SMBus Control Method Interface Specification

Version 1.0
December 10, 1999

Questions and comments regarding this specification may be forwarded to:
smbus@sbs-forum.org

For additional information on SMBus Specification, visit the SBS-IF at:
http://www.sbs-forum.org/smbus/

Document Revision History

Revision	Date	Author	Reason for Changes
Rev. 1.0	10 Dec 1999	SBS-IF	Initial release.

Table of Contents

1 Introduction 1
  1.1 Target Audience 1
  1.2 Related Documents 1
  1.3 Data Format 1
  1.4 Terminology 1
2 Overview 3
  2.1 Goals 3
  2.2 Problem Statement 3
  2.3 Solution 3
    2.3.1 SMBus CMI Objects 4
    2.3.2 Namespace Hierarchy 5
  2.4 Hardware Alerting 5
    2.5 Packet Error Checking 5
3 SMBus CMI Objects 6
  3.1 Overview 6
  3.2 ASL Definition 6
    3.2.1 Requirements 6
    3.2.2 Name 7
    3.2.3 Device Identification 7
  3.3 Control Methods 7
    3.3.1 SMBus Information (_SBI) 8
    3.3.2 SMBus Data Read (_SBR) 8
    3.3.3 SMBus Data Write (_SBW) 9
    3.3.4 SMBus Data Transfer (_SBT) 9
    3.3.5 SMBus Alert Information (_SBA) 10
  3.4 Protocol Mappings 10
    3.4.1 _SBR Protocols 10
    3.4.2 _SBW Protocols 11
    3.4.3 _SBT Protocols 11
  3.5 Data Structures 12
    3.5.1 SMB_INFO 12
    3.5.2 SMB_DEVICE 13
Appendix A - Data Types 16
  A.1 Data Format 16
  A.2 Device Addressing 16
  A.3 Bus Protocols 16
  A.4 Status Codes 17
Appendix B - Sample ASL 19
  B.1 Mobile Example 19
    B.1.1 ASL Overview 19
    B.1.2 EC-SMBus Sample ASL 20
  B.2 Desktop Example 32
    B.2.1 ASL Overview 32
    B.2.2 PIIX4 SMBus Sample ASL 32

List of Tables

Table 1: _SBR Protocol Mapping 10
Table 2: _SBW Protocol Mapping 11
Table 3: _SBT Protocol Mapping 11
Table 4: Protocol Values 17
Table 5: Status Code Values 18

List of Figures

Figure 1: SMBus CMI Architecture 4
Figure 2: Example ACPI Namespace Hierarchy 5
Figure 3: Data Format 16
Figure 4: Slave Address Encoding 16
Figure 5: Protocol Encoding 17
Figure 6: Status Code Encoding 17
Figure 7: Mobile Example System 19
Figure 8: Desktop Example System 32

1 Introduction

This specification defines a System Management Bus (SMBus) interface for Advanced Configuration and Power Interface (ACPI). This interface, referred to as the *SMBus Control Method Interface (CMI)*, allows the capabilities of an SMBus segment to be easily utilized regardless of hardware origin or implementation.

This document's scope is limited to the definition and use of the SMBus CMI. The appendix provides sample ACPI Source Language (ASL) code illustrating how to implement this interface for several example systems.

1.1 Target Audience

This specification is intended for use by the following audience:

- OEMs and ISVs developing platform firmware.
- OEMs and IHVs developing SMBus devices.
- Others interested in accessing SMBus devices in the ACPI environment.

Basic understanding of ACPI and the SMBus is assumed.

1.2 Related Documents

[1] *Advanced Configuration and Power Interface Specification v1.0b*, ©1996, 1997, 1998 Intel Corporation, Microsoft Corporation, Toshiba Corporation. This specification and other ACPI documentation are available at: http://www.teleport.com/~acpi/

[2] *System Management Bus Specification*, Revision 1.1, SBS-Implementers Forum, ©December, 1998. This specification is available at: http://www.sbs-forum.org/smbus/index.html

The following Smart Battery Specifications are available at: http://www.sbs-forum.org/

[3] *Smart Battery Data Specification*, Revision 1.1, SBS-Implementers Forum, ©December, 1998
[4] *Smart Battery Charger Specification*, Revision 1.1, SBS-Implementers Forum, ©December, 1998
[5] *Smart Battery Selector Specification*, Revision 1.1, SBS-Implementers Forum, ©December, 1998
[6] *Smart Battery System Manager Specification*, Revision 1.1, SBS-Implementers Forum, ©December, 1998

1.3 Data Format

All numbers specified in this document are in decimal format unless otherwise indicated. A number preceded by '0x' indicates hexadecimal format, and a number followed by the letter 'b' indicates binary format. For example, the numbers 10, 0x0A, and 1010b are equivalent.

1.4 Terminology

Acronym	Description
ACPI	Advanced Configuration and Power Interface. See http://www.teleport.com/~acpi/.
AML	ACPI Machine Language. See http://www.teleport.com/~acpi/.
ASIC	Application-Specific Integrated Circuit.
ASL	ACPI Source Language. See http://www.teleport.com/~acpi/.
BIOS	Basic Input / Output System.
CM	Control Method.
CMI	Control Method Interface.

Acronym	Description
EC	Embedded Controller.
HC	SMBus Segment Host Controller.
ICH	Intel I/O Hub Controller.
OEM	Original Equipment Manufacturer.
OS	Operating System.
PIIX4	Intel Chipset with an SMBus Host Controller.
SCI	System Control Interrupt.
SMBus	System Management Bus. See http://www.sbs-forum.org/smbus/index.html.

2 Overview

The ACPI specification defines an SMBus interface based solely on an embedded controller (EC) implementation. The limited nature of this definition prohibits its use for SMBus host controllers that are not EC-based. This specification defines a new ACPI-based SMBus interface (the *SMBus CMI*) that facilitates access to all types of SMBus host controller hardware.

The SMBus definition presented in this specification varies greatly from its ACPI-native counterpart. As implied in the name, SMBus CMI objects utilize ACPI control methods to provide a similar (but more advanced) feature set. For example, the interface defined in this specification supports packet error checking protocols, exposes the hardware capabilities of an SMBus controller, and facilitates the enumeration of SMBus devices on a given segment – all of which are unavailable in the native ACPI model.

It is important to note that a complete SMBus solution requires driver support from the target operating system. This driver would facilitate the use of SMBus CMI objects by the OS, system software, and user applications. In conjunction to this specification, the Smart Battery System Implementers Forum (SBS-IF) is developing the required driver support for ACPI-compatible Windows™ operating systems. For more information see the SBS-IF web at: http://www.sbs-forum.org/smbus/.

2.1 Goals

The primary goals of this specification are to:

- Provide an efficient, robust, and well-integrated ACPI interface that supports both EC- and non-EC-based SMBus host controller hardware.
- Facilitate system-wide synchronization for access to SMBus resources.
- Ensure that this new interface is compliant with versions 1.0 and 1.1 of the SMBus specification, and can be easily extended to support features targeted for future versions.

2.2 Problem Statement

The lack of native support for non-EC-based SMBus host controllers in ACPI is the primary problem being addressed by this specification. The absence of a well-defined ACPI interface prohibits the use of a standard OS software stack, which in turn deters upper-level software from utilizing the services available from these SMBus segments.

Not having a well-defined interface also makes it difficult to synchronize access to SMBus segments from their many consumers (firmware, ACPI, OS, system software, applications, etc.). For example, legacy manageability software that directly accesses an SMBus-based thermal sensor may be unaware that ACPI uses the same bus/device in its management of a thermal zone. This contention could result in (best case) data corruption or (worst case) a complete system lockup.

2.3 Solution

The solution presented in this specification addresses the above problems by providing a flexible interface that is capable of abstracting all types of SMBus host controller hardware. Unlike the SMBus interface defined in the ACPI specification (which relies on a special SMBus operation region), this definition uses a set of control methods whose implementation is determined by the OEM.

The SMBus CMI includes a feature set equivalent to ACPI's SMBus interface by supporting the capabilities defined in version 1.0 of the SMBus specification. This includes support for all device communication protocols (e.g. *read byte*), SMBus alerts, and other basic elements. The SMBus CMI also provides features not available in ACPI's EC-SMBus interface, exposing:

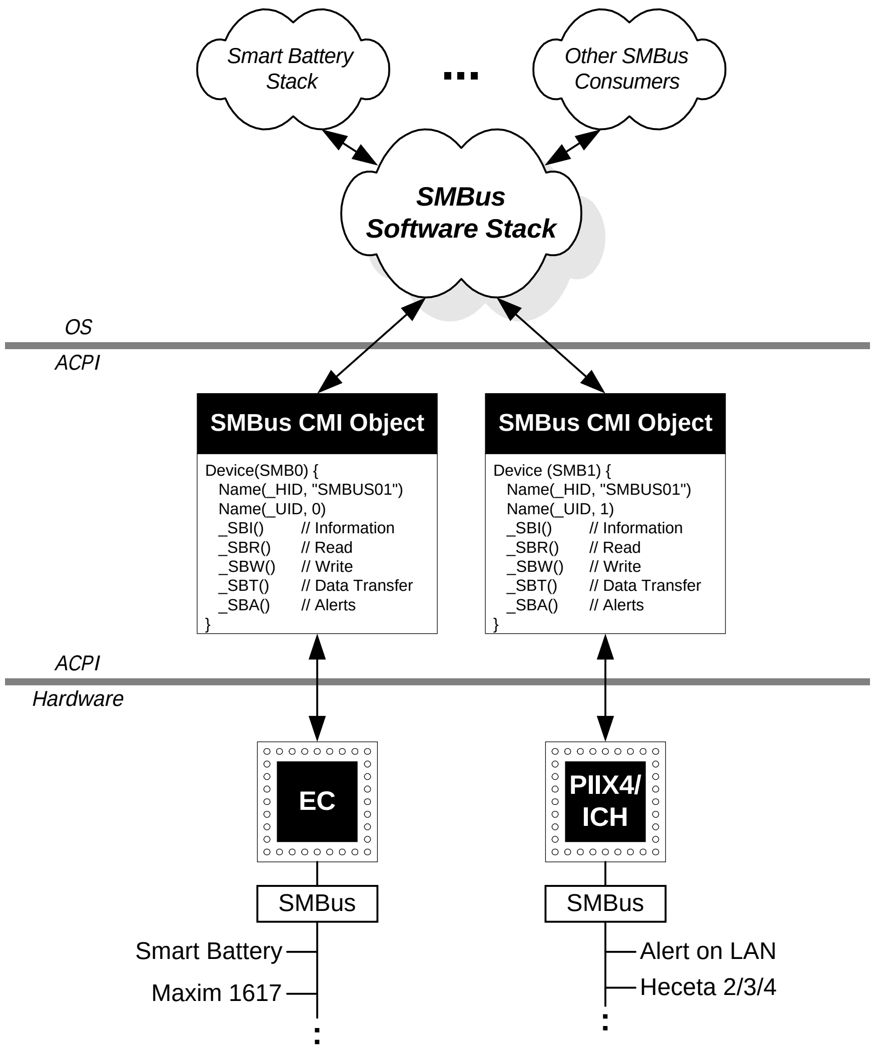
- Packet error checking.

- Hardware capabilities of an SMBus host controller.
- Information on each fixed-address device residing on a segment.

2.3.1 SMBus CMI Objects

OEMs implement the interface defined in this specification by defining *SMBus CMI Objects* in ASL. Each SMBus CMI object represents a single SMBus segment and is treated as a unique device in the ACPI namespace. Each SMBus CMI object implements the set of control methods required by this specification, but does so in the manner best suited to the particular hardware. Figure 1 illustrates the SMBus environment resulting from this specification for an example system.

Figure 1: SMBus CMI Architecture



```
Device(SMB0) {
   Name(_HID, "SMBUS01")
   Name(_UID, 0)
   _SBI()       // Information
   _SBR()       // Read
   _SBW()       // Write
   _SBT()       // Data Transfer
   _SBA()       // Alerts
}
```

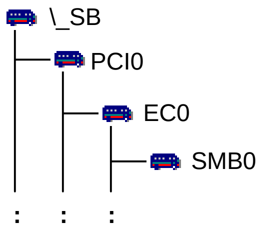
```
Device (SMB1) {
   Name(_HID, "SMBUS01")
   Name(_UID, 1)
   _SBI()       // Information
   _SBR()       // Read
   _SBW()       // Write
   _SBT()       // Data Transfer
   _SBA()       // Alerts
}
```

As implied by this diagram, the interface can support any number of SMBus segments. Each segment's definition is unique, giving OEMs full flexibility in the design of their platforms. For example, a notebook system may use an EC-based SMBus for the Smart Battery System (SBS) to ensure security, but opt to install the Alert on LAN* and other manageability ASICs off of the chipset's built-in SMBus controller. Additionally, EC-based segments may be modeled using the ACPI-native (operation region) interface, while other segments would be modeled as SMBus CMI objects.

2.3.2 Namespace Hierarchy

As with other devices, SMBus CMI objects are added to the ACPI namespace in a manner that represents the functional hierarchy of the system. For example, the first SMBus host controller shown in Figure 1 happens to be connected via a PCI-based EC. The corresponding SMBus CMI object (`‘SMB0’`) would therefore be located in the ACPI namespace as a child of EC and PCI devices, as shown in Figure 2.

Figure 2: Example ACPI Namespace Hierarchy



2.4 Hardware Alerting

The ability of SMBus host controller hardware to asynchronously notify consumers of a pending SMBus alert is referred to by this specification as *hardware alerting*. This highly desirable capability relieves upper-level software from implementing poll-based policies and generally results in a much more responsive and accurate environment.

The ACPI-defined mechanism for generating asynchronous notifications is through a system control interrupt (SCI). An SCI occurs whenever one or more non-masked bits are set in the ACPI event status register. Each bit in the event status register has associated AML that serve as the SCI interrupt handler. It is the job of this AML to determine the root cause of the SCI and perform the required action. See the ACPI specification for more information.

The AML code that handles an SCI for an SMBus alert simply issues a `Notify` command targeting the associated SMBus CMI object. For example, a smart battery may generate an SMBus alert when its remaining charge drops below some predefined level. The EC-SMBus host controller would receive this alert, set the corresponding GPE bit, and trigger an SCI. ACPI gets the SCI, realizes that the GPE bit has been set, and calls the AML ‘interrupt handler’ control method associated with this bit. Assuming the EC-SMBus is modeled in the namespace as `‘\_SB.EC0.SMB0’`, the AML code would simply issue the command:

```
   Notify(\_SB.EC0.SMB0, 0x80)
```

2.5 Packet Error Checking

Section 7.4 of the *System Management Bus Specification (Version 1.1)* describes how packet error code (PEC) is implemented for SMBus devices to provide improved communication reliability and robustness. Part of this implementation requires the transmission of an frame check sequence (FCV), which is calculated using an 8-bit cyclic redundancy check (CRC-8).

This capability is optional for the SMBus segments (and devices residing on these segments) compliant with version 1.1 of the SMBus specification. This specification allows segments and devices to advertise their support for packet error checking through a ‘hardware capability’ bitmap (see sections 3.5.1.1 and 3.5.2.2).

This specification defines an interface that enables software to discover if a particular SMBus segment (controller) and SMBus device support PEC. Software can then specify if the SMBus controller should use PEC protocols but the actual CRC checking is handled by hardware.

3 SMBus CMI Objects

This chapter provides details on how to abstract SMBus segments through the use of SMBus Control Method Interface object definitions.

3.1 Overview

Below is an overview of the process used to develop SMBus CMI objects in ASL. Note that the device definition and implementation of the `_SBI` (SMBus Information) control method are required for all SMBus CMI objects. The implementation of other control methods depends on the capabilities of the host controller and devices existing on the segment.

- Define a unique device (e.g. `'SMB0'`) for each SMBus segment. As stated previously, these devices should be located in the ACPI namespace in a manner that represents the functional hierarchy of the system. Each device must specify `'SMBUS01'` as its `_HID` and use a unique `_UID` value.
- Implement `_SBI` (SMBus Information) control method. The data returned by this control method identifies the characteristics of the segment, including the hardware capabilities of the host controller and a listing of all fixed-address devices connected to the segment.
- Implement the bus protocols required for communication to the SMBus devices that exist on the given segment. Read protocols are implemented using the `_SBR` control method, write protocols are implemented using the `_SBW` control method, and the *process call* protocol is implemented using the `_SBT` control method.
- Implement the `_SBA` (SMBus Alerting) control method if the segment supports SMBus alerts.

3.2 ASL Definition

The ACPI definition for SMBus CMI objects is provided below. Details on control methods are provided in section 3.2.2. Details on SMBus properties are provided in section 3.4.

```
Device(SMB<id>){
   Name(_HID, "SMBUS01")   // Hardware ID (PnP ID)
   Name(_UID, <uid>)       // Unique Identification
   Method(_SBI, 0) {…}     // SMBus Information
   Method(_SBR, 3) {…}     // SMBus Data Read
   Method(_SBW, 6) {…}     // SMBus Data Write
   Method(_SBT, 6) {…}     // SMBus Data Transfer
   Method(_SBA, 0) {…}     // SMBus Alert Information
}
```

Note: Required elements are marked in **bold** in the ASL definition above.

3.2.1 Requirements

Each SMBus CMI object must comply with the device definition specified in sections 3.2.2 and 3.2.3, and must fully implement the `_SBI` control method. The implementation of other control methods depends on the capabilities of the host controller and devices existing on the segment. Required elements are marked in **bold** in the ASL definition (above).

It should be noted that although `_SBR`, `_SBW`, and `_SBT` are not required, it is impossible for software to communicate with an SMBus device without the implementation of at least one of these control methods. Within each of these control methods, support is needed only for the protocols required to communicate with the devices on the segment. For example, assume that a single device exists on an SMBus segment and it only communicates using the *read byte* protocol. For this segment the `_SBR` control method would be implemented to support this protocol, but implementations of `_SBW` or `_SBT` would not be needed. Additionally, `_SBR` could return the *unsupported protocol* status code for all read protocols other than *read byte*.

Implementation of `_SBA` is only necessary if the segment is capable of generating SMBus alerts. This includes both segments that must be polled for alert detection and those that can generate ACPI-visible asynchronous notifications.

3.2.2 Name

SMBus CMI object names, as with any object in ACPI, must follow the ASL naming convention defined in section 15.1.2 of the ACPI Specification. Below is the *recommended* format for specifying the name of an SMBus CMI object.

`Device(SMB<id>)`

`<id>` A single character identifier that creates a device name that is unique throughout all ACPI device objects. This specification recommends using a zero-based numeric value. For example, `'SMB0'` would be used as the device name for the first SMBus segment.

3.2.3 Device Identification

SMBus CMI objects utilize the _HID and _UID device identification objects to facilitate Plug and Play enumeration, providing the capability for the 'automatic' enumeration by the OS.

`Name(_HID, "SMBUS01")`

Used to specify the Plug and Play hardware ID for SMBus CMI objects.

`Name(_UID, <uid>)`

`<uid>` A 32-bit value used to specify a unique identifier for this SMBus CMI object. This specification recommends using the same value as used for the `<id>` field of the device name. For example, a _UID value of 0 (zero) would be used for the device `'SMB0'`.

3.3 Control Methods

By convention ACPI control methods can accept multiple input arguments as separate objects, but can return only a single object. The single object returned may be a packet. In this specification the SMBus control methods always return output arguments as a package of one or more elements. For example, the `_SBI` control method output argument is a package containing the CMI version and a `SMB_INFO` structure. The `_SBW` control method output argument is a package with the 1st element containing the status code.

When a control method returns an error code it is placed in the status code element of the output package and all other elements of the output package must be '0' (zero).

3.3.1 SMBus Information (_SBI)

Description:	Returns a `SMB_INFO` structure describing the general properties of an SMBus segment.	
Input Argument(s)	<none>	
Output Arguments(s):	SMBus CMI Version (Integer)	The version of the SMBus Control Method Interface Specification that the SMBus CMI objects are compliant with. The major version is specified in the high nibble, the minor version in the low nibble. For example, the value 0x10 identifies the interface defined in version 1.0 of this specification. This byte value is encoded into an ASL integer as described in section A.1.
	SMBus Information (Buffer)	A `SMB_INFO` structure as a byte array. See 3.5.1.
Notes:	This control method returns information on the general characteristics of the SMBus segment. This includes the hardware capabilities of the host controller and basic information on each fixed-address device existing on the segment.	

3.3.2 SMBus Data Read (_SBR)

Description:	Reads byte, word, or block data from a device residing on an SMBus segment.	
Input Argument(s):	Protocol Value (Integer)	The bus protocol to use during an SMBus request. This byte value is encoded into an ASL integer as described in section A.3.
	Slave Address (Integer)	The slave address of the target device. This byte value is encoded into an ASL integer as described in section A.2.
	Command Code (Integer)	The device-specific command code. This byte value is encoded into an ASL integer as described in section A.1.
Output Argument(s):	Status Code (Integer)	The return status code. This byte value is encoded into an ASL integer as described in section A.4.
	Data Length (Integer)	The length (in bytes) of the data read from the device. This byte value is encoded into an ASL integer as described in section A.1.
	Data (Integer | Buffer)	The data read from the device. This byte, word, or block data is encoded into an ASL integer (byte/word) or buffer (block) as described in section A.1 – where the return data type depends on the protocol used.
Notes:	This control method is used to implement the *read quick*, *receive byte*, *read byte*, *read word*, and *read block* protocols. See section 3.4 for additional details on this control method's implementation.	

3.3.3 SMBus Data Write (_SBW)

Description:	Writes byte, word, or block data to a device residing on an SMBus segment.	
Input Argument(s):	Protocol Value (Integer)	The bus protocol to use during an SMBus request. This byte value is encoded into an ASL integer as described in section A.3.
	Slave Address (Integer)	The slave address of the target device. This byte value is encoded into an ASL integer as described in section A.2.
	Command Code (Integer)	The device-specific command code. This byte value is encoded into an ASL integer as described in section A.1.
	Data Length (Integer)	The length (in bytes) of the data to be written. This byte value is encoded into an ASL integer as described in section A.1.
	Data (Integer | Buffer)	The data to be written. This byte, word, or block data is encoded into an ASL integer (byte/word) or buffer (block) as described in section A.1 – where the data type depends on the protocol being used.
Output Argument(s):	Status Code (Integer)	The return status code. This byte value is encoded into an ASL integer as described in section A.4.
Notes:	This control method is used to implement the *write quick*, *send byte*, *write byte*, *write word*, and *write block* protocols. See section 3.4 for additional details on this control method's implementation.	

3.3.4 SMBus Data Transfer (_SBT)

Description:	Performs data transfers to/from a device residing on an SMBus segment.	
Input Argument(s):	Protocol Value (Integer)	The bus protocol to use during an SMBus request. This byte value is encoded into an ASL integer as described in section A.3.
	Slave Address (Integer)	The slave address of the target device. This byte value is encoded into an ASL integer as described in section A.2.
	Data Length (Integer)	The length (in bytes) of the data to be written. This byte value is encoded into an ASL integer as described in section A.1.
	Data (Integer | Buffer)	The data to be written. This word or block data is encoded into an ASL integer (word) or buffer (block) as described in section A.1 – where the data type depends on the protocol used.
Output Argument(s):	Status Code (Integer)	The return status code. This byte value is encoded into an ASL integer as described in section A.4.
	Data Length (Integer)	The length (in bytes) of the data read from the device. This byte value is encoded into an ASL integer as described in section A.1.
	Data (Integer | Buffer)	The data read from the device. This word or block data is encoded into an ASL integer (word) or buffer (block) as described in section A.1 – where the data type depends on the protocol used.
Notes:	This control method is used to implement the *process call* protocol. See section 3.4 for additional details on this control method's implementation.	

3.3.5 SMBus Alert Information (_SBA)

Description:	Returns information on an SMBus alert.	
Input Argument(s):	<none>	
Output Argument(s):	Status Code (Integer)	The return status code. This byte value is encoded into an ASL integer as described in section A.4. Valid status codes for this control method are: 0x00 – Indicates and outstanding alert was active and the alert information was retrieved successfully. 0x01 – Indicates that there were no outstanding alerts. 0x07 – Indicates a general failure. Note that all other output argument fields should be set to 0 (zero) whenever a status code of 0x01 or 0x07 is returned.
	Slave Address (Integer)	The slave address that generated this alert. This byte value is encoded into an ASL integer as described in section A.2.
	Data Length (Integer)	The data length is a byte value encoded in an ASL integer as described in section A.1. This field must be 0 (zero) for alerts that do not return data (e.g. devices using the SMBALERT# line).
	Data (Integer)	The data field consists of word data encoded in an ASL integer as described in section A.1. This field must be 0 (zero) for alerts that do not return data (e.g. devices using the SMBALERT# line).
Notes:	This control method is used to surface SMBus alerts. It may be polled (called repeatedly at some predefined interval) by upper-level software for segments that don't support ACPI-visible asynchronous alert notifications.	

3.4 Protocol Mappings

This section describes the mapping between the SMBus command protocols and the input/output arguments values for the read (`_SBR`), write (`_SBW`), and data transfer (`_SBT`) control methods.

3.4.1 _SBR Protocols

Table 1 shows the mapping between the SMBus 'read' protocols and the `_SBR` control method. Read protocols include *read quick*, *receive byte*, and *read byte/word/block*.

Table 1: _SBR Protocol Mapping

Protocol	***Input Arguments***			***Return Arguments***		
	Protocol Value	Slave Address	Command Code	Status Code	Data Length	Data
Read Quick	0x03	Byte	0	Byte	0	0
Receive Byte (with PEC)	0x05 0x85		0		1	Byte
Read Byte (with PEC)	0x07 0x87		Byte		1	Byte
Read Word (with PEC)	0x09 0x89				2	Word
Read Block (with PEC)	0x0B 0x8B				0-32	Buffer

3.4.2 _SBW Protocols

Table 2 shows the mapping between the SMBus 'write' protocols and the `_SBW` control method. Write protocols include *write quick*, *send byte*, and *write byte/word/block*.

Table 2: _SBW Protocol Mapping

Protocol	***Input Arguments***					***Return Arguments***
	Protocol Value	Slave Address	Command	Data Length	Data	Status Code
Write Quick	0x02	Byte	0	0	0	Byte
Send Byte (with PEC)	0x04 0x84	Byte	Byte	0	0	Byte
Write Byte (with PEC)	0x06 0x86	Byte	Byte	1	Byte	Byte
Write Word (with PEC)	0x08 0x88	Byte	Byte	2	Word	Byte
Write Block (with PEC)	0x0A 0x8A	Byte	Byte	1-32	Buffer	Byte

3.4.3 _SBT Protocols

Table 3 shows the mapping between the SMBus 'data transfer' protocols and the `_SBT` control method. The only data transfer protocol currently defined is *process call*.

Table 3: _SBT Protocol Mapping

Protocol	***Input Arguments***				***Return Arguments***		
	Protocol Value	Slave Address	Data Length	Data	Status Code	Data Length	Data
Process Call (with PEC)	0x0C 0x8C	Byte	2	Word	Byte	2	Word

3.5 Data Structures

3.5.1 SMB_INFO

This data structure specifies the general characteristics of an SMBus CMI object.

Offset	Name	Length	Value	Description
0x00	SMB_INFO Structure Version	BYTE	0x10	This field specifies the version of the SMB_INFO structure. The major version is specified in the high nibble, the minor version in the low nibble. For example, the value 0x10 identifies the interface defined in version 1.0 of this specification.
0x01	SMBus Specification Version	BYTE	Varies	This field specifies the version of the SMBus Specification that the SMBus hardware (represented by this object) is compliant with. The major version is specified in the high nibble, the minor version in the low nibble. For example, the value 0x10 indicates that the SMBus host controller is compliant with SMBus version 1.0.
0x02	Segment Hardware Capability	BYTE	Bit Field	This field specifies the basic hardware capabilities of this SMBus segment. See 3.5.1.1.
0x03	Alert Polling Interval	BYTE	Varies	This field specifies the polling interval in seconds for segments that need to be polled in order for higher level software to detect SMBus alerts. A zero (0) indicates that the segment does not require polling because it either does not have devices that produce alerts or the host controller is capable of generating ACPI-visible asynchronous notifications (e.g. SCI).
0x04	Device Count	BYTE	Varies	The number `(n)` of `SMB_DEVICE` elements existing in the property array.
0x05 + (n-1) * 18	Device Array	Varies	Varies	An array of 18-byte `SMB_DEVICE` elements describing the fixed-address devices connected to this SMBus segment. See 3.5.2.

The following is a ‘C’-style definition of the `SMB_INFO` structure.

```
#define ANYSIZE_ARRAY 1

struct SMB_INFO
{
   BYTE SMBInfoStructureVersion;
   BYTE SMBusSpecificationVersion;
   BYTE HardwareCapability;
   BYTE AlertPollingInterval;
   BYTE DeviceCount;
   SMB_DEVICE DeviceArray[ANYSIZE_ARRAY];
};
```

See section 3.5.2 for details concerning the `SMB_DEVICE` structure. The use of the `ANYSIZE_ARRAY` is simply for ‘C’ syntactical correctness.

3.5.1.1 Segment Hardware Capability

This 8-bit value specifies the hardware capabilities of this SMBus segment. Possible values for this bit field are defined below. A set bit (1) indicates that the segment supports the associated hardware capability, while a cleared bit (0) indicates that the capability is not supported.

Bits	Name	Description
Bit 0	Segment supports Packet Error Checking?	This bit indicates whether this SMBus segment supports packet error code (PEC) as defined in the SMBus v1.1 specification.
Bit 1	Segment may contain ARP devices?	This bit indicates whether this SMBus segment may contain devices which need Address Resolution Protocol (ARP) to be run in order to assign slave addresses.
Bits 2:7	<Reserved>	Cleared (0).

3.5.2 SMB_DEVICE

The `SMB_DEVICE` structure is used to specify details of each fixed-address device attached to a SMBus segment.

Offset	Name	Length	Value	Description
0x00	Slave Address	BYTE	Varies	The 7-bit SMBus slave address of the device. Note that the address is specified using bits 0:6 of this byte field (non-shifted). See section A.2.
0x01	<Reserved>	BYTE	0x00	Cleared (0).
0x02	Device UDID	BYTE	Bit Field	This 16-byte (128-bit) value specifies the unique device ID for this SMBus device. See 3.5.2.1.

The following is a 'C'-style definition of the `SMB_DEVICE` structure.

```
struct SMB_DEVICE
{
   BYTE SlaveAddress;
   BYTE Reserved;
   SMB_UDID DeviceUDID;
}
```

3.5.2.1 SMB_UDID

Offset	Name	Length	Value	Description
0x00	Device Hardware Capabilities	BYTE	Varies	This field specifies the hardware capabilities of this SMBus device. See 3.5.2.2.
0x01	Version/ Revision	BYTE	Varies	This field specifies the UDID version and silicon revision ID for this SMBus device. See 3.5.2.3.
0x02	Vendor ID	WORD	Varies	This field specifies the device manufacturer's vendor ID as assigned by the SBS-IF.
0x04	Device ID	WORD	Varies	This field specifies the device ID assigned by the device manufacturer.
0x06	Interface	WORD	Varies	This field specifies the SMBus version for this device. See 3.5.2.4.
0x08	Subsystem Vendor ID	WORD	Varies	This field specifies the subsystem interface as assigned by the SBS-IF. This field, in combination with the Subsystem Device ID can be used to identify a company, organization or industry group that has defined a common device interface specification. If no subsystem interface is defined this field must be zero (0) and the Subsystem Device ID must also be zero (0).
0x0A	Subsystem Device ID	WORD	Varies	This field specifies a particular interface, implementation, or device as defined by the subsystem vendor or industry group. If the Subsystem Vendor ID field is zero (0) this field must also be zero (0).
0x0C	<Reserved>	BYTE[4]	0x00 0x00 0x00 0x00	Reserved. Must be zero (0x00000000).

The following is a ‘C’-style definition of the `SMB_UDID` structure.

```
struct SMB_UDID
{
   BYTE   DeviceCapability;
   BYTE   UDIDversionSiRevision;
   WORD   VendorID;
   WORD   DeviceID;
   WORD   Interface;
   WORD   SubsystemVendorID;
   WORD   SubsystemID;
   BYTE   Reserved[4];
}
```

The byte encoding of WORD values in an ASL buffer is geared towards ASL readability. Consider the following ASL which describes an example SMB_UDID structure:

```
Buffer ()
{
   0x00,                              // Device capabilities
   0x00,                              // UDID version/Silicon revision ID
   0x80, 0x86,                        // Vendor ID 0x8086
   0x00, 0x01,                        // Device ID 0x0001
   0x00, 0x00,                        // SMBus interface
   0x00, 0x00,                        // Subsystem Vendor ID 0x0000
   0x00, 0x00,                        // Subsystem ID 0x0000
   0x00, 0x00, 0x00, 0x00             // reserved
}
```

Note the byte ordering of the WORD values. This ordering is expected by the device driver when interfacing to the SMBus control methods. Also note that all structures defined in this specification assume single-byte alignment [e.g. `#pragma pack(1)`].

3.5.2.2 Device Hardware Capability

This 8-bit value specifies the hardware capabilities of this SMBus device. Possible values for this bit field are defined below. A set bit (1) indicates that the device supports the associated hardware capability, while a cleared bit (0) indicates that the capability is not supported.

Bits	Name	Description
Bit 0	Device supports Packet Error Checking?	This bit indicates whether this device supports packet error code (PEC) on all commands supported by the device. If this bit is cleared (0) then the ability of the device to support PEC is unknown.
Bits 1:7	<Reserved>	Cleared (0)

3.5.2.3 Version/Revision

This 8-bit value specifies the version, revision and hardware capabilities of this SMBus device. Possible values for this bit field are defined below.

Bits	Name	Description
Bits 0:2	Silicon Revision ID	These bits designate the silicon revision level for this SMBus device.
Bits 3:5	SMBus UDID Version	These bits designate the SMBus UDID version for this device. For this version of the UDID these bits must be cleared (0).
Bit 6:7	<Reserved>	Cleared (0).

3.5.2.4 Interface

This 16-bit value specifies the SMBus version for this device.

Bits	Name	Description
Bits 0:3	SMBus Version	These bits define the SMBus version for this device. Possible values are 0000b for SMBus version 1.0 and 0001b for SMBus version 1.1. All other values are reserved.
Bits 4:15	<Reserved>	Cleared (0).

3.5.2.5 SMB_UDID Considerations

For most devices the values for the `SMB_UDID` fields are straightforward. However for certain devices the proper values are not obvious. This section describes how these should be handled.

The Subsystem Vendor ID for the SBS-IF industry group is 0x5342. Smart battery system devices that conform to the *Smart Battery Data Specification v1.0 or v1.1* as published by the Smart Battery System Implementers Forum must have the ‘Subsystem Vendor ID’ field set to 0x5342 and the ‘Subsystem Device ID’ set to 0x5309 for smart battery chargers, 0x530A for smart battery selectors and 0x530B for smart battery devices.

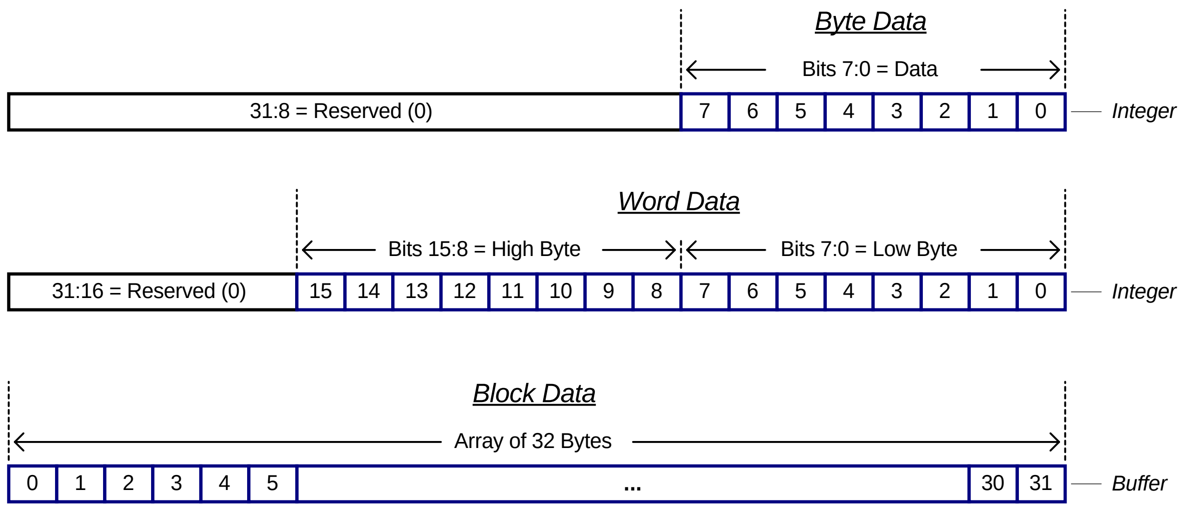
Cases are known to exist where a fixed slave address may have one of several SMBus devices attached or none at all at any point in time.   For example some mobile platforms have common docking station models containing additional SMBus devices (i.e. several docks, each with different combinations of devices) so the addresses must be reserved.   If the device is not present or it’s characteristics not known then the entire 16 bytes of the UDID structure must be set to all zeros (16 bytes of 0x00).

Appendix A - Data Types

A.1 Data Format

SMBus protocols require data to be transferred in byte, word, or block format – where ASL supports integer, string, and buffer data types. Figure 3 illustrates how data is mapped between these types. Note that byte and word SMBus data is mapped to the integer ASL type, and block SMBus data is mapped to the buffer ASL type.

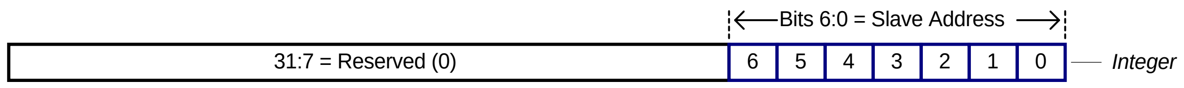
Figure 3: Data Format



A.2 Device Addressing

Slave addresses are presented in this specification using an integer-encoded, 7-bit, non-shifted notation, as illustrated in Figure 4. For example, the slave address of the Smart Battery Selector device would be specified as 0x0A (1010b), not 0x14 (10100b) as might be found in other documents.

Figure 4: Slave Address Encoding



A.3 Bus Protocols

SMBus protocols are presented in this specification using an integer-encoded, 8-bit notation, as illustrated in Figure 5. Note that bit 7 of the protocol value is used to indicate whether packet error checking should be employed. A value of 1 (one) in this bit indicates that PEC format should be used for the specified protocol, and a value of 0 (zero) indicates the standard (non-PEC) format should be used.

Figure 5: Protocol Encoding

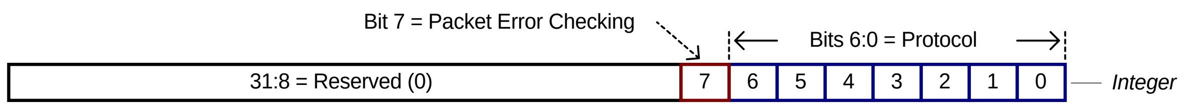


Table 4 lists the bus protocols defined in the SMBus Specification and their associated value. Note that the values listed match those defined for the EC SMBus in the ACPI specification.

Table 4: Protocol Values

Value	Description
0x00, 0x01	<Reserved>
0x02	Write Quick
0x03	Read Quick
0x04	Send Byte
0x05	Receive Byte
0x06	Write Byte
0x07	Read Byte
0x08	Write Word
0x09	Read Word
0x0A	Write Block
0x0B	Read Block
0x0C	Process Call
0x0D – 0xFF	<Reserved>

For example, the protocol value of 0x09 would be used to communicate to a device that supported the standard *read word* protocol. If this device supports packet error checking for this protocol, a value of 0x89 could also be used.

A.4 Status Codes

Status codes are presented in this specification using an integer-encoded, 8-bit notation, as illustrated in Figure 6.

Figure 6: Status Code Encoding

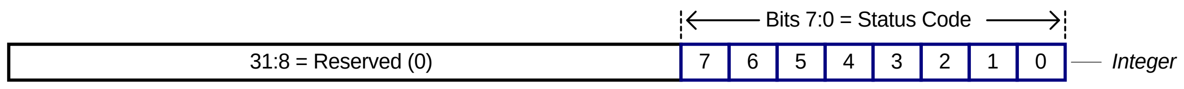


Table 5 lists the success/failure codes defined by this specification. The values listed match those defined for the EC SMBus in the ACPI Specification, with an additional status code to indicate a PEC error (0x1F).

Table 5: Status Code Values

Value	Description
0x00	OK (Success)
0x07	Unknown Failure
0x10	Address Not Acknowledged
0x11	Device Error
0x12	Command Access Denied
0x13	Unknown Error
0x17	Device Access Denied
0x18	Timeout
0x19	Unsupported Protocol
0x1A	Bus Busy
0x1F	PEC (CRC-8) Error
All other values	<Reserved>

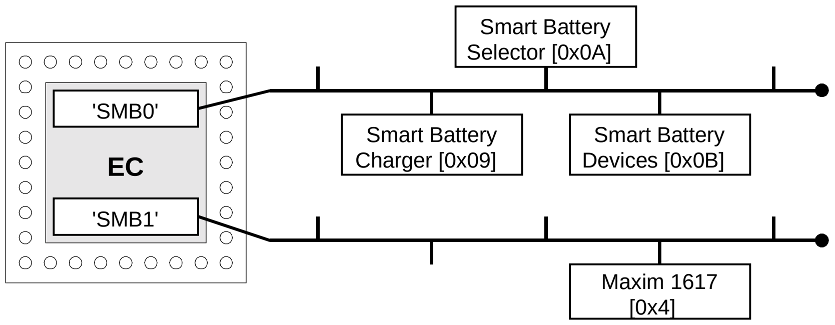
For example, an SMBus control method would return the status value 0x19 (*unsupported protocol*) for requests made for protocols not supported by the SMBus segment or slave device.

Appendix B - Sample ASL

B.1 Mobile Example

The sample ASL presented in this section was tested on an Intel Mobile Reference Platform. This platform includes an ACPI-compliant embedded controller (EC) with support for two SMBus segments: one at offset 0x04 in EC-space and one at offset 0x30. Devices connected to the first segment include the Smart Battery Subsystem (SBS) (charger, selector, and battery devices). Devices connected the second segment include a Maxim 1617 (local/remote thermal sensor). Figure 7 illustrates a logical view of this SMBus configuration.

Figure 7: Mobile Example System


Smart Battery Selector [0x0A]
'SMB0'
EC
'SMB1'
Smart Battery Charger [0x09]
Smart Battery Devices [0x0B]
Maxim 1617 [0x4]

Information on the embedded controller is available in chapter 13 of the ACPI specification:

http://www.teleport.com/~acpi/

Information on the Smart Battery Subsystem is available at:
http://www.sbs-forum.org/

Information on the Maxim 1617 is available at:
http://209.1.238.250/arpdf/1855.pdf

B.1.1 ASL Overview

The mobile sample ASL presented in section B.1.1 includes two SMBus CMI objects: `SMB0` represents the first SMBus segment and `SMB1` represents the second. Each of these ACPI devices is assigned a `_HID` value of `'SMBUS01'` and unique `_UID` values to allow enumeration of these SMBus CMI objects by the OS and higher-level software.

A mutex is declared for each device (`SBX0`, `SBX1`) to ensure *transactional synchronization*. This guarantees that a request that is currently being executed will complete before another (overlapping) request is started. Note that synchronization is only required within a segment.

The EC exposes a set of registers for each SMBus host controller. These registers are accessed via ACPI's `OperationRegion` primitive. `SMB0` is located at an offset of 0x04 in EC-space, while `SMB1` is located at offset 0x30.

The `_SBI` control method is required and thus was implemented for each segment. Although not required, all of the 'read' and 'write' protocols were implemented in the `_SBR` and `_SBW` control methods due to the simplicity of the EC-SMBus hardware interface. The `_SBA` control method was implemented for both segments, as the devices on each are capable of generating SMBus alerts.

Note that testing was performed on only the read/write byte, read/write word, and read block protocols – corresponding to the capabilities of the devices on these SMBus segments.

Since this platform includes an ACPI-compliant EC, the sample ASL presented in section B.1.1 can be used with minimal modifications on other EC-based platforms. Items that would need to be changed are the offset of the SMBus registers in EC-space and the information returned in the _SBI control method (e.g. device list).

Note that the `_GLK` (global lock) method, as defined in section 6.5.6 of the *ACPI Specification* (1.0b) should not be defined for devices nested in SMBus CMI devices (such as smart battery). Since the sample ASL's `Field` definitions' lock rule is set to "`Lock`", the global lock will automatically be acquired during the ASL's field accesses, and therefore `_GLK` should not be used.

B.1.2 EC-SMBus Sample ASL

```
////////////////////////////////////////////////////////////////////////////
// Device:      SMB0
// ----------------------------------------------------------------------
// Description: The first EC-SMBus segment (includes the SBS).
////////////////////////////////////////////////////////////////////////////
Device(SMB0)
{
    Name(_HID, "SMBUS01")
    Name(_UID, 0)

    Mutex(SBX0, 0)          // SMBus transactional synchronization.

    //
    // Smart Battery Subsystem
    // -----------------------
    Device(SBS0)
    {
        Name(_HID,"ACPI0002")      // HID for Smart Battery
        Name(_SBS,0x2)             // 2 batteries in system
    }


    //
    // OperationRegion:
    // ----------------
    // This SMBus resides at offset 0x04 in EC space.  See the ACPI
    // Specification for information on the EC-SMBus register interface.
    //
    OperationRegion(SMB0,EmbeddedControl,0x04,0x40)
    Field(SMB0,ByteAcc,Lock,Preserve)    // _GLK not used in SBS because 'Lock' is here
    {
        PRTC,   8,     // Protocol
        STS,    8,     // Status
        ADDR,   8,     // Address
        CMD,    8,     // Command
        DATA,   256,   // SMBus Data Bytes (Block)
        BCNT,   8,     // Block Count
        AADR,   8,     // Alarm Address
        ADB0,   8,     // Alarm Data Byte 0
        ADB1,   8      // Alarm Data Byte 1
    }
    Field(SMB0,ByteAcc,Lock,Preserve)
    {
        Offset(4),     // Move to byte offset 4 (beginning of data).
        DAT0, 8,       // 8-bit data register for byte/word access.
        DAT1, 8        // 8-bit data register for word access.
    }

    //
    // _SBI (SMBus Information):
```

```
// -------------------------
// Returns a SMB_INFO structure describing the properties of this
// SMBus segment.
//
// Parameters:
//  <none>
//
// Return Value (Package):
//  (0)     = SMBus Control Method Interface (CMI) version (Integer)
//  (1)     = SMB_INFO data structure (Buffer)
//
Method(_SBI)
{
    //
    // Local0 is the return package.  The CMI version is set
    // to v1.0 (0x10).
    //
    Store(Package(2){0x10,0x00}, Local0)

    Store(Buffer()
    {
        0x10,                   // SMB_INFO structure Version (v1.0)
        0x10,                   // SMBus Specification Version (v1.0)
        0x00,                   // Segment Hardware Capability
        0x00,                   // Alert Polling Interval (Supports Async Notifications)
        0x03,                   // Device Count
        0x09, 0x00,             // SMBus Device #1: SBS Charger
        0x00, 0x00, 0x80, 0x86, //   SMB_UDID... (Vendor ID is a placeholder)
        0x00, 0x01, 0x00, 0x00,
        0x53, 0x42, 0x53, 0x09, //   Conforms to SBS-IF Smart Battery Charger spec
        0x00, 0x00, 0x00, 0x00,
        0x0A, 0x00,             // SMBus Device #2: SBS Selector
        0x00, 0x00, 0x80, 0x86, //   SMB_UDID... (Vendor ID is a placeholder)
        0x00, 0x02, 0x00, 0x00,
        0x53, 0x42, 0x53, 0x0A, //   Conforms to SBS-IF Smart Battery Selector spec
        0x00, 0x00, 0x00, 0x00,
        0x0B, 0x00,             // SMBus Device #3: SBS Battery Devices
        0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Vendor ID is a placeholder)
        0x00, 0x03, 0x00, 0x00,
        0x53, 0x42, 0x53, 0x0B, //   Conforms to SBS-IF Smart Battery Data spec
        0x00, 0x00, 0x00, 0x00
    }, Index(Local0, 1))
    Return(Local0)
} // _SBI

//
// SWTC (Wait for Transaction Complete):
// --------------------------------------------
// Wait until the previous SMBus transaction has completed.
//
// Parameters:
//  Arg0    = Timeout Value (in ms)
//
// Return Value:
//  0x00    = OK
//  0x07    = Unknown Failure
//  0x10    = Address Not Acknowledged
//  0x11    = Device Error
//  0x12    = Command Access Denied
//  0x13    = Unknown Error
//  0x17    = Device Access Denied
//  0x18    = Timeout
//  0x19    = Unsupported Protocol
//  0x1A    = Bus Busy
//  0x1F    = PEC (CRC-8) Error
//
Method(SWTC, 1)
{
    Store(Arg0, Local0)
    Store(0x07, Local2)

    //
    // The previous command has completed when the protocol
    // register is equal to 0 (zero).  Wait <timeout> ms
    // (in 10ms chunks) for this to occur.
    //
```

```
        Store(1, Local1)
        While(LEqual(Local1, 1))
        {
            If(LEqual(PRTC, 0))
            {
                And(STS, 0x1F, Local2)      // Store status code.
                Store(0x00, Local1)         // Terminate loop.
            }
            Else
            {
                //
                // Transaction isn't complete.  Check for timeout, and if not,
                // sleep 10ms and loop again.
                //
                If(LLess(Local0, 10))
                {
                    Store(0x18, Local2)     // ERROR: Timeout occurred.
                    Store(0x00, Local1)     // Terminate loop.
                }
                Else
                {
                    Sleep(10)
                    Subtract(Local0, 10, Local0)
                }
            }
        }

        Return(Local2)
    } // Method(SWTC)

    //
    // _SBR (SMBus Read):
    // ------------------
    //
    // Parameters:
    //  Arg0    = Protocol (Integer)
    //  Arg1    = Slave Address (Integer)
    //  Arg2    = Command (Integer)
    //
    // Return Value (Package):
    //  (0)     = Status (Integer)
    //  (1)     = Data Length (Integer)
    //  (2)     = Data (Integer | Buffer)
    //
    Method(_SBR, 3)
    {
        //
        // Local0 is the return package.  The status code is defaulted
        // to 'unknown failure' (0x07).
        //
        Store(Package(3){0x07,0x00,0x00}, Local0)

        //
        // Make sure the protocol is valid, if not return the
        // 'invalid protocol' status code.  Note that this segment
        // does not support packet error checking.
        //
        If(LNotEqual(Arg0, 0x03))                   // Read Quick
        {
            If(LNotEqual(Arg0, 0x05))               // Receive Byte
            {
                If(LNotEqual(Arg0, 0x07))           // Read Byte
                {
                    If(LNotEqual(Arg0, 0x09))       // Read Word
                    {
                        If(LNotEqual(Arg0, 0x0B))   // Read Block
                        {
                            Store(0x19, Index(Local0, 0))
                            Return(Local0)
                        }
                    }
                }
            }
        }

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX0, 0xFFFF), 0))
        {
            //
            // Make sure the SMBus is ready for this transaction.  If not,
            // return a 'bus busy' error code.  Note that 'we' should be
            // the only consumer...
            //
            If(LNotEqual(PRTC, 0))
            {
                Store(0x1A, Index(Local0, 0))   // ERROR: Bus is busy.
            }
            Else
            {
                //
                // Initiate the transaction by writing the slave address,
                // command, and protocol registers.  Note that the command
                // code is always written, even if the protocol (e.g. 'read
                // quick') doesn't require it (it will be ignored).  Note also
                // that the “Read Block" always return 32-byte buffer regardless
                // of the actual block length. This is not a requirement but
                // is implementation specific to this sample ASL.
                //
                Store(ShiftLeft(Arg1, 1), ADDR)
                Store(Arg2, CMD)
                Store(Arg0, PRTC)

                //
                // Wait for completion.  Save the status code, data size,
                // and data into the return package (if required by the protocol).
                //
                Store(SWTC(1000), Index(Local0, 0))

                If(LEqual(Arg0, 0x05))          // Receive Byte
                {
                    Store(1, Index(Local0, 1))
                    Store(DAT0, Index(Local0, 2))
                }
                If(LEqual(Arg0, 0x07))          // Read Byte
                {
                    Store(1, Index(Local0, 1))
                    Store(DAT0, Index(Local0, 2))
                }
                If(LEqual(Arg0, 0x09))          // Read Word
                {
                    Store(2, Index(Local0, 1))
                    Store(DAT1, Local1)
                    ShiftLeft(Local1, 8, Local1)
                    Add(Local1, DAT0, Local1)
                    Store(Local1, Index(Local0, 2))
                }
                If(LEqual(Arg0, 0x0B))          // Read Block
                {
                    Store(BCNT, Index(Local0, 1))
                    Store(DATA, Index(Local0, 2))
                }
            }

            Release(SBX0)
        }

        Return(Local0)
    } // _SBR()

    //
    // _SBW (SMBus Write):
    // ------------------
    //
    // Parameters:
    //  Arg0    = Protocol Value (Integer)
    //  Arg1    = Slave Address (Integer)
    //  Arg2    = Command Code (Integer)
    //  Arg3    = Data Length (Integer)
    //  Arg4    = Data (Integer | Buffer)
    //
```

```
// Return Value (Package):
//  (0)     = Status (Integer)
//
Method(_SBW, 5)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(1){0x07}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking or the 'process call'
    // protocol.
    //
    If(LNotEqual(Arg0, 0x02))                   // Write Quick
    {
        If(LNotEqual(Arg0, 0x04))               // Send Byte
        {
            If(LNotEqual(Arg0, 0x06))           // Write Byte
            {
                If(LNotEqual(Arg0, 0x08))       // Write Word
                {
                    If(LNotEqual(Arg0, 0x0A))   // Write Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional synchronization.
    //
    If(LEqual(Acquire(SBX0, 0xFFFF), 0))
    {
        //
        // Make sure the SMBus is ready for this transaction.  If not,
        // return a 'bus busy' error code.  Note that 'we' should be
        // the only consumer...
        //
        If(LNotEqual(PRTC, 0))
        {
            Store(0x1A, Local0)
        }
        Else
        {
            //
            // Initiate the transaction by writing the slave address,
            // command, and protocol registers. Note that the command
            // code and data length are always written, even if the
            // protocol (e.g. 'write quick') doesn't require it (it
            // will be ignored).
            //
            Store(ShiftLeft(Arg1, 1), ADDR)
            Store(Arg2, CMD)
            Store(Arg3, BCNT)

            If(LEqual(Arg0, 0x06))          // Write Byte
            {
                Store(Arg4, DAT0)
            }
            If(LEqual(Arg0, 0x08))          // Write Word
            {
                And(Arg4, 0x00FF, DAT0)
                ShiftRight(Arg4, 8, DAT1)
            }
            If(LEqual(Arg0, 0x0A))          // Write Block
            {
                Store(Arg4, DATA)
            }
```

```
                Store(Arg0, PRTC)

                //
                // Wait for completion.
                //
                Store(SWTC(1000), Local0)
            }
            Release(SBX0)
        }

        Return(Local0)
    } // _SBW()

    //
    // _SBA (SMBus Alert Information):
    // -------------------------------
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = Status Code (Integer) -> {0x00=Success | 0x01=No Active Alert
    //                                      | 0x07=Unknown Failure}
    //  (1)     = Slave Address (Integer)
    //  (2)     = Data Length (Integer)
    //  (3)     = Data (Integer)
    //
    Method(_SBA, 0)
    {
        //
        // Local0 is the return package.  The status code is defaulted
        // to 'unknown failure' (0x07).
        //
        Store(Package(4){0x07,0x00,0x00,0x00}, Local0)

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX0, 0xFFFF), 0))
        {
            //
            // Make sure there's an active (non-consumed) alarm by
            // checking bit 6 of the status register.  If not, the
            // return package already indicates that there isn't an
            // active alarm.
            //
            If(And(STS, 0x40))
            {
                Store(0x00, Index(Local0, 0))       // Success

                ShiftRight(AADR, 1, Index(Local0, 1))       // Slave Address

                Store(2, Index(Local0, 2))          // Data Length

                Store(ShiftLeft(ADB1, 8), Local1)   // Data
                Add(Local1, ADB0, Local1)
                Store(Local1, Index(Local0, 3))

                //
                // Clear the alarm by resetting the status register.
                //
                Store(0x00, STS)
            }
            Else
            {
                Store(0x01, Index(Local0, 0))       // Status = No Alert
            }

            Release(SBX0)
        }

        Return(Local0)
    } // _SBA()

} // Device(SMB0)
```

```
////////////////////////////////////////////////////////////////////////////
// Device:      SMB1
// -----------------------------------------------------------------------
// Description: The second EC-SMBus segment (includes the Maxim 1617).
////////////////////////////////////////////////////////////////////////////
Device(SMB1)
{
    Name(_HID, "SMBUS01")
    Name(_UID, 1)

    Mutex(SBX1, 0)          // SMBus transactional synchronization.

    //
    // OperationRegion:
    // ----------------
    // This SMBus resides at offset 0x30 in EC space.  See the ACPI
    // Specification for information on the EC-SMBus register interface.
    //
    OperationRegion(SMB1,EmbeddedControl,0x30,0x40)
    Field(SMB1,ByteAcc,Lock,Preserve)
    {
        PRTC,   8,     // Protocol
        STS,    8,     // Status
        ADDR,   8,     // Address
        CMD,    8,     // Command
        DATA,   256,   // SMBus Data Bytes (Block)
        BCNT,   8,     // Block Count
        AADR,   8,     // Alarm Address
        ADB0,   8,     // Alarm Data Byte 0
        ADB1,   8      // Alarm Data Byte 1
    }
    Field(SMB1,ByteAcc,Lock,Preserve)
    {
        Offset(4),     // Move to byte offset 4 (beginning of data).
        DAT0, 8,       // 8-bit data register for byte/word access.
        DAT1, 8        // 8-bit data register for word access.
    }

    //
    // _SBI (SMBus Information):
    // -------------------------
    // Returns a SMB_INFO structure describing the properties of this
    // SMBus segment.
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = SMBus Control Method Interface (CMI) version (Integer)
    //  (1)     = SMB_INFO data structure (Buffer)
    //
    Method(_SBI)
    {
        //
        // Local0 is the return package.  The CMI version is set
        // to v1.0 (0x10).
        //
        Store(Package(2){0x10,0x00}, Local0)

        Store(Buffer()
        {
            0x10,                   // SMB_INFO structure Version (v1.0)
            0x10,                   // SMBus Specification Version (v1.0)
            0x00,                   // Segment Hardware Capability
            0x0A,                   // Alert Polling Interval (poll every 10 seconds)
            0x01,                   // Device Count
            0x09, 0x00,             // SMBus Device #1: Maxim 1617
            0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Vendor ID is a placeholder)
            0x00, 0x01, 0x00, 0x00, //                  (Device ID is a placeholder)
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00
        }, Index(Local0, 1))
        Return(Local0)
    } // _SBI
```

```
// SWTC (Wait for Transaction Complete):
// ------------------------------------------
// Wait until the previous SMBus transaction has completed.
//
// Parameters:
//  Arg0    = Timeout Value (in ms)
//
// Return Value:
//  0x00    = OK
//  0x07    = Unknown Failure
//  0x10    = Address Not Acknowledged
//  0x11    = Device Error
//  0x12    = Command Access Denied
//  0x13    = Unknown Error
//  0x17    = Device Access Denied
//  0x18    = Timeout
//  0x19    = Unsupported Protocol
//  0x1A    = Bus Busy
//  0x1F    = PEC (CRC-8) Error
//
Method(SWTC, 1)
{
    Store(Arg0, Local0)
    Store(0x07, Local2)

    //
    // The previous command has completed when the protocol
    // register is equal to 0 (zero).  Wait <timeout> ms
    // (in 10ms chunks) for this to occur.
    //
    Store(1, Local1)
    While(LEqual(Local1, 1))
    {
        If(LEqual(PRTC, 0))
        {
            And(STS, 0x1F, Local2)      // Store status code.
            Store(0x00, Local1)         // Terminate loop.
        }
        Else
        {
            //
            // Transaction isn't complete.  Check for timeout, and if not,
            // sleep 10ms and loop again.
            //
            If(LLess(Local0, 10))
            {
                Store(0x18, Local2)     // ERROR: Timeout occurred.
                Store(0x00, Local1)     // Terminate loop.
            }
            Else
            {
                Sleep(10)
                Subtract(Local0, 10, Local0)
            }
        }
    }

    Return(Local2)
} // Method(SWTC)

//
// _SBR (SMBus Read):
// ------------------
//
// Parameters:
//  Arg0    = Protocol (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command (Integer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//  (1)     = Data Length (Integer)
//  (2)     = Data (Integer | Buffer)
//
Method(_SBR, 3)
{
```

```
//
// Local0 is the return package.  The status code is defaulted
// to 'unknown failure' (0x07).
//
Store(Package(3){0x07,0x00,0x00}, Local0)

//
// Make sure the protocol is valid, if not return the
// 'invalid protocol' status code.  Note that this segment
// does not support packet error checking.
//
If(LNotEqual(Arg0, 0x03))                   // Read Quick
{
    If(LNotEqual(Arg0, 0x05))               // Receive Byte
    {
        If(LNotEqual(Arg0, 0x07))           // Read Byte
        {
            If(LNotEqual(Arg0, 0x09))       // Read Word
            {
                If(LNotEqual(Arg0, 0x0B))   // Read Block
                {
                    Store(0x19, Index(Local0, 0))
                    Return(Local0)
                }
            }
        }
    }
}

//
// Acquire the SMBus mutex to ensure transactional synchronization.
//
If(LEqual(Acquire(SBX1, 0xFFFF), 0))
{
    //
    // Make sure the SMBus is ready for this transaction.  If not,
    // return a 'bus busy' error code.  Note that 'we' should be
    // the only consumer...
    //
    If(LNotEqual(PRTC, 0))
    {
        Store(0x1A, Index(Local0, 0))   // ERROR: Bus is busy.
    }
    Else
    {
        //
        // Initiate the transaction by writing the slave address,
        // command, and protocol registers.  Note that the command
        // code is always written, even if the protocol (e.g. 'read
        // quick') doesn't require it (it will be ignored). Note also
        // that the “Read Block" always return 32-byte buffer regardless
        // of the actual block length. This is not a requirement but
        // is implementation specific to this sample ASL.
        //
        Store(ShiftLeft(Arg1, 1), ADDR)
        Store(Arg2, CMD)
        Store(Arg0, PRTC)

        //
        // Wait for completion.  Save the status code, data size,
        // and data into the return package (if required by the
        // protocol).
        //
        Store(SWTC(1000), Index(Local0, 0))

        If(LEqual(Arg0, 0x05))          // Receive Byte
        {
            Store(1, Index(Local0, 1))
            Store(DAT0, Index(Local0, 2))
        }
        If(LEqual(Arg0, 0x07))          // Read Byte
        {
            Store(1, Index(Local0, 1))
            Store(DAT0, Index(Local0, 2))
        }
        If(LEqual(Arg0, 0x09))          // Read Word
```

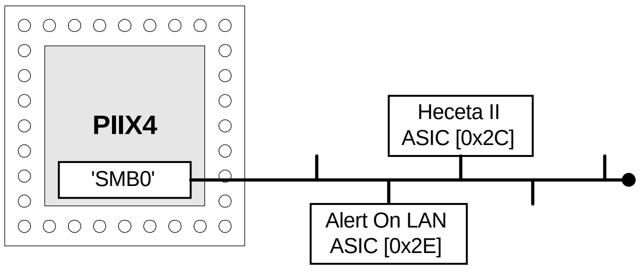
```
            {
                Store(2, Index(Local0, 1))
                Store(DAT1, Local1)
                ShiftLeft(Local1, 8, Local1)
                Add(Local1, DAT0, Local1)
                Store(Local1, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x0B))          // Read Block
            {
                Store(BCNT, Index(Local0, 1))
                Store(DATA, Index(Local0, 2))
            }
        }

        Release(SBX1)
    }

    Return(Local0)
} // _SBR()

//
// _SBW (SMBus Write):
// ------------------
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//  Arg3    = Data Length (Integer)
//  Arg4    = Data (Integer | Buffer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//
Method(_SBW, 5)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(1){0x07}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking or the 'process call'
    // protocol.
    //
    If(LNotEqual(Arg0, 0x02))                   // Write Quick
    {
        If(LNotEqual(Arg0, 0x04))               // Send Byte
        {
            If(LNotEqual(Arg0, 0x06))           // Write Byte
            {
                If(LNotEqual(Arg0, 0x08))       // Write Word
                {
                    If(LNotEqual(Arg0, 0x0A))   // Write Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional synchronization.
    //
    If(LEqual(Acquire(SBX1, 0xFFFF), 0))
    {
        //
        // Make sure the SMBus is ready for this transaction.  If not,
        // return a 'bus busy' error code.  Note that 'we' should be
        // the only consumer...
        //
```

```
            If(LNotEqual(PRTC, 0))
            {
                Store(0x1A, Local0)
            }
            Else
            {
                //
                // Initiate the transaction by writing the slave address,
                // command, and protocol registers. Note that the command
                // code and data length are always written, even if the
                // protocol (e.g. 'write quick') doesn't require it (it
                // will be ignored).
                //
                Store(ShiftLeft(Arg1, 1), ADDR)
                Store(Arg2, CMD)
                Store(Arg3, BCNT)

                If(LEqual(Arg0, 0x06))          // Write Byte
                {
                    Store(Arg4, DAT0)
                }
                If(LEqual(Arg0, 0x08))          // Write Word
                {
                    And(Arg4, 0x00FF, DAT0)
                    ShiftRight(Arg4, 8, DAT1)
                }
                If(LEqual(Arg0, 0x0A))          // Write Block
                {
                    Store(Arg4, DATA)
                }

                Store(Arg0, PRTC)

                //
                // Wait for completion.
                //
                Store(SWTC(1000), Local0)
            }
            Release(SBX1)
        }

        Return(Local0)
    } // _SBW()

    //
    // _SBA (SMBus Alert Information):
    // -------------------------------
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = Status Code (Integer) -> {0x00=Success | 0x01=No Active Alert | 0x07=Unknown
Failure}
    //  (1)     = Slave Address (Integer)
    //  (2)     = Data Length (Integer)
    //  (3)     = Data (Integer)
    //
    Method(_SBA, 0)
    {
        //
        // Local0 is the return package.  The status code is defaulted
        // to 'unknown failure' (0x07).
        //
        Store(Package(3){0x07,0x00,0x00}, Local0)

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX1, 0xFFFF), 0))
        {
            //
            // Make sure there's an active (non-consumed) alarm by
            // checking bit 6 of the status register.  If not, the
            // return package already indicates that there isn't an
            // active alarm.
            //
            If(And(STS, 0x40))
            {
                Store(0x00, Index(Local0, 0))       // Status = Success

                ShiftRight(AADR, 1, Index(Local0, 1))       // Slave Address

                Store(2, Index(Local0, 2))          // Data Length

                Store(ShiftLeft(ADB1, 8), Local1)   // Data
                Add(Local1, ADB0, Local1)
                Store(Local1, Index(Local0, 3))

                //
                // Clear the alarm by resetting the status register.
                //
                Store(0x00, STS)
            }
            Else
            {
                Store(0x01, Index(Local0, 0))       // Status = No Alert
            }

            Release(SBX1)
        }

        Return(Local0)
    } // _SBA()

} // Device(SMB1)
```

B.2 Desktop Example

The sample ASL presented in this section was tested on an Intel desktop system. This platform uses the built-in SMBus controller available on the PIIX4 chipset. Devices connect to this segment include the Heceta II and Alert on LAN manageability ASICs. Figure 8 illustrates a logical view of this SMBus configuration.

Figure 8: Desktop Example System



Information on the PIIX4 ASIC is available at:
    http://developer.intel.com/design/intarch/DATASHTS/209562.htm

A functional overview of the Heceta II is available at:
    http://developer.intel.com/ial/wfm/wfm20/design/sensdt/HEC2FUNC.HTM

Information on the Alert on LAN ASIC is available at:
    http://developer.intel.com/design/network/datashts/69281801.pdf

B.2.1 ASL Overview

The desktop sample ASL presented in section B.2.2 includes a single SMBus CMI object (`SMB0`) representing the PIIX4's built-in SMBus host controller. This device is assigned a `_HID` value of `'SMBUS01'` and unique `_UID` value to allow enumeration by the OS and higher-level software.

As with the mobile example, a mutex is declared (`SBX0`) to ensure *transactional synchronization* for accesses to this segment. The PIIX4 exposes a set of registers for the SMBus host controller at 0x7000 in system IO space, which is mapped using the `OperationRegion` primitive.

The `_SBI` control method is required and thus was implemented for the segment. Although not required, all of the 'read' and 'write' protocols (except *read/write block*) were implemented in the _SBR and `_SBW` control methods due to the simplicity of the PIIX4 SMBus hardware interface. The _SBA control method was not implemented, as neither the Heceta II nor Alert on LAN devices are capable of generating SMBus alerts.

Note that testing was performed on only the read/write byte protocols – corresponding to the capabilities of the devices on this segment.

B.2.2 PIIX4 SMBus Sample ASL

```
////////////////////////////////////////////////////////////////////////////////
// Device:      SMB0
// ----------------------------------------------------------------------------
// Description: The PIIX4 SMBus segment (includes the ADM9240 and AOL ASIC).
//              This ASL assumes the SMBus is enabled and its registers are
//              located at offset 0x7000.
////////////////////////////////////////////////////////////////////////////////
Device(SMB0)
{
```

```
Name(_HID, "SMBUS01")
Name(_UID, 0)

Mutex(SBX0, 0)          // SMBus transactional synchronization.

//
// OperationRegion:
// ----------------
// The PIIX4 SMBus registers reside at offset 0x7000 in system IO space.
//
OperationRegion(SMB0,SystemIO,0x7000,0x0C)
Field(SMB0,ByteAcc,NoLock,Preserve)
{
    HSTS,   8,  // Host Status
    SSTS,   8,  // Slave Status
    HCNT,   8,  // Host Control
    HCMD,   8,  // Host Command
    HADD,   8,  // Host Address
    DAT0,   8,  // Host Data Byte 0
    DAT1,   8,  // Host Data Byte 1
    BLKD,   8,  // Host Block Data
    SCNT,   8,  // Slave Count
    SCMD,   8,  // Shadow Command
    SEVT,   8,  // Slave Event
    SDAT,   8   // Slave Data

}

//
// _SBI (SMBus Information):
// -------------------------
// Returns a SMB_INFO structure describing the properties of this
// SMBus segment.
//
// Parameters:
//  <none>
//
// Return Value (Package):
//  (0)     = SMBus Control Method Interface (CMI) version (Integer)
//  (1)     = SMB_INFO data structure (Buffer)
//
Method(_SBI)
{
    //
    // Local0 is the return package.  The CMI version is set
    // to v1.0 (0x10).
    //
    Store(Package(2){0x10,0x00}, Local0)

    Store(Buffer()
    {
        0x10,                   // SMB_INFO structure Version (v1.0)
        0x10,                   // SMBus Specification Version (v1.0)
        0x00,                   // Segment Hardware Capability
        0x00,                   // Alert Polling Interval (no alert capable devices)
        0x02,                   // Device Count
        0x2C, 0x00,             // SMBus Device #1: ADM9240
        0x00, 0x00, 0x11, 0xD4, //      SMB_UDID... (Device ID is a placeholder)
        0x00, 0x01, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00,
        0x2E, 0x00,             // SMBus Device #2: AOL ASIC
        0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Device ID is a placeholder)
        0x00, 0x04, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00
    }, Index(Local0, 1))
    Return(Local0)
} // _SBI

//
// SWTC (Wait for Transaction Complete):
// -----------------------------------------
// Wait until the previous SMBus transaction has completed.
//
// Parameters:
```

```
//  Arg0    = Timeout Value (in ms)
//
// Return Value:
//  0x00    = OK
//  0x07    = Unknown Failure
//  0x10    = Address Not Acknowledged
//  0x11    = Device Error
//  0x12    = Command Access Denied
//  0x13    = Unknown Error
//  0x17    = Device Access Denied
//  0x18    = Timeout
//  0x19    = Unsupported Protocol
//  0x1A    = Bus Busy
//  0x1F    = PEC (CRC-8) Error
//
Method(SWTC, 1)
{
    Store(Arg0, Local0)
    Store(0x07, Local2)

    //
    // The previous command has completed when bit 1 ('interrupt status')
    // of the host status (HSTS) register is set or an error occurs.
    // Wait <timeout> ms (in 10ms chunks) for this to occur.
    //
    Store(1, Local1)
    While(LEqual(Local1, 1))
    {
        //
        // Read the hosts status (HSTS) register, mask off bits 4:1, and
        // check to see if this transaction has completed.  Note that the
        // transaction is being processed while these bits are non-zero.
        //
        Store(And(HSTS, 0x1E), Local3)
        If(LNotEqual(Local3, 0))
        {
            //
            // See if this transaction was successful.  Note that errors
            // are reported in bits 4:2.
            //
            If(LEqual(Local3, 0x02))
            {
                Store(0x00, Local2)     // Success.
            }
            Else
            {
                Store(0x07, Local2)     // ERROR: Unknown Error.
            }

            Store(0, Local1)            // Terminate loop.
        }
        Else
        {
           //
           // Transaction isn't complete.  Check for timeout, and if not,
            // sleep 10ms and loop again.
           //
           If(LLess(Local0, 10))
           {
              Store(0x18, Local2)       // ERROR: Timeout occurred.
              Store(0x00, Local1)       // Terminate loop.
           }
           Else
           {
              Sleep(10)
              Subtract(Local0, 10, Local0)
           }
        }
    }

    Return(Local2)
} // Method(SWTC)

//
// _SBR (SMBus Read):
// ------------------
```

```
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//
// Return Value (Package):
//  (0)     = Status Code (Integer)
//  (1)     = Data Length (Integer)
//  (2)     = Data (Integer | Buffer)
//
Method(_SBR, 3)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(3){0x07,0x00,0x00}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking.
    //
    If(LNotEqual(Arg0, 0x03))                   // Read Quick
    {
        If(LNotEqual(Arg0, 0x05))               // Receive Byte
        {
            If(LNotEqual(Arg0, 0x07))           // Read Byte
            {
                If(LNotEqual(Arg0, 0x09))       // Read Word
                {
                    If(LNotEqual(Arg0, 0x0B))   // Read Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional
    // synchronization.
    //
    If(LEqual(Acquire(SBX0, 0xFFFF), 0))
    {
        //
        // Translate the slave address (shift left + set 'read' bit) and
        // write this and the command byte to the SMBus registers.  Clear
        // the host status register (preserving bits 7:5).
        //
        Store(Or(ShiftLeft(Arg1, 0x01), 0x01), HADD)
        Store(Arg2, HCMD)
        Store(Or(HSTS, 0x1F), HSTS)

        //
        // Specify the protocol using bits 4:2 of the host control
        // register (preserving bits 5 & 7) and start the transaction
        // by writing a '1' to bit 6 of the host control register.
        //
        If(LEqual(Arg0, 0x03))          // Read Quick
        {
            Store(Or(And(HCNT, 0xA0), 0x40), HCNT)
        }
        If(LEqual(Arg0, 0x05))          // Receive Byte
        {
            Store(Or(And(HCNT, 0xA0), 0x44), HCNT)
        }
        If(LEqual(Arg0, 0x07))          // Read Byte
        {
            Store(Or(And(HCNT, 0xA0), 0x48), HCNT)
        }
        If(LEqual(Arg0, 0x09))          // Read Word
        {
            Store(Or(And(HCNT, 0xA0), 0x4C), HCNT)
        }
        If(LEqual(Arg0, 0x0B))          // Read Block
        {
            Store(Or(And(HCNT, 0xA0), 0x54), HCNT)
        }

        //
        // Wait (up to 1 second) for the transaction to complete.  Store
        // the status code and, if successful, the data length and data
        // into the return package.
        //
        Store(SWTC(1000), Local1)
        Store(Local1, Index(Local0, 0))

        If(LEqual(Local1, 0))
        {
            If(LEqual(Arg0, 0x05))          // Receive Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x07))          // Read Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x09))          // Read Word
            {
                Store(2, Index(Local0, 1))
                Store(DAT1, Local2)
                ShiftLeft(Local2, 8, Local2)
                Add(Local2, DAT0, Local2)
                Store(Local2, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x0B))          // Read Block
            {
                // TBD...
            }
        }

        Release(SBX0)
    }

    Return(Local0)
} // _SBR()

//
// _SBW (SMBus Write):
// ------------------
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//  Arg3    = Data Length (Integer)
//  Arg4    = Data (Integer | Buffer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//
Method(_SBW, 5)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(1){0x07}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking or the 'process call'
    // protocol.
    //
    If(LNotEqual(Arg0, 0x02))                   // Write Quick
```

```
        {
            If(LNotEqual(Arg0, 0x04))               // Send Byte
            {
                If(LNotEqual(Arg0, 0x06))           // Write Byte
                {
                    If(LNotEqual(Arg0, 0x08))       // Write Word
                    {
                        If(LNotEqual(Arg0, 0x0A))   // Write Block
                        {
                            Store(0x19, Index(Local0, 0))
                            Return(Local0)
                        }
                    }
                }
            }
        }

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX0, 0xFFFF), 0))
        {
            //
            // Translate the slave address (shift left) and write this and
            // the command byte to the SMBus registers.  Clear the host status
            // register (preserving bits 7:5).
            //
            Store(ShiftLeft(Arg1, 0x01), HADD)
            Store(Arg2, HCMD)
            Store(Or(HSTS, 0x1F), HSTS)

            //
            // Store the data to be written (if any), specify the protocol
            // using bits 4:2 of the host control register, and start the
            // transaction by writing a '1' to bit 6 of the host control
            // register.
            //
            If(LEqual(Arg0, 0x02))          // Write Quick
            {
                Store(Or(And(HCNT, 0xA0), 0x40), HCNT)
            }
            If(LEqual(Arg0, 0x04))          // Send Byte
            {
                Store(Or(And(HCNT, 0xA0), 0x44), HCNT)
            }
            If(LEqual(Arg0, 0x06))          // Write Byte
            {
                Store(Arg4, DAT0)
                Store(Or(And(HCNT, 0xA0), 0x48), HCNT)
            }
            If(LEqual(Arg0, 0x08))          // Write Word
            {
                And(Arg4, 0x00FF, DAT0)
                ShiftRight(Arg4, 8, DAT1)
                Store(Or(And(HCNT, 0xA0), 0x4C), HCNT)
            }
            If(LEqual(Arg0, 0x0A))          // Write Block
            {
                // TBD...
                Store(Or(And(HCNT, 0xA0), 0x54), HCNT)
            }

            //
            // Wait (up to 1 second) for the transaction to complete.  Store
            // the status code into the return package.
            //
            Store(SWTC(1000), Index(Local0, 0))

            Release(SBX0)
        }

        Return(Local0)
    } // _SBW()

} // Device(SMB0)
```