The Linux IPMI Driver

Author: 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 https://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. You can also enable these dynamically by setting the module parameter named "panic_op" in the ipmi_msghandler module to "event" or "string". Setting that parameter to "none" disables this function.

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_ssif - 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_powernv - A driver for access BMCs on POWERNV systems.

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.

bt-bmc - This is not part of the main driver, but instead a driver for accessing a BMC-side interface of a BT interface. It is used on BMCs running Linux to provide an interface to the host.

These are all individually selectable via configuration options.

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

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 going through the BMC 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.

There is also an IPMB direct address for a situation where the sender is directly on an IPMB bus and doesn't have to go through the BMC. You can send messages to a specific management controller (MC) on the IPMB using the IPMI_IPMB_DIRECT_ADDR_TYPE with the following format:

```
struct ipmi_ipmb_direct_addr
{
      int           addr_type;
      short         channel;
      unsigned char slave_addr;
      unsigned char rq_lun;
      unsigned char rs_lun;
};
```

The channel is always zero. You can also receive commands from other MCs that you have registered to handle and respond to them, so you can use this to implement a management controller on a bus..

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.

Watching For Interfaces

When your code comes up, the IPMI driver may or may not have detected if IPMI devices exist. So you might have to defer your setup until the device is detected, or you might be able to do it immediately. To handle this, and to allow for discovery, you register an SMI watcher with ipmi_smi_watcher_register() to iterate over interfaces and tell you when they come and go.

Creating the User

To use 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_settime() 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.

To respond to a received command, set the response bit in the returned netfn, use the address from the received message, and use the same msgid that you got in the receive message.

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 KCS, BT, and SMIC interfaces to be configured in the system. It discovers interfaces through a host of different methods, depending on the system.

You can specify up to four interfaces on the module load line and control some module parameters:

```
modprobe ipmi_si.o type=<type1>,<type2>....
     ports=<port1>,<port2>... addrs=<addr1>,<addr2>...
     irqs=<irq1>,<irq2>...
     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]
     trydmi=[0|1] tryacpi=[0|1]
     tryplatform=[0|1] trypci=[0|1]
```

Each of these except try... items 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 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 ports as non-zero for an interface, the driver will use the I/O port given as the device address.

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

The other try... items disable discovery by their corresponding names. These are all enabled by default, set them to zero to disable them. The tryplatform disables openfirmware.

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.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.

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 (SSIF)

The SMBus driver allows up to 4 SMBus devices to be configured in the system. By default, the driver will only register with something it finds in DMI or ACPI tables. You can change this at module load time (for a module) with:

```
modprobe ipmi_ssif.o
      addr=<i2caddr1>[,<i2caddr2>[,...]]
```

```
        adapter=<adapter1>[,<adapter2>[...]]
        dbg=<flags1>,<flags2>...
        slave_addrs=<addr1>,<addr2>,...
        tryacpi=[0|1] trydmi=[0|1]
        [dbg_probe=1]
        alerts_broken
```

The addresses are normal I2C addresses. The adapter is the string name of the adapter, as shown in /sys/class/i2c-adapter/i2c-<n>/name. It is *NOT* i2c-<n> itself. Also, the comparison is done ignoring spaces, so if the name is "This is an I2C chip" you can say adapter_name=ThisisanI2cchip. This is because it's hard to pass in spaces in kernel parameters.

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

The tryxxx parameters can be used to disable detecting interfaces from various sources.

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

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.

alerts_broken does not enable SMBus alert for SSIF. Otherwise SMBus alert will be enabled on supported hardware.

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 unless you have DMI or ACPI data to tell the driver what to use.

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

```
ipmb_ssif.addr=<i2caddr1>[,<i2caddr2>[...]]
      ipmi_ssif.adapter=<adapter1>[,<adapter2>[...]]
      ipmi_ssif.dbg=<flags1>[,<flags2>[...]]
      ipmi_ssif.dbg_probe=1
      ipmi_ssif.slave_addrs=<addr1>[,<addr2>[...]]
      ipmi_ssif.tryacpi=[0|1] ipmi_ssif.trydmi=[0|1]
```

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

The I2C driver does not support non-blocking access or polling, so this driver cannod to IPMI panic events, extend the watchdog at panic time, or other panic-related IPMI functions without special kernel patches and driver modifications. You can get those at the openipmi web page.

The driver supports a hot add and remove of interfaces through the I2C sysfs interface.

The IPMI IPMB Driver

This driver is for supporting a system that sits on an IPMB bus; it allows the interface to look like a normal IPMI interface. Sending system interface addressed messages to it will cause the message to go to the registered BMC on the system (default at IPMI address 0x20).

It also allows you to directly address other MCs on the bus using the ipmb direct addressing. You can receive commands from other MCs on the bus and they will be handled through the normal received command mechanism described above.

Parameters are:

```
ipmi_ipmb.bmcaddr=<address to use for system interface addresses messages>
      ipmi_ipmb.retry_time_ms=<Time between retries on IPMB>
      ipmi_ipmb.max_retries=<Number of times to retry a message>
```

Loading the module will not result in the driver automatcially starting unless there is device tree information setting it up. If you want to instantiate one of these by hand, do:

```
echo ipmi-ipmb <addr> > /sys/class/i2c-dev/i2c-<n>/device/new_device
```

Note that the address you give here is the I2C address, not the IPMI address. So if you want your MC address to be 0x60, you put 0x30 here. See the I2C driver info for more details.

Command bridging to other IPMB busses through this interface does not work. The receive message queue is not implemented, by design. There is only one receive message queue on a BMC, and that is meant for the host drivers, not something on the IPMB bus.

A BMC may have multiple IPMB busses, which bus your device sits on depends on how the system is wired. You can fetch the channels with "ipmitool channel info <n>" where <n> is the channel, with the channels being 0-7 and try the IPMB channels.

Other Pieces

Get the detailed info related with the IPMI device

Some users need more detailed information about a device, like where the address came from or the raw base device for the IPMI interface. You can use the IPMI smi_watcher to catch the IPMI interfaces as they come or go, and to grab the information, you can use the function ipmi_get_smi_info(), which returns the following structure:

```
struct ipmi_smi_info {
      enum ipmi_addr_src addr_src;
      struct device *dev;
      union {
              struct {
                      void *acpi_handle;
              } acpi_info;
      } addr_info;
};
```

Currently special info for only for SI_ACPI address sources is returned. Others may be added as necessary.

Note that the dev pointer is included in the above structure, and assuming ipmi_smi_get_info returns success, you must call put_device on the dev pointer.

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 panic_wdt_timeout=<t>
```

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 panic_wdt_timeout is the value of timeout which is set on kernel panic, in order to let actions such as kdump to occur during panic.

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
      ipmi_watchdog.panic_wdt_timeout=<t>
```

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.