SMBus Device Driver External Architecture 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 SMBus Forum at:
http://www.sbs-forum.org/smbus/index.html/

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 Target Audience

This specification is intended for use by the following audience:

- ISVs and OEMs developing software applications that access SMBus.
- Others interested in accessing SMBus devices.

Basic understanding of SMBus and Windows Driver Model (WDM) is assumed.

1.2 Related Documents

[1] *SMBus Control Method Interface Specification v1.0,* SBS-Implementers Forum, ©1999.

[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

[3] *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/

[4] *Windows 2000 DDK,* ©1999 Microsoft Corporation.

1.3 Data Conventions

1.3.1 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.
EC	Embedded Controller.
HC	SMBus Segment Host Controller.
ICHx	Intel I/O Hub Controller.
IOCTL	Input/Output Control.

Acronym	Description
IRP	Interrupt Request Packet.
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.
WDM	Windows Driver Model

2 Architecture Overview

The SMBus device driver is implemented as a class and mini-port driver pair. The class driver is `SMBCMICL.SYS` and the mini-port is `SMBCMIHC.SYS`. The SMBus device driver receives requests in the form of IOCTLs (I/O Control) and converts them into calls to the appropriate ACPI control method. The ACPI SMBus control methods resident in AML on the platform perform the actual interfacing with the SMBus host controller. The AML code is responsible for actually interfacing with the physical SMBus hardware. The driver also monitors SMBus segments for alerts, and forwards notifications to any entities that have registered to receive them.

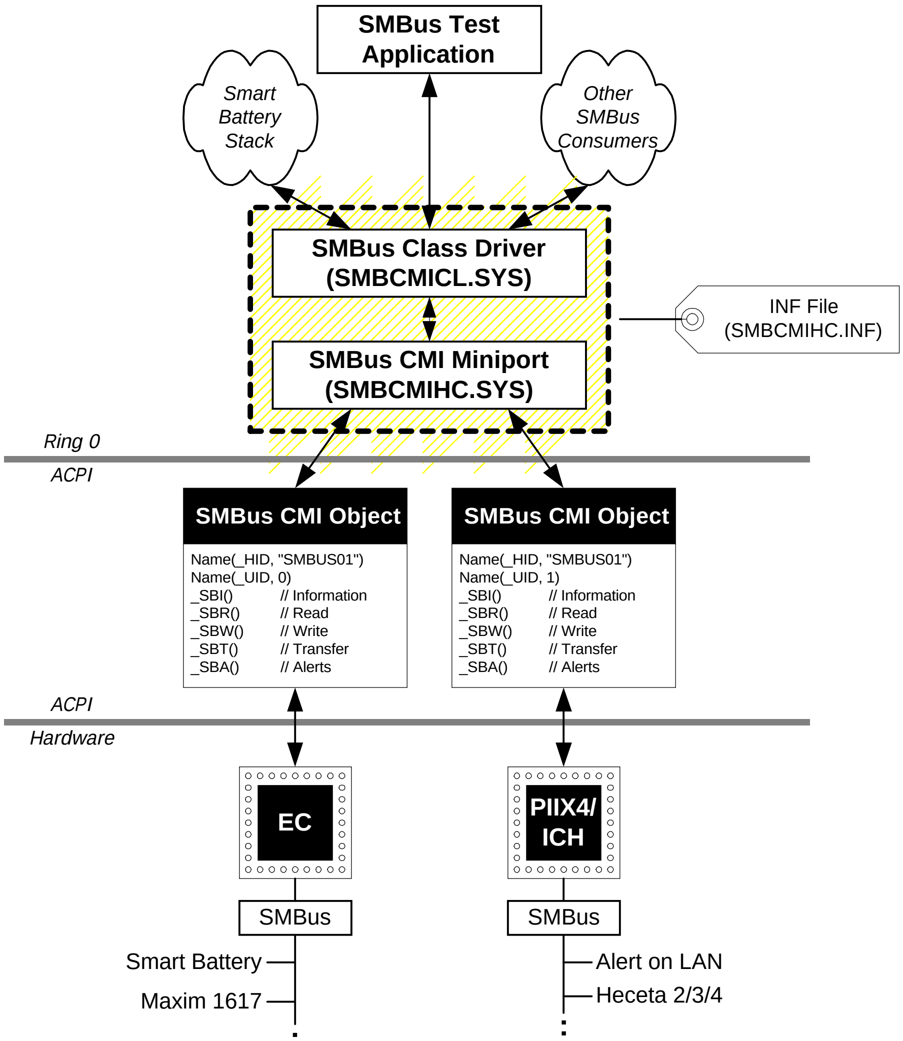


Figure 1: SMBus Driver and CMI Architecture

The SMBus device driver supports Windows 98 and Windows 2000. Because the driver uses ACPI control methods to interface with the SMBus host controllers, ACPI must be enabled. The SMBus device driver conforms to the *System Management Bus Specification v1.0, and v1.1*.

3 SMBus Device Driver Interface

This document defines the exposed external interfaces of the SMBus device driver. The SMBus device driver is intended to operate on both Windows 98 and Windows 2000. The device driver conforms to the Windows Driver Model (WDM) as set forth by Microsoft. The device driver uses ACPI SMBus control methods to communicate with SMBus hardware as defined in the *SMBus Control Method Interface Specification*.

The SMBus device driver provides a kernel mode (ring 0) internal IOCTL interface that supports the following operations:

- `SMB_SEGMENT_INFORMATION` – Enumerates an SMBus segment and its attached devices
- `SMB_BUS_REQUEST` – Initiates read, write, etc. requests to a specific SMBus device
- `SMB_REGISTER_ALARM_NOTIFY` – Registers the caller to receive alarm notifications
- `SMB_DEREGISTER_ALARM_NOTIFY` – Unregisters the caller from receiving alarm notifications

Additionally the SMBus driver will notify a client of an alert on a registered slave address by calling the client's notify function `SMB_ALARM_NOTIFY`.

3.1 Obtaining a Handle to the SMBus

The SMBus device driver creates a device object for each ACPI enumerated SMBus segment. Higher-level drivers enumerate the SMBus device interface by using the **IoGetDeviceInterfaces** kernel function using the SMB_GUID. This function returns a pointer to the SMBus device object that may be passed IOCTLs via the **IoCallDriver** kernel function.

```
// GUID_SMB {EF2CEA02L-64FC-11D2-9EE7-00AA0009E4E6}
DEFINE_GUID (GUID_SMB,
               0xef2cea02L, 0x64fc, 0x11d2, 0x9e, 0xe7, 0x0, 0xaa, 0x0, 0x9, 0xe4, 0xe6);
```

See section 3.2.5.1 for programming details.

3.2 Enumerating SMBus Information

Once a handle is obtained to a valid SMBus segment device, an upper-level driver can initiate a `SMB_SEGMENT_INFORMATION` IOCTL to obtain information about the segment and all of the slave devices attached to it.

3.2.1 IOCTL Code

```
#define SMB_SEGMENT_INFORMATION  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x04, METHOD_NEITHER,
FILE_ANY_ACCESS)
```

See section 3.2.5.1 for programming details.

3.2.2 Input

`Irp->Parameters.DeviceIoControl.Type3InputBuffer` points to an allocated memory region large enough to hold a `SMB_INFO` data structure (see 3.2.4.1) and zero or more `SMB_DEVICE` structures (one for each slave device that resides on the segment).

`Irp->Parameters.DeviceIoControl.InputBufferLength` specifies the size (in bytes) of the memory allocated by the caller for the `Type3InputBuffer`.

3.2.3 Output

`IoStatus.Status` is set to `STATUS_SUCCESS` if the operation succeeded, `STATUS_PENDING` if the operation has been queued, `STATUS_BUFFER_TOO_SMALL` if the `Type3InputBuffer` allocated was too small, or another error code (e.g. `STATUS_INVALID_PARAMETER`) to indicate failure.

`IoStatus.Information` is set to the size of the `SMB_INFO` structure returned in the `Type3InputBuffer`. If the IRP is pending, this field is set to 0 (zero). If the input buffer is too small (`STATUS_BUFFER_TOO_SMALL`), this field is set to the minimum required length of the input buffer.

`Irp->Parameters.DeviceIoControl.Type3InputBuffer` points to a valid `SMB_INFO` structure describing the SMBus segment and all slave devices if the operation was successful (`STATUS_SUCCESS`).

3.2.4 Data Structures

```
#define ANYSIZE_ARRAY 1

struct SMB_INFO
{
    BYTE SMB_INFOVersion;
    BYTE SMBusSpecificationVersion;
    BYTE SegmentHWCapability;
    BYTE Reserved;
    BYTE DeviceCount;
    SMB_DEVICE DeviceArray[ANYSIZE_ARRAY];
};

struct SMB_DEVICE
{
    BYTE    SlaveAddress;
    BYTE    Reserved;
    SMB_UDID UDID;
}

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

Note that all structures defined in this specification assume single-byte alignment [e.g. `#pragma pack(1)`].

3.2.4.1 SMB_INFO Structure

The `SMB_INFO` data structure specifies the general characteristics of an SMBus Segment.

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 structure.
0x01	SMBus Specification Version	BYTE	Varies	This field specifies the version of the SMBus Specification that the SMBus host controller 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 version 1.0 of the SMBus Specification.
0x02	Segment Hardware Capability	BYTE	Bit Field	This field specifies the basic hardware capabilities of this SMBus segment. See 3.2.4.2.
0x03	Reserved	BYTE	Varies	Reserved.
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 3.2.4.3.

3.2.4.2 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 checking as defined in the SMBus v1.1 specification.
Bits 1:7	<Reserved>	Cleared (0).

3.2.4.3 SMB_Device Structure

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).
0x01	<Reserved>	BYTE	0x00	Cleared (0).
0x02	Device UDID	BYTE	Bit Field	This 16-byte (128-bit) value specifies the device ID for this SMBus device. See 3.2.4.4.

3.2.4.4 SMB_UDID

Offset	Name	Length	Value	Description
0x00	Device Hardware Capabilities	BYTE	Varies	This field specifies the hardware capabilities of this SMBus device. See 3.2.4.5.
0x01	Version/ Revision	BYTE	Varies	This field specifies the UDID version and silicon revision ID for this SMBus device. See 3.2.4.6.
0x02	Vendor ID	WORD	Varies	This field specifies the device manufacturer's ID as assigned by the Smart Battery System Implementers Forum.
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 3.2.4.7.
0x08	Subsystem Vendor ID	WORD	Varies	This field specifies the subsystem interface ID as assigned by the Smart Battery System Implementers Forum. 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.

3.2.4.5 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)

3.2.4.6 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.
Bit 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).
Bits 6:7	<Reserved>	Cleared (0).

3.2.4.7 Interface

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

Bits	Name	Description
Bits 0:3	SMBus Version	These bits designate 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).

3.2.5 Process

The client needs to allocate a memory region large enough to hold the `SMB_INFO` structure and an array of `SMB_DEVICE` structures (one for each slave device on the segment), initialize this memory region (zeroed out), and point the IRP's `Type3InputBuffer` to this region. The `InputBufferLength` should be set to the size (in bytes) of this memory region.

Next the client should initiate an `SMB_SEGMENT_INFORMATION` request and wait until the command has completed (`IoStatus.Status != STATUS_PENDING`). The size of the returned data will be available in `IoStatus.Information`.

Since these operations are executed from within a kernel mode (ring 0) process, synchronization is accomplished by the process itself.

```
Status = IoCallDriver(pLowerDO, pIrp);
if (Status == STATUS_PENDING)
{
    KeWaitForSingleObject(&Event, Executive, KernelMode, FALSE, NULL);
    Status = ioStatus.Status;
}
```

If successful, the `Type3InputBuffer` can be recast as a `SMB_INFO` pointer and parsed. Note that the number of device structures (size of the device array) is given in the `DeviceCount` field of the `SMB_INFO` structure, but can also be calculated using the structure sizes for validation purposes.

3.2.5.1 Example: SMBus Enumeration

```
#define MYVENDORID 0x8086
#define MYDEVICEID 0x0001

//
// Irp Completion Routine
//
NTSTATUS
SynchFunction(
              IN PDEVICE_OBJECT DeviceObject,
              IN PIRP Irp,
              IN PVOID Context
              )
{
    PKEVENT     Event;

    Event = (PKEVENT) Context;
    KeSetEvent(Event, 0, FALSE);
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

```
// Sample function which gets smbus segments, and shows how
// a higher-level driver would enumerate its devices.
//
NTSTATUS
FindDevicesOnSMBus()
{
    PWSTR               pInterfaceList;
    PWSTR               pInterfaceListWalk;

    PSMB_INFORMATION    pSmbInfo;
    NTSTATUS            status;
    //
    // Enumerate buses
    //
    status = IoGetDeviceInterfaces(&GUID_SMB,
        NULL,
        0,
        &pInterfaceList);

    if (!NT_SUCCESS(status))
    {
        return status;
    }

    //
    // we need to keep the original ptr
    // around to free the mem, so walk the list with this
    //
    pInterfaceListWalk = pInterfaceList;

    pSmbInfo = ExAllocatePool(PagedPool, sizeof(SMB_INFORMATION));

    if (!pSmbInfo)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    while (*pInterfaceListWalk != UNICODE_NULL)
    {
        PDEVICE_OBJECT      pTargetDevice;
        UNICODE_STRING      TargetDeviceName;
        PIRP                pIrp;
        KEVENT              Event;
        IO_STATUS_BLOCK     Iosb;

        PIO_STACK_LOCATION  pIrpSp;

        PFILE_OBJECT        pFileObject;

        //
        // Create counted string version of
        // target device name.
        //
        RtlInitUnicodeString(&TargetDeviceName, pInterfaceList);

        KeInitializeEvent(&Event, NotificationEvent, FALSE);

        //
        // Get a pointer to its Device object
        //
        status = IoGetDeviceObjectPointer(
            &TargetDeviceName,
            FILE_ALL_ACCESS,
            &pFileObject,
            &pTargetDevice);

        if (!NT_SUCCESS(status))
        {
            return status;
        }

        //
        // Decrement reference count on unused
        // File object
        //
```

```
        ObDereferenceObject(pFileObject);

        RtlZeroMemory(pSmbInfo, sizeof(SMB_INFORMATION) );
        pIrp = IoAllocateIrp(pTargetDevice->StackSize, FALSE);

        pIrpSp = IoGetNextIrpStackLocation(pIrp);

        pIrp->UserBuffer = pSmbInfo;
        pIrpSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;

        pIrpSp->Parameters.DeviceIoControl.IoControlCode = SMB_INFORMATION_REQUEST;
        pIrpSp->Parameters.DeviceIoControl.InputBufferLength = sizeof(SMB_INFORMATION);
        pIrpSp->Parameters.DeviceIoControl.Type3InputBuffer = pSmbInfo;
        pIrpSp->Parameters.DeviceIoControl.OutputBufferLength = sizeof(SMB_INFORMATION);

        IoSetCompletionRoutine(pIrp, SynchFunction, &Event, TRUE, TRUE, TRUE);

        //
        // Call SMB CMI driver to get segment/device information
        //
        status = IoCallDriver(pTargetDevice, pIrp);

        if (status == STATUS_PENDING)
        {
            // Wait for the IRP to be completed, then grab the real status code
            KeWaitForSingleObject(
                &Event,
                Executive,
                KernelMode,
                FALSE,
                NULL);

            status = pIrp->IoStatus.Status;
        }

        if (NT_SUCCESS(status))
        {
            //
            // Walk the buffer we got back and see if there are any of our
            // devices on this segment
            //

            UCHAR i;

            for (i = 0; i < pSmbInfo->SmbInfo.DeviceCount; i++)
            {
                if (pSmbInfo->SmbInfo.DeviceArray[i].Device.VendorID == MYVENDORID &&
                    pSmbInfo->SmbInfo.DeviceArray[i].Device.DeviceID == MYDEVICEID)
                {
                    // TODO
                    //
                    // we've got a device, so create a device object and save
                    // the slave address and lower device object pointer
                    // to the device extension
                }
            }
        }
        IoFreeIrp(pIrp);

        //
        // Advance to next interface string
        //
        pInterfaceListWalk += wcslen(pInterfaceListWalk) + 1;
    }

    ExFreePool(pSmbInfo);
    ExFreePool(pInterfaceList);

    return status;
}
```

3.3 Initiating SMBus Requests

Once a handle is obtained to a valid SMBus segment device, an upper-level driver can initiate a SMB_BUS_REQUEST IOCTL to communicate directly to a slave device residing on the segment.

3.3.1 IOCTL Code

```
#define SMB_BUS_REQUEST  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x00, METHOD_NEITHER, FILE_ANY_ACCESS)
```

3.3.2 Input

Irp->Parameters.DeviceIoControl.Type3InputBuffer points to an allocated memory region large enough to hold a SMB_REQUEST data structure (see 3.3.4).

Irp->Parameters.DeviceIoControl.InputBufferLength specifies the size (in bytes) of the memory allocated by the caller for the Type3InputBuffer.

3.3.3 Output

IoStatus.Status is set to STATUS_SUCCESS if the operation succeeded, STATUS_PENDING if the operation has been queued, STATUS_BUFFER_TOO_SMALL if the Type3InputBuffer allocated was too small, or another error code (e.g. STATUS_INVALID_PARAMETER) to indicate failure.

IoStatus.Information is set to the size of the SMB_REQUEST structure returned in the Type3InputBuffer. If the IRP is pending, this field is set to 0 (zero). If the input buffer is too small (STATUS_BUFFER_TOO_SMALL), this field is set to the minimum required length of the input buffer.

Irp->Parameters.DeviceIoControl.Type3InputBuffer points to a SMB_REQUEST data structure describing the results of the SMBus request.

3.3.4 Data Structures

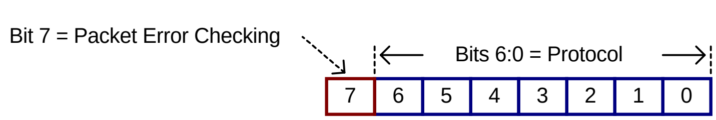
```
#define SMB_MAX_DATA_SIZE 32;

struct SMB_REQUEST
{
    BYTE Status;
    BYTE Protocol;
    BYTE Address;
    BYTE Command;
    BYTE BlockLength;
    BYTE Data[SMB_MAX_DATA_SIZE];
};
```

3.3.4.1 Protocol

SMBus protocols are presented in this specification using an integer-encoded, 8-bit notation, as illustrated in Figure 2. 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 2: Protocol Encoding

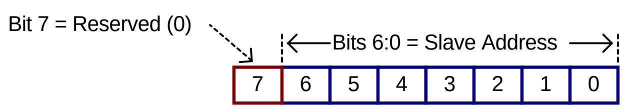


Client software can instruct the SMBus driver to use of packet error checking[1] (PEC) when making a SMBus request by setting bit 7 of the protocol value. PEC capability is optional for the SMBus segments (and devices residing on these segments) that are compliant with version 1.1 of the SMBus specification. The `SegmentHWCapability` and `DeviceHWCapability` fields returned by `SMB_SEGMENT_INFORMATION` enable client 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. A value of 1 (one) in bit 7 of the protocol 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. The actual CRC checking is handled by hardware.

3.3.4.2 Address

Slave addresses are presented in this specification using an integer-encoded, 7-bit, non-shifted notation, as illustrated in Figure 3. 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 3: Slave Address Encoding



3.3.5 Process

The client needs to allocate a memory region large enough to hold the SMB_REQUEST structure, initialize this memory region (zeroed out), and specify the type of request by filling in the protocol, address, and command fields. On write requests, the client specifies the data length and data using the `BlockLength` and `Data` fields. Note that the block length should always be specified on write requests – even when implied by the protocol (e.g. ‘write byte’).

The `Type3InputBuffer` must point to this `SMB_REQUEST` structure, and the size of this structure needs to be specified in the `InputBufferLength` field.

Next the client should initiate an `SMB_BUS_REQUEST` and wait until the command has completed (`IoStatus.Status != STATUS_PENDING`). The size of the returned data will be available in `IoStatus.Informaiton`.

After the request has completed, the `Type3InputBuffer` can be recast as a `SMB_REQUEST` pointer and parsed. The Status field indicates the success/failure code of the request, and should be checked first. On read requests, the returned data length and data are provided in the `BlockLength` and `Data` fields. Note that the data length will always be returned on a read request – even when implied by the protocol (e.g. ‘read byte’).

---

[1] Section 7.4 of the *System Management Bus Specification (Version 1.1)* describes packet error checking (PEC) for improved communication reliability and robustness.

Table 1: SMB_BUS_REQUEST Structure – Input/Output Parameters

SMB_BUS_REQUEST Type	Input					Output		
	Procotol	Address	Command	Block Length	Data	Status	Block Length	Data
SMB_WRITE_QUICK	0x00	0-127	-	0	-	Varies	0	-
SMB_READ_QUICK	0x01	0-127	-	0	-	Varies	0	-
SMB_SEND_BYTE	0x02	0-127	-	1	Input	Varies	0	-
SMB_RECEIVE_BYTE	0X03	0-127	-	0	-	Varies	1	Output
SMB_WRITE_BYTE	0X04	0-127	0-255	1	Input	Varies	-	-
SMB_READ_BYTE	0X05	0-127	0-255	0	-	Varies	1	Output
SMB_WRITE_WORD	0X06	0-127	0-255	2	Input	Varies	-	-
SMB_READ_WORD	0X07	0-127	0-255	0	-	Varies	2	Output
SMB_WRITE_BLOCK	0X08	0-127	0-255	0-32	Input	Varies	-	-
SMB_READ_BLOCK	0X09	0-127	0-255	0	-	Varies	0-32	Output
SMB_PROCESS_CALL	0x0A	0-127	0-255	2	Input	Varies	2	Output

Table 2: 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>

3.3.5.1 Example: SMBus Request (code fragment)

```
SMB_REQUEST SmbRequest;
.
.
.
RtlZeroMemory(&SmbRequest, sizeof(SMB_REQUEST));

pIrp = IoAllocateIrp(pTargetDevice->StackSize, FALSE);

pIrpSp = IoGetNextIrpStackLocation(pIrp);

pIrpSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;

pIrpSp->Parameters.DeviceIoControl.IoControlCode = SMB_BUS_REQUEST;
pIrpSp->Parameters.DeviceIoControl.InputBufferLength = sizeof(SMB_REQUEST);
pIrpSp->Parameters.DeviceIoControl.Type3InputBuffer = &SmbRequest;
IoSetCompletionRoutine(pIrp, SynchFunction, &Event, TRUE, TRUE, TRUE);
```

```
pSmbRequest->Protocol = SMB_READ_WORD;
pSmbRequest->Address  = 0xA;
pSmbRequest->Command  = 0x1;

status = IoCallDriver(pTargetDevice, pIrp);

if (status == STATUS_PENDING)
{
    // Wait for the IRP to be completed, then grab the real status code
    KeWaitForSingleObject(
        &Event,
        Executive,
        KernelMode,
        FALSE,
        NULL);

    status = pIrp->IoStatus.Status;
}

DbgPrint("Status is %x\n", SmbRequest.Status);

IoFreeIrp(pIrp);
.
.
.
```

3.4 SMBus Alarm Registration

3.4.1 IOCTL Code

```
#define SMB_REGISTER_ALARM_NOTIFY   CTL_CODE(FILE_DEVICE_UNKNOWN, 0x01, METHOD_NEITHER,
                                             FILE_ANY_ACCESS)
```

3.4.2 Input

`Parameters.DeviceIoControl.Type3InputBuffer` points to an `SMB_REGISTER_ALARM` structure.

`Parameters.DeviceIoControl.InputBufferLength` specifies the length of the `SMB_REGISTER_ALARM` structure (see 3.4.4).

`Parameters.DeviceIoControl.OutputBufferLength` specifies the number of bytes allocated by the caller for the returned handle, which should equal the size of a VOID pointer (e.g. `sizeof(PVOID)`).

3.4.3 Output

`Irp->UserBuffer` points to a handle to be used when unregistering the alarm.

3.4.4 Data Structures

```
typedef VOID (*SMB_ALARM_NOTIFY)
(
    PVOID Context,
    BYTE Address,
    WORD Data
);

struct SMB_REGISTER_ALARM
{
    BYTE MinAddress;
    BYTE MaxAddress;
    SMB_ALARM_NOTIFY NotifyFunction;
    PVOID NotifyContext;
};
```

3.4.5 Process

The client needs to allocate a memory region large enough to hold the `SMB_REGISTER_ALARM` structure, initialize this memory region (zeroed out), and specify the range of SMBus slave addresses to register notification for (`MinAddress`, `MaxAddress`) inclusive. The client must specify a function pointer in the `NotifyFunction` field, which gets called by the SMBus driver stack whenever an alert occurs on a slave device within the specified range on the given segment. The `UserBuffer` should also be initialized to zero.

The `Type3InputBuffer` must point to this `SMB_REGISTER_ALARM` structure, the size of this structure needs to be specified in the `InputBufferLength` field, and `OutputBufferLength` field must be set to the size allocated by the caller for the `UserBuffer` field (e.g. `sizeof(PVOID)`).

Next the client should initiate a `SMB_REGISTER_ALARM_NOTIFY` request and wait until the command has completed (`IoStatus.Status != STATUS_PENDING`). If the operation was successful, the `UserBuffer` will contain a handle that will be used during deregistration – this value should be retained by the client for future use.

3.4.5.1 Example: SMBus Alarm Registration (code fragment)

```
//
// Function which will be called when the device has an alarm
//
VOID
AlarmFunc(
  PVOID       Context,
  UCHAR       Address,
  USHORT      Data
)
{
    DbgPrint("Alarm on device at %x, Data is %x\n", Address, Data);
}

.
.
.

SMB_REGISTER_ALARM SmbAlarm;
PVOID AlarmHandle;

RtlZeroMemory(&SmbAlarm, sizeof(SMB_REGISTER_ALARM));

pIrp = IoAllocateIrp(pTargetDevice->StackSize, FALSE);

pIrp->UserBuffer = &AlarmHandle;

pIrpSp = IoGetNextIrpStackLocation(pIrp);

pIrpSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;

pIrpSp->Parameters.DeviceIoControl.IoControlCode = SMB_REGISTER_ALARM_NOTIFY;
pIrpSp->Parameters.DeviceIoControl.InputBufferLength = sizeof(SMB_REGISTER_ALARM);
pIrpSp->Parameters.DeviceIoControl.Type3InputBuffer = &SmbAlarm;
pIrpSp->Parameters.DeviceIoControl.InputBufferLength = sizeof(AlarmHandle);
IoSetCompletionRoutine(pIrp, SynchFunction, &Event, TRUE, TRUE, TRUE);

SmbAlarm.MinAddress = 0x20;
SmbAlarm.MaxAddress = 0x30;
SmbAlarm.NotifyFunction = AlarmFunc;
SmbAlarm.NotifyContext = pDeviceExtension;

status = IoCallDriver(pTargetDevice, pIrp);

if (status == STATUS_PENDING)
{
    // Wait for the IRP to be completed, then grab the real status code
    KeWaitForSingleObject(
        &Event,
```

```
            Executive,
            KernelMode,
            FALSE,
            NULL);

        status = pIrp->IoStatus.Status;
    }

    DbgPrint("Status is %x\n", SmbRequest.Status);

    // Save the AlarmHandle here for when you deregister

    IoFreeIrp(pIrp);
    .
    .
    .
```

3.5 SMBus Alarm Deregistration

3.5.1 IOCTL Codes

```
#define SMB_DEREGISTER_ALARM_NOTIFY CTL_CODE(FILE_DEVICE_UNKNOWN, 0x02, METHOD_NEITHER,
                                             FILE_ANY_ACCESS)
```

3.5.2 Input

`Parameters.DeviceIoControl.Type3InputBuffer` points to a handle previously returned by `SMB_REGISTER_ALARM_NOTIFY`.

`Parameters.DeviceIoControl.InputBufferLength` specifies the size of the handle (e.g. `sizeof(PVOID)`)..

3.5.3 Output

None.

3.5.4 Data Structures

None.

3.5.5 Process

The handle that was returned during the alarm registration process (see 3.3.5.1) is used to deregister for alarms. The client should place the handle value in the `Type3InputBuffer`, and its size in the `InputBufferLength`.

Next the client should initiate a `SMB_DEREGISTER_ALARM_NOTIFY` request and wait until the command has completed (`IoStatus.Status != STATUS_PENDING`).

Note that each alarm registration is given a unique handle and thus requires an associated deregistration.

3.5.5.1 Example: SMBus Alarm Deregistration

```
    .
    .
    .
    pIrp = IoAllocateIrp(pTargetDevice->StackSize, FALSE);

    pIrpSp = IoGetNextIrpStackLocation(pIrp);

    pIrpSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
```

```
pIrpSp->Parameters.DeviceIoControl.IoControlCode = SMB_DEREGISTER_ALARM_NOTIFY;
// AlarmHandle is a PVOID previously returned from registering for the alarm.
pIrpSp->Parameters.DeviceIoControl.InputBufferLength = sizeof(AlarmHandle);
pIrpSp->Parameters.DeviceIoControl.Type3InputBuffer = &AlarmHandle;
IoSetCompletionRoutine(pIrp, SynchFunction, &Event, TRUE, TRUE, TRUE);

status = IoCallDriver(pTargetDevice, pIrp);

if (status == STATUS_PENDING)
{
    // Wait for the IRP to be completed, then grab the real status code
    KeWaitForSingleObject(
        &Event,
        Executive,
        KernelMode,
        FALSE,
        NULL);

    status = pIrp->IoStatus.Status;
}

IoFreeIrp(pIrp);
.
.
.
```

3.6 SMBus Alarm Notification

Following a successful alarm (alert) registration, the SMBus stack will notify a client of an alert on a registered slave address by calling the client's notify function. The prototype for this function is shown below (and is identical to `SMB_ALARM_NOTIFY` structure shown in section 3.4.4).

```
typedef VOID (*SMB_ALARM_NOTIFY)
(
    PVOID Context,
    BYTE Address,
    WORD Data
);
```

The `Context` field contains the registration instance handle (in case multiple registrations were made by the client), the slave address that caused the alert is provided in the `Address` field, and the alert data (if any) is provided in the `Data` field.