MontaVista IPMI Driver
Theory of Operation

Corey Minyard <minyard@mvista.com>

June 5, 2002

Abstract

This document describes the design and function of the IPMI driver designed by MontaVista Software. It shows how the driver is divided up, the individual pieces of the driver and what they do, and how the pieces fit together. Note that this document assumes some familiarity with IPMI.

1 Basic Design

Figure 1 on the following page shows the basic pieces of the IPMI driver. The message handler (ipmi_msghandler.c) provides the central function for the IPMI driver. It provides an in-kernel user interface so other functions in the kernel can send IPMI requests, receive the responses, receive events, and receive incoming IPMI commands.

An SMI (System Management Interface) driver such as the KCS driver can bind into the message handler as a lower layer. The SMI driver provides the low-level interface for talking to a BMC. In the future, an SMIC, BT, or other driver could easily be written to bind in to the message handler. Multiple KCS drivers may bind into the message handler; the message handler provides separate access to each using an interface number.

Several modules use the IPMI interface provided by the message handler.

2 Why a New Interface?

Several interfaces already exist for IPMI, why not use one of them them? Unfortunately, they all have problems that would prevent them from being very useful or prevent them from being included into the mainstream Linux kernel. An evaluation has already been done, the team chose Radisys driver as the most functional driver, so this discussion will only focus on that driver (indeed, the Radisys driver was much better than the others).

The Radisys driver has a fundamentally good structure. However, as they say, "The devil is in the details."

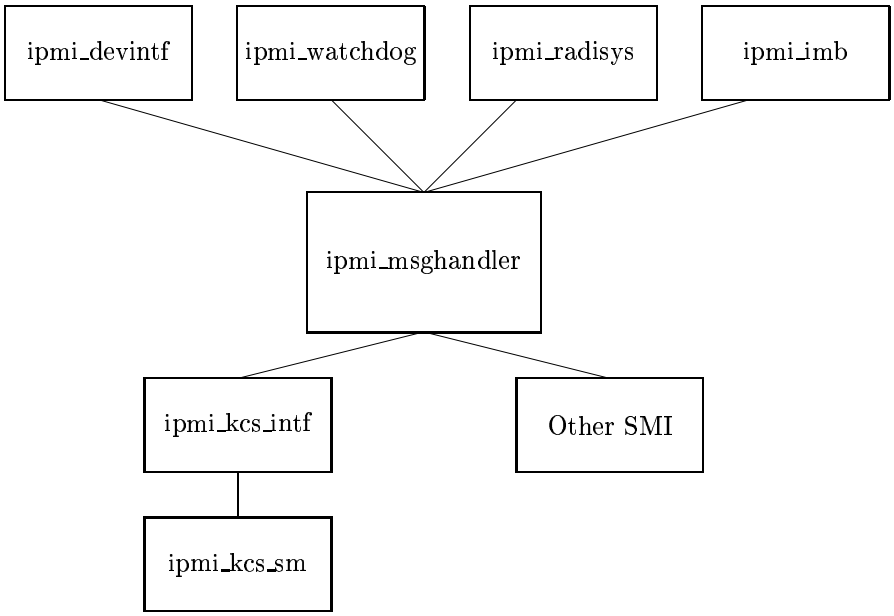

Figure 1: Parts of the MontaVista IPMI driver

- The driver has significant name space issues. You can't create types named ULONG and expect the Linux kernel maintainers to be happy about it. You can't create an include file named "typedefs.h" and put it in the main kernel include directory.
- The IOCTL interface structure has kernel-specific stuff in it. This is not allowed in user interfaces; the kernel code should completely hide it's internal data structures.
- It's so far off the Linux coding standards that there's no way it would be accepted.
- It's tightly bound to IPMB with no provisions for different interface types (the address is hard-coded as an IPMB slave address). If future IPMI busses come out (say, IPMI over 802.3) then the driver would require interface changes to handle this and all old code would be broken, or you would end up with a cumbersome interface.
- They data structures are not coded with forward-compatability in mind (the data structures cannot be extended if the maximum message size gets bigger, it will break old executables).
- There is no way to do select() on the interface to wait for data from the driver. You have to poll the interfaces, which is inefficient.
- The interfaces are timing-sensitive. The driver requires that you register to receive your responses with the response information; it holds this registration for a limited amount of time. On a busy real-time system the responses could be lost since the IPMI user would undoubtedly be lower priority than the real-time portions of the system.
- Having to register to receive your own responses is cumbersome, and can result messages going to the wrong user in some instances.
- The driver can only have one user that receive incoming commands. The applications running on a system could not be split by function.

Because of these issues, I decided that a new interface would be the best option.

3 The Message Handler

Since the message handler handles the central function of the IPMI stack, it gets first billing. The message handler handles all messages in the IPMI stack, it acts as the central router for messages. The elements "above" the message handler use the "user" interface to the message handler. The elements "below" the message handler use the "SMI" interface.

The message handler could easily be ported or made portable to multiple operating systems, it does not rely on any low-level functions besides locks and memory allocation.

3.1 The User Interface

Internal kernel users bind into the message handler for an interface and are given an opaque user data structure to work with. The user data structure only applies to one interface (sort of like a minor device only applies to one physical device). The user may then send messages to various channels on the BMC, either to the BMC itself or on the IPMB busses (or other channels) hooked to the BMC. The user may register to receive incoming events from the event message buffer and register to receive incoming IPMI commands from other devices. Appendix A on page 7 describes this interface.

All incoming messages are delivered to the user via a callback mechanism. The user supplies the callback when binding to the message handler. All incoming messages come in a common data structure, a field in the structure identifies the messages as a response, and incoming command, or an IPMI event.

In all messages, the message handler strips the header information from the message. It puts the addressing information in the address and just the message contents in the data. This way, the user doesn't have to understand the header information to get the message contents.

3.1.1 Addressing

The driver can handle different types of addresses. Currently it handles the "unused" address type (for talking directly to the BMC) and IPMB addresses. The user supplies the address as a pointer (much like a socket interface). In the future, new types of addresses can be easily added in the future.

The IPMI interface closely resembles a network[1], it might be better to provide a socket interface to IPMI in the future.

3.1.2 Sending Commands

The user may send commands to the BMC or other IPMI devices. The user provides an ID in the message, when the response comes back the user will be sent the response automatically. The message handler assigns the sequence numbers and handles formatting the message header with the proper addressing contents. This way, the user does not have to register to receive it's incoming responses, if they send the command, they will get the response.

3.1.3 Incoming Events

Incoming IPMI events are fetched by the message handler automatically and delivered to users that have called the ipmi_set_gets_events() function and set their event receiver value to true. Once the user sets this to true, they will receive all incoming IPMI events.

The message handlers asks the SMI interfaces to check for events in their event message buffers. It does this once a second. The SMI interfaces will wait

[1] Well, maybe it *is* a network.

until they are idle and do the request. The SMI interfaces also may detect that the event message buffer has filled using the ATN mechanism; if this happens they will automatically empty the event message buffer.

Since this may happen before an IPMI user has attached and asked for events, the message handler will queue up to 25 events and deliver them when the first user asks for events.

3.1.4 Incoming Commands

The user may register against a specific NetFN and command; if it does it will receive those commands. Only one user may be registered against a specific NetFN/command, but different users may be simultaneously registered against different ones.

If no user has registered against a command, the message handler will automatically return the proper error response.

3.2 The SMI Interface

In order to abstract the message handler from the low-level details of taking to the BMC, it provides a generic interface for low-level drivers. The include file in appendix C on page 17 describes this interface.

3.3 Panic Handling

The panic handler is the only real Linux-specific portion of the message handlers. It ties into the panic notifier chain and delivers a panic event to the BMC's event buffer so the panic information can be held between reboots. Unfortunately it only has a few bytes to store panic information (a bigger event would be VERY useful). But it emulates the function that already exists to do this.

4 Drivers

This section describes the various SMI drivers. Since we only have a KCS driver, it's not much of a section, but maybe it will get bigger.

4.1 The KCS Driver

The KCS driver consists of two pieces, the state machine (ipmi_kcs_sm.c) and the interface (ipmi_kcs_intf.c). The state machine provides a generic KCS state machine that could be easily ported to other operating systems. The interface interfaces the state machine to the operating system and to the message handler.

5 IPMI Users

Several users of the IPMI interface have already been written, this section describes them.

5.1 The MontaVista IPMI Driver Interface

The MontaVista IPMI driver interface (ipmi_devintf.c) provides a userland interface to the IPMI message handler. It provides all the functions of the message handler via IOCTLs, except that incoming messages are queued and the user may wait for them with select() or poll() and fetch them with an IOCTL call.

5.2 The Watchdog Timer

The watchdog timer (ipmi_watchdog.c) uses the IPMI message handler to access the IPMI watchdog function. It provides the Linux-standard watchdog device interface.

5.3 The Radisys Interface Emulator

The Radisys emulator (ipmi_radisys.c) emulates the Radisys driver on top of the message handler.

5.4 The IMB Interface Emulator

The IMB emulator (ipmi_imb.c) emulates Intel's IMB driver on top of the message handler.

A The main IPMI user interface include file

```
/*
 * ipmi.h
 *
 * MontaVista IPMI interface
 *
 * Author: MontaVista Software, Inc.
 *         Corey Minyard <minyard@mvista.com>
 *         source@mvista.com
 *
 * Copyright 2002 MontaVista Software Inc.
 *
 *  This program is free software; you can redistribute it and/or modify it
 *  under the terms of the GNU General Public License as published by the
 *  Free Software Foundation; either version 2 of the License, or (at your
 *  option) any later version.
 *
 *
 *  THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED
 *  WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 *  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 *  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 *  OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 *  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
 *  TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 *  USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with this program; if not, write to the Free Software Foundation, Inc.,
 *  675 Mass Ave, Cambridge, MA 02139, USA.
 */

#ifndef __LINUX_IPMI_H
#define __LINUX_IPMI_H

#include <linux/ipmi_msgdefs.h>

/*
 * This file describes an interface to an IPMI driver.  You have to
 * have a fairly good understanding of IPMI to use this, so go read
 * the specs first before actually trying to do anything.
 *
 * With that said, this driver provides a multi-user interface to the
```

```
 * IPMI driver, and it allows multiple IPMI physical interfaces below
 * the driver.  The physical interfaces bind as a lower layer on the
 * driver.  They appear as interfaces to the application using this
 * interface.
 *
 * Multi-user means that multiple applications may use the driver,
 * send commands, receive responses, etc.  The driver keeps track of
 * commands the user sends and tracks the responses.  The responses
 * will go back to the application that send the command.  If the
 * response doesn't come back in time, the driver will return a
 * timeout error response to the application.  Asynchronous events
 * from the BMC event queue will go to all users bound to the driver.
 * The incoming event queue in the BMC will automatically be flushed
 * if it becomes full and it is queried once a second to see if
 * anything is in it.  Incoming commands to the driver will get
 * delivered as commands.
 *
 * This driver provides two main interfaces: one for in-kernel
 * applications and another for userland applications.  The
 * capabilities are basically the same for both interface, although
 * the interfaces are somewhat different.  The stuff in the
 * #ifdef KERNEL below is the in-kernel interface.  The userland
 * interface is defined later in the file.  */



/* This is an overlay for all the address types, so it's easy to
   determine the actual address type.  This is kind of like addresses
   work for sockets. */
#define IPMI_MAX_ADDR_SIZE 32
typedef struct ipmi_addr_s
{
        int   addr_type;
        short channel;
        char  data[IPMI_MAX_ADDR_SIZE];
} ipmi_addr_t;

/* When the address is not used, the type will be set to this value.
   The channel is the BMC's channel number for the channel (usually
   0), or IPMC_BMC_CHANNEL if communicating directly with the BMC. */
#define IPMI_UNUSED_ADDR_TYPE   0
typedef struct ipmi_unused_addr_s
{
        int   addr_type;
        short channel;
} ipmi_unused_addr_t;
```

```
/* An IPMB Address. */
#define IPMI_IPMB_ADDR_TYPE     1
typedef struct ipmi_ipmb_addr_s
{
        int           addr_type;
        short         channel;
        unsigned char slave_addr;
        unsigned char lun;
} ipmi_ipmb_addr_t;


/* Channel for talking directly with the BMC.  When using this
   channel, This is for the unused address type only.  FIXME - is this
   right, or should we use -1? */
#define IPMI_BMC_CHANNEL  0xf
#define IPMI_NUM_CHANNELS 0x10


/* A raw IPMI message without any addressing.  This covers both
   commands and responses.  The completion code is always the first
   byte of data in the response (as the spec shows the messages laid
   out). */
typedef struct ipmi_msg_s
{
        unsigned char netfn;
        unsigned char lun;
        unsigned char cmd;
        unsigned char *data;
        int           data_len;
} ipmi_msg_t;

/*
 * Various defines that are useful for IPMI applications.
 */
#define IPMI_INVALID_CMD_COMPLETION_CODE        0xC1
#define IPMI_TIMEOUT_COMPLETION_CODE            0xC3
#define IPMI_UNKNOWN_ERR_COMPLETION_CODE        0xff


/* Response types for messages coming from the receive interface.
   This is used for the receive in-kernel interface and in the receive
   IOCTL. */
#define IPMI_RESPONSE_RECV_TYPE         1 /* A response to a command */
#define IPMI_ASYNC_EVENT_RECV_TYPE      2 /* Something from the event queue */
#define IPMI_CMD_RECV_TYPE              3 /* A command from somewhere else */
```

```
#ifdef __KERNEL__

/*
 * The in-kernel interface.
 */
#include <linux/list.h>

/* Opaque type for a IPMI message user.  One of these is needed to
   send and receive messages. */
typedef struct ipmi_user_s ipmi_user_t;

/* Stuff coming from the recieve interface comes as one of these.
   They are allocated, the receiver must free them with
   ipmi_free_recv_msg() when done with the message.  The link is not
   used after the message is delivered, so the upper layer may use the
   link to build a linked list, if it likes. */
typedef struct ipmi_recv_msg_s
{
        struct list_head link;

        int         recv_type;
        ipmi_user_t *user;
        ipmi_addr_t addr;
        long        msgid;
        ipmi_msg_t  msg;

        /* Place-holder for the data, don't make any assumptions about
           the size or existance of this, since it may change. */
        unsigned char   msg_data[IPMI_MAX_MSG_LENGTH];
} ipmi_recv_msg_t;

/* Free the receive message. */
void ipmi_free_recv_msg(ipmi_recv_msg_t *msg);
ipmi_recv_msg_t *ipmi_alloc_recv_msg(void);

/* Routine type to call when a message needs to be routed to the upper
   layer.  This will be called with some locks held, the only IPMI
   routines that can be called are ipmi_request and the alloc/free
   operations. */
typedef void (*ipmi_recv_hndl_t)(ipmi_recv_msg_t *msg,
                                 void            *handler_data);

/* Create a new user of the IPMI layer on the given interface number. */
```

```
int ipmi_create_user(unsigned int     if_num,
                     ipmi_recv_hndl_t handler,
                     void             *handler_data,
                     ipmi_user_t      **user);

/* Destroy the given user of the IPMI layer. */
int ipmi_destroy_user(ipmi_user_t *user);

/* Send a command request from the given user.  The address is the
   proper address for the channel type.  If this is a command, then
   the message response comes back, the receive handler for this user
   will be called with the given msgid value in the recv msg.  If this
   is a response to a command, then the msgid will be used as the
   sequence number for the response (truncated if necessary), so when
   sending a response you should use the sequence number you received
   in the msgid field of the received command. */
int ipmi_request(ipmi_user_t *user,
                 ipmi_addr_t *addr,
                 long        msgid,
                 ipmi_msg_t  *msg);

/* When commands come in to the SMS, the user can register to receive
   them.  Only one user can be listening on a specific netfn/cmd pair
   at a time, you will get an EBUSY error if the command is already
   registered.  If a command is received that does not have a user
   registered, the driver will automatically return the proper
   error. */
int ipmi_register_for_cmd(ipmi_user_t   *user,
                          unsigned char netfn,
                          unsigned char cmd);
int ipmi_unregister_for_cmd(ipmi_user_t   *user,
                            unsigned char netfn,
                            unsigned char cmd);

/* When the user is created, it will not receive IPMI events by default.
   The user must set this to TRUE to get incoming events.  The first
   user that sets this to TRUE will receive all events that have been
   queued while no one was waiting for events. */
int ipmi_set_gets_events(ipmi_user_t *user, int val);

/* Register the given user to handle all received IPMI commands.  This
   will fail if anyone is registered as a command receiver or if
   another is already registered to receive all commands.  NOTE THAT
   THIS IS FOR EMULATION USERS ONLY, DO NOT USER THIS FOR NORMAL
   STUFF. */
int ipmi_register_all_cmd_rcvr(ipmi_user_t *user);
```

```
int ipmi_unregister_all_cmd_rcvr(ipmi_user_t *user);


/* The following are various helper functions for dealing with IPMI
   addresses. */

/* Return the maximum length of an IPMI address given it's type. */
int ipmi_addr_length(int addr_type);

/* Validate that the given IPMI address is valid. */
int ipmi_validate_addr(ipmi_addr_t *addr, int len);

/* Return 1 if the given addresses are equal, 0 if not. */
int ipmi_addr_equal(ipmi_addr_t *addr1, ipmi_addr_t *addr2);

#endif /* __KERNEL__ */


/*
 * The userland interface
 */

/* The userland interface for the IPMI driver is a standard character
   device, with each instance of an interface registered as a minor
   number under the major character device.

   The read and write calls do not work, to get messages in and out
   requires ioctl calls because of the complexity of the data.  select
   and poll do work, so you can wait for input using the file
   descriptor, you just can use read to get it.

   In general, you send a command down to the interface and receive
   responses back.  You can use the msgid value to correlate commands
   and responses, the driver will take care of figuring out which
   incoming messages are for which command and find the proper msgid
   value to report.  You will only receive reponses for commands you
   send.  Asynchronous events, however, go to all open users, so you
   must be ready to handle these (or ignore them if you don't care).

   The address type depends upon the channel type.  When talking
   directly to the BMC (IPMC_BMC_CHANNEL), the address is ignored
   (IPMI_UNUSED_ADDR_TYPE).  When talking to an IPMB channel, you must
   supply a valid IPMB address with the addr_type set properly.

   When talking to normal channels, the driver takes care of the
   details of formatting and sending messages on that channel.  You do
```

```
   not, for instance, have to format a send command, you just send
   whatever command you want to the channel, the driver will create
   the send command, automatically issue receive command and get even
   commands, and pass those up to the proper user.

*/


/* The magic IOCTL value for this interface. */
#define IPMI_IOC_MAGIC 'i'


/* Messages sent to the interface are this format. */
typedef struct ipmi_req_s
{
        unsigned char *addr; /* Address to send the message to. */
        int           addr_len;

        long    msgid; /* The sequence number for the message.  This
                          exact value will be reported back in the
                          response to this request if it is a command.
                          If it is a response, this will be used as
                          the sequence value for the response.  */

        ipmi_msg_t msg;
} ipmi_req_t;
#define IPMICTL_SEND_COMMAND            _IOR(IPMI_IOC_MAGIC, 13,        \
                                             ipmi_req_t)

/* Messages received from the interface are this format. */
typedef struct ipmi_recv_s
{
        int     recv_type; /* Is this a command, response or an
                              asyncronous event. */

        unsigned char *addr;    /* Address the message was from is put
                                   here.  The caller must supply the
                                   memory. */
        int           addr_len; /* The size of the address buffer.
                                   The caller supplies the full buffer
                                   length, this value is updated to
                                   the actual message length when the
                                   message is received. */

        long    msgid; /* The sequence number specified in the request
                          if this is a response.  If this is a command,
```

```
                        this will be the sequence number from the
                        command. */

        ipmi_msg_t msg; /* The data field must point to a buffer.  The
                           data_size field must be set to the size of
                           the message buffer.  The caller supplies
                           the full buffer length, this value is
                           updated to the actual message length when
                           the message is received. */
} ipmi_recv_t;
#define IPMICTL_RECEIVE_MSG             _IOWR(IPMI_IOC_MAGIC, 12,       \
                                              ipmi_recv_t)

/* Register to get commands from other entities on this interface. */
typedef struct ipmi_cmdspec_s
{
        unsigned char netfn;
        unsigned char cmd;
} ipmi_cmdspec_t;
#define IPMICTL_REGISTER_FOR_CMD        _IOR(IPMI_IOC_MAGIC, 14,        \
                                             ipmi_cmdspec_t)
#define IPMICTL_UNREGISTER_FOR_CMD      _IOR(IPMI_IOC_MAGIC, 15,        \
                                             ipmi_cmdspec_t)

#define IPMICTL_SET_GETS_EVENTS_CMD     _IOR(IPMI_IOC_MAGIC, 16, int)

#endif /* __LINUX_IPMI_H */
```

B Include file for various IPMI message definitions

```
/*
 * ipmi_smi.h
 *
 * MontaVista IPMI system management interface
 *
 * Author: MontaVista Software, Inc.
 *         Corey Minyard <minyard@mvista.com>
 *         source@mvista.com
 *
 * Copyright 2002 MontaVista Software Inc.
 *
 *  This program is free software; you can redistribute it and/or modify it
 *  under the terms of the GNU General Public License as published by the
 *  Free Software Foundation; either version 2 of the License, or (at your
 *  option) any later version.
 *
 *
 *  THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED
 *  WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 *  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 *  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 *  OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 *  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
 *  TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 *  USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with this program; if not, write to the Free Software Foundation, Inc.,
 *  675 Mass Ave, Cambridge, MA 02139, USA.
 */

#ifndef __LINUX_IPMI_MSGDEFS_H
#define __LINUX_IPMI_MSGDEFS_H

/* Various definitions for IPMI messages used by almost everything in
   the IPMI stack. */

#define IPMI_NETFN_APP_REQUEST  0x06
#define IPMI_NETFN_APP_RESPONSE 0x07
```

```
#define IPMI_BMC_SLAVE_ADDR     0x20

#define IPMI_GET_MSG_FLAGS_CMD          0x31
#define IPMI_SEND_MSG_CMD               0x34
#define IPMI_GET_MSG_CMD                0x33

#define IPMI_SET_BMC_GLOBAL_ENABLES_CMD 0x2e
#define IPMI_GET_BMC_GLOBAL_ENABLES_CMD 0x2f
#define IPMI_READ_EVENT_MSG_BUFFER_CMD  0x35

#define IPMI_MAX_MSG_LENGTH     40

#endif /* __LINUX_IPMI_MSGDEFS_H */
```

C The SMI include file

```
/*
 * ipmi_smi.h
 *
 * MontaVista IPMI system management interface
 *
 * Author: MontaVista Software, Inc.
 *         Corey Minyard <minyard@mvista.com>
 *         source@mvista.com
 *
 * Copyright 2002 MontaVista Software Inc.
 *
 *  This program is free software; you can redistribute it and/or modify it
 *  under the terms of the GNU General Public License as published by the
 *  Free Software Foundation; either version 2 of the License, or (at your
 *  option) any later version.
 *
 *
 *  THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED
 *  WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 *  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 *  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 *  OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 *  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
 *  TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 *  USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with this program; if not, write to the Free Software Foundation, Inc.,
 *  675 Mass Ave, Cambridge, MA 02139, USA.
 */

#ifndef __LINUX_IPMI_SMI_H
#define __LINUX_IPMI_SMI_H

#include <linux/ipmi_msgdefs.h>

/* This files describes the interface for IPMI system management interface
   drivers to bind into the IPMI message handler. */

/* Structure for the low-level drivers. */
typedef struct ipmi_smi_s ipmi_smi_t;
```

```
/* Messages to/from the lower layer.  The smi interface will take one
   of these to send. After the send has occurred and a response has
   been received, it will report this same data structure back up to
   the upper layer.  If an error occurs, it should fill in the
   response with an error code in the completion code location. When
   asyncronous data is received, one of these is allocated, the
   data_size is set to zero and the response holds the data from the
   get message or get event command that the interface initiated.
   Note that it is the interfaces responsibility to detect
   asynchronous data and messages and request them from the
   interface. */
typedef struct ipmi_smi_msg_s
{
        struct list_head link;

        long    msgid;
        void    *user_data;

        int           data_size;
        unsigned char data[IPMI_MAX_MSG_LENGTH];

        int           rsp_size;
        unsigned char rsp[IPMI_MAX_MSG_LENGTH];
} ipmi_smi_msg_t;

typedef struct ipmi_smi_handlers_s
{
        /* Called to enqueue an SMI message to be sent.  This
           operation is not allowed to fail.  If an error occurs, it
           should report back the error in a received message.  It
           may do this in the current call context, since no write
           locks are held when this is run. */
        void (*sender)(void           *send_info,
                       ipmi_smi_msg_t *msg);

        /* Called by the upper layer to request that we try to get
           events from the BMC we are attached to. */
        void (*request_events)(void *send_info);

        /* Called when someone is using the interface, so the module can
           adjust it's use count. */
        void (*new_user)(void *send_info);

        /* Called when someone is no longer using the interface, so the
           module can adjust it's use count. */
        void (*user_left)(void *send_info);
```

```
        /* Called when the interface should go into "run to
           completion" mode.  If this call sets the value to true, the
           interface should make sure that all messages are flushed
           out and that none are pending, and any new requests are run
           to completion immediately. */
        void (*set_run_to_completion)(void *send_info, int run_to_completion);
} ipmi_smi_handlers_t;

/* Add a low-level interface to the IPMI driver. */
int ipmi_register_smi(ipmi_smi_handlers_t *handlers,
                      void                *send_info,
                      ipmi_smi_t          **intf);

/* Remove a low-level interface from the IPMI driver.  This will return an
   error if the interface is still in use by a user. */
int ipmi_unregister_smi(ipmi_smi_t *intf);

/* The lower layer reports received messages through this interface.
   The data_size should be zero if this is an asyncronous message.  If
   the lower layer gets an error sending a message, it should format
   an error response in the message response. */
void ipmi_smi_msg_received(ipmi_smi_t     *intf,
                           ipmi_smi_msg_t *msg);

ipmi_smi_msg_t *ipmi_alloc_smi_msg(void);
void ipmi_free_smi_msg(ipmi_smi_msg_t *msg);

#endif /* __LINUX_IPMI_SMI_H */
```