Intelligent Management Bus (IMB) Driver (v8.10) Specification

Jan 10, 2005

Revision 1.0

Revision History

Date	Rev	Modifications
	1.0	Initial Version

Disclaimers

This Intel IPMI Driver documentation, and the software described in it, are furnished under license and may only be used or copied in accordance with the terms of the license. Except as permitted by such license (which is provided with the software), no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

This Document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this Document. Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Neither this documentation nor the software described in it are intended for use in medical, life saving, or life sustaining applications.

Table Of Contents

This page intentionally left blank

1 Introduction

The IMB driver is a reference implementation of an IPMI driver for the Windows operating system. It is a driver that supports IPMB messaging to server management firmware on IPMI conformant servers.

1.1 Features

The IMB driver provides the following features:

- Common source base across platforms
  Only a small part of the driver is hardware interface specific. A common source base allows bug fixes and enhancements to be shared by all platform implementations.
- Common source base across OS’s
  Although the OS specific portion of the driver is significant, the IPMB and interface portions of the driver are common across OS’s allowing bug fixes and enhancements to be shared by all OS specific drivers.
- Interface type discovery
  The driver allows multiple interface modules to be linked in and provides a mechanism for discovery of the appropriate one at run time. I.e., the same driver object file, compiled for a specific OS will run on all platforms with one of the supported hardware interfaces. The driver supports both the IPMI SMIC and the IPMI KCS style interface specifications.
- Asynchronous messaging
  The asynchronous messaging capability required by the ICMB feature is provided by this driver. This allows for the reception of unsolicited messages.

Note: The IMB driver package contains Windows 2000 and UnixWare specific code which is not described in this document. This document along with the associated code is providing the reference implementation of the IMB driver for the Windows NT operating system only.

1.2 References

The following were used as references in creating this document.

- *Intelligent Platform Management Interface Specification v 1.5.* Revision 1.0

2 Design

This section provides an overview of the driver design. The following three sections provide detailed information on the specific subsystem implementations.

2.1 Structure

The IMB driver is divided into three main modules each of which exports specific functions to be used by the others:

- OS Specific
  This module encapsulates OS specific functions and provides the external interface for access to the driver. It exports to the other modules functions for synchronization and resource management and uses IPMB generic module functions to send and receive IPMB messages.
- IMB Generic
  This module encapsulates all the common IMB related functions which involve composing and sending IMB messages, receiving and validating IMB messages, and managing the asynchronous incoming message queue.
- Interface Specific
  This module layer implements hardware interface specific functions. It exports functions to discover the interface, send requests and get responses from the interface, and get status on the availability of incoming messages. The driver implementation supports systems that conform to the IPMI 0.9 specification as well as to the IMPI 1.0 specification.

The software stack is pictured in Figure 2-1.

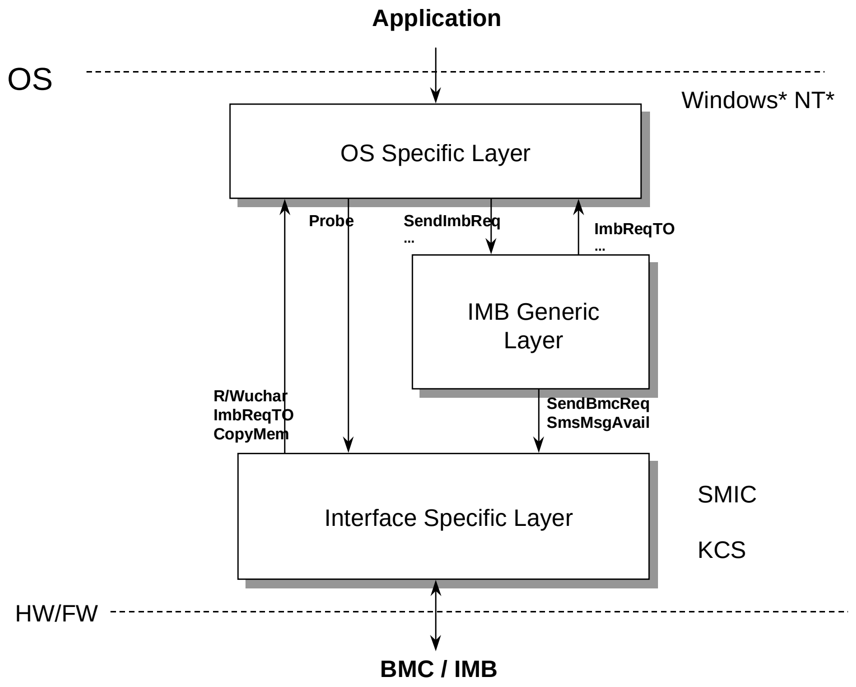

Figure 2-1: IMB Driver Software Stack

3 IMB Generic Code

3.1 IMB Messaging

The IMB driver sends messages either directly to BMC or through the BMC to other controllers, and constructs IMB packets as necessary for the BMC to forward. The current implementation only allows one IMB request, to either the BMC or another controller, to be outstanding at any time. Although the BMC will only allow one outstanding request, it is theoretically possible to have more than one outstanding request to another controller (e.g. an FPC or ICMB bridge).

Asynchronous sends are supported, in which the request is considered completed as soon as it is sent. No response data is returned.

3.1.1 Files

The files containing the IMB Generic code are:

imb_drv.c	– functions to format requests and read responses
imb_drv.h	– defines structures and constants for the generic layer.
imb_async.c	– implements the generic part of the async messaging feature.

3.1.2 API

The generic code exports two IMB messaging functions to the OS specific layer:

SendImbRequest:

```
IMB_STATUS SendImbRequest (
   ImbRequestBuffer *      req,          // IN – request to send
   ImbResponseBuffer*      resp,         // OUT – where to put response
   DWORD *                 respLength,   // IN/OUT – max response and real size
   void *                  context       // IN – info used by OS specific routines
   );
```

This function takes the request as provided by the application and translates it into a request to the interface layer.

If the request is destined for the BMC, the user’s request is directly passed to the interface layer for delivery (after first being copied into a BmcRequest structure). The call to the interface layer is synchronous in that it doesn’t return until either it has the BMC’s response or a request timeout has occurred. The appropriate data and/or status are returned to the caller.

If the request is to a controller other than the BMC, a WriteRead I^{2}C command is constructed and passed to the interface layer to be delivered to the BMC. If the NO_RESPONSE_EXPECTED flag had been set in the request buffer, the routine immediately returns. Otherwise, it waits for the associated IMB response by periodically polling for an available SMS message. SMS messages are received from the BMC until either the expected response is received or a request timeout occurs. Any unexpected SMS messages received are passed to the Asynchronous messaging subsystem.

3.2 System Shutdown Support

The driver provides OS independent support for system shutdown – reset/power off. This is implemented by the following IMB generic module function.

ImbShutdownSystem:

```
void ImbShutdownSystem (
   DWORD     shutdownCode,
   DWORD     delayTime,
   void *    context
   );
```

This function implements the IMB functions to be executed when the system shuts down. Currently, it sends a command to the firmware watchdog timer to configure it as appropriate to the shutdown code and delay time provided. It then sends a command to the watchdog to trigger its action. Asynchronous Messaging

One of the features of this driver is the ability to handle incoming requests or responses that have no associated outstanding outgoing requests. This feature requires that the driver periodically poll the interface to see if there are any new incoming messages available and that it deal with unrelated messages when expecting a response to a previously sent IMB request. The IMB driver provides a set of API calls related to this feature.

The polling loop is implemented in the OS specific code, but the IMB generic code provides the functions to check for and process the asynchronous messages as well as the code to read messages from the queue.

The feature is implemented by storing received SMS messages in a circular queue.

As messages come in, if they are not associated with an outstanding request, they are placed in the queue, overwriting older messages as necessary. Each message has a “sequence number”, different from the IMB sequence number, that can be used to determine if an application has seen a particular message or not.

An application reads a received message by providing the sequence number of the last message read, requesting the next one available. A timeout can be provided so that an application can wait until the next message arrives (or a timeout occurs).

Access to the queue, and its state variables, is synchronized through a semaphore (maintained by the OS specific code) to avoid problems with access by multiple threads.

The following figure demonstrates the organization of the asynchronous messaging feature.

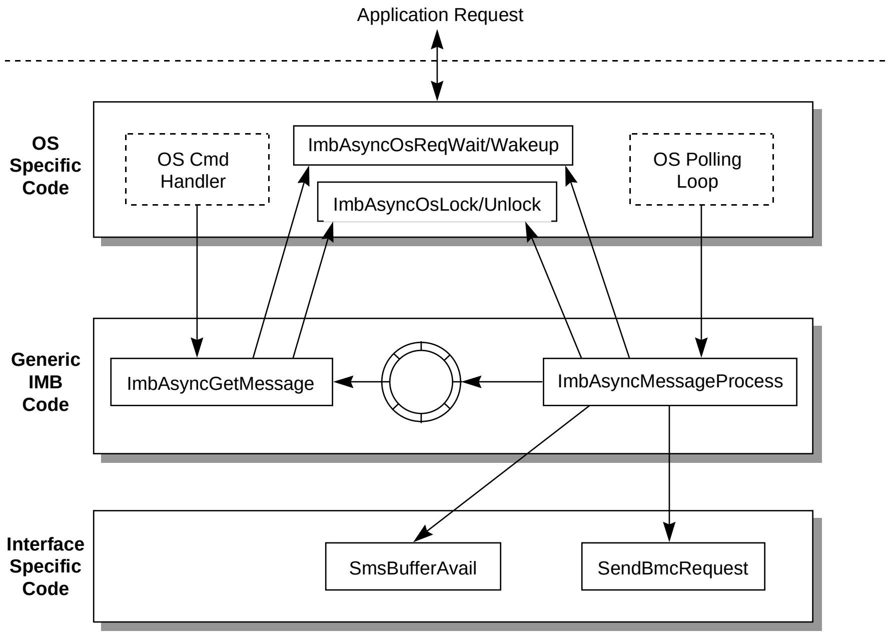

Figure 3-1: Async Messaging Organization

The following functions are exported from the generic IMB code to the OS specific layer:

- **ImbAsyncGetMessage**

```
ACCESS_STATUS ImbAsyncGetMessage (
   ImbAsyncRequest *    req,          // his request
   ImbAsyncResponse *   buf,          // data buffer
   DWORD *              length,       // IN - size of buf and OUT - size of msg
   ImbAsyncQueueInfo *  asyncQ        // ptr to queue resources
   DWORD                channelNumber // BMC channel number that received the message
                                      // 0 - IMBP channel number
                                      // 1 - EMP channel number
                                      // 2 - LAN channel number
   );
```

  The above function checks the queue for a message with a higher sequence number than that indicated in the request structure. The message with the lowest sequence number less than the supplied value will be returned. If there are none and the timeout indicated in the request structure is not zero, then the thread will block waiting until either a message comes in or the timeout has occurred.

- **ImbAsyncMessageProcess**

```
void ImbAsyncMessageProcess (
   void * context       // OS dependent handle
);
```

  The OS polling loop will call the ImbAsyncMessageProcess function periodically to see if an SMS message is waiting to be read from the BMC. If so, this function will read the message and place it in the queue. This will result in waking up any threads waiting for such a message.

4 OS Specifics

This section details the implementation of the OS specific layer including the functions that are exported to the generic IMB and interface specific layers.

The following functions are exported by the OS specific implementation to the IMB generic and Interface specific layers:

- **ImbOsStartReq**

  ```
  void ImbOsStartReq (
     DWORD  timeOut,      // in uSec units
     void * context       // OS dependent handle
     );
  ```

  The ImbOsStartReq function is called at the start of request processing. It starts timing the current request and acquires exclusive access to the interface.

- **ImbOsEndReq**

  ```
  void ImbOsEndReq (
     void * context       // OS dependent handle
     );
  ```

  After the request has succeeded, failed, or timed out, the ImbOsEndReq function is called to cancel the current request timer (if necessary) and release the interface access lock.

- **ImbReqTimedOut**

  ```
  BOOL ImbReqTimedOut (
     void *context        // OS dependent handle
     );
  ```

  This function is called periodically to determine if the current request has timed out.

- **DelayRetryFunc**

  ```
  VOID DelayRetryFunc (
     DWORD  retryCount,   // number of times called while waiting
     DWORD  delay,        // time to delay (in uSecs)
     void * context       // OS dependent handle
     );
  ```

  DelayRetryFunc is called during request processing when the driver needs to wait for a time before checking for status or changing to the next state. It uses a heuristic to determine if it will really delay or immediately return. If retryCount is less than an OS specific value, the routine will immediately return. Otherwise, the function will delay for the specified time. If retryCount is equal to FORCE_DELAY, then the function is guaranteed to delay.

- **GetNextImbSeq**

  ```
  BYTE GetNextImbSeq (
     void * context       // OS dependent handle
     );
  ```

  This function returns the current IMB sequence number for frame being built. It is used when I^2C frames are being built for requests for controllers other than the BMC.

- **ImbIf**

  ```
  ImbInterface * ImbIf (
     void *context       // OS dependent handle
     );
  ```

  A pointer to the interface function vector appropriate for this platform is returned.

- **ImbAsyncQueue**

  ```
  ImbAsyncQueueInfo * ImbAsyncQueue(
     void *context       // OS dependent handle
     );
  ```

  A pointer to the async queue structure is returned.

- **ImbAsyncOsInit**

  ```
  void ImbAsyncOsInit (
     ImbAsyncQueueInfo *info
     );
  ```

  This function initializes OS specific resources associated with Async queue management.

- **ImbAsyncOsReqWait**

  ```
  void ImbAsyncOsReqWait (
     ImbAsyncQueueInfo * info,
     DWORD               timeOut
     );
  ```

  A request for an SMS message from the Async queue can sleep waiting for a message to come in. This function implements that function. It will return either when the timeout period has elapsed or the thread has been woken up by a call to ImbAsyncOsReqWakeup().

- **ImbAsyncOsReqWakeup**

  ```
  void ImbAsyncOsReqWakeup (
     ImbAsyncQueueInfo *info
     );
  ```

  This function will wake up all threads waiting for incoming async messages.

- **ImbAsyncOsLock**

  ```
  BOOL ImbAsyncOsLock (
     ImbAsyncQueueInfo *info
     );
  ```

  Because the Async queue data structures are potentially read and modified by multiple threads, their access must be protected so that each thread sees a coherent view of the queue state. This function synchronizes access to the queue and is called before accessing it.

- **ImbAsyncOsUnlock**

  ```
  BOOL ImbAsyncOsUnLock (
     ImbAsyncQueueInfo *info
     );
  ```

  As with the previous function the ImbAsyncOsUnlock function releases exclusive access to the Async queue.

Many of these functions take a void *context argument. This pointer represents OS specific driver data that is passed back to the OS specific layer during up-calls. For Windows NT, this is the device extension structure. Windows NT

This section describes the Windows NT specific layer implementation

4.1.1 Files

- The Windows NT specific sources are divided into five files.imb_nt.c
  – driver entry points and generic support
- imb_os.h – Windows NT specific data structure and constant definitions
- imb_log.h – error/status return value definitions
- memmap.c – support for physical memory access (SM/DM BIOS)
- memif.h – data structures and constants for above feature

4.1.2 Driver Entry Points

The implementation of the Windows NT specific code is structured with the standard Windows NT driver entry points:

- DriverEntry – DriverEntry()

This function allocates and initializes driver specific resources such as:

- Dispatch semaphore
- Request semaphore
- Request timer (DPC)
- Async queue access semaphore (via ImbAsyncInit())
- Async messaging event

It also registers with the OS and creates file system namespace links.

- OpenCloseDispatch – ImbOpenCloseDispatch()
  This function is effectively a no-op, always returning success.

- DeviceControl – ImbDeviceControl()
  All requests for service come through this routine. Incoming IMB requests are queued for processing by a driver thread. Async messaging requests directly call ImbAsyncGetMessage() to get a message or wait for one to become available.

Shutdown state and memory mapping requests are handled as before.

- Shutdown – ImbShutdown()
  This entry point is called when Windows NT is shutting down and it calls the generic function ImbShutdownSystem() which sets up and triggers the BMC watchdog timer.

- Unload – ImbUnload()
  When the driver is about to be unloaded, the system calls this function. It signals the request processing thread to exit and waits for that to happen. It also releases driver held resources such as semaphores and events.

4.1.3 Driver Structure

Driver data and other resources are held in the per-driver instance device extension structure. This includes the request queue, synchronization resources (semaphores and events), and the async queue state. The queue data itself is allocated at driver initialization and pointed to by the device extension.

The Windows NT driver supports unloading and registers for shutdown notification.

As in the previous Windows NT drivers, there is a driver thread that processes the IMB requests. In this driver, the driver thread waits for new requests to be posted and also wakes up periodically to check to see if there are any asynchronous messages to be read from the SMS buffer. The generic async messaging routine ImbAsyncMessageProcess() is called to deal with this.

Async queue synchronization is implemented by a Windows NT semaphore to lock the queue (syncSem) and a Windows NT event to wait for messages (waitEvent).

The following figure demonstrates the connections between the major Windows NT specific code functions and the other layers.

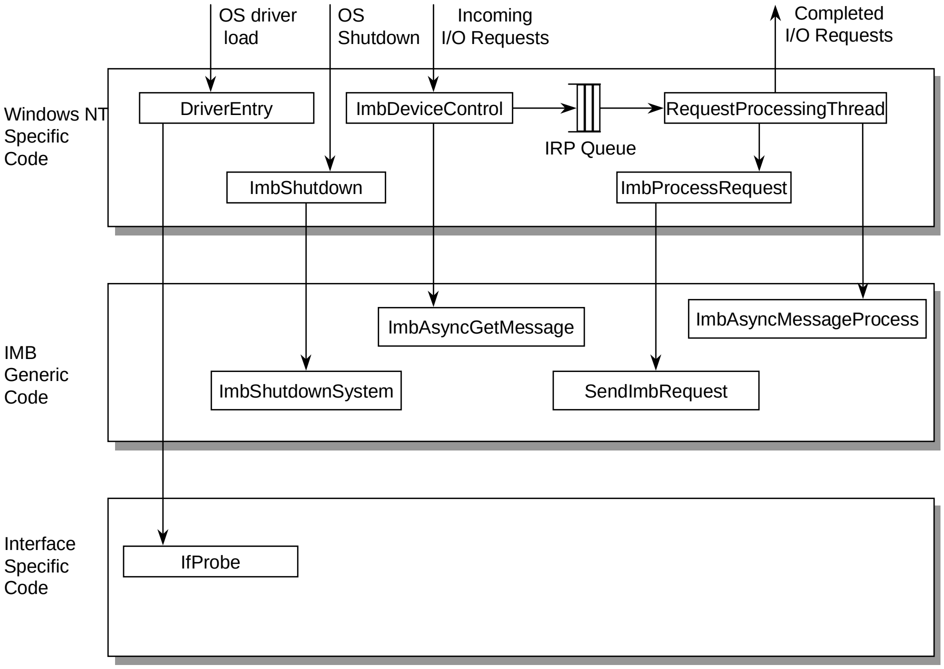

Figure 4-1: Windows NT Code Structure

4.1.4 Initialization

During initialization, the driver allocates the data structures it needs and registers with the kernel, creating the appropriate symbolic link so that applications can find it. It also calls the Interface specific probe routine for each interface type until one returns success. Once interface is identified, driver attempts to identify capabilities of underlying platform by finding IPMI version it supports. This is done based on SMBIOS specification. If driver fails to find IPMI version, it assumes default version as 1.0

4.1.5 Request processing

All requests are IOCTLs. No read or write support is provided. Requests to send IMB messages, read an async message, or map/unmap memory come in to the ImbDeviceControl entry point. IMB message request IRP structures are queued for disposition by the driver thread. All other requests are handled directly.

Once a request IRP is put on the device’s queue, the driver thread is signaled.

The driver thread is in a loop where it waits for either a signal or timeout. If either of these occur, it will wakeup and query the queue for work.

If there is work to do, it will de-queue the oldest request and pass it down to the IMB generic layer (SendImbRequest) for processing. The request will be marked completed with appropriate status when the call into the IMB generic layer returns.

The thread will then call into the IMB generic layer (ImbAsyncMessageProcess) to check to see if there is an SMS message (async message) to process, and if so process it.

After all this is over, the thread will go back to waiting for work.

5 Interface Specific Layer

An interface module provides access to the IMB interface it supports in an OS and interface independent way. The following assumptions are made about the driver.

- Each interface is through the baseboard BMC and requests to other controllers must be made through the BMC.
- The BMC supports an SMS message buffer and the commands to read it. This buffer receives IMB responses from other controllers and any other messages directed to the SMS Lun on the BMC.
- Only one IMB interface exists on a platform

Each interface-specific module exports a structure that contains pointers to three functions. The structures of all interface modules are pointed to by an array ImbInterfaces, which is walked at driver initialization time when looking for the appropriate interface module for the platform.

The driver supports the SMIC and KCS interfaces described in the IPMI specification. The functions provided are:

- **ifProbe**

  ```
  BOOL ifProbe ( void );
  ```

  The ifProbe function searches for an instance of it type of interface. It returns TRUE if the interface is present.

- **sendBmcRequest**

  ```
  IMB_STATUS sendBmcRequest (
     BmcRequest *        request,
     DWORD               reqLength,
     BmcResponse *       response,
     DWORD *             respLength,
     void *              context
     );
  ```

  This function sends the specified request to the BMC and collects its response. The response length argument *respLength* should be set to the maximum size of the response buffer on entry and will contain the actual response size on return. The value is undefined if the return status is not IMB_SUCCESS. When sending requests to other controllers than the BMC, a WriteI2C command is first sent via this interface. This function will return as soon as it has gotten the response from the BMC that it has sent the request. A second call to this function must be made with a ReadSMSBuffer command to retrieve the response from the other controller (after verifying that the response is available via the smsBufferAvail function).

- **smsBufferAvail**

  ```
  BOOL smsBufferAvail ( void );
  ```

  The smsBufferAvail function provides an indication of whether there is a message available in the BMC's SMS buffer. It returns TRUE if such a message is available.

5.1 SMIC Interface

This interface module is state machine driven and follows the IPMI SMIC interface specification.

5.2 KCS Interface

This interface module is state machine driven and follows the IPMI Keyboard Controller Style interface specification. KCS interface address is dynamically fetched during initialization.

Appendix 1: API Calls

This section describes the API calls exported by the driver.

SendTimedImbpRequest

Send a specified IPMI protocol based request on I2c bus. Retry until specified timeout.

```
ACCESS_STATUS SendTimedImbpRequest(
   IMBPREQUESTDATA  *requestDataPtr,
   DWORD            timeout,
   BYTE             *responseDataPtr,
   DWORD            *responseDataLength,
   BYTE             *completionCode
   )
```

Parameters

`requestDataPtr`	pointer to the request related data structure
`timeout`	retry timeout (in milliseconds)
`responseDataPtr`	pointer to the response data buffer. On call, a NULL pointer would indicate that no response is expected. On return this will point to response data(Framing information stripped off). The calling application should allocate sufficient space for the returned data.
`responseDataLength`	pointer to the length of the response data. On call, this parameter points to the size of the allocated buffer for response data. The driver will use this number to check for overflow condition of the received data. On return it will point to the actual number of data bytes received.
`completionCode`	Pointer to the completion code. On return it will point to the completion code returned by the firmware. For details of the completion code types refer to the IPMI documents.

Data Type Definitions

```
   typedef struct {

      unsigned char     *data;      // command body
      int               dataLength; // body size
      unsigned char     cmdType;    // IMB command
      unsigned char     rsSa;       // command destination address
      unsigned char     busType;    // not used
      unsigned char     netFn;      // IMB command class
      unsigned char     rsLun;      // subsystem on destination

   } IMBPREQUESTDATA;
```

Return Value

Returns the driver access status code.

SendTimedI2cRequest

Send a raw message on I2c bus. Retry until specified timeout.

```
ACCESS_STATUS SendTimedI2cRequest(
   I2CREQUESTDATA   *reqPtr,
   DWORD            timeout,
   BYTE             *responseDataPtr,
   DWORD            *responseDataLength
   BYTE             *completionCode
   )
```

Parameters

`reqPtr`	Pointer to the request block
`timeout`	retry timeout (in milliseconds)
`responseDataPtr`	Pointer to the response buffer. A NULL pointer would indicate that no response is expected.
`responseDataLength`	Length of the response message
`completionCode`	Pointer to the completion code returned by the firmware

Data Type Definitions

```
typedef struct {

   unsigned char *data;                      // write data
   int         dataLength;                   // write data size
   unsigned char     rsSa;                   // device address
   unsigned char     busType;                // which I2C bus
   unsigned char     numberOfBytesToRead;// how much data to return

} I2CREQUESTDATA;
```

Return Value

Returns driver access status code.

GetAsyncImbpMessage

This API call returns the Asynchronous message received by the driver on a specific BMC channel.

```
ACCESS_STATUS
GetAsyncImbpMessage (
   BYTE *    msgPtr,         // request info and data
   DWORD *   msgLen,         // IN – length of buffer,
                             // OUT - msg len
   DWORD     timeOut,        // how long to wait for the message
   DWORD *   seqNo,          // IN/OUT - pointer to previously returned sequence
                             // number To get the first message start with 0
   DWORD     channelNumber   // IN - number of the BMC channel that received the message
                             // 0 –IPMB channel
   )
```

For the asynchronous messages received on the IMBP channel, the ‘msgPtr’ points to the message data that is structured as follows:

```
//
// This is the generic IMB packet format, the final checksum can't be
// represented in this structure and will show up as the last data byte
//
typedef struct {

      BYTE rsSa;
      BYTE nfLn;
      BYTE cSum1;
      BYTE rqSa;
      BYTE seqLn;
      BYTE cmd;
      BYTE data[1];

} ImbPacket;
```

Return Value

Returns driver access status code.

MapPhysicalMemory

This api maps a given range of physical memory into the address space of the calling process. The driver will handle only one mapping at any time. It is the responsibility of the calling process to unmap the memory before a next mapping call can be processed. An error status code will be returned to the calling process if an active mapping call already exists with the driver.

```
ACCESS_STATUS MapPhysicalMemory  (
   DWORD startAddress,
   DWORD length,
   DWORD *virtualAddress
   )
```

Parameters

`startAddress`	starting address of the physical memory to be mapped
`length`	length of the physical memory to be mapped
`virtualAddress`	pointer to the virtual address to be mapped. On return, it will point to the virtual address that is mapped to the physical memory.

Return Value

Returns driver access status code.

UnmapPhysicalMemory

This API call un-maps a given range of physical memory that was mapped by the driver into the calling processe’s address space. Calling process should use this call to unmap the physical memory that was mapped using ‘MapPhysicalMemory’

```
ACCESS_STATUS MapPhysicalMemory (
   DWORD virtualAddress
   )
```

Parameters

`virtualAddress`	The mapped virtual address. This was returned by ‘MapPhysicalMemory’.

Return Value

Returns driver access status code.

Driver Access Structure Type Definition

```
typedef enum {

   ACCESS_OK,
   ACCESS_ERROR,
   ACCESS_OUT_OF_RANGE,
   ACCESS_END_OF_DATA,
   ACCESS_UNSUPPORTED,
   ACCESS_INVALID_TRANSACTION,
   ACCESS_TIMED_OUT

   } ACCESS_STATUS;
```

Appendix 4: Glossary

Term	**Definition**
BMC	Baseboard Management Controller
FPC	Front Panel Controller
ICMB	Inter Chassis Management Bus
IPMI	Intelligent Platform Management Interface
IPMB	Intelligent Platform management Bus
I^2C bus	Inter Integrated Circuit bus. A 2-wire bi-directional serial bus developed by Philips for an independent communications path between embedded ICs on printed circuit boards and subsystems. The I^2C bus is used on many servers for system management and diagnostics.
KCS	Keyboard Controller Style Interface
SMIC	Server Management Interface Chip - an ASIC that provides a parallel I/O mapped interface between the “ISA” bus and the BMC.