Intel® Platform Innovation Framework for EFI SMBus Host Controller Protocol Specification

Draft for Review

Version 0.9

April 1, 2004

Intel order number xxxxxx-001

Revision History

Revision	Revision History	Date
0.9	First public release.	4/1/04

Contents

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the System Management Bus (SMBus) Host Controller Protocol of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This protocol is used by code, typically early chipset drivers, and SMBus bus drivers that are running in the EFI Boot Services environment to perform data transactions over the SMBus. This specification does the following:

- Describes the basic components of the SMBus Host Controller Protocol
- Provides code definitions for the SMBus Host Controller Protocol and the SMBus-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

Overview

This document describes the System Management Bus (SMBus) Host Controller Protocol. This protocol provides an I/O abstraction for an SMBus host controller. An SMBus host controller is a hardware component that interfaces to an SMBus. It moves data between system memory and devices on the SMBus by processing data structures and generating transactions on the SMBus. The following use this protocol:

- An SMBus bus driver to perform all data transactions over the SMBus
- Early chipset drivers that need to manage devices that are required early in the Driver Execution Environment (DXE) phase, before the Boot Device Selection (BDS) phase

This protocol should be used only by drivers that require direct access to the SMBus.

Considerable discussion has been done to understand the usage model of the EFI Driver Model in the SMBus. Although, the EFI Driver Model concepts can be applied to SMBus, only the SMBus Host Controller Protocol was created for now for the following reasons:

- The EFI Driver Model is designed primarily for boot devices. Boot devices are unlikely to be connected to the SMBus because of SMBus-intrinsic capability. They are slow and not enumerable.
- The current usage model of SMBus is to enable and configure devices early during the boot phase, before BDS.

If some of these assumptions become obsolete and require being revisited in the future, this specification is extensible to convert to the EFI Driver Model.

Related Information

The following publications and sources of information may be useful to you or are referred to by this specification. See Related Information from Intel in the master Framework help system for the URLs for EFI specifications and other documentation from Intel.

Industry Specifications

- *System Management Bus (SMBus) Specification,* version 2.0, SBS Implementers Forum, August 3, 2000:
  http://www.smbus.org*
- *PCI Local Bus Specification,* revision 2.2, PCI Special Interest Group: See Industry Specifications in the master Framework help system for the URL for specifications from the PCI SIG.

SMBus Host Controller Protocol Terms

The following terms are used throughout this document to describe the model for constructing SMBus Host Controller Protocol instances in the DXE environment. See the Glossary in the master Framework help system for explanations of Framework-specific terms.

PEC

Packet Error Code. It is similar to a checksum data of the data coming across the SMBus wire.

SMBus

System Management Bus.

SMBus host controller

Provides a mechanism for the processor to initiate communications with SMBus slave devices. This controller can be connected to a main I/O bus such as PCI.

SMBus master device

Any device that initiates SMBus transactions and drives the clock.

SMBus slave device

The target of an SMBus transaction, which is driven by some master.

UDID

Unique Device Identifier. A 128-bit value that a device uses during the Address Resolution Protocol (ARP) process to uniquely identify itself.

SMBus Host Controller Protocol Overview

The interfaces that are provided in the **EFI_SMBUS_HC_PROTOCOL** are used to manage data transactions on the SMBus. The **EFI_SMBUS_HC_PROTOCOL** is designed to support SMBus 1.0– and 2.0–compliant host controllers.

Each instance of the **EFI_SMBUS_HC_PROTOCOL** corresponds to an SMBus host controller in a platform. To provide support for early drivers that need to communicate on the SMBus, this protocol is available before the Boot Device Selection (BDS) phase. During BDS, this protocol can be attached to the device handle of an SMBus host controller that is created by a device driver for the SMBus host controller's parent bus type. For example, an SMBus controller that is implemented as a PCI device would require a PCI device driver to produce an instance of the **EFI_SMBUS_HC_PROTOCOL**.

See SMBus Host Controller Protocol in Code Definitions for the definition of this protocol.

3
Code Definitions

Introduction

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

- **EFI_SMBUS_HC_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_SMBUS_NOTIFY_FUNCTION**

SMBus Host Controller Protocol

EFI_SMBUS_HC_PROTOCOL

Summary

Provides basic SMBus host controller management and basic data transactions over the SMBus.

GUID

```
#define EFI_SMBUS_HC_PROTOCOL_GUID  \
{0xe49d33ed, 0x513d, 0x4634, 0xb6, 0x98, 0x6f, 0x55, 0xaa, 0x75,
0x1c, 0x1b}
```

Protocol Interface Structure

```
typedef struct _EFI_SMBUS_HC_PROTOCOL {
  EFI_SMBUS_HC_EXECUTE_OPERATION    Execute;
  EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE  ArpDevice;
  EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP GetArpMap;
  EFI_SMBUS_HC_PROTOCOL_NOTIFY      Notify;
} EFI_SMBUS_HC_PROTOCOL;
```

Parameters

Execute

Executes the SMBus operation to an SMBus slave device. See the **Execute()** function description.

ArpDevice

Allows an SMBus 2.0 device(s) to be Address Resolution Protocol (ARP). See the **ArpDevice()** function description.

GetArpMap

Allows a driver to retrieve the address that was allocated by the SMBus host controller during enumeration/ARP. See the **GetArpMap()** function description.

Notify

Allows a driver to register for a callback to the SMBus host controller driver when the bus issues a notification to the bus controller driver. See the **Notify()** function description.

Description

The **EFI_SMBUS_HC_PROTOCOL** provides SMBus host controller management and basic data transactions over SMBus. There is one **EFI_SMBUS_HC_PROTOCOL** instance for each SMBus host controller.

Early chipset drivers can communicate with specific SMBus slave devices by calling this protocol directly. Also, for drivers that are called during the Boot Device Selection (BDS) phase, the device driver that wishes to manage an SMBus bus in a system retrieves the **EFI_SMBUS_HC_PROTOCOL** instance that is associated with the SMBus bus to be managed. A device handle for an SMBus host controller will minimally contain an **EFI_DEVICE_PATH_PROTOCOL** instance and an **EFI_SMBUS_HC_PROTOCOL** instance.

EFI_SMBUS_HC_PROTOCOL.Execute()

Summary

Executes an SMBus operation to an SMBus controller. Returns when either the command has been executed or an error is encountered in doing the operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_EXECUTE_OPERATION) (
  IN      EFI_SMBUS_HC_PROTOCOL          *This,
  IN      EFI_SMBUS_DEVICE_ADDRESS       SlaveAddress,
  IN      EFI_SMBUS_DEVICE_COMMAND       Command,
  IN      EFI_SMBUS_OPERATION            Operation,
  IN      BOOLEAN                        PecCheck,
  IN OUT  UINTN                          *Length,
  IN OUT  VOID                           *Buffer
  );
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

The SMBus slave address of the device with which to communicate. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification*.

Command

This command is transmitted by the SMBus host controller to the SMBus slave device and the interpretation is SMBus slave device specific. It can mean the offset to a list of functions inside an SMBus slave device. Not all operations or slave devices support this command's registers. Type **EFI_SMBUS_DEVICE_COMMAND** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification.*

Operation

Signifies which particular SMBus hardware protocol instance that it will use to execute the SMBus transactions. This SMBus hardware protocol is defined by the *SMBus Specification* and is not related to EFI. Type **EFI_SMBUS_OPERATION** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification.*

PecCheck

Defines if Packet Error Code (PEC) checking is required for this operation.

Length

Signifies the number of bytes that this operation will do. The maximum number of bytes can be revision specific and operation specific. This field will contain the actual number of bytes that are executed for this operation. Not all operations require this argument.

Buffer

Contains the value of data to execute to the SMBus slave device. Not all operations require this argument. The length of this buffer is identified by *Length*.

Description

The **Execute()** function provides a standard way to execute an operation as defined in the *System Management Bus (SMBus) Specification*. The resulting transaction will be either that the SMBus slave devices accept this transaction or that this function returns with error.

Status Codes Returned

EFI_SUCCESS	The last data that was returned from the access matched the poll exit criteria.
EFI_CRC_ERROR	Checksum is not correct (PEC is incorrect)
EFI_TIMEOUT	*Timeout* expired before the operation was completed. *Timeout* is determined by the SMBus host controller device.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_DEVICE_ERROR	The request was not completed because a failure that was reflected in the Host Status Register bit. Device errors are a result of a transaction collision, illegal command field, unclaimed cycle (host initiated), or bus errors (collisions).
EFI_INVALID_PARAMETER	*Operation* is not defined in **EFI_SMBUS_OPERATION**.
EFI_INVALID_PARAMETER	*Length*/*Buffer* is **NULL** for operations except for **EfiSmbusQuickRead** and **EfiSmbusQuickWrite**. *Length* is outside the range of valid values.
EFI_UNSUPPORTED	The SMBus operation or PEC is not supported.
EFI_BUFFER_TOO_SMALL	*Buffer* is not sufficient for this operation.

EFI_SMBUS_HC_PROTOCOL.ArpDevice()

Summary

Sets the SMBus slave device addresses for the device with a given unique ID or enumerates the entire bus.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE) (
  IN      EFI_SMBUS_HC_PROTOCOL          *This,
  IN      BOOLEAN                        ArpAll,
  IN      EFI_SMBUS_UDID                 *SmbusUdid, OPTIONAL
  IN OUT  EFI_SMBUS_DEVICE_ADDRESS       *SlaveAddress OPTIONAL
);
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

ArpAll

A Boolean expression that indicates if the host drivers need to enumerate all the devices or enumerate only the device that is identified by *SmbusUdid*. If *ArpAll* is **TRUE**, *SmbusUdid* and *SlaveAddress* are optional. If *ArpAll* is **FALSE**, *ArpDevice* will enumerate *SmbusUdid* and the address will be at *SlaveAddress*.

SmbusUdid

The Unique Device Identifier (UDID) that is associated with this device. Type **EFI_SMBUS_UDID** is defined in **EFI_PEI_SMBUS_PPI.ArpDevice()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification*.

SlaveAddress

The SMBus slave address that is associated with an SMBus UDID. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification.*

Description

The **ArpDevice()** function provides a standard way for a device driver to enumerate the entire SMBus or specific devices on the bus.

Status Codes Returned

EFI_SUCCESS	The last data that was returned from the access matched the poll exit criteria.
EFI_CRC_ERROR	Checksum is not correct (PEC is incorrect)
EFI_TIMEOUT	*Timeout* expired before the operation was completed. *Timeout* is determined by the SMBus host controller device.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_DEVICE_ERROR	The request was not completed because a failure was reflected in the Host Status Register bit. Device Errors are a result of a transaction collision, illegal command field, unclaimed cycle (host initiated), or bus errors (collisions).

EFI_SMBUS_HC_PROTOCOL.GetArpMap()

Summary

Returns a pointer to the Address Resolution Protocol (ARP) map that contains the ID/address pair of the slave devices that were enumerated by the SMBus host controller driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP) (
  IN      EFI_SMBUS_HC_PROTOCOL    *This,
  IN OUT  UINTN                    *Length,
  IN OUT  EFI_SMBUS_DEVICE_MAP     **SmbusDeviceMap
);
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

Length

Size of the buffer that contains the SMBus device map.

SmbusDeviceMap

The pointer to the device map as enumerated by the SMBus controller driver. Type **EFI_SMBUS_DEVICE_MAP** is defined in **EFI_PEI_SMBUS_PPI.GetArpMap()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification*.

Description

The **GetArpMap()** function returns the mapping of all the SMBus devices that were enumerated by the SMBus host driver.

Status Codes Returned

EFI_SUCCESS	The SMBus returned the current device map.

EFI_SMBUS_HC_PROTOCOL.Notify()

Summary

Allows a device driver to register for a callback when the bus driver detects a state that it needs to propagate to other drivers that are registered for a callback.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_PROTOCOL_NOTIFY) (
  IN      EFI_SMBUS_HC_PROTOCOL          *This,
  IN      EFI_SMBUS_DEVICE_ADDRESS       SlaveAddress,
  IN      UINTN                          Data,
  IN      EFI_SMBUS_NOTIFY_FUNCTION      NotifyFunction
);
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

Address that the host controller detects as sending a message and calls all the registered function. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification*.

Data

Data that the host controller detects as sending a message and calls all the registered function.

NotifyFunction

The function to call when the bus driver detects the *SlaveAddress* and *Data* pair. Type **EFI_SMBUS_NOTIFY_FUNCTION** is defined in "Related Definitions" below.

Description

The **Notify()** function registers all the callback functions to allow the bus driver to call these functions when the *SlaveAddress*/*Data* pair happens.

Related Definitions

```
//*******************************************************
// EFI_SMBUS_NOTIFY_FUNCTION
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_NOTIFY_FUNCTION) (
  IN      EFI_SMBUS_DEVICE_ADDRESS          SlaveAddress,
  IN      UINTN                             Data
);
```

SlaveAddress

The SMBUS hardware address to which the SMBUS device is preassigned or allocated. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification*.

Data

Data of the SMBus host notify command that the caller wants to be called.

Status Codes Returned

EFI_SUCCESS	*NotifyFunction* was registered.