pio_cycle_timing;
  UINT16  reserved_52;
  UINT16  field_validity;
  UINT16  current_cylinders;
  UINT16  current_heads;
  UINT16  current_sectors;
  UINT16  CurrentCapacityLsb;
  UINT16  CurrentCapacityMsb;
  UINT16  reserved_59;
  UINT16  user_addressable_sectors_lo;
  UINT16  user_addressable_sectors_hi;
  UINT16  reserved_62;
  UINT16  multi_word_dma_mode;
  UINT16  advanced_pio_modes;
  UINT16  min_multi_word_dma_cycle_time;
  UINT16  rec_multi_word_dma_cycle_time;
  UINT16  min_pio_cycle_time_without_flow_control;
  UINT16  min_pio_cycle_time_with_flow_control;
  UINT16  reserved_69_79[11];
```

```
    UINT16  major_version_no;
    UINT16  minor_version_no;
    UINT16  command_set_supported_82; // word 82
    UINT16  command_set_supported_83; // word 83
    UINT16  command_set_feature_extn; // word 84
    UINT16  command_set_feature_enb_85; // word 85
    UINT16  command_set_feature_enb_86; // word 86
    UINT16  command_set_feature_default; // word 87
    UINT16  ultra_dma_mode; // word 88
    UINT16  reserved_89_127[39];
    UINT16  security_status;
    UINT16  vendor_data_129_159[31];
    UINT16  reserved_160_255[96];
  } EFI_ATA_IDENTIFY_DATA;
  #pragma pack()



  //*********************************************************
  // EFI_ATAPI_IDENTIFY_DATA
  //*********************************************************
  #pragma pack(1)
  typedef struct {
      UINT16  config;              // General Configuration
      UINT16  obsolete_1;
      UINT16  specific_config;
      UINT16  obsolete_3;
      UINT16  retired_4_5[2];
      UINT16  obsolete_6;
      UINT16  cfa_reserved_7_8[2];
      UINT16  retired_9;
      CHAR8   SerialNo[20];        // ASCII
      UINT16  retired_20_21[2];
      UINT16  obsolete_22;
      CHAR8   FirmwareVer[8];      // ASCII
      CHAR8   ModelName[40];       // ASCII
      UINT16  multi_sector_cmd_max_sct_cnt;
      UINT16  reserved_48;
      UINT16  capabilities_49;
      UINT16  capabilities_50;
      UINT16  obsolete_51_52[2];
      UINT16  field_validity;
      UINT16  obsolete_54_58[5];
      UINT16  mutil_sector_setting;
      UINT16  user_addressable_sectors_lo;
      UINT16  user_addressable_sectors_hi;
      UINT16  obsolete_62;
      UINT16  multi_word_dma_mode;
      UINT16  advanced_pio_modes;
```

```
        UINT16  min_multi_word_dma_cycle_time;
        UINT16  rec_multi_word_dma_cycle_time;
        UINT16  min_pio_cycle_time_without_flow_control;
        UINT16  min_pio_cycle_time_with_flow_control;
        UINT16  reserved_69_74[6];
        UINT16  queue_depth;
        UINT16  reserved_76_79[4];
        UINT16  major_version_no;
        UINT16  minor_version_no;
        UINT16  cmd_set_support_82;
        UINT16  cmd_set_support_83;
        UINT16  cmd_feature_support;
        UINT16  cmd_feature_enable_85;
        UINT16  cmd_feature_enable_86;
        UINT16  cmd_feature_default;
        UINT16  ultra_dma_select;
        UINT16  time_required_for_sec_erase;
        UINT16  time_required_for_enhanced_sec_erase;
        UINT16  current_advanced_power_mgmt_value;
        UINT16  master_pwd_revison_code;
        UINT16  hardware_reset_result;
        UINT16  current_auto_acoustic_mgmt_value;
        UINT16  reserved_95_99[5];
        UINT16  max_user_lba_for_48bit_addr[4];
        UINT16  reserved_104_126[23];
        UINT16  removable_media_status_notification_support;
        UINT16  security_status;
        UINT16  vendor_data_129_159[31];
        UINT16  cfa_power_mode;
        UINT16  cfa_reserved_161_175[15];
        UINT16  current_media_serial_no[30];
        UINT16  reserved_206_254[49];
        UINT16  integrity_word;
    } EFI_ATAPI_IDENTIFY_DATA;
    #pragma pack()
```

Status Codes Returned

EFI_SUCCESS	The information was accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()

Summary

Disqualifies specific modes for an IDE device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_DISQUALIFY_MODE) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN UINT8                              Channel,
  IN UINT8                              Device,
  IN EFI_ATA_COLLECTIVE_MODE            *BadModes
  );
```

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

BadModes

The modes that the device does not support and that should be disqualified. Type **EFI_ATA_COLLECTIVE_MODE** is defined in "Related Definitions" below.

Description

This function allows the IDE bus driver or other drivers (such as platform drivers) to reject certain timing modes and request the IDE controller driver to recalculate modes. This function allows the IDE bus driver and the IDE controller driver to negotiate the timings on a per-device basis. This function is useful in the case of drives that lie about their capabilities. An example is when the IDE device fails to accept the timing modes that are calculated by the IDE controller driver based on the response to the Identify Drive command.

If the IDE bus driver does not want to limit the ATA timing modes and leave that decision to the IDE controller driver, it can either not call this function for the given device or call this function and set the *Valid* flag to **FALSE** for all modes that are listed in **EFI_ATA_COLLECTIVE_MODE**.

The IDE bus driver may disqualify modes for a device in any order and any number of times.

This function can be called multiple times to invalidate multiple modes of the same type (e.g., Programmed Input/Output [PIO] modes 3 and 4). See the ATA/ATAPI specification for more information on PIO modes.

For Serial ATA (SATA) controllers, this member function can be used to disqualify a higher transfer rate mode on a given channel. For example, a platform driver may inform the IDE controller driver to not use second-generation (Gen2) speeds for a certain SATA drive.

Related Definitions

```
//*******************************************************
// EFI_ATA_COLLECTIVE_MODE
//*******************************************************
typedef struct {
  EFI_ATA_MODE            PioMode;
  EFI_ATA_MODE            SingleWordDmaMode;
  EFI_ATA_MODE            MultiWordDmaMode;
  EFI_ATA_MODE            UdmaMode;
  UINT32                  ExtModeCount;
  EFI_ATA_EXTENDED_MODE   ExtMode[1];
} EFI_ATA_COLLECTIVE_MODE;
```

PioMode

This field specifies the PIO mode. PIO modes are defined in the ATA/ATAPI specification. The ATA/ATAPI specification defines the enumeration. In other words, a value of 1 in this field means PIO mode 1. The actual meaning of PIO mode 1 is governed by the ATA/ATAPI specification. Type **EFI_ATA_MODE** is defined below.

SingleWordDmaMode

This field specifies the single word DMA mode. Single word DMA modes are defined in the ATA/ATAPI specification, versions 1 and 2. Single word DMA support was obsoleted in the ATA/ATAPI specification, version 3; therefore, most devices and controllers will not support this transfer mode. The ATA/ATAPI specification defines the enumeration. In other words, a value of 1 in this field means single word DMA mode 1. The actual meaning of single word DMA mode 1 is governed by the ATA/ATAPI specification.

MultiWordDmaMode

This field specifies the multiword DMA mode. Various multiword DMA modes are defined in the ATA/ATAPI specification. A value of 1 in this field means multiword DMA mode 1. The actual meaning of multiword DMA mode 1 is governed by the ATA/ATAPI specification.

UdmaMode

This field specifies the ultra DMA (UDMA) mode. UDMA modes are defined in the ATA/ATAPI specification. A value of 1 in this field means UDMA mode 1. The actual meaning of UDMA mode 1 is governed by the ATA/ATAPI specification.

ExtModeCount

The number of extended-mode bitmap entries. Extended modes describe transfer protocols beyond PIO, single word DMA, multiword DMA, and UDMA. This field can be zero and provides extensibility.

ExtMode

ExtModeCount number of entries. Each entry represents a transfer protocol other than the ones defined above (i.e., PIO, single word DMA, multiword DMA, and UDMA). This field is defined for extensibility. At this time, only one extended transfer protocol is defined to cover SATA transfers. Type **EFI_ATA_EXTENDED_MODE** is defined below.

```
//*******************************************************
// EFI_ATA_MODE
//*******************************************************
typedef struct {
  BOOLEAN   Valid;
  UINT32    Mode;
} EFI_ATA_MODE;
```

Valid

TRUE if *Mode* is valid.

Mode

The actual ATA mode. This field is not a bit map.

```
//*******************************************************
// EFI_ATA_EXTENDED_MODE
//*******************************************************
typedef struct {
  EFI_ATA_EXT_TRANSFER_PROTOCOL  TransferProtocol;
  UINT32                         Mode;
} EFI_ATA_EXTENDED_MODE;
```

TransferProtocol

An enumeration defining various transfer protocols other than the protocols that exist at the time this specification was developed (i.e., PIO, single word DMA, multiword DMA, and UDMA). Each transfer protocol is associated with a mode. The various transfer protocols are defined by the ATA/ATAPI specification. This enumeration makes the interface extensible because we can support new transport protocols beyond UDMA. Type **EFI_ATA_EXT_TRANSFER_PROTOCOL** is defined below.

Mode

The mode for operating the transfer protocol that is identified by *TransferProtocol*.

```
//*******************************************************
// EFI_ATA_EXT_TRANSFER_PROTOCOL
//*******************************************************
//
// This extended mode describes the SATA physical protocol.
// SATA physical layers can operate at different speeds.
// These speeds are defined below. Various PATA protocols
// and associated modes are not applicable to SATA devices.
//
typedef enum {
  EfiAtaSataTransferProtocol
} EFI_ATA_EXT_TRANSFER_PROTOCOL;

#define  EFI_SATA_AUTO_SPEED  0
#define  EFI_SATA_GEN1_SPEED  1
#define  EFI_SATA_GEN2_SPEED  2
```

Following is a description of the fields in the above definition.

EFI_SATA_AUTO_SPEED	Automatically detects the optimum SATA speed.
EFI_SATA_GEN1_SPEED	Indicates a first-generation (Gen1) SATA speed.
EFI_SATA_GEN2_SPEED	Indicates a second-generation (Gen2) SATA speed.

Status Codes Returned

EFI_SUCCESS	The modes were accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.
EFI_INVALID_PARAMETER	*IdentifyData* is **NULL**.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()

Summary

Returns the information about the optimum modes for the specified IDE device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_CALCULATE_MODES) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN UINT8                              Channel,
  IN UINT8                              Device,
  OUT EFI_ATA_COLLECTIVE_MODE           **SupportedModes
  );
```

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

SupportedModes

The optimum modes for the device. Type **EFI_ATA_COLLECTIVE_MODE** is defined in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()**.

Description

This function is used by the IDE bus driver to obtain the optimum ATA modes for a specific device. The IDE controller driver takes into account the following while calculating the mode:

- The *IdentifyData* inputs to **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()**
- The *BadModes* inputs to **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()**

The IDE bus driver is required to call **SubmitData()** for all the devices that belong to an enumeration group before calling **CalculateMode()** for any device in the same group.

The IDE controller driver will use controller- and possibly platform-specific algorithms to arrive at *SupportedModes*. The IDE controller may base its decision on user preferences and other considerations as well. This function may be called multiple times because the IDE bus driver may renegotiate the mode with the IDE controller driver using **DisqualifyMode()**.

The IDE bus driver may collect timing information for various devices in any order. The IDE bus driver is responsible for making sure that all the dependencies are satisfied; for example, the

SupportedModes information for device A that was previously returned may become stale after a call to **DisqualifyMode()** for device B.

The buffer *SupportedModes* is allocated by the callee because the caller does not necessarily know the size of the buffer. The type **EFI_ATA_COLLECTIVE_MODE** is defined in a way that allows for future extensibility and can be of variable length. This memory pool should be deallocated by the caller when it is no longer necessary.

The IDE controller driver for a Serial ATA (SATA) controller can use this member function to force a lower speed (first-generation [Gen1] speeds on a second-generation [Gen2]–capable hardware). The IDE controller driver can also allow the IDE bus driver to stay with the speed that has been negotiated by the physical layer.

Status Codes Returned

EFI_SUCCESS	*SupportedModes* was returned.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.
EFI_INVALID_PARAMETER	*SupportedModes* is **NULL**.
EFI_NOT_READY	Modes cannot be calculated due to a lack of data. This error may happen if **SubmitData()** and **DisqualifyData()** were not called for at least one drive in the same enumeration group.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming()

Summary

Commands the IDE controller driver to program the IDE controller hardware so that the specified device can operate at the specified mode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_SET_TIMING) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN UINT8                              Channel,
  IN UINT8                              Device,
  IN EFI_ATA_COLLECTIVE_MODE            *Modes
  );
```

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

Modes

The modes to set. Type **EFI_ATA_COLLECTIVE_MODE** is defined in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()**.

Description

This function is used by the IDE bus driver to instruct the IDE controller driver to program the IDE controller hardware to the specified modes. This function can be called only once for a particular device. For a Serial ATA (SATA) Advanced Host Controller Interface (AHCI) controller, no controller-specific programming may be required.

Status Codes Returned

EFI_SUCCESS	The command was accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.
EFI_NOT_READY	*Modes* cannot be set at this time due to lack of data.
EFI_DEVICE_ERROR	*Modes* cannot be set due to hardware failure. The IDE bus driver should not use this device.