EFI 1.1
SCSI Driver Model

Draft for Review

Version 0.3

May 9, 2002

Revision History

Revision	Revision History	Date
0.01	Initial Draft.	9/28/01
0.1	Update according to feedback.	10/8/01
0.2	Update the SCSI I/O Protocol according to the SCSI Pass Thru Protocol Document's update.	10/22/01
0.21	Add ATAPI device path example, Fibre Channel device path example, InfiniBand device path example.	10/29/01
0.22	Add GetDeviceLocation() API.	11/5/01
0.23	Modify the Fibre Channel device path example and InfiniBand device path example, make them easier to understand.	12/12/01
0.3	Add HostAdapter status and Target Status definition.	5/9/02

Contents

Figures

Tables

1 Introduction

1.1 Scope

This document describes the SCSI Driver Model. This includes the behavior of SCSI Bus Drivers, the behavior of a SCSI Device Drivers, and a detailed description of the SCSI I/O Protocol. This document provides enough material to implement a SCSI Bus Driver, and the tools required to design and implement a SCSI Device Drivers. It does not provide any information on specific SCSI devices.

The material contained in this document is designed to extend the *EFI Specification* and the *EFI Driver Model Specification* in a way that supports SCSI device drivers and SCSI bus drivers. These extensions are provided in the form of SCSI specific protocols. This document provides the information required to implement a SCSI Bus Driver in system firmware. The document also contains the information required by driver writers to design and implement SCSI Device Drivers that a platform may need to boot an EFI compliant OS.

A full understanding of the *EFI Specification*, the *EFI Driver Model Specification*, and the *SCSI Pass Thru Protocol Specification* is assumed throughout this document. The SCSI Driver Model described here is intended to be a foundation on which a SCSI Bus Driver and a wide variety of SCSI Device Drivers can be created.

1.2 Target Audience

This document is intended for the following readers:

- IHVs that design and manufacture SCSI adapters and peripherals that are used in Intel architecture-based platforms.
- OEMs who will be creating Intel architecture-based platforms intended to boot shrink-wrap operating systems.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in Intel architecture-based products.
- Operating system developers who will be adapting their shrink-wrap operating system products to run on Intel architecture-based platforms.

1.3 Related Information

The following publications and sources of information may be useful to you or are referred to by this specification:

- *EFI Specification Version 1.02*, Intel Corporation, 2000, http://developer.intel.com/technology/efi.
- *EFI 1.1 Specification Draft 0.95*, Intel Corporation, 2002, http://developer.intel.com/technology/efi.
- *Information technology – Small Computer System Interface –2 Revision 10L*
- *InfiniBand Architecture Specification Volume 1,Release 1.0.a*
- *InfiniBand Architecture Specification Volume 2,Release 1.0.a*
- *Fibre Channel Framing and Signaling (FC-FS) Rev1.30*

1.4 Terms

Hardware architectures used for the design of I/O systems in Intel Architecture computers can be described by a set of buses and a set of devices. In order to access these different buses and devices, bus drivers and device drivers are required. The following terms are used throughout this document to describe the model for construction of SCSI Bus Drivers and SCSI Device Drivers in the EFI environment:

SCSI Host Controller: A device that produces one or more physical SCSI bus(es).

SCSI Host Adapter: In this document, it has the same meaning as the *SCSI Host Controller*. Both indicate the physical component that produces SCSI bus(es).

SCSI Bus: A collection of *SCSI Devices* that share the same physical SCSI bus. All devices on a SCSI Bus share the same bandwidth of the SCSI Bus.

SCSI Channel: Another representation of *SCSI Bus*.

SCSI Device: A SCSI peripheral that is physically attached to the *SCSI Bus*.

Initiator: In the SCSI spec, it represents a SCSI device that requests an I/O process to be performed by another SCSI device (a target). In this document, only the *SCSI Host Controller* (or say, *SCSI Host Adapter*) can act as an initiator.

Target: A SCSI device that performs an operation requested by an initiator.

SCSI Bus Driver: Software that enumerates and creates a handle for every *SCSI Controller* on a *SCSI Bus* and installs both the *SCSI I/O Protocol* and the *Device Path Protocol* onto that handle.

SCSI Device Driver: Software that manages one or more *SCSI Controllers* of a specific type. A driver will use the *SCSI I/O Protocol* to produce a device I/O abstraction in the form of another protocol (i.e. Block I/O, Simple Network, Simple Input, Simple Text Output, Serial I/O, Load File).

SCSI I/O Protocol: A software interface that provides services to manage a *SCSI Controller*, and services to move data between a *SCSI Controller* and system memory.

SCSI Enumeration: The process of searching the presence of any *SCSI Device* on a given *SCSI Bus*.

1.5 Conventions Used in This Document

This document uses typographic and illustrative conventions described below.

1.5.1 Data Structure Descriptions

The Intel Architecture processors of the IA-32 family are "little endian" machines. This means that the low-order byte of a multi-byte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Itanium Processor Family (IPF) 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.

1.5.2 Protocol Descriptions

A protocol description generally has the following format:

Protocol:	The formal name of the protocol interface.
Summary:	A brief description of the protocol interface.
GUID:	The 128 bit unique identifier for the protocol interface.
Revision Number:	The revision of the protocol interface.
Protocol Interface Struct	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.
Related Definitions:	The type declarations and constants that are used in the protocol interface structure or any of its procedures.
Description:	A description of the functionality provided by the protocol interface including any limitations and caveats the caller should be aware of.

1.5.3 Procedure Descriptions

A procedure description generally has 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:	The parameters defined in the template are described in further detail.
Related Definitions:	The type declarations and constants that are only used by this procedure.
Description:	A description of the functionality provided by the interface including any limitations and caveats the caller should be aware of.
Status Codes Returned:	A description of the codes returned by the interface.

1.5.4 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 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 *EFI Specification*.

1.5.5 Typographic Conventions

The following typographic conventions are used in this document to illustrate programming concepts:

`Prototype`	This typeface is use to indicate prototype code.
`Argument`	This typeface is used to indicate arguments.
`Name`	This typeface is used to indicate actual code or a programming construct.
register	This typeface is used to indicate a processor register.

2
SCSI Driver Model

2.1 SCSI Driver Model Overview

The EFI SCSI Driver Stack includes the SCSI Pass Thru Driver, SCSI Bus Driver and individual SCSI Device Drivers.

SCSI Pass Thru Driver: A SCSI Pass Through Driver manages a SCSI Host Controller that contains one or more SCSI Buses. It creates SCSI Bus Controller Handles for each SCSI Bus, and attaches SCSI Pass Thru Protocol and Device Path Protocol to each handle the driver produced. Please refer to *EFI1.1 SCSI Pass Thru Protocol, Version0.8* for details about the protocol.

SCSI Bus Driver: A SCSI Bus Driver manages a SCSI Bus Controller Handle that is created by SCSI Pass Thru Driver. It creates SCSI Device Handles for each SCSI Device Controller detected during SCSI Bus Enumeration, and attaches SCSI I/O Protocol and Device Path Protocol to each handle the driver produced.

SCSI Device Driver: A SCSI Device Driver manages one kind of SCSI Devices. Device handles for SCSI Devices are created by SCSI Bus Drivers. A SCSI Device Driver could be a bus driver itself, and may create child handles. But most SCSI Device Drivers will be device drivers that do not create new handles. For the pure device driver, it attaches protocol instance to the device handle of the SCSI Device. These protocol instances are I/O abstractions that allow the SCSI Device to be used in the pre-boot environment. The most common I/O abstractions are used to boot an EFI compliant OS.

2.2 SCSI Bus Drivers

A SCSI Bus Driver manages a SCSI Bus Controller Handle. A SCSI Bus Controller Handle is created by a SCSI Pass Thru Driver and is abstracted in software with the SCSI Pass Thru Protocol. A SCSI Bus Driver will manage handles that contain this protocol. Figure 2-1 shows an example device handle for a SCSI Bus handle. It contains a Device Path Protocol instance and a SCSI Pass Thru Protocol Instance.

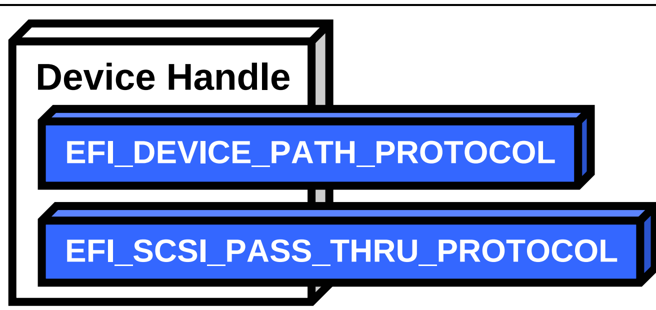

Figure 2-1 Device Handle for a SCSI Bus Controller

2.2.1 Driver Binding Protocol for SCSI Bus Drivers

The Driver Binding Protocol contains three services. These are **Supported()**, **Start()**, and **Stop()**. **Supported()** tests to see if the SCSI Bus Driver can manage a device handle. A SCSI Bus Driver can only manage device handle that contain the Device Path Protocol and the SCSI Pass Thru Protocol, so a SCSI Bus Driver must look for these two protocols on the device handle that is being tested.

The **Start()** function tells the SCSI Bus Driver to start managing a device handle. The device handle should support the protocols shown in Figure 2-1. The SCSI Pass Thru Protocol provides information about a SCSI Channel and the ability to communicate with any SCSI devices attached to that SCSI Channel.

The SCSI Bus Driver has the option of creating all of its children in one call to **Start()**, or spreading it across several calls to **Start()**. In general, if it is possible to design a bus driver to create one child at a time, it should do so to support the rapid boot capability in the EFI Driver Model. Each of the child device handles created in **Start()** must contain a Device Path Protocol instance, and a SCSI I/O protocol instance. The SCSI I/O Protocol is described in Section 2.4 and Section 2.5. The format of device paths for SCSI Devices is described in Section 2.6. Figure 2-2 shows an example child device handle that is created by a SCSI Bus Driver for a SCSI Device.

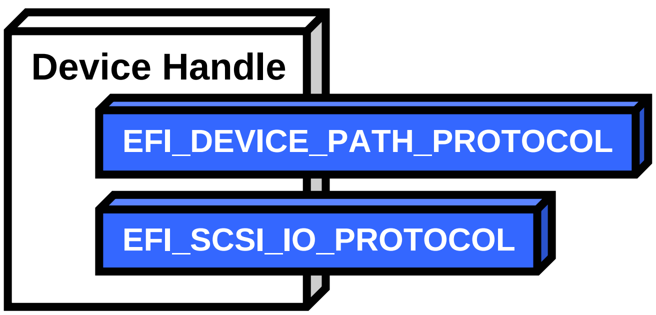

Figure 2-2 Child Handle Created by a SCSI Bus Driver

A SCSI Bus Driver must perform several steps to manage a SCSI Bus.

- Scan for the SCSI Devices on the SCSI Channel that connected to the SCSI Bus Controller. If a request is being made to scan only one SCSI Device, then only looks for the one specified. Create a device handle for the SCSI Device found.
- Install a Device Path Protocol instance and a SCSI I/O Protocol instance on the device handle created for each SCSI Device.

The **Stop()** function tells the SCSI Bus Driver to stop managing a SCSI Bus. The **Stop()** function can destroy one or more of the device handles that were created on a previous call to **Start()**. If all of the child device handles have been destroyed, then **Stop()** will place the SCSI Bus Controller in a quiescent state. The functionality of **Stop()** mirrors **Start()**.

2.2.2 SCSI Enumeration

The purpose of the SCSI Enumeration is only to scan for the SCSI Devices attached to the specific SCSI channel. The SCSI Bus driver need not allocate resources for SCSI Devices (like PCI Bus Drivers do), nor need it connect a SCSI Device with its Device Driver (like USB Bus Drivers do). The details of the SCSI Enumeration is implementation specific, thus is out of the scope of this document.

2.3 SCSI Device Drivers

SCSI Device Drivers manage SCSI Devices. Device handles for SCSI Devices are created by SCSI Bus Drivers. A SCSI Device Driver could be a bus driver itself, and may create child handles. But most SCSI Device Drivers will be device drivers that do not create new handles. For the pure device driver, it attaches protocol instance to the device handle of the SCSI Device. These protocol instances are I/O abstractions that allow the SCSI Device to be used in the pre-boot environment. The most common I/O abstractions are used to boot an EFI compliant OS.

2.3.1 Driver Binding Protocol for SCSI Device Drivers

The Driver Binding Protocol contains three services. These are **`Supported()`**, **`Start()`**, and **`Stop()`**. **`Supported()`** tests to see if the SCSI Device Driver can manage a device handle. A SCSI Device Driver can only manage device handle that contain the Device Path Protocol and the SCSI I//O Protocol, so a SCSI Device Driver must look for these two protocols on the device handle that is being tested. In addition, it needs to check to see if the device handle represents a SCSI Device that SCSI Device Driver knows how to manage. This is typically done by using the services of the SCSI I/O Protocol to see whether the device information retrieved is supported by the device driver.

The **`Start()`** function tells the SCSI Device Driver to start managing a SCSI Device. A SCSI Device Driver could be a bus driver itself, and may create child handles. But most SCSI Device Drivers will be device drivers that do not create new handles. For the pure device driver, it installs one or more addition protocol instances on the device handle for the SCSI Device.

The **`Stop()`** function mirrors the **`Start()`** function, so the **`Stop()`** function completes any outstanding transactions to the SCSI Device and removes the protocol interfaces that were installed in **`Start()`**.

2.4 EFI SCSI I/O Protocol Overview

This section defines the EFI SCSI I/O protocol. This protocol is used by code, typically drivers, running in the EFI boot services environment to access SCSI devices. In particular, functions for managing devices on SCSI buses are defined here.

The interfaces provided in the **`EFI_SCSI_IO_PROTOCOL`** are for performing basic operations to access SCSI devices.

2.5 EFI SCSI I/O Protocol

This section provides a detailed description of the **EFI_SCSI_IO_PROTOCOL**.

Summary

Provides services to manage and communicate with SCSI devices.

GUID

```
#define EFI_SCSI_IO_PROTOCOL_GUID  \

{0x403cd195,0xf233,0x48ec,0x84,0x55,0xb2,0xe5,0x2f,0x1d,0x9e,0x2}
```

Protocol Interface Structure

```
typedef struct _EFI_SCSI_IO_PROTOCOL {
  EFI_SCSI_IO_PROTOCOL_GET_DEVICE_TYPE        GetDeviceType;
  EFI_SCSI_IO_PROTOCOL_GET_DEVICE_LOCATION    GetDeviceLocation;
  EFI_SCSI_IO_PROTOCOL_RESET_BUS              ResetBus;
  EFI_SCSI_IO_PROTOCOL_RESET_DEVICE           ResetDevice;
  EFI_SCSI_IO_PROTOCOL_EXECUTE_SCSI_COMMAND   ExecuteScsiCommand;
} EFI_SCSI_IO_PROTOCOL;
```

Parameters

GetDeviceType	Retrieves the information of the device type which the SCSI device belongs to. See Section 2.5.1.
GetDeviceLocation	Retrieves the device location information in the SCSI bus. See Section 2.5.2.
ResetBus	Resets the entire SCSI bus the SCSI device attaches to. See Section 2.5.3.
ResetDevice	Resets the SCSI Device that is specified by the device handle the SCSI I/O protocol attaches. See Section 2.5.4.
ExecuteSCSICommand	Sends a SCSI command to the SCSI device and waits for the execution completion until an exit condition is met, or a timeout occurs. See Section 2.5.5.

Description

The **EFI_SCSI_IO_PROTOCOL** provides the basic functionalities to access and manage a SCSI Device. There is one **EFI_SCSI_IO_PROTOCOL** instance for each SCSI Device on a SCSI Bus. A device driver that wishes to manage a SCSI Device in a system will have to retrieve the **EFI_SCSI_IO_PROTOCOL** instance that is associated with the SCSI Device. A device handle for a SCSI Device will minimally contain an **EFI_DEVICE_PATH_PROTOCOL** instance and an **EFI_SCSI_IO_PROTOCOL** instance.

2.5.1 EFI_SCSI_IO_PROTOCOL.GetDeviceType()

Summary

Retrieves the device type information of the SCSI Device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_GET_DEVICE_TYPE) (
  IN  EFI_SCSI_IO_PROTOCOL      *This,
  OUT UINT8                     *DeviceType
  );
```

Parameters

This	A pointer to the **EFI_SCSI_IO_PROTOCOL** instance. Type **EFI_SCSI_IO_PROTOCOL** is defined in Section 2.5.
DeviceType	A pointer to the device type information retrieved from the SCSI Device. See "Related Definitions" for the possible returned values of this parameter.

Description

This function is used to retrieve the SCSI device type information. This function is typically used for SCSI Device Drivers to quickly recognize whether the SCSI Device could be managed by it.

If *DeviceType* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the device type is returned in *DeviceType* and **EFI_SUCCESS** is returned.

Related Definitions

```
// Peripheral Device Type Definitions  (SCSI-2)
//
#define EFI_SCSI_IO_TYPE_DISK            0x00 // Disk device
#define EFI_SCSI_IO_TYPE_TAPE            0x01 // Tape device
#define EFI_SCSI_IO_TYPE_PRINTER         0x02 // Printer
#define EFI_SCSI_IO_TYPE_PROCESSOR       0x03 // Processor
#define EFI_SCSI_IO_TYPE_WORM            0x04 // Write-once read-multiple
#define EFI_SCSI_IO_TYPE_CDROM           0x05 // CD-ROM device
#define EFI_SCSI_IO_TYPE_SCANNER         0x06 // Scanner device
#define EFI_SCSI_IO_TYPE_OPTICAL         0x07 // Optical memory device
#define EFI_SCSI_IO_TYPE_MEDIUMCHANGER   0x08 // Medium Changer device
#define EFI_SCSI_IO_TYPE_COMMUNICATION   0x09 // Communications device
#define EFI_SCSI_IO_TYPE_RESERVED_LOW    0x0A // Reserved (low)
#define EFI_SCSI_IO_TYPE_RESERVED_HIGH   0x1E // Reserved (high)
#define EFI_SCSI_IO_TYPE_UNKNOWN         0x1F // Unknown no device type
```

Status Codes Returned

EFI_SUCCESS	Retrieves the device type information successfully.
EFI_INVALID_PARAMETER	The *DeviceType* is **NULL**.

2.5.2 EFI_SCSI_IO_PROTOCOL. GetDeviceLocation()

Summary

Retrieves the SCSI device location in the SCSI channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_GET_DEVICE_LOCATION) (
  IN   EFI_SCSI_IO_PROTOCOL     *This,
  OUT  UINT32                   *Target,
  OUT  UINT64                   *Lun
  );
```

Parameters

This	A pointer to the **EFI_SCSI_IO_PROTOCOL** instance. Type **EFI_SCSI_IO_PROTOCOL** is defined in Section 2.5.
Target	A pointer to the Target ID of the SCSI device on the SCSI channel.
Lun	A pointer to the Logical Unit Number of the SCSI device on the SCSI channel.

Description

This function is used to retrieve the SCSI device location in the SCSI bus. The device location is definitely determined by (Target, Lun) pair. This function would allow a SCSI Device Driver to retrieve its location in the SCSI channel, and may use the SCSI Pass Thru Protocol to access the SCSI device directly.

If *Target* or *Lun* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the device location is returned in *Target* and *Lun*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	Retrieves the device location successfully.
EFI_INVALID_PARAMETER	*Target* or *Lun* is **NULL**.

2.5.3 EFI_SCSI_IO_PROTOCOL. ResetBus()

Summary

Resets the SCSI Bus that the SCSI Device is attached to.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_RESET_BUS) (
  IN  EFI_SCSI_IO_PROTOCOL      *This
  );
```

Parameters

This	A pointer to the **EFI_SCSI_IO_PROTOCOL** instance. Type **EFI_SCSI_IO_PROTOCOL** is defined in Section 2.5.

Description

This function provides the mechanism to reset the whole SCSI bus that the specified SCSI Device is connected to. Some SCSI Host Controller may not support bus reset, if so, **EFI_UNSUPPORTED** is returned. If a device error occurs while executing that bus reset operation, then **EFI_DEVICE_ERROR** is returned. If a timeout occurs during the execution of the bus reset operation, then **EFI_TIMEOUT** is returned. If the bus reset operation is completed, then **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SCSI bus is reset successfully.
EFI_DEVICE_ERROR	Errors encountered when resetting the SCSI bus.
EFI_UNSUPPORTED	The bus reset operation is not supported by the SCSI Host Controller.
EFI_TIMEOUT	A timeout occurred while attempting to reset the SCSI bus.

2.5.4 EFI_SCSI_IO_PROTOCOL.ResetDevice()

Summary

Resets the SCSI Device that is specified by the device handle that the SCSI I/O Protocol is attached.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_RESET_DEVICE) (
  IN  EFI_SCSI_IO_PROTOCOL     *This
  );
```

Parameters

This A pointer to the **EFI_SCSI_IO_PROTOCOL** instance. Type **EFI_SCSI_IO_PROTOCOL** is defined in Section 2.5.

Description

This function provides the mechanism to reset the SCSI Device. If the SCSI bus does not support a device reset operation, then **EFI_UNSUPPORTED** is returned. If a device error occurs while executing that device reset operation, then **EFI_DEVICE_ERROR** is returned. If a timeout occurs during the execution of the device reset operation, then **EFI_TIMEOUT** is returned. If the device reset operation is completed, then **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	Reset the SCSI Device successfully.
EFI_DEVICE_ERROR	Errors are encountered when resetting the SCSI Device.
EFI_UNSUPPORTED	The SCSI bus does not support a device reset operation.
EFI_TIMEOUT	A timeout occurred while attempting to reset the SCSI Device.

2.5.5 EFI_SCSI_IO_PROTOCOL. ExecuteScsiCommand()

Summary

Sends a SCSI Request Packet to the SCSI Device for execution.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_EXECUTE_SCSI_COMMAND) (
  IN       EFI_SCSI_IO_PROTOCOL                 *This,
  IN OUT   EFI_SCSI_IO_SCSI_REQUEST_PACKET      *Packet,
  IN       EFI_EVENT                            Event    OPTIONAL
  );
```

Parameters

This — A pointer to the **EFI_SCSI_IO_PROTOCOL** instance. Type **EFI_SCSI_IO_PROTOCOL** is defined in Section 2.5.

Packet — The SCSI request packet to send to the SCSI Device specified by the device handle. See "Related Definitions" for a description of **EFI_SCSI_IO_SCSI_REQUEST_PACKET**.

Event — If the SCSI bus where the SCSI device is attached does not support non-blocking I/O, then *Event* is ignored, and blocking I/O is performed. If *Event* is **NULL**, then blocking I/O is performed. If *Event* is not **NULL** and non-blocking I/O is supported, then non-blocking I/O is performed, and *Event* will be signaled when the SCSI Request Packet completes.

Related Definitions

```
typedef struct {
  UINT64       Timeout;
  VOID         *DataBuffer;
  VOID         *SenseData;
  VOID         *Cdb;
  UINT32       TransferLength;
  UINT8        CdbLength;
  UINT8        DataDirection;
  UINT8        HostAdapterStatus;
  UINT8        TargetStatus;
  UINT8        SenseDataLength;
}EFI_SCSI_IO_SCSI_REQUEST_PACKET;
```

Timeout	The timeout, in 100 ns units, to use for the execution of this SCSI Request Packet. A *Timeout* value of 0 means that this function will wait indefinitely for the SCSI Request Packet to execute. If *Timeout* is greater than zero, then this function will return **EFI_TIMEOUT** if the time required to execute the SCSI Request Packet is greater than *Timeout*.
DataBuffer	A pointer to the data buffer to transfer from or to the SCSI device.
SenseData	A pointer to the sense data that was generated by the execution of the SCSI Request Packet.
Cdb	A pointer to buffer that contains the Command Data Block to send to the SCSI device.
TransferLength	On input, the size, in bytes, of *DataBuffer*. On output, the number of bytes actually transferred. If *TransferLength* is larger than the SCSI Host Controller can handle, then the SCSI Host Controller will transfer its maximum amount, and will update *TransferLength* with the number of bytes actually transferred.
CdbLength	The length, in bytes, of the buffer *Cdb*. The standard values are 6, 10, 12, and 16, but other values are possible if a variable length CDB is used.
DataDirection	The direction of the data transfer. 0 for reads, 1 for writes. All other values are reserved, and must not be used.
HostAdapterStatus	The status of the SCSI Host Controller that produces the SCSI bus where the SCSI device attached when the SCSI Request Packet was executed on the SCSI Controller. See the possible values listed below.
TargetStatus	The status returned by the SCSI device when the SCSI Request Packet was executed. See the possible values listed below.
SenseDataLength	On input, the length in bytes of the *SenseData* buffer. On output, the number of bytes written to the *SenseData* buffer.

```
//
// HostAdapterStatus
//
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_OK
0x00
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_TIMEOUT_COMMAND
0x09
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_TIMEOUT
0x0b
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_MESSAGE_REJECT
0x0d
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_BUS_RESET
0x0e
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_PARITY_ERROR
0x0f
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_REQUEST_SENSE_FAILED
0x10
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_SELECTION_TIMEOUT
0x11
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_DATA_OVERRUN_UNDERRUN
0x12
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_BUS_FREE
0x13
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_PHASE_ERROR
0x14
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_OTHER
0x7f

//
// TargetStatus
//
#define EFI_SCSI_IO_STATUS_TARGET_GOOD                          0x00
#define EFI_SCSI_IO_STATUS_TARGET_CHECK_CONDITION               0x02
#define EFI_SCSI_IO_STATUS_TARGET_CONDITION_MET                 0x04
#define EFI_SCSI_IO_STATUS_TARGET_BUSY                          0x08
#define EFI_SCSI_IO_STATUS_TARGET_INTERMEDIATE                  0x10
#define EFI_SCSI_IO_STATUS_TARGET_INTERMEDIATE_CONDITION_MET 0x14
#define EFI_SCSI_IO_STATUS_TARGET_RESERVATION_CONFLICT          0x18
#define EFI_SCSI_IO_STATUS_TARGET_COMMAND_TERMINATED            0x22
#define EFI_SCSI_IO_STATUS_TARGET_QUEUE_FULL                    0x28
```

Description

This function sends the SCSI Request Packet specified by *Packet* to the SCSI Device.

If the SCSI Bus supports non-blocking I/O and *Event* is not **NULL**, then this function will return immediately after the command is sent to the SCSI Device, and will later signal *Event* when the command has completed. If the SCSI Bus supports non-blocking I/O and *Event* is **NULL**, then this function will send the command to the SCSI Device and block until it is complete. If the SCSI Bus does not support non-blocking I/O, the *Event* parameter is ignored, and the function will send the command to the SCSI Device and block until it is complete.

If *Packet* is successfully sent to the SCSI Device, then **EFI_SUCCESS** is returned.

If *Packet* cannot be sent because there are too many packets already queued up, then **EFI_NOT_READY** is returned. The caller may retry *Packet* at a later time.

If a device error occurs while sending the *Packet*, then **EFI_DEVICE_ERROR** is returned.

If a timeout occurs during the execution of *Packet*, then **EFI_TIMEOUT** is returned.

If any field of *Packet* is invalid, then **EFI_INVALID_PARAMETER** is returned.

If the data buffer described by *DataBuffer* and *TransferLength* is too big to be transferred in a single command, then **EFI_WARN_BUFFER_TOO_SMALL** is returned. The number of bytes actually transferred is returned in *TransferLength*.

If the command described in *Packet* is not supported by the SCSI Host Controller that produces the SCSI bus, then **EFI_UNSUPPORTED** is returned.

If **EFI_SUCCESS,EFI_WARN_BUFFER_TOO_SMALL,EFI_DEVICE_ERROR**, or **EFI_TIMEOUT** is returned, then the caller must examine the status fields in *Packet* in the following precedence order: *HostAdapterStatus* followed by *TargetStatus* followed by *SenseDataLength*, followed by *SenseData*. If non-blocking I/O is being used, then the status fields in *Packet* will not be valid until the *Event* associated with *Packet* is signaled.

If **EFI_NOT_READY**, **EFI_INVALID_PARAMETER** or **EFI_UNSUPPORTED** is returned, then *Packet* was never sent, so the status fields in *Packet* are not valid. If non-blocking I/O is being used, the *Event* associated with *Packet* will not be signaled.

Status Codes Returned

EFI_SUCCESS	The SCSI Request Packet was sent by the host successfully, and *TransferLength* bytes were transferred to/from *DataBuffer*. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.
EFI_WARN_BUFFER_TOO_SMALL	The SCSI Request Packet was executed, but the entire *DataBuffer* could not be transferred. The actual number of bytes transferred is returned in *TransferLength*. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.
EFI_NOT_READY	The SCSI Request Packet could not be sent because there are too many SCSI Command Packets already queued. The caller may retry again later.
EFI_DEVICE_ERROR	A device error occurred while attempting to send the SCSI Request Packet. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.
EFI_INVALID_PARAMETER	The contents of *CommandPacket* are invalid. The SCSI Request Packet was not sent, so no additional status information is available.
EFI_UNSUPPORTED	The command described by the SCSI Request Packet is not supported by the SCSI initiator (i.e., SCSI Host Controller). The SCSI Request Packet was not sent, so no additional status information is available.
EFI_TIMEOUT	A timeout occurred while waiting for the SCSI Request Packet to execute. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.

2.6 SCSI Device Paths

An **EFI_SCSI_IO_PROTOCOL** must be installed on a handle for its services to be available to SCSI device drivers. In addition to the **EFI_SCSI_IO_PROTOCOL**, an **EFI_DEVICE_PATH_PROTOCOL** must also be installed on the same handle. See Chapter 5 of the *EFI Specification* for detailed description of the **EFI_DEVICE_PATH_PROTOCOL**.

The SCSI Driver Model defined in this document can support the SCSI channel generated or emulated by multiple architectures, such as SCSI-I, SCSI-II, SCSI-III, ATAPI, Fibre Channel, InfiniBand*, and other future channel types. In this section, there are four example device paths provided, including SCSI device path, ATAPI device path, Fibre Channel device path and InfiniBand device path.

2.6.1 SCSI Device Path Example

Table 2-1 shows an example device path for a SCSI device controller on a desktop platform. This SCSI device controller is connected to a SCSI channel that is generated by a PCI SCSI host controller. The PCI SCSI host controller generates a single SCSI channel, it is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. The SCSI device controller is assigned SCSI Id 2, and its LUN is 0.

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, a SCSI Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

ACPI(PNP0A03,0)/PCI(7|0)/SCSI(2,0).

Table 2-1. SCSI Device Path Examples

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x07	PCI Function
0x11	0x01	0x00	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x13	0x01	0x02	Sub type – SCSI
0x14	0x02	0x08	Length – 0x08 bytes
0x16	0x02	0x0002	Target ID on the SCSI bus, PUN

Byte Offset	Byte Length	Data	Description
0x18	0x02	0x0000	Logical Unit Number, LUN
0x1A	0x01	0Xff	**Generic Device Path Header** – Type End of Hardware Device Path
0x1B	0x01	0xFF	Sub type – End of Entire Device Path
0x1C	0x02	0x04	Length – 0x04 bytes

2.6.2 ATAPI Device Path Example

Table 2-2 shows an example device path for an ATAPI device on a desktop platform. This ATAPI device is connected to the IDE bus on Primary channel, and is configured as the Master device on the channel. The IDE bus is generated by the IDE controller that is a PCI device. It is located at PCI device number 0x1F and PCI function 0x01, and is directly attached to a PCI root bridge.

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, an ATAPI Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|0)/ATAPI(Primary,Master).
```

Table 2-2. ATAPI Device Path Examples

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x07	PCI Function
0x11	0x01	0x00	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x13	0x01	0x01	Sub type – ATAPI
0x14	0x02	0x08	Length – 0x08 bytes
0x16	0x01	0x00	PrimarySecondary – Set to zero for primary or one for secondary.
0x17	0x01	0x00	SlaveMaster – set to zero for master or one for slave.
0x18	0x02	0x0000	Logical Unit Number,LUN.
0x1A	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x1B	0x01	0xFF	Sub type – End of Entire Device Path
0x1C	0x02	0x04	Length – 0x04 bytes

2.6.3 Fibre Channel Device Path Example

Table 2-3 shows an example device path for an SCSI device that is connected to a Fibre Channel Port on a desktop platform. The Fibre Channel Port is a PCI device that is located at PCI device number 0x08 and PCI function 0x00, and is directly attached to a PCI root bridge. The Fibre Channel Port is addressed by the World Wide Number, and is assigned as X (X is a 64bit value); the SCSI device's Logical Unit Number is 0.

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, a Fibre Channel Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(8|0)/Fibre(X,0).
```

Table 2-3. Fibre Channel Device Path Examples

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x08	PCI Function
0x11	0x01	0x00	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x13	0x01	0x02	Sub type – Fibre Channel
0x14	0x02	0x24	Length – 0x24 bytes
0x16	0x04	0x00	Reserved
0x1A	0x08	X	Fibre Channel World Wide Number
0x22	0x08	0x00	Fibre Channel Logical Unit Number.
0x2A	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x2B	0x01	0xFF	Sub type – End of Entire Device Path
0x2C	0x02	0x04	Length – 0x04 bytes

2.6.4 InfiniBand* Device Path Example

Table 2-4 shows an example device path for a SCSI device in an InfiniBand Network. This SCSI device is connected to a single SCSI channel generated by a SCS Host Adapter, and the SCSI Host Adapter is an end node in the InfiniBand Network. The SCSI Host Adapter is a PCI device that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. The SCSI device is addressed by the (IOU X, IOC Y, DeviceId Z) in the InfiniBand Network. (X, Y, Z are EUI-64 compliant identifiers).

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, an InfiniBand Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|0)/Infiniband(X,Y,Z).
```

Table 2-4. InfiniBand Device Path Examples

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x07	PCI Function
0x11	0x01	0x00	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x13	0x01	0x09	Sub type – InfiniBand
0x14	0x02	0x20	Length – 0x20 bytes
0x16	0x04	0x00	Reserved
0x1A	0x08	X	64bit node GUID of the IOU
0x22	0x08	Y	64bit GUID of the IOC
0x2A	0x08	Z	64bit persistent ID of the device.
0x32	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x33	0x01	0xFF	Sub type – End of Entire Device Path
0x34	0x02	0x04	Length – 0x04 bytes