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

<u>Draft for Review</u>

Version 0.3

August 9, 2004

Revision History

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

Contents

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the Platform IDE Initialization Protocol of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This protocol abstracts the platform aspects of the IDE channels that the IDE-controller-specific driver cannot know and is used by an IDE controller driver to obtain platform-specific information. This specification does the following:

- Describes the basic components of the Platform IDE Initialization Protocol
- Provides code definitions for the Platform IDE Initialization Protocol and platform-IDE-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

Platform IDE Initialization Protocol Overview

The Platform IDE Initialization Protocol is used by an IDE controller driver to obtain platform-specific information. This protocol abstracts the platform aspects of the IDE channels that the IDE-controller-specific driver cannot know. This protocol is not tied to any specific bus or any specific IDE controller hardware. This protocol is optional.

See the Code Definitions section in this specification for the definition of **`EFI_PLATFORM_IDE_INIT_PROTOCOL`**.

See the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification* for additional background information and design discussion about the IDE subsystem.

Platform IDE Initialization Protocol Related Information

The following sources of information are referenced in this specification or may be useful to you. See References in the Framework master 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/*
- *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*, version 0.9

Platform IDE Initialization Protocol Terms

See Design Discussion > IDE Controller Terms in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification* for definitions of IDE-related terms.

3 Code Definitions

Introduction

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

- `EFI_PLATFORM_IDE_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_CABLE_TYPE`
- `EFI_ATA_COLLECTIVE_MODE_BITMAP`
- `EFI_ATA_MODE_BITMAP`
- `EFI_ATA_EXTENDED_MODE_BITMAP`

Platform IDE Initialization Protocol

EFI_PLATFORM_IDE_INIT_PROTOCOL

Summary

Provides the basic interfaces to abstract the platform-specific aspects of the IDE bus. **This protocol is optional.**

GUID

```
#define EFI_PLATFORM_IDE_INIT_PROTOCOL_GUID \
{ 0x377c66a3, 0x8fe7, 0x4ee8, 0x85, 0xb8, 0xf1, 0xa2, 0x82, 0x56,
0x9e, 0x3b };
```

Protocol Interface Structure

```
typedef struct _EFI_PLATFORM_IDE_INIT_PROTOCOL {
  EFI_PLATFORM_IDE_GET_CHANNEL_INFO     GetChannelInfo;
  EFI_PLATFORM_IDE_NOTIFY_PHASE         NotifyPhase;
  EFI_PLATFORM_IDE_SUBMIT_DATA          SubmitData;
  EFI_PLATFORM_IDE_OVERRIDE_MODES       OverrideModes;
} EFI_PLATFORM_IDE_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.

OverrideModes

Overrides the default mode settings by the IDE controller driver. See the **OverrideModes()** function description.

Description

The **EFI_PLATFORM_IDE_INIT_PROTOCOL** provides platform-specific information to the IDE controller driver. There cannot be more than one instance of **EFI_PLATFORM_IDE_INIT_PROTOCOL** in a system. This protocol is installed on a separate handle.

A platform-specific driver produces the **EFI_PLATFORM_IDE_INIT_PROTOCOL**. Note that this functionality can be part of a platform driver that produces multiple platform-specific protocols. The driver that produces **EFI_PLATFORM_IDE_INIT_PROTOCOL** does not follow the EFI Driver Model.

This protocol is optional and may not be required. For example, if all the channels are always enabled and the IDE controller can find out the cable type or does not need to find out the cable type, **EFI_PLATFORM_IDE_INIT_PROTOCOL** may not be needed. The IDE controller driver consumes the **EFI_PLATFORM_IDE_INIT_PROTOCOL**. The IDE controller driver and other drivers that participate in IDE configuration are described in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

EFI_PLATFORM_IDE_INIT_PROTOCOL.GetChannelInfo()

Summary

Returns information about the specified IDE channel behind a specified controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PLATFORM_IDE_GET_CHANNEL_INFO) (
  IN EFI_PLATFORM_IDE_INIT_PROTOCOL   *This,
  IN EFI_HANDLE                       Controller,
  IN UINT8                            Channel,
  OUT BOOLEAN                         *Enabled,
  OUT UINT8                           *MaxDevices,
  OUT EFI_IDE_CABLE_TYPE              *CableType
  );
```

Parameters

This

Pointer to the **EFI_PLATFORM_IDE_INIT_PROTOCOL** instance.

Controller

The handle that corresponds to the IDE controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

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-6 specification, 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.

CableType

The type of cable that is detected on this channel. Type **EFI_IDE_CABLE_TYPE** is defined in "Related Definitions" below.

Description

This member function can be used to obtain information about a particular channel on a particular IDE controller. The IDE controller driver will pass this information to the IDE bus driver. The IDE bus driver uses this information during the enumeration process. The information that is returned by this call cannot change during the boot process because the consumer may call this function once for every controller and cache this information.

This member function is similar to **`EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo()`** except for the addition of another input parameter, *`Controller`*, and the output parameter *`CableType`*. The **`EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo()`** function is defined in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

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 collection of devices behind a port multiplier constitute a channel. See the "Design Discussion" section in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification* for more information on port multipliers and SATA configurations.

Related Definitions

```
//*******************************************************
// EFI_IDE_CABLE_TYPE
//*******************************************************
typedef enum {
  EfiIdeCableTypeUnknown,
  EfiIdeCableType40pin,
  EfiIdeCableType80Pin,
  EfiIdeCableTypeSerial,
  EfiIdeCableTypeMaximum
} EFI_IDE_CABLE_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The information was returned without any errors.
EFI_UNSUPPORTED	The platform does not have any information about this controller. The controller driver should use defaults.
EFI_INVALID_PARAMETER	*`Channel`* is invalid for this *`Controller`*.
EFI_INVALID_PARAMETER	*`Controller`* is not a valid handle.

EFI_PLATFORM_IDE_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_PLATFORM_IDE_NOTIFY_PHASE) (
  IN EFI_PLATFORM_IDE_INIT_PROTOCOL   *This,
  IN EFI_HANDLE                       Controller,
  IN EFI_IDE_CONTROLLER_ENUM_PHASE    Phase,
  IN UINT8                            Channel
  );
```

Parameters

This

Pointer to the **EFI_PLATFORM_IDE_INIT_PROTOCOL** instance.

Controller

The handle that corresponds to the IDE controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Phase

The phase during enumeration. Type **EFI_IDE_CONTROLLER_ENUM_PHASE** is defined in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()** in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

Channel

Zero-based channel number.

Description

This member function can be used to notify the platform IDE driver to perform specific actions, including any platform-specific initialization, so that the platform is ready to enter the next phase. Seven notification points are defined at this time. These notification points are the same as those defined in the IDE Controller Initialization Protocol. See "Related Definitions" in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()** in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification* for the definition of various notification points.

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

This member function is similar to **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()** except for the addition of another input parameter, *Controller*.

`EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()` calls **`EFI_PLATFORM_IDE_INIT_PROTOCOL.NotifyPhase()`** before performing any controller-specific actions.

Status Codes Returned

EFI_SUCCESS	The notification was accepted without any errors.
EFI_INVALID_PARAMETER	*`Phase`* is invalid.
EFI_INVALID_PARAMETER	*`Controller`* is not a handle.
EFI_INVALID_PARAMETER	*`Channel`* is invalid for this *`Controller`*.
EFI_NOT_READY	This phase cannot be entered at this time.

EFI_PLATFORM_IDE_INIT_PROTOCOL.SubmitData()

Summary

Submits the device information to the IDE platform driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PLATFORM_IDE_SUBMIT_DATA) (
  IN EFI_PLATFORM_IDE_INIT_PROTOCOL  *This,
  IN EFI_HANDLE                      Controller,
  IN UINT8                           Channel,
  IN UINT8                           Device,
  IN EFI_IDENTIFY_DATA               *IdentifyData
  );
```

Parameters

This

Pointer to the **EFI_PLATFORM_IDE_INIT_PROTOCOL** instance.

Controller

The handle that corresponds to the IDE controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

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 **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()** in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

Description

This member function is used by the IDE controller driver to pass detailed information about a particular device to the platform driver. The information is obtained by the IDE bus driver 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 caller, and the IDE platform 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 at a later point.

The platform driver may consult various fields of the **EFI_IDENTIFY_DATA** structure and make platform-specific decisions. For example, the IDE controller driver may examine the vendor and type/mode field to match known bad drives.

If the device does not exist, *IdentifyData* will be set to **NULL**.

This member function is similar to **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()** except for the addition of another input parameter, *Controller*. **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()** is defined in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

The IDE controller driver must call **EFI_PLATFORM_IDE_INIT_PROTOCOL.SubmitData()** when the bus driver calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()**.

Status Codes Returned

EFI_SUCCESS	The information was accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid for this *Controller*.
EFI_INVALID_PARAMETER	*Controller* is not a handle.
EFI_INVALID_PARAMETER	*Device* is invalid.

EFI_PLATFORM_IDE_INIT_PROTOCOL.OverrideModes()

Summary

Submits the device information to the IDE platform driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PLATFORM_IDE_OVERRIDE_MODES) (
  IN  EFI_PLATFORM_IDE_INIT_PROTOCOL      *This,
  IN  EFI_HANDLE                          Controller,
  IN  UINT8                               Channel,
  IN  UINT8                               Device,
  IN OUT EFI_ATA_COLLECTIVE_MODE_BITMAP   *SupportedModes
  );
```

Parameters

This

Pointer to the **EFI_PLATFORM_IDE_INIT_PROTOCOL** instance.

Controller

The handle that corresponds to the IDE controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

SupportedModes

The bit map representing the various modes that the IDE device supports. Type **EFI_ATA_COLLECTIVE_MODE_BITMAP** is defined in "Related Definitions" below.

Description

This member function is used by the platform to override the default mode selection algorithm in the IDE controller driver. The platform may use this member function to do the following:

- Force incompatible devices to operate at a lower mode.
- Implement user-configurable mode settings.

The platform may also use this function in case the platform design prevents the certain mode, even though the controller supports that mode.

When **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** is called, the IDE controller driver calls **EFI_PLATFORM_IDE_INIT_PROTOCOL.OverrideModes()** and thereby provides the platform the opportunity to override the mode settings for various transfer protocols.

On input, the parameter *SupportedModes* lists all the transfer protocols and the supported modes for each of the protocols. The IDE controller driver calculates the input settings of *SupportedModes*. If the platform does not wish to override the IDE controller selection, it will not update the contents of the *SupportedModes* buffer. If the platform wishes to override the IDE controller selection, it will update the contents of the *SupportedModes* buffer. The platform can remove mode selections from the *SupportedModes* buffer only by clearing bits. The platform cannot add any selections by setting any bits in the *SupportedModes* buffer. The platform driver must not change the ordering within *SupportedModes.ExtModeBitmap*. The IDE controller driver will choose the fastest mode settings within the output buffer *SupportedModes*.

This member function may be called multiple times for an IDE device because **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** can be called multiple times for an IDE device. **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** is defined in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

Related Definitions

```
//*******************************************************
// EFI_ATA_COLLECTIVE_MODE_BITMAP
//*******************************************************
typedef struct {
  EFI_ATA_MODE_BITMAP            PioModeBitmap;
  EFI_ATA_MODE_BITMAP            SingleWordDmaModeBitmap;
  EFI_ATA_MODE_BITMAP            MultiWordDmaModeBitmap;
  EFI_ATA_MODE_BITMAP            UdmaModeBitmap;
  UINT32                         ExtModeCount;
  EFI_ATA_EXTENDED_MODE_BITMAP   ExtModeBitmap[1];
} EFI_ATA_COLLECTIVE_MODE_BITMAP;
```

PioModeBitmap

Specifies the Programmed Input/Output (PIO) mode bit map. PIO modes are defined in the ATA/ATAPI specification. Type **EFI_ATA_MODE_BITMAP** is defined below.

SingleWordDmaModeBitmap

Specifies the single word DMA mode bit map.

MultiWordDmaModeBitmap

Specifies the multiword DMA mode bit map.

UdmaModeBitmap

Specifies the ultra DMA (UDMA) mode bit map.

ExtModeCount

The number of entries in the extended-mode bit map. Can be zero. This field provides extensibility.

ExtModeBitmap

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. Type **EFI_ATA_EXTENDED_MODE_BITMAP** is defined below.

```
//***********************************************
// EFI_ATA_MODE_BITMAP
//***********************************************
typedef  UINT64    EFI_ATA_MODE_BITMAP;
```

EFI_ATA_MODE_BITMAP is a bit map representing various supported modes. It can convey more than one mode per transfer protocol by setting more than one bit. Mode x is supported if bit x is set.

```
//***********************************************
// EFI_ATA_EXTENDED_MODE_BITMAP
//***********************************************
typedef struct {
  EFI_ATA_EXT_TRANSFER_PROTOCOL   TransferProtocol;
  EFI_ATA_MODE_BITMAP             ModeBitmap;
} EFI_ATA_EXTENDED_MODE_BITMAP;
```

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 in "Related Definitions" in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()** in the *Intel® Platform Innovation Framework for EFI IDE Controller Initialization Protocol Specification*.

ModeBitmap

A bit map representing various supported modes. Type **EFI_ATA_MODE_BITMAP** is defined above.

Status Codes Returned

EFI_SUCCESS	The information was accepted without any errors.
EFI_INVALID_PARAMETER	*`Channel`* is invalid.
EFI_INVALID_PARAMETER	*`Controller`* is not a handle.
EFI_INVALID_PARAMETER	*`Device`* is invalid.