The Linux IPMI Driver

Corey Minyard
<minyard@mvista.com>
<minyard@acm.org>

The Intelligent Platform Management Interface, or IPMI, is a standard for controlling intelligent devices that monitor a system. It provides for dynamic discovery of sensors in the system and the ability to monitor the sensors and be informed when the sensor's values change or go outside certain boundaries. It also has a standardized database for field-replaceable units (FRUs) and a watchdog timer.

To use this, you need an interface to an IPMI controller in your system (called a Baseboard Management Controller, or BMC) and management software that can use the IPMI system.

This document describes how to use the IPMI driver for Linux. If you are not familiar with IPMI itself, see the web site at http://www.intel.com/design/servers/ipmi/index.htm. IPMI is a big subject and I can't cover it all here!

Configuration

The Linux IPMI driver is modular, which means you have to pick several things to have it work right depending on your hardware. Most of these are available in the 'Character Devices' menu then the IPMI menu.

No matter what, you must pick 'IPMI top-level message handler' to use IPMI. What you do beyond that depends on your needs and hardware.

The message handler does not provide any user-level interfaces. Kernel code (like the watchdog) can still use it. If you need access from userland, you need to select 'Device interface for IPMI' if you want access through a device driver.

The driver interface depends on your hardware. If your system properly provides the SMBIOS info for IPMI, the driver will detect it and just work. If you have a board with a standard interface (These will generally be either "KCS", "SMIC", or "BT", consult your hardware manual), choose the 'IPMI SI handler' option. A driver also exists for direct I2C access to the IPMI management controller. Some boards support this, but it is unknown if it will work on every board. For this, choose 'IPMI SMBus handler', but be ready to try to do some figuring to see if it will work on your system if the SMBIOS/APCI information is wrong or not present. It is fairly safe to have both these enabled and let the drivers auto-detect what is present.

You should generally enable ACPI on your system, as systems with IPMI can have ACPI tables describing them.

If you have a standard interface and the board manufacturer has done their job correctly, the IPMI controller should be automatically detected (via ACPI or SMBIOS tables) and should just work. Sadly, many boards do not have this information. The driver attempts standard defaults, but they may not work. If you fall into this situation, you need to read the section below named 'The SI Driver' or "The SMBus Driver" on how to hand-configure your system.

IPMI defines a standard watchdog timer. You can enable this with the 'IPMI Watchdog Timer' config option. If you compile the driver into the kernel, then via a kernel command-line option you can have the watchdog timer start as soon as it initializes. It also have a lot of other options, see the 'Watchdog' section below for more details. Note that you can also have the watchdog continue to run if it is closed (by default it is disabled on close). Go into the 'Watchdog Cards' menu, enable 'Watchdog Timer Support', and enable the option 'Disable watchdog shutdown on close'.

IPMI systems can often be powered off using IPMI commands. Select 'IPMI Poweroff' to do this. The driver will auto-detect if the system can be powered off by IPMI. It is safe to enable this even if your system doesn't support this option. This works on ATCA systems, the Radisys CPI1 card, and any IPMI system that supports standard chassis management commands.

If you want the driver to put an event into the event log on a panic, enable the 'Generate a panic event to all BMCs on a panic' option. If you want the whole panic string put into the event log using OEM events, enable the 'Generate OEM events containing the panic string' option.

Basic Design

The Linux IPMI driver is designed to be very modular and flexible, you only need to take the pieces you need and you can use it in many different ways. Because of that, it's broken into many chunks of code. These chunks (by module name) are:

ipmi_msghandler - This is the central piece of software for the IPMI system. It handles all messages, message timing, and responses. The IPMI users tie into this, and the IPMI physical interfaces (called System Management Interfaces, or SMIs) also tie in here. This provides the kernelland interface for IPMI, but does not provide an interface for use by application processes.

ipmi_devintf - This provides a userland IOCTL interface for the IPMI driver, each open file for this device ties in to the message handler as an IPMI user.

ipmi_si - A driver for various system interfaces. This supports KCS, SMIC, and BT interfaces. Unless you have an SMBus interface or your own custom interface, you probably need to use this.

ipmi_smb - A driver for accessing BMCs on the SMBus. It uses the I2C kernel driver's SMBus interfaces to send and receive IPMI messages over the SMBus.

ipmi_watchdog - IPMI requires systems to have a very capable watchdog
timer. This driver implements the standard Linux watchdog timer
interface on top of the IPMI message handler.

ipmi_poweroff - Some systems support the ability to be turned off via
IPMI commands.

These are all individually selectable via configuration options.

Note that the KCS-only interface has been removed. The af_ipmi driver
is no longer supported and has been removed because it was impossible
to do 32 bit emulation on 64-bit kernels with it.

Much documentation for the interface is in the include files. The
IPMI include files are:

net/af_ipmi.h - Contains the socket interface.

linux/ipmi.h - Contains the user interface and IOCTL interface for IPMI.

linux/ipmi_smi.h - Contains the interface for system management interfaces
(things that interface to IPMI controllers) to use.

linux/ipmi_msgdefs.h - General definitions for base IPMI messaging.

Addressing

The IPMI addressing works much like IP addresses, you have an overlay
to handle the different address types. The overlay is:

```
  struct ipmi_addr
  {
	int   addr_type;
	short channel;
	char  data[IPMI_MAX_ADDR_SIZE];
  };
```

The addr_type determines what the address really is. The driver
currently understands two different types of addresses.

"System Interface" addresses are defined as:

```
  struct ipmi_system_interface_addr
  {
	int   addr_type;
	short channel;
  };
```

and the type is IPMI_SYSTEM_INTERFACE_ADDR_TYPE. This is used for talking
straight to the BMC on the current card. The channel must be
IPMI_BMC_CHANNEL.

Messages that are destined to go out on the IPMB bus use the

IPMI_IPMB_ADDR_TYPE address type. The format is

```
  struct ipmi_ipmb_addr
  {
        int           addr_type;
        short         channel;
        unsigned char slave_addr;
        unsigned char lun;
  };
```

The "channel" here is generally zero, but some devices support more than one channel, it corresponds to the channel as defined in the IPMI spec.

Messages

Messages are defined as:

```
struct ipmi_msg
{
        unsigned char netfn;
        unsigned char lun;
        unsigned char cmd;
        unsigned char *data;
        int           data_len;
};
```

The driver takes care of adding/stripping the header information. The data portion is just the data to be send (do NOT put addressing info here) or the response. Note that the completion code of a response is the first item in "data", it is not stripped out because that is how all the messages are defined in the spec (and thus makes counting the offsets a little easier :-).

When using the IOCTL interface from userland, you must provide a block of data for "data", fill it, and set data_len to the length of the block of data, even when receiving messages. Otherwise the driver will have no place to put the message.

Messages coming up from the message handler in kernelland will come in as:

```
  struct ipmi_recv_msg
  {
        struct list_head link;

        /* The type of message as defined in the "Receive Types"
           defines above. */
        int         recv_type;

        ipmi_user_t      *user;
        struct ipmi_addr addr;
        long             msgid;
        struct ipmi_msg  msg;
        /* Call this when done with the message.  It will presumably free
           the message and do any other necessary cleanup. */
        void (*done)(struct ipmi_recv_msg *msg);

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

You should look at the receive type and handle the message appropriately.

The Upper Layer Interface (Message Handler)

The upper layer of the interface provides the users with a consistent view of the IPMI interfaces. It allows multiple SMI interfaces to be addressed (because some boards actually have multiple BMCs on them) and the user should not have to care what type of SMI is below them.

Creating the User

To user the message handler, you must first create a user using ipmi_create_user. The interface number specifies which SMI you want to connect to, and you must supply callback functions to be called when data comes in. The callback function can run at interrupt level, so be careful using the callbacks. This also allows to you pass in a piece of data, the handler_data, that will be passed back to you on all calls.

Once you are done, call ipmi_destroy_user() to get rid of the user.

From userland, opening the device automatically creates a user, and closing the device automatically destroys the user.

Messaging

To send a message from kernel-land, the ipmi_request() call does pretty much all message handling. Most of the parameter are self-explanatory. However, it takes a "msgid" parameter. This is NOT the sequence number of messages. It is simply a long value that is passed back when the response for the message is returned. You may use it for anything you like.

Responses come back in the function pointed to by the ipmi_recv_hndl field of the "handler" that you passed in to ipmi_create_user(). Remember again, these may be running at interrupt level. Remember to look at the receive type, too.

From userland, you fill out an ipmi_req_t structure and use the IPMICTL_SEND_COMMAND ioctl. For incoming stuff, you can use select() or poll() to wait for messages to come in. However, you cannot use read() to get them, you must call the IPMICTL_RECEIVE_MSG with the ipmi_recv_t structure to actually get the message. Remember that you must supply a pointer to a block of data in the msg.data field, and you must fill in the msg.data_len field with the size of the data. This gives the receiver a place to actually put the message.

If the message cannot fit into the data you provide, you will get an EMSGSIZE error and the driver will leave the data in the receive queue. If you want to get it and have it truncate the message, us the IPMICTL_RECEIVE_MSG_TRUNC ioctl.

When you send a command (which is defined by the lowest-order bit of the netfn per the IPMI spec) on the IPMB bus, the driver will automatically assign the sequence number to the command and save the command. If the response is not receive in the IPMI-specified 5 seconds, it will generate a response automatically saying the command timed out. If an unsolicited response comes in (if it was after 5 seconds, for instance), that response will be ignored.

In kernelland, after you receive a message and are done with it, you MUST call ipmi_free_recv_msg() on it, or you will leak messages. Note that you should NEVER mess with the "done" field of a message, that is required to properly clean up the message.

Note that when sending, there is an ipmi_request_supply_msgs() call that lets you supply the smi and receive message. This is useful for pieces of code that need to work even if the system is out of buffers (the watchdog timer uses this, for instance). You supply your own buffer and own free routines. This is not recommended for normal use, though, since it is tricky to manage your own buffers.

Events and Incoming Commands

The driver takes care of polling for IPMI events and receiving commands (commands are messages that are not responses, they are commands that other things on the IPMB bus have sent you). To receive these, you must register for them, they will not automatically be sent to you.

To receive events, you must call ipmi_set_gets_events() and set the "val" to non-zero. Any events that have been received by the driver since startup will immediately be delivered to the first user that registers for events. After that, if multiple users are registered for events, they will all receive all events that come in.

For receiving commands, you have to individually register commands you want to receive. Call ipmi_register_for_cmd() and supply the netfn and command name for each command you want to receive. You also specify a bitmask of the channels you want to receive the command from (or use IPMI_CHAN_ALL for all channels if you don't care). Only one user may be registered for each netfn/cmd/channel, but different users may register for different commands, or the same command if the channel bitmasks do not overlap.

From userland, equivalent IOCTLs are provided to do these functions.

The Lower Layer (SMI) Interface

As mentioned before, multiple SMI interfaces may be registered to the
message handler, each of these is assigned an interface number when
they register with the message handler. They are generally assigned
in the order they register, although if an SMI unregisters and then
another one registers, all bets are off.

The ipmi_smi.h defines the interface for management interfaces, see
that for more details.

The SI Driver

The SI driver allows up to 4 KCS or SMIC interfaces to be configured
in the system. By default, scan the ACPI tables for interfaces, and
if it doesn't find any the driver will attempt to register one KCS
interface at the spec-specified I/O port 0xca2 without interrupts.
You can change this at module load time (for a module) with:

```
  modprobe ipmi_si.o type=<type1>,<type2>....
       ports=<port1>,<port2>... addrs=<addr1>,<addr2>...
       irqs=<irq1>,<irq2>... trydefaults=[0|1]
       regspacings=<sp1>,<sp2>,... regsizes=<size1>,<size2>,...
       regshifts=<shift1>,<shift2>,...
       slave_addrs=<addr1>,<addr2>,...
       force_kipmid=<enable1>,<enable2>,...
       kipmid_max_busy_us=<ustime1>,<ustime2>,...
       unload_when_empty=[0|1]
```

Each of these except si_trydefaults is a list, the first item for the
first interface, second item for the second interface, etc.

The si_type may be either "kcs", "smic", or "bt". If you leave it blank, it
defaults to "kcs".

If you specify si_addrs as non-zero for an interface, the driver will
use the memory address given as the address of the device. This
overrides si_ports.

If you specify si_ports as non-zero for an interface, the driver will
use the I/O port given as the device address.

If you specify si_irqs as non-zero for an interface, the driver will
attempt to use the given interrupt for the device.

si_trydefaults sets whether the standard IPMI interface at 0xca2 and
any interfaces specified by ACPE are tried. By default, the driver
tries it, set this value to zero to turn this off.

The next three parameters have to do with register layout. The
registers used by the interfaces may not appear at successive

locations and they may not be in 8-bit registers. These parameters
allow the layout of the data in the registers to be more precisely
specified.

The regspacings parameter give the number of bytes between successive
register start addresses. For instance, if the regspacing is set to 4
and the start address is 0xca2, then the address for the second
register would be 0xca6. This defaults to 1.

The regsizes parameter gives the size of a register, in bytes. The
data used by IPMI is 8-bits wide, but it may be inside a larger
register. This parameter allows the read and write type to specified.
It may be 1, 2, 4, or 8. The default is 1.

Since the register size may be larger than 32 bits, the IPMI data may not
be in the lower 8 bits. The regshifts parameter give the amount to shift
the data to get to the actual IPMI data.

The slave_addrs specifies the IPMI address of the local BMC. This is
usually 0x20 and the driver defaults to that, but in case it's not, it
can be specified when the driver starts up.

The force_ipmid parameter forcefully enables (if set to 1) or disables
(if set to 0) the kernel IPMI daemon. Normally this is auto-detected
by the driver, but systems with broken interrupts might need an enable,
or users that don't want the daemon (don't need the performance, don't
want the CPU hit) can disable it.

If unload_when_empty is set to 1, the driver will be unloaded if it
doesn't find any interfaces or all the interfaces fail to work. The
default is one. Setting to 0 is useful with the hotmod, but is
obviously only useful for modules.

When compiled into the kernel, the parameters can be specified on the
kernel command line as:

```
  ipmi_si.type=<type1>,<type2>...
       ipmi_si.ports=<port1>,<port2>... ipmi_si.addrs=<addr1>,<addr2>...
       ipmi_si.irqs=<irq1>,<irq2>... ipmi_si.trydefaults=[0|1]
       ipmi_si.regspacings=<sp1>,<sp2>,...
       ipmi_si.regsizes=<size1>,<size2>,...
       ipmi_si.regshifts=<shift1>,<shift2>,...
       ipmi_si.slave_addrs=<addr1>,<addr2>,...
       ipmi_si.force_kipmid=<enable1>,<enable2>,...
       ipmi_si.kipmid_max_busy_us=<ustime1>,<ustime2>,...
```

It works the same as the module parameters of the same names.

By default, the driver will attempt to detect any device specified by
ACPI, and if none of those then a KCS device at the spec-specified
0xca2. If you want to turn this off, set the "trydefaults" option to
false.

If your IPMI interface does not support interrupts and is a KCS or
SMIC interface, the IPMI driver will start a kernel thread for the
interface to help speed things up. This is a low-priority kernel

thread that constantly polls the IPMI driver while an IPMI operation
is in progress. The force_kipmid module parameter will all the user to
force this thread on or off. If you force it off and don't have
interrupts, the driver will run VERY slowly. Don't blame me,
these interfaces suck.

Unfortunately, this thread can use a lot of CPU depending on the
interface's performance. This can waste a lot of CPU and cause
various issues with detecting idle CPU and using extra power. To
avoid this, the kipmid_max_busy_us sets the maximum amount of time, in
microseconds, that kipmid will spin before sleeping for a tick. This
value sets a balance between performance and CPU waste and needs to be
tuned to your needs. Maybe, someday, auto-tuning will be added, but
that's not a simple thing and even the auto-tuning would need to be
tuned to the user's desired performance.

The driver supports a hot add and remove of interfaces. This way,
interfaces can be added or removed after the kernel is up and running.
This is done using /sys/modules/ipmi_si/parameters/hotmod, which is a
write-only parameter. You write a string to this interface. The string
has the format:

```
   <op1>[:op2[:op3...]]
```

The "op"s are:

```
   add|remove,kcs|bt|smic,mem|i/o,<address>[,<opt1>[,<opt2>[,...]]]
```

You can specify more than one interface on the line. The "opt"s are:

```
   rsp=<regspacing>
   rsi=<regsize>
   rsh=<regshift>
   irq=<irq>
   ipmb=<ipmb slave addr>
```

and these have the same meanings as discussed above. Note that you
can also use this on the kernel command line for a more compact format
for specifying an interface. Note that when removing an interface,
only the first three parameters (si type, address type, and address)
are used for the comparison. Any options are ignored for removing.

The SMBus Driver

The SMBus driver allows up to 4 SMBus devices to be configured in the
system. By default, the driver will register any SMBus interfaces it finds
in the I2C address range of 0x20 to 0x4f on any adapter. You can change this
at module load time (for a module) with:

```
  modprobe ipmi_smb.o
        addr=<adapter1>,<i2caddr1>[,<adapter2>,<i2caddr2>[,...]]
        dbg=<flags1>,<flags2>...
        [defaultprobe=1] [dbg_probe=1]
```

The addresses are specified in pairs, the first is the adapter ID and the
second is the I2C address on that adapter.

The debug flags are bit flags for each BMC found, they are:
IPMI messages: 1, driver state: 2, timing: 4, I2C probe: 8

Setting smb_defaultprobe to zero disabled the default probing of SMBus

interfaces at address range 0x20 to 0x4f. This means that only the
BMCs specified on the smb_addr line will be detected.

Setting smb_dbg_probe to 1 will enable debugging of the probing and
detection process for BMCs on the SMBusses.

Discovering the IPMI compliant BMC on the SMBus can cause devices
on the I2C bus to fail. The SMBus driver writes a "Get Device ID" IPMI
message as a block write to the I2C bus and waits for a response.
This action can be detrimental to some I2C devices. It is highly recommended
that the known I2c address be given to the SMBus driver in the smb_addr
parameter. The default address range will not be used when a smb_addr
parameter is provided.

When compiled into the kernel, the addresses can be specified on the
kernel command line as:

```
  ipmb_smb.addr=<adapter1>,<i2caddr1>[,<adapter2>,<i2caddr2>[,...]]
        ipmi_smb.dbg=<flags1>,<flags2>...
        ipmi_smb.defaultprobe=0 ipmi_smb.dbg_probe=1
```

These are the same options as on the module command line.

Note that you might need some I2C changes if CONFIG_IPMI_PANIC_EVENT
is enabled along with this, so the I2C driver knows to run to
completion during sending a panic event.

Other Pieces

Watchdog

A watchdog timer is provided that implements the Linux-standard
watchdog timer interface. It has three module parameters that can be
used to control it:

```
  modprobe ipmi_watchdog timeout=<t> pretimeout=<t> action=<action type>
      preaction=<preaction type> preop=<preop type> start_now=x
      nowayout=x ifnum_to_use=n
```

ifnum_to_use specifies which interface the watchdog timer should use.
The default is -1, which means to pick the first one registered.

The timeout is the number of seconds to the action, and the pretimeout
is the amount of seconds before the reset that the pre-timeout panic will
occur (if pretimeout is zero, then pretimeout will not be enabled). Note
that the pretimeout is the time before the final timeout. So if the
timeout is 50 seconds and the pretimeout is 10 seconds, then the pretimeout
will occur in 40 second (10 seconds before the timeout).

The action may be "reset", "power_cycle", or "power_off", and
specifies what to do when the timer times out, and defaults to
"reset".

The preaction may be "pre_smi" for an indication through the SMI
interface, "pre_int" for an indication through the SMI with an
interrupts, and "pre_nmi" for a NMI on a preaction. This is how
the driver is informed of the pretimeout.

The preop may be set to "preop_none" for no operation on a pretimeout,
"preop_panic" to set the preoperation to panic, or "preop_give_data"
to provide data to read from the watchdog device when the pretimeout
occurs. A "pre_nmi" setting CANNOT be used with "preop_give_data"
because you can't do data operations from an NMI.

When preop is set to "preop_give_data", one byte comes ready to read
on the device when the pretimeout occurs. Select and fasync work on
the device, as well.

If start_now is set to 1, the watchdog timer will start running as
soon as the driver is loaded.

If nowayout is set to 1, the watchdog timer will not stop when the
watchdog device is closed. The default value of nowayout is true
if the CONFIG_WATCHDOG_NOWAYOUT option is enabled, or false if not.

When compiled into the kernel, the kernel command line is available
for configuring the watchdog:

```
  ipmi_watchdog.timeout=<t> ipmi_watchdog.pretimeout=<t>
	ipmi_watchdog.action=<action type>
	ipmi_watchdog.preaction=<preaction type>
	ipmi_watchdog.preop=<preop type>
	ipmi_watchdog.start_now=x
	ipmi_watchdog.nowayout=x
```

The options are the same as the module parameter options.

The watchdog will panic and start a 120 second reset timeout if it
gets a pre-action. During a panic or a reboot, the watchdog will
start a 120 timer if it is running to make sure the reboot occurs.

Note that if you use the NMI preaction for the watchdog, you MUST NOT
use the nmi watchdog. There is no reasonable way to tell if an NMI
comes from the IPMI controller, so it must assume that if it gets an
otherwise unhandled NMI, it must be from IPMI and it will panic
immediately.

Once you open the watchdog timer, you must write a 'V' character to the
device to close it, or the timer will not stop. This is a new semantic
for the driver, but makes it consistent with the rest of the watchdog
drivers in Linux.

Panic Timeouts

The OpenIPMI driver supports the ability to put semi-custom and custom
events in the system event log if a panic occurs. if you enable the
'Generate a panic event to all BMCs on a panic' option, you will get

one event on a panic in a standard IPMI event format. If you enable
the 'Generate OEM events containing the panic string' option, you will
also get a bunch of OEM events holding the panic string.

The field settings of the events are:
* Generator ID: 0x21 (kernel)
* EvM Rev: 0x03 (this event is formatting in IPMI 1.0 format)
* Sensor Type: 0x20 (OS critical stop sensor)
* Sensor #: The first byte of the panic string (0 if no panic string)
* Event Dir | Event Type: 0x6f (Assertion, sensor-specific event info)
* Event Data 1: 0xa1 (Runtime stop in OEM bytes 2 and 3)
* Event data 2: second byte of panic string
* Event data 3: third byte of panic string
See the IPMI spec for the details of the event layout. This event is
always sent to the local management controller. It will handle routing
the message to the right place

Other OEM events have the following format:
Record ID (bytes 0-1): Set by the SEL.
Record type (byte 2): 0xf0 (OEM non-timestamped)
byte 3: The slave address of the card saving the panic
byte 4: A sequence number (starting at zero)
The rest of the bytes (11 bytes) are the panic string. If the panic string
is longer than 11 bytes, multiple messages will be sent with increasing
sequence numbers.

Because you cannot send OEM events using the standard interface, this
function will attempt to find an SEL and add the events there. It
will first query the capabilities of the local management controller.
If it has an SEL, then they will be stored in the SEL of the local
management controller. If not, and the local management controller is
an event generator, the event receiver from the local management
controller will be queried and the events sent to the SEL on that
device. Otherwise, the events go nowhere since there is nowhere to
send them.

Poweroff

If the poweroff capability is selected, the IPMI driver will install
a shutdown function into the standard poweroff function pointer. This
is in the ipmi_poweroff module. When the system requests a powerdown,
it will send the proper IPMI commands to do this. This is supported on
several platforms.

There is a module parameter named "poweroff_powercycle" that may
either be zero (do a power down) or non-zero (do a power cycle, power
the system off, then power it on in a few seconds). Setting
ipmi_poweroff.poweroff_control=x will do the same thing on the kernel
command line. The parameter is also available via the proc filesystem
in /proc/sys/dev/ipmi/poweroff_powercycle. Note that if the system
does not support power cycling, it will always do the power off.

The "ifnum_to_use" parameter specifies which interface the poweroff code should use. The default is -1, which means to pick the first one registered.

Note that if you have ACPI enabled, the system will prefer using ACPI to power off.