

## [PATCH] IPMI driver updates, part 2

**Source:** <http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-02/5880.html>

---

**From:** Corey Minyard ([minyard\\_at\\_acm.org](mailto:minyard_at_acm.org))

**Date:** 02/23/04

Date: Mon, 23 Feb 2004 10:00:24 -0600

To: lkml <[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)>, Linus Torvalds <[torvalds@osdl.org](mailto:torvalds@osdl.org)>

It has been far too long since the last IPMI driver updates, but now all the planets have aligned and all the pieces I needed are in and all seem to be working. This update is coming as four parts that must be applied in order, but the later parts do not have to be applied for the former parts to work.

This second part adds the "System Interface" driver. The previous driver for system interfaces only supported the KCS interface, this driver supports all system interfaces defined in the IPMI standard. It also does a much better job of handling ACPI and SMBIOS tables for detecting IPMI system interfaces.

FYI, IPMI is a standard for monitoring and maintaining a system. It provides interfaces for detecting sensors (voltage, temperature, etc.) in the system and monitoring those sensors. Many systems have extended capabilities that allow IPMI to control the system, doing things like lighting leds and controlling hot-swap. This driver allows access to the IPMI system.

-Corey

```
diff -urN linux-a1/Documentation/IPMI.txt linux-a2/Documentation/IPMI.txt
--- linux-a1/Documentation/IPMI.txt 2004-02-23 08:22:00.000000000 -0600
+++ linux-a2/Documentation/IPMI.txt 2004-02-23 08:35:19.000000000 -0600
@@ -40,8 +40,13 @@
```

main menu. This provides a socket interface to IPMI. You may select both of these at the same time, they will both work together.

-There is also a KCS-only driver interface supplied, most IPMI systems support KCS, so you need taht.

+The driver interface depends on your hardware. 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.

+

+There is also a KCS–only driver interface supplied, but it is  
+depracted in favor of the SI interface.

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

@@ –84,8 +89,13 @@

driver, each open file for this device ties in to the message handler  
as an IPMI user.

+ipmi\_si\_drv – A driver for various system interfaces. This supports  
+KCS, SMIC, and may support BT in the future. Unless you have your own  
+custom interface, you probably need to use this.

+

ipmi\_kcs\_drv – A driver for the KCS SI. Most systems have a KCS  
–interface for IPMI.

+interface for IPMI. This is deprecated, ipmi\_si\_drv supports KCS and  
+SMIC interfaces.

Much documentation for the interface is in the include files. The

@@ –308,6 +318,64 @@

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:

+

+ insmod ipmi\_si\_drv.o si\_type=<type1>,<type2>....

+ si\_ports=<port1>,<port2>... si\_addr=<addr1>,<addr2>

+ si\_irqs=<irq1>,<irq2>... si\_trydefaults=[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\_addr 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

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

+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.  
+  
+When compiled into the kernel, the addresses can be specified on the  
+kernel command line as:  
+  
+ ipmi\_si=[<type>,<bmc1>:<irq1>,<type>,<bmc2>:<irq2>.....,<nodefault>]  
+  
+The type is optional and may be either "kcs" for KCS, "smic" for  
+SMIC, or "bt" for BT. If not specified, it defaults to KCS.  
+The <bmcx> values is either "p<port>" or "m<addr>" for port or memory  
+addresses. So for instance, a KCS interface at port 0xca2 using  
+interrupt 9 and a SMIC memory interface at address 0xf9827341 with no  
+interrupt would be specified "ipmi\_si=k,p0xca2:9,s,m0xf9827341". If you  
+specify zero for in irq or don't specify it, the driver will run polled  
+unless the software can detect the interrupt to use in the ACPI tables.  
+  
+By default, the driver will attempt to detect a KCS device at the  
+spec-specified 0xca2 address and any address specified by ACPI. If  
+you want to turn this off, use the "nodefault" option.  
+  
+If you have high-res timers compiled into the kernel, the driver will  
+use them to provide much better performance. Note that if you do not  
+have high-res timers enabled in the kernel and you don't have  
+interrupts enabled, the driver will run VERY slowly. Don't blame me,  
+these interfaces suck.

### The KCS Driver

```
diff -urN linux-a1/drivers/char/ipmi/ipmi_bt_sm.c linux-a2/drivers/char/ipmi/ipmi_bt_sm.c
--- linux-a1/drivers/char/ipmi/ipmi_bt_sm.c 1969-12-31 18:00:00.000000000 -0600
+++ linux-a2/drivers/char/ipmi/ipmi_bt_sm.c 2004-02-23 08:27:14.000000000 -0600
@@ -0,0 +1,511 @@
+/*
+ * ipmi_bt_sm.c
+ *
+ * The state machine for an Open IPMI BT sub-driver under ipmi_si.c, part
+ * of the driver architecture at http://sourceforge.net/project/openipmi
+ *
+ * Author: Rocky Craig <first.last@hp.com>
+ *
+ * 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.
+ *
```

## Linux–Kernel: [PATCH] IPMI driver updates, part 2

```
+ * 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. */
+
+#include <linux/kernel.h> /* For printk. */
+#include <linux/string.h>
+#include <linux/ipmi_msgdefs.h> /* for completion codes */
+#include "ipmi_si_sm.h"
+
+#define IPMI_BT_VERSION "v30"
+
+static int bt_debug = 0x00; /* Production value 0, see following flags */
+
+#define BT_DEBUG_ENABLE 1
+#define BT_DEBUG_MSG 2
+#define BT_DEBUG_STATES 4
+
+/* Typical "Get BT Capabilities" values are 2–3 retries, 5–10 seconds,
+ and 64 byte buffers. However, one HP implementation wants 255 bytes of
+ buffer (with a documented message of 160 bytes) so go for the max.
+ Since the Open IPMI architecture is single–message oriented at this
+ stage, the queue depth of BT is of no concern. */
+
+#define BT_NORMAL_TIMEOUT 2000000 /* seconds in microseconds */
+#define BT_RETRY_LIMIT 2
+#define BT_RESET_DELAY 6000000 /* 6 seconds after warm reset */
+
+enum bt_states {
+ BT_STATE_IDLE,
+ BT_STATE_XACTION_START,
+ BT_STATE_WRITE_BYTES,
+ BT_STATE_WRITE_END,
+ BT_STATE_WRITE_CONSUME,
+ BT_STATE_B2H_WAIT,
+ BT_STATE_READ_END,
+ BT_STATE_RESET1, /* These must come last */
+ BT_STATE_RESET2,
+ BT_STATE_RESET3,
+ BT_STATE_RESTART,
+ BT_STATE_HOSED
```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+};
+
+struct si_sm_data {
+ enum bt_states state;
+ enum bt_states last_state; /* assist printing and resets */
+ unsigned char seq; /* BT sequence number */
+ struct si_sm_io *io;
+ unsigned char write_data[IPMI_MAX_MSG_LENGTH];
+ int write_count;
+ unsigned char read_data[IPMI_MAX_MSG_LENGTH];
+ int read_count;
+ int truncated;
+ long timeout;
+ unsigned int error_retries; /* end of "common" fields */
+ int nonzero_status; /* hung BMCs stay all 0 */
+};
+
+#define BT_CLR_WR_PTR 0x01 /* See IPMI 1.5 table 11.6.4 */
+#define BT_CLR_RD_PTR 0x02
+#define BT_H2B_ATN 0x04
+#define BT_B2H_ATN 0x08
+#define BT_SMS_ATN 0x10
+#define BT_OEM0 0x20
+#define BT_H_BUSY 0x40
+#define BT_B_BUSY 0x80
+
+/* Some bits are toggled on each write: write once to set it, once
+ more to clear it; writing a zero does nothing. To absolutely
+ clear it, check its state and write if set. This avoids the "get
+ current then use as mask" scheme to modify one bit. Note that the
+ variable "bt" is hardcoded into these macros. */
+
+#define BT_STATUS bt->io->inputb(bt->io, 0)
+#define BT_CONTROL(x) bt->io->outputb(bt->io, 0, x)
+
+#define BMC2HOST bt->io->inputb(bt->io, 1)
+#define HOST2BMC(x) bt->io->outputb(bt->io, 1, x)
+
+#define BT_INTMASK_R bt->io->inputb(bt->io, 2)
+#define BT_INTMASK_W(x) bt->io->outputb(bt->io, 2, x)
+
+/* Convenience routines for debugging. These are not multi-open safe!
+ Note the macros have hardcoded variables in them. */
+
+static char *state2txt(unsigned char state)
+{
+ switch (state) {
+ case BT_STATE_IDLE: return("IDLE");
+ case BT_STATE_XACTION_START: return("XACTION");
+ case BT_STATE_WRITE_BYTES: return("WR_BYTES");
+ case BT_STATE_WRITE_END: return("WR_END");
```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ case BT_STATE_WRITE_CONSUME: return("WR_CONSUME");
+ case BT_STATE_B2H_WAIT: return("B2H_WAIT");
+ case BT_STATE_READ_END: return("RD_END");
+ case BT_STATE_RESET1: return("RESET1");
+ case BT_STATE_RESET2: return("RESET2");
+ case BT_STATE_RESET3: return("RESET3");
+ case BT_STATE_RESTART: return("RESTART");
+ case BT_STATE_HOSED: return("HOSED");
+ }
+ return("BAD STATE");
+}
+#define STATE2TXT state2txt(bt->state)
+
+static char *status2txt(unsigned char status)
+{
+ static char msg[40];
+ strcpy(msg, "[ ");
+ if (status & BT_B_BUSY) strcat(msg, "B_BUSY ");
+ if (status & BT_H_BUSY) strcat(msg, "H_BUSY ");
+ if (status & BT_OEM0) strcat(msg, "OEM0 ");
+ if (status & BT_SMS_ATN) strcat(msg, "SMS ");
+ if (status & BT_B2H_ATN) strcat(msg, "B2H ");
+ if (status & BT_H2B_ATN) strcat(msg, "H2B ");
+ strcat(msg, "]");
+ return msg;
+}
+#define STATUS2TXT status2txt(status)
+
+/* This will be called from within this module on a hosed condition */
+#define FIRST_SEQ 0
+static unsigned int bt_init_data(struct si_sm_data *bt, struct si_sm_io *io)
+{
+ bt->state = BT_STATE_IDLE;
+ bt->last_state = BT_STATE_IDLE;
+ bt->seq = FIRST_SEQ;
+ bt->io = io;
+ bt->write_count = 0;
+ bt->read_count = 0;
+ bt->error_retries = 0;
+ bt->nonzero_status = 0;
+ bt->truncated = 0;
+ bt->timeout = BT_NORMAL_TIMEOUT;
+ return 3; /* We claim 3 bytes of space; ought to check SPMI table */
+}
+
+static int bt_start_transaction(struct si_sm_data *bt,
+ unsigned char *data,
+ unsigned int size)
+{
+ unsigned int i;
+
+ }
```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ if ((size < 2) || (size > IPMI_MAX_MSG_LENGTH)) return -1;
+
+ if ((bt->state != BT_STATE_IDLE) && (bt->state != BT_STATE_HOSED))
+ return -2;
+
+ if (bt_debug & BT_DEBUG_MSG) {
+ printk(KERN_WARNING "++++++++++++++++++++++++++++++++++++\n");
+ printk(KERN_WARNING "BT: write seq=0x%02X:", bt->seq);
+ for (i = 0; i < size; i++) printk (" %02x", data[i]);
+ printk("\n");
+ }
+ bt->write_data[0] = size + 1; /* all data plus seq byte */
+ bt->write_data[1] = *data; /* NetFn/LUN */
+ bt->write_data[2] = bt->seq;
+ memcpy(bt->write_data + 3, data + 1, size - 1);
+ bt->write_count = size + 2;
+
+ bt->error_retries = 0;
+ bt->nonzero_status = 0;
+ bt->read_count = 0;
+ bt->truncated = 0;
+ bt->state = BT_STATE_XACTION_START;
+ bt->last_state = BT_STATE_IDLE;
+ bt->timeout = BT_NORMAL_TIMEOUT;
+ return 0;
+}
+
+/* After the upper state machine has been told SI_SM_TRANSACTION_COMPLETE
+ it calls this. Strip out the length and seq bytes. */
+
+static int bt_get_result(struct si_sm_data *bt,
+ unsigned char *data,
+ unsigned int length)
+{
+ int i, msg_len;
+
+ msg_len = bt->read_count - 2; /* account for length & seq */
+ /* Always NetFn, Cmd, cCode */
+ if (msg_len < 3 || msg_len > IPMI_MAX_MSG_LENGTH) {
+ printk(KERN_WARNING "BT results: bad msg_len = %d\n", msg_len);
+ data[0] = bt->write_data[1] | 0x4; /* Kludge a response */
+ data[1] = bt->write_data[3];
+ data[2] = IPMI_ERR_UNSPECIFIED;
+ msg_len = 3;
+ } else {
+ data[0] = bt->read_data[1];
+ data[1] = bt->read_data[3];
+ if (length < msg_len) bt->truncated = 1;
+ if (bt->truncated) { /* can be set in read_all_bytes() */
+ data[2] = IPMI_ERR_MSG_TRUNCATED;
+ msg_len = 3;
+ }
+ }
```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```

+ } else memcpy(data + 2, bt->read_data + 4, msg_len - 2);
+
+ if (bt_debug & BT_DEBUG_MSG) {
+ printk(KERN_WARNING "BT: res (raw)");
+ for (i = 0; i < msg_len; i++) printk(" %02x", data[i]);
+ printk ("\n");
+ }
+ }
+ bt->read_count = 0; /* paranoia */
+ return msg_len;
+ }
+
+ /* This bit's functionality is optional */
+ #define BT_BMC_HWRST 0x80
+
+ static void reset_flags(struct si_sm_data *bt)
+ {
+ if (BT_STATUS & BT_H_BUSY) BT_CONTROL(BT_H_BUSY);
+ if (BT_STATUS & BT_B_BUSY) BT_CONTROL(BT_B_BUSY);
+ BT_CONTROL(BT_CLR_WR_PTR);
+ BT_CONTROL(BT_SMS_ATN);
+ BT_INTMASK_W(BT_BMC_HWRST);
+ #ifdef DEVELOPMENT_ONLY_NOT_FOR_PRODUCTION
+ if (BT_STATUS & BT_B2H_ATN) {
+ int i;
+ BT_CONTROL(BT_H_BUSY);
+ BT_CONTROL(BT_B2H_ATN);
+ BT_CONTROL(BT_CLR_RD_PTR);
+ for (i = 0; i < IPMI_MAX_MSG_LENGTH + 2; i++) BMC2HOST;
+ BT_CONTROL(BT_H_BUSY);
+ }
+ #endif
+ }
+
+ static inline void write_all_bytes(struct si_sm_data *bt)
+ {
+ int i;
+
+ if (bt_debug & BT_DEBUG_MSG) {
+ printk(KERN_WARNING "BT: write %d bytes seq=0x%02X",
+ bt->write_count, bt->seq);
+ for (i = 0; i < bt->write_count; i++)
+ printk (" %02x", bt->write_data[i]);
+ printk ("\n");
+ }
+ for (i = 0; i < bt->write_count; i++) HOST2BMC(bt->write_data[i]);
+ }
+
+ static inline int read_all_bytes(struct si_sm_data *bt)
+ {
+ unsigned char i;

```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```

+
+ bt->read_data[0] = BMC2HOST;
+ bt->read_count = bt->read_data[0];
+ if (bt_debug & BT_DEBUG_MSG)
+ printk(KERN_WARNING "BT: read %d bytes:", bt->read_count);
+
+ /* minimum: length, NetFn, Seq, Cmd, cCode == 5 total, or 4 more
+ following the length byte. */
+ if (bt->read_count < 4 || bt->read_count >= IPMI_MAX_MSG_LENGTH) {
+ if (bt_debug & BT_DEBUG_MSG)
+ printk("bad length %d\n", bt->read_count);
+ bt->truncated = 1;
+ return 1; /* let next XACTION START clean it up */
+ }
+ for (i = 1; i <= bt->read_count; i++) bt->read_data[i] = BMC2HOST;
+ bt->read_count++; /* account for the length byte */
+
+ if (bt_debug & BT_DEBUG_MSG) {
+ for (i = 0; i < bt->read_count; i++)
+ printk (" %02x", bt->read_data[i]);
+ printk ("\n");
+ }
+ if (bt->seq != bt->write_data[2]) /* idiot check */
+ printk(KERN_WARNING "BT: internal error: sequence mismatch\n");
+
+ /* per the spec, the (NetFn, Seq, Cmd) tuples should match */
+ if ((bt->read_data[3] == bt->write_data[3]) && /* Cmd */
+ (bt->read_data[2] == bt->write_data[2]) && /* Sequence */
+ ((bt->read_data[1] & 0xF8) == (bt->write_data[1] & 0xF8)))
+ return 1;
+
+ if (bt_debug & BT_DEBUG_MSG) printk(KERN_WARNING "BT: bad packet: "
+ "want 0x(%02X, %02X, %02X) got (%02X, %02X, %02X)\n",
+ bt->write_data[1], bt->write_data[2], bt->write_data[3],
+ bt->read_data[1], bt->read_data[2], bt->read_data[3]);
+ return 0;
+ }
+
+ /* Modifies bt->state appropriately, need to get into the bt_event() switch */
+
+ static void error_recovery(struct si_sm_data *bt, char *reason)
+ {
+ volatile unsigned char status;
+
+ bt->timeout = BT_NORMAL_TIMEOUT; /* various places want to retry */
+
+ status = BT_STATUS;
+ printk(KERN_WARNING "BT: %s in %s %s ", reason, STATE2TXT, STATUS2TXT);
+
+ (bt->error_retries)++;
+ if (bt->error_retries > BT_RETRY_LIMIT) {

```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ printk("retry limit (%d) exceeded\n", BT_RETRY_LIMIT);
+ bt->state = BT_STATE_HOSED;
+ if (!bt->nonzero_status)
+ printk(KERN_ERR "IPMI: BT stuck, try power cycle\n");
+ else if (bt->seq == FIRST_SEQ + BT_RETRY_LIMIT) {
+ /* most likely during insmod */
+ printk(KERN_WARNING "IPMI: BT reset (takes 5 secs)\n");
+ bt->state = BT_STATE_RESET1;
+ }
+ return;
+ }
+
+ /* Sometimes the BMC queues get in an "off-by-one" state...*/
+ if ((bt->state == BT_STATE_B2H_WAIT) && (status & BT_B2H_ATN)) {
+ printk("retry B2H_WAIT\n");
+ return;
+ }
+
+ printk("restart command\n");
+ bt->state = BT_STATE_RESTART;
+ }
+
+ /* Check the status and (possibly) advance the BT state machine. The
+ default return is SI_SM_CALL_WITH_DELAY. */
+
+static enum si_sm_result bt_event(struct si_sm_data *bt, long time)
+{
+ volatile unsigned char status;
+ int i;
+
+ status = BT_STATUS;
+ bt->nonzero_status |= status;
+
+ if ((bt_debug & BT_DEBUG_STATES) && (bt->state != bt->last_state))
+ printk(KERN_WARNING "BT: %s %s TO=%ld - %ld\n",
+ STATE2TXT,
+ STATUS2TXT,
+ bt->timeout,
+ time);
+ bt->last_state = bt->state;
+
+ if (bt->state == BT_STATE_HOSED) return SI_SM_HOSED;
+
+ if (bt->state != BT_STATE_IDLE) { /* do timeout test */
+
+ /* Certain states, on error conditions, can lock up a CPU
+ because they are effectively in an infinite loop with
+ CALL_WITHOUT_DELAY (right back here with time == 0).
+ Prevent infinite lockup by ALWAYS decrementing timeout. */
+
+ /* FIXME: bt_event is sometimes called with time > BT_NORMAL_TIMEOUT
```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ (noticed in ipmi_smic_sm.c January 2004) */
+
+ if ((time <= 0) || (time >= BT_NORMAL_TIMEOUT)) time = 100;
+ bt->timeout -= time;
+ if ((bt->timeout < 0) && (bt->state < BT_STATE_RESET1)) {
+ error_recovery(bt, "timed out");
+ return SI_SM_CALL_WITHOUT_DELAY;
+ }
+ }
+
+ switch (bt->state) {
+
+ case BT_STATE_IDLE: /* check for asynchronous messages */
+ if (status & BT_SMS_ATN) {
+ BT_CONTROL(BT_SMS_ATN); /* clear it */
+ return SI_SM_ATTEN;
+ }
+ return SI_SM_IDLE;
+
+ case BT_STATE_XACTION_START:
+ if (status & BT_H_BUSY) {
+ BT_CONTROL(BT_H_BUSY);
+ break;
+ }
+ if (status & BT_B2H_ATN) break;
+ bt->state = BT_STATE_WRITE_BYTES;
+ return SI_SM_CALL_WITHOUT_DELAY; /* for logging */
+
+ case BT_STATE_WRITE_BYTES:
+ if (status & (BT_B_BUSY | BT_H2B_ATN)) break;
+ BT_CONTROL(BT_CLR_WR_PTR);
+ write_all_bytes(bt);
+ BT_CONTROL(BT_H2B_ATN); /* clears too fast to catch? */
+ bt->state = BT_STATE_WRITE_CONSUME;
+ return SI_SM_CALL_WITHOUT_DELAY; /* it MIGHT sail through */
+
+ case BT_STATE_WRITE_CONSUME: /* BMCs usually blow right thru here */
+ if (status & (BT_H2B_ATN | BT_B_BUSY)) break;
+ bt->state = BT_STATE_B2H_WAIT;
+ /* fall through with status */
+
+ /* Stay in BT_STATE_B2H_WAIT until a packet matches. However, spinning
+ hard here, constantly reading status, seems to hold off the
+ generation of B2H_ATN so ALWAYS return CALL_WITH_DELAY. */
+
+ case BT_STATE_B2H_WAIT:
+ if (!(status & BT_B2H_ATN)) break;
+
+ /* Assume ordered, uncached writes: no need to wait */
+ if (!(status & BT_H_BUSY)) BT_CONTROL(BT_H_BUSY); /* set */
+ BT_CONTROL(BT_B2H_ATN); /* clear it, ACK to the BMC */
```



## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ default: /* HOSED is supposed to be caught much earlier */
+ error_recovery(bt, "internal logic error");
+ break;
+ }
+ return SI_SM_CALL_WITH_DELAY;
+}
+
+static int bt_detect(struct si_sm_data *bt)
+{
+ /* It's impossible for the BT status and interrupt registers to be
+ all 1's, (assuming a properly functioning, self-initialized BMC)
+ but that's what you get from reading a bogus address, so we
+ test that first. The calling routine uses negative logic. */
+
+ if ((BT_STATUS == 0xFF) && (BT_INTMASK_R == 0xFF)) return 1;
+ reset_flags(bt);
+ return 0;
+}
+
+static void bt_cleanup(struct si_sm_data *bt)
+{
+}
+
+static int bt_size(void)
+{
+ return sizeof(struct si_sm_data);
+}
+
+struct si_sm_handlers bt_smi_handlers =
+{
+ .version = IPMI_BT_VERSION,
+ .init_data = bt_init_data,
+ .start_transaction = bt_start_transaction,
+ .get_result = bt_get_result,
+ .event = bt_event,
+ .detect = bt_detect,
+ .cleanup = bt_cleanup,
+ .size = bt_size,
+};
diff -urN linux-a1/drivers/char/ipmi/ipmi_si.c linux-a2/drivers/char/ipmi/ipmi_si.c
--- linux-a1/drivers/char/ipmi/ipmi_si.c 1969-12-31 18:00:00.000000000 -0600
+++ linux-a2/drivers/char/ipmi/ipmi_si.c 2004-02-23 08:27:14.000000000 -0600
@@ -0,0 +1,2077 @@
+/*
+ * ipmi_si.c
+ *
+ * The interface to the IPMI driver for the system interfaces (KCS, SMIC,
+ * BT in the future).
+ *
+ * Author: MontaVista Software, Inc.
+ * Corey Minyard <minyard@mvista.com>
```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ * 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.
+ */
+
+/*
+ * This file holds the "policy" for the interface to the SMI state
+ * machine. It does the configuration, handles timers and interrupts,
+ * and drives the real SMI state machine.
+ */
+
+#include <linux/config.h>
+#include <linux/module.h>
+#include <asm/system.h>
+#include <linux/sched.h>
+#include <linux/timer.h>
+#include <linux/errno.h>
+#include <linux/spinlock.h>
+#include <linux/slab.h>
+#include <linux/delay.h>
+#include <linux/list.h>
+#include <linux/pci.h>
+#include <linux/ioport.h>
+#ifdef CONFIG_HIGH_RES_TIMERS
+#include <linux/hrtimer.h>
+# if defined(schedule_next_int)
+/* Old high-res timer code, do translations. */
+# define get_arch_cycles(a) quick_update_jiffies_sub(a)
+# define arch_cycles_per_jiffy cycles_per_jiffies
+# endif
```

## Linux–Kernel: [PATCH] IPMI driver updates, part 2

```
+static inline void add_usec_to_timer(struct timer_list *t, long v)
+{
+ t->sub_expires += nsec_to_arch_cycle(v * 1000);
+ while (t->sub_expires >= arch_cycles_per_jiffy)
+ {
+ t->expires++;
+ t->sub_expires -= arch_cycles_per_jiffy;
+ }
+}
+
+#endif
+#include <linux/interrupt.h>
+#include <linux/rcupdate.h>
+#include <linux/ipmi_smi.h>
+#include <asm/io.h>
+#include <asm/irq.h>
+#include "ipmi_si_sm.h"
+#include <linux/init.h>
+
+#define IPMI_SI_VERSION "v30"
+
+/* Measure times between events in the driver. */
+#undef DEBUG_TIMING
+
+/* Call every 10 ms. */
+#define SI_TIMEOUT_TIME_USEC 10000
+#define SI_USEC_PER_JIFFY (1000000/HZ)
+#define SI_TIMEOUT_JIFFIES (SI_TIMEOUT_TIME_USEC/SI_USEC_PER_JIFFY)
+#define SI_SHORT_TIMEOUT_USEC 250 /* .25ms when the SM request a
+ short timeout */
+
+enum si_intf_state {
+ SI_NORMAL,
+ SI_GETTING_FLAGS,
+ SI_GETTING_EVENTS,
+ SI_CLEARING_FLAGS,
+ SI_CLEARING_FLAGS_THEN_SET_IRQ,
+ SI_GETTING_MESSAGES,
+ SI_ENABLE_INTERRUPTS1,
+ SI_ENABLE_INTERRUPTS2
+ /* FIXME – add watchdog stuff. */
+};
+
+enum si_type {
+ SI_KCS, SI_SMIC, SI_BT
+};
+
+struct smi_info
+{
+ ipmi_smi_t intf;
+ struct si_sm_data *si_sm;
+ struct si_sm_handlers *handlers;
+};
```

```

+ enum si_type si_type;
+ spinlock_t si_lock;
+ spinlock_t msg_lock;
+ struct list_head xmit_msgs;
+ struct list_head hp_xmit_msgs;
+ struct ipmi_smi_msg *curr_msg;
+ enum si_intf_state si_state;
+
+ /* Used to handle the various types of I/O that can occur with
+ IPMI */
+ struct si_sm_io io;
+ int (*io_setup)(struct smi_info *info);
+ void (*io_cleanup)(struct smi_info *info);
+ int (*irq_setup)(struct smi_info *info);
+ void (*irq_cleanup)(struct smi_info *info);
+ unsigned int io_size;
+
+ /* Flags from the last GET_MSG_FLAGS command, used when an ATTN
+ is set to hold the flags until we are done handling everything
+ from the flags. */
+#define RECEIVE_MSG_AVAIL 0x01
+#define EVENT_MSG_BUFFER_FULL 0x02
+#define WDT_PRE_TIMEOUT_INT 0x08
+ unsigned char msg_flags;
+
+ /* If set to true, this will request events the next time the
+ state machine is idle. */
+ atomic_t req_events;
+
+ /* If true, run the state machine to completion on every send
+ call. Generally used after a panic to make sure stuff goes
+ out. */
+ int run_to_completion;
+
+ /* The I/O port of an SI interface. */
+ int port;
+
+ /* zero if no irq; */
+ int irq;
+
+ /* The timer for this si. */
+ struct timer_list si_timer;
+
+ /* The time (in jiffies) the last timeout occurred at. */
+ unsigned long last_timeout_jiffies;
+
+ /* Used to gracefully stop the timer without race conditions. */
+ volatile int stop_operation;
+ volatile int timer_stopped;
+
+ /* The driver will disable interrupts when it gets into a

```

```

+ situation where it cannot handle messages due to lack of
+ memory. Once that situation clears up, it will re-enable
+ interrupts. */
+ int interrupt_disabled;
+
+ unsigned char ipmi_si_dev_rev;
+ unsigned char ipmi_si_fw_rev_major;
+ unsigned char ipmi_si_fw_rev_minor;
+ unsigned char ipmi_version_major;
+ unsigned char ipmi_version_minor;
+
+ /* Counters and things for the proc filesystem. */
+ spinlock_t count_lock;
+ unsigned long short_timeouts;
+ unsigned long long_timeouts;
+ unsigned long timeout_restarts;
+ unsigned long idles;
+ unsigned long interrupts;
+ unsigned long attentions;
+ unsigned long flag_fetches;
+ unsigned long hosed_count;
+ unsigned long complete_transactions;
+ unsigned long events;
+ unsigned long watchdog_pretimeouts;
+ unsigned long incoming_messages;
+ };
+
+ static void si_restart_short_timer(struct smi_info *smi_info);
+
+ static void deliver_recv_msg(struct smi_info *smi_info,
+ struct ipmi_smi_msg *msg)
+ {
+ /* Deliver the message to the upper layer with the lock
+ released. */
+ spin_unlock(&(smi_info->si_lock));
+ ipmi_smi_msg_received(smi_info->intf, msg);
+ spin_lock(&(smi_info->si_lock));
+ }
+
+ static void return_hosed_msg(struct smi_info *smi_info)
+ {
+ struct ipmi_smi_msg *msg = smi_info->curr_msg;
+
+ /* Make it a reponse */
+ msg->rsp[0] = msg->data[0] | 4;
+ msg->rsp[1] = msg->data[1];
+ msg->rsp[2] = 0xFF; /* Unknown error. */
+ msg->rsp_size = 3;
+
+ smi_info->curr_msg = NULL;
+ deliver_recv_msg(smi_info, msg);

```

```

+}
+
+static enum si_sm_result start_next_msg(struct smi_info *smi_info)
+{
+ int rv;
+ struct list_head *entry = NULL;
+#ifdef DEBUG_TIMING
+ struct timeval t;
+#endif
+
+ /* No need to save flags, we already have interrupts off and we
+ already hold the SMI lock. */
+ spin_lock(&(smi_info->msg_lock));
+
+ /* Pick the high priority queue first. */
+ if (!list_empty(&(smi_info->hp_xmit_msgs))) {
+ entry = smi_info->hp_xmit_msgs.next;
+ } else if (!list_empty(&(smi_info->xmit_msgs))) {
+ entry = smi_info->xmit_msgs.next;
+ }
+
+ if (!entry) {
+ smi_info->curr_msg = NULL;
+ rv = SI_SM_IDLE;
+ } else {
+ int err;
+
+ list_del(entry);
+ smi_info->curr_msg = list_entry(entry,
+ struct ipmi_smi_msg,
+ link);
+#ifdef DEBUG_TIMING
+ do_gettimeofday(&t);
+ printk("***Start2: %d.%9d\n", t.tv_sec, t.tv_usec);
+#endif
+ err = smi_info->handlers->start_transaction(
+ smi_info->si_sm,
+ smi_info->curr_msg->data,
+ smi_info->curr_msg->data_size);
+ if (err) {
+ return_hosed_msg(smi_info);
+ }
+
+ rv = SI_SM_CALL_WITHOUT_DELAY;
+ }
+ spin_unlock(&(smi_info->msg_lock));
+
+ return rv;
+}
+
+static void start_enable_irq(struct smi_info *smi_info)

```

```

+{
+ unsigned char msg[2];
+
+ /* If we are enabling interrupts, we have to tell the
+ BMC to use them. */
+ msg[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ msg[1] = IPMI_GET_BMC_GLOBAL_ENABLES_CMD;
+
+ smi_info->handlers->start_transaction(smi_info->si_sm, msg, 2);
+ smi_info->si_state = SI_ENABLE_INTERRUPTS1;
+}
+
+static void start_clear_flags(struct smi_info *smi_info)
+{
+ unsigned char msg[3];
+
+ /* Make sure the watchdog pre-timeout flag is not set at startup. */
+ msg[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ msg[1] = IPMI_CLEAR_MSG_FLAGS_CMD;
+ msg[2] = WDT_PRE_TIMEOUT_INT;
+
+ smi_info->handlers->start_transaction(smi_info->si_sm, msg, 3);
+ smi_info->si_state = SI_CLEARING_FLAGS;
+}
+
+/* When we have a sitaion where we run out of memory and cannot
+ allocate messages, we just leave them in the BMC and run the system
+ polled until we can allocate some memory. Once we have some
+ memory, we will re-enable the interrupt. */
+static inline void disable_si_irq(struct smi_info *smi_info)
+{
+ if ((smi_info->irq) && (!smi_info->interrupt_disabled)) {
+ disable_irq_nosync(smi_info->irq);
+ smi_info->interrupt_disabled = 1;
+ }
+}
+
+static inline void enable_si_irq(struct smi_info *smi_info)
+{
+ if ((smi_info->irq) && (smi_info->interrupt_disabled)) {
+ enable_irq(smi_info->irq);
+ smi_info->interrupt_disabled = 0;
+ }
+}
+
+static void handle_flags(struct smi_info *smi_info)
+{
+ if (smi_info->msg_flags & WDT_PRE_TIMEOUT_INT) {
+ /* Watchdog pre-timeout */
+ spin_lock(&smi_info->count_lock);
+ smi_info->watchdog_pretimeouts++;

```

```

+ spin_unlock(&smi_info->count_lock);
+
+ start_clear_flags(smi_info);
+ smi_info->msg_flags &= ~WDT_PRE_TIMEOUT_INT;
+ spin_unlock(&(smi_info->si_lock));
+ ipmi_smi_watchdog_pretimeout(smi_info->intf);
+ spin_lock(&(smi_info->si_lock));
+ } else if (smi_info->msg_flags & RECEIVE_MSG_AVAIL) {
+ /* Messages available. */
+ smi_info->curr_msg = ipmi_alloc_smi_msg();
+ if (!smi_info->curr_msg) {
+ disable_si_irq(smi_info);
+ smi_info->si_state = SI_NORMAL;
+ return;
+ }
+ enable_si_irq(smi_info);
+
+ smi_info->curr_msg->data[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ smi_info->curr_msg->data[1] = IPMI_GET_MSG_CMD;
+ smi_info->curr_msg->data_size = 2;
+
+ smi_info->handlers->start_transaction(
+ smi_info->si_sm,
+ smi_info->curr_msg->data,
+ smi_info->curr_msg->data_size);
+ smi_info->si_state = SI_GETTING_MESSAGES;
+ } else if (smi_info->msg_flags & EVENT_MSG_BUFFER_FULL) {
+ /* Events available. */
+ smi_info->curr_msg = ipmi_alloc_smi_msg();
+ if (!smi_info->curr_msg) {
+ disable_si_irq(smi_info);
+ smi_info->si_state = SI_NORMAL;
+ return;
+ }
+ enable_si_irq(smi_info);
+
+ smi_info->curr_msg->data[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ smi_info->curr_msg->data[1] = IPMI_READ_EVENT_MSG_BUFFER_CMD;
+ smi_info->curr_msg->data_size = 2;
+
+ smi_info->handlers->start_transaction(
+ smi_info->si_sm,
+ smi_info->curr_msg->data,
+ smi_info->curr_msg->data_size);
+ smi_info->si_state = SI_GETTING_EVENTS;
+ } else {
+ smi_info->si_state = SI_NORMAL;
+ }
+ }
+
+static void handle_transaction_done(struct smi_info *smi_info)

```

```

+{
+ struct ipmi_smi_msg *msg;
+#ifdef DEBUG_TIMING
+ struct timeval t;
+
+ do_gettimeofday(&t);
+ printk("***Done: %d.%09d\n", t.tv_sec, t.tv_usec);
+#endif
+ switch (smi_info->si_state) {
+ case SI_NORMAL:
+ if (!smi_info->curr_msg)
+ break;
+
+ smi_info->curr_msg->rsp_size
+ = smi_info->handlers->get_result(
+ smi_info->si_sm,
+ smi_info->curr_msg->rsp,
+ IPMI_MAX_MSG_LENGTH);
+
+ /* Do this here because deliver_rcv_msg() releases the
+ lock, and a new message can be put in during the
+ time the lock is released. */
+ msg = smi_info->curr_msg;
+ smi_info->curr_msg = NULL;
+ deliver_rcv_msg(smi_info, msg);
+ break;
+
+ case SI_GETTING_FLAGS:
+ {
+ unsigned char msg[4];
+ unsigned int len;
+
+ /* We got the flags from the SMI, now handle them. */
+ len = smi_info->handlers->get_result(smi_info->si_sm, msg, 4);
+ if (msg[2] != 0) {
+ /* Error fetching flags, just give up for
+ now. */
+ smi_info->si_state = SI_NORMAL;
+ } else if (len < 3) {
+ /* Hmm, no flags. That's technically illegal, but
+ don't use uninitialized data. */
+ smi_info->si_state = SI_NORMAL;
+ } else {
+ smi_info->msg_flags = msg[3];
+ handle_flags(smi_info);
+ }
+ break;
+ }
+
+ case SI_CLEARING_FLAGS:
+ case SI_CLEARING_FLAGS_THEN_SET_IRQ:

```

```

+ {
+ unsigned char msg[3];
+
+ /* We cleared the flags. */
+ smi_info->handlers->get_result(smi_info->si_sm, msg, 3);
+ if (msg[2] != 0) {
+ /* Error clearing flags */
+ printk(KERN_WARNING
+ "ipmi_si: Error clearing flags: %2.2x\n",
+ msg[2]);
+ }
+ if (smi_info->si_state == SI_CLEARING_FLAGS_THEN_SET_IRQ)
+ start_enable_irq(smi_info);
+ else
+ smi_info->si_state = SI_NORMAL;
+ break;
+ }
+
+ case SI_GETTING_EVENTS:
+ {
+ smi_info->curr_msg->rsp_size
+ = smi_info->handlers->get_result(
+ smi_info->si_sm,
+ smi_info->curr_msg->rsp,
+ IPMI_MAX_MSG_LENGTH);
+
+ /* Do this here because deliver_rcv_msg() releases the
+ lock, and a new message can be put in during the
+ time the lock is released. */
+ msg = smi_info->curr_msg;
+ smi_info->curr_msg = NULL;
+ if (msg->rsp[2] != 0) {
+ /* Error getting event, probably done. */
+ msg->done(msg);
+
+ /* Take off the event flag. */
+ smi_info->msg_flags &= ~EVENT_MSG_BUFFER_FULL;
+ } else {
+ spin_lock(&smi_info->count_lock);
+ smi_info->events++;
+ spin_unlock(&smi_info->count_lock);
+
+ deliver_rcv_msg(smi_info, msg);
+ }
+ handle_flags(smi_info);
+ break;
+ }
+
+ case SI_GETTING_MESSAGES:
+ {
+ smi_info->curr_msg->rsp_size

```

```

+ = smi_info->handlers->get_result(
+ smi_info->si_sm,
+ smi_info->curr_msg->rsp,
+ IPMI_MAX_MSG_LENGTH);
+
+ /* Do this here because deliver_rcv_msg() releases the
+ lock, and a new message can be put in during the
+ time the lock is released. */
+ msg = smi_info->curr_msg;
+ smi_info->curr_msg = NULL;
+ if (msg->rsp[2] != 0) {
+ /* Error getting event, probably done. */
+ msg->done(msg);
+
+ /* Take off the msg flag. */
+ smi_info->msg_flags &= ~RECEIVE_MSG_AVAIL;
+ } else {
+ spin_lock(&smi_info->count_lock);
+ smi_info->incoming_messages++;
+ spin_unlock(&smi_info->count_lock);
+
+ deliver_rcv_msg(smi_info, msg);
+ }
+ handle_flags(smi_info);
+ break;
+ }
+
+ case SI_ENABLE_INTERRUPTS1:
+ {
+ unsigned char msg[4];
+
+ /* We got the flags from the SMI, now handle them. */
+ smi_info->handlers->get_result(smi_info->si_sm, msg, 4);
+ if (msg[2] != 0) {
+ printk(KERN_WARNING
+ "ipmi_si: Could not enable interrupts"
+ ", failed get, using polled mode.\n");
+ smi_info->si_state = SI_NORMAL;
+ } else {
+ msg[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ msg[1] = IPMI_SET_BMC_GLOBAL_ENABLES_CMD;
+ msg[2] = msg[3] | 1; /* enable msg queue int */
+ smi_info->handlers->start_transaction(
+ smi_info->si_sm, msg, 3);
+ smi_info->si_state = SI_ENABLE_INTERRUPTS2;
+ }
+ break;
+ }
+
+ case SI_ENABLE_INTERRUPTS2:
+ {

```

```

+ unsigned char msg[4];
+
+ /* We got the flags from the SMI, now handle them. */
+ smi_info->handlers->get_result(smi_info->si_sm, msg, 4);
+ if (msg[2] != 0) {
+ printk(KERN_WARNING
+ "ipmi_si: Could not enable interrupts"
+ ", failed set, using polled mode.\n");
+ }
+ smi_info->si_state = SI_NORMAL;
+ break;
+ }
+ }
+ }
+
+ /* Called on timeouts and events. Timeouts should pass the elapsed
+ time, interrupts should pass in zero. */
+ static enum si_sm_result smi_event_handler(struct smi_info *smi_info,
+ int time)
+ {
+ enum si_sm_result si_sm_result;
+
+ restart:
+ /* There used to be a loop here that waited a little while
+ (around 25us) before giving up. That turned out to be
+ pointless, the minimum delays I was seeing were in the 300us
+ range, which is far too long to wait in an interrupt. So
+ we just run until the state machine tells us something
+ happened or it needs a delay. */
+ si_sm_result = smi_info->handlers->event(smi_info->si_sm, time);
+ time = 0;
+ while (si_sm_result == SI_SM_CALL_WITHOUT_DELAY)
+ {
+ si_sm_result = smi_info->handlers->event(smi_info->si_sm, 0);
+ }
+
+ if (si_sm_result == SI_SM_TRANSACTION_COMPLETE)
+ {
+ spin_lock(&smi_info->count_lock);
+ smi_info->complete_transactions++;
+ spin_unlock(&smi_info->count_lock);
+
+ handle_transaction_done(smi_info);
+ si_sm_result = smi_info->handlers->event(smi_info->si_sm, 0);
+ }
+ else if (si_sm_result == SI_SM_HOSED)
+ {
+ spin_lock(&smi_info->count_lock);
+ smi_info->hosed_count++;
+ spin_unlock(&smi_info->count_lock);
+ }
+ }

```

```

+ if (smi_info->curr_msg != NULL) {
+ /* If we were handling a user message, format
+ a response to send to the upper layer to
+ tell it about the error. */
+ return_hosed_msg(smi_info);
+ }
+ si_sm_result = smi_info->handlers->event(smi_info->si_sm, 0);
+ smi_info->si_state = SI_NORMAL;
+ }
+
+ /* We prefer handling attn over new messages. */
+ if (si_sm_result == SI_SM_ATTN)
+ {
+ unsigned char msg[2];
+
+ spin_lock(&smi_info->count_lock);
+ smi_info->attentions++;
+ spin_unlock(&smi_info->count_lock);
+
+ /* Got a attn, send down a get message flags to see
+ what's causing it. It would be better to handle
+ this in the upper layer, but due to the way
+ interrupts work with the SMI, that's not really
+ possible. */
+ msg[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ msg[1] = IPMI_GET_MSG_FLAGS_CMD;
+
+ smi_info->handlers->start_transaction(
+ smi_info->si_sm, msg, 2);
+ smi_info->si_state = SI_GETTING_FLAGS;
+ goto restart;
+ }
+
+ /* If we are currently idle, try to start the next message. */
+ if (si_sm_result == SI_SM_IDLE) {
+ spin_lock(&smi_info->count_lock);
+ smi_info->idles++;
+ spin_unlock(&smi_info->count_lock);
+
+ si_sm_result = start_next_msg(smi_info);
+ if (si_sm_result != SI_SM_IDLE)
+ goto restart;
+ }
+
+ if ((si_sm_result == SI_SM_IDLE)
+ && (atomic_read(&smi_info->req_events)))
+ {
+ /* We are idle and the upper layer requested that I fetch
+ events, so do so. */
+ unsigned char msg[2];
+

```

```

+ spin_lock(&smi_info->count_lock);
+ smi_info->flag_fetches++;
+ spin_unlock(&smi_info->count_lock);
+
+ atomic_set(&smi_info->req_events, 0);
+ msg[0] = (IPMI_NETFN_APP_REQUEST << 2);
+ msg[1] = IPMI_GET_MSG_FLAGS_CMD;
+
+ smi_info->handlers->start_transaction(
+ smi_info->si_sm, msg, 2);
+ smi_info->si_state = SI_GETTING_FLAGS;
+ goto restart;
+ }
+
+ return si_sm_result;
+}
+
+static void sender(void *send_info,
+ struct ipmi_smi_msg *msg,
+ int priority)
+{
+ struct smi_info *smi_info = (struct smi_info *) send_info;
+ enum si_sm_result result;
+ unsigned long flags;
+ #ifdef DEBUG_TIMING
+ struct timeval t;
+ #endif
+
+ spin_lock_irqsave(&(smi_info->msg_lock), flags);
+ #ifdef DEBUG_TIMING
+ do_gettimeofday(&t);
+ printk("***Enqueue: %d.%9d\n", t.tv_sec, t.tv_usec);
+ #endif
+
+ if (smi_info->run_to_completion) {
+ /* If we are running to completion, then throw it in
+ the list and run transactions until everything is
+ clear. Priority doesn't matter here. */
+ list_add_tail(&(msg->link), &(smi_info->xmit_msgs));
+
+ /* We have to release the msg lock and claim the smi
+ lock in this case, because of race conditions. */
+ spin_unlock_irqrestore(&(smi_info->msg_lock), flags);
+
+ spin_lock_irqsave(&(smi_info->si_lock), flags);
+ result = smi_event_handler(smi_info, 0);
+ while (result != SI_SM_IDLE) {
+ udelay(SI_SHORT_TIMEOUT_USEC);
+ result = smi_event_handler(smi_info,
+ SI_SHORT_TIMEOUT_USEC);
+ }

```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```
+ spin_unlock_irqrestore(&(smi_info->si_lock), flags);
+ return;
+ } else {
+ if (priority > 0) {
+ list_add_tail(&(msg->link), &(smi_info->hp_xmit_msgs));
+ } else {
+ list_add_tail(&(msg->link), &(smi_info->xmit_msgs));
+ }
+ }
+ spin_unlock_irqrestore(&(smi_info->msg_lock), flags);
+
+ spin_lock_irqsave(&(smi_info->si_lock), flags);
+ if ((smi_info->si_state == SI_NORMAL)
+ && (smi_info->curr_msg == NULL))
+ {
+ start_next_msg(smi_info);
+ si_restart_short_timer(smi_info);
+ }
+ spin_unlock_irqrestore(&(smi_info->si_lock), flags);
+}
+
+static void set_run_to_completion(void *send_info, int i_run_to_completion)
+{
+ struct smi_info *smi_info = (struct smi_info *) send_info;
+ enum si_sm_result result;
+ unsigned long flags;
+
+ spin_lock_irqsave(&(smi_info->si_lock), flags);
+
+ smi_info->run_to_completion = i_run_to_completion;
+ if (i_run_to_completion) {
+ result = smi_event_handler(smi_info, 0);
+ while (result != SI_SM_IDLE) {
+ udelay(SI_SHORT_TIMEOUT_USEC);
+ result = smi_event_handler(smi_info,
+ SI_SHORT_TIMEOUT_USEC);
+ }
+ }
+
+ spin_unlock_irqrestore(&(smi_info->si_lock), flags);
+}
+
+static void request_events(void *send_info)
+{
+ struct smi_info *smi_info = (struct smi_info *) send_info;
+
+ atomic_set(&smi_info->req_events, 1);
+}
+
+static int initialized = 0;
+
```

```

+/* Must be called with interrupts off and with the si_lock held. */
+static void si_restart_short_timer(struct smi_info *smi_info)
+{
+#if defined(CONFIG_HIGH_RES_TIMERS)
+ unsigned long flags;
+ unsigned long jiffies_now;
+
+ if (del_timer(&(smi_info->si_timer))) {
+ /* If we don't delete the timer, then it will go off
+ immediately, anyway. So we only process if we
+ actually delete the timer. */
+
+ /* We already have irqsave on, so no need for it
+ here. */
+ read_lock(&xtime_lock);
+ jiffies_now = jiffies;
+ smi_info->si_timer.expires = jiffies_now;
+ smi_info->si_timer.sub_expires = get_arch_cycles(jiffies_now);
+
+ add_usec_to_timer(&smi_info->si_timer, SI_SHORT_TIMEOUT_USEC);
+
+ add_timer(&(smi_info->si_timer));
+ spin_lock_irqsave(&smi_info->count_lock, flags);
+ smi_info->timeout_restarts++;
+ spin_unlock_irqrestore(&smi_info->count_lock, flags);
+ }
+#endif
+}
+
+static void smi_timeout(unsigned long data)
+{
+ struct smi_info *smi_info = (struct smi_info *) data;
+ enum si_sm_result smi_result;
+ unsigned long flags;
+ unsigned long jiffies_now;
+ unsigned long time_diff;
+#ifdef DEBUG_TIMING
+ struct timeval t;
+#endif
+
+ if (smi_info->stop_operation) {
+ smi_info->timer_stopped = 1;
+ return;
+ }
+
+ spin_lock_irqsave(&(smi_info->si_lock), flags);
+#ifdef DEBUG_TIMING
+ do_gettimeofday(&t);
+ printk("***Timer: %d.%9d\n", t.tv_sec, t.tv_usec);
+#endif
+ jiffies_now = jiffies;

```

## Linux-Kernel: [PATCH] IPMI driver updates, part 2

```

+ time_diff = ((jiffies_now - smi_info->last_timeout_jiffies)
+ * SI_USEC_PER_JIFFY);
+ smi_result = smi_event_handler(smi_info, time_diff);
+
+ spin_unlock_irqrestore(&(smi_info->si_lock), flags);
+
+ smi_info->last_timeout_jiffies = jiffies_now;
+
+ if ((smi_info->irq) && (! smi_info->interrupt_disabled)) {
+ /* Running with interrupts, only do long timeouts. */
+ smi_info->si_timer.expires = jiffies + SI_TIMEOUT_JIFFIES;
+ spin_lock_irqsave(&smi_info->count_lock, flags);
+ smi_info->long_timeouts++;
+ spin_unlock_irqrestore(&smi_info->count_lock, flags);
+ goto do_add_timer;
+ }
+
+ /* If the state machine asks for a short delay, then shorten
+ the timer timeout. */
+ if (smi_result == SI_SM_CALL_WITH_DELAY) {
+ spin_lock_irqsave(&smi_info->count_lock, flags);
+ smi_info->short_timeouts++;
+ spin_unlock_irqrestore(&smi_info->count_lock, flags);
+ #if defined(CONFIG_HIGH_RES_TIMERS)
+ read_lock(&xtime_lock);
+ smi_info->si_timer.expires = jiffies;
+ smi_info->si_timer.sub_expires
+ = get_arch_cycles(smi_info->si_timer.expires);
+ read_unlock(&xtime_lock);
+ add_usec_to_timer(&smi_info->si_timer, SI_SHORT_TIMEOUT_USEC);
+ #else
+ smi_info->si_timer.expires = jiffies + 1;
+ #endif
+ } else {
+ spin_lock_irqsave(&smi_info->count_lock, flags);
+ smi_info->long_timeouts++;
+ spin_unlock_irqrestore(&smi_info->count_lock, flags);
+ smi_info->si_timer.expires = jiffies + SI_TIMEOUT_JIFFIES;
+ #if defined(CONFIG_HIGH_RES_TIMERS)
+ smi_info->si_timer.sub_expires = 0;
+ #endif
+ }
+
+ do_add_timer:
+ add_timer(&(smi_info->si_timer));
+ }
+
+ static irqreturn_t si_irq_handler(int irq, void *data, struct pt_regs *regs)
+ {
+ struct smi_info *smi_info = (struct smi_info *) data;
+ unsigned long flags;

```

```

+ #ifdef DEBUG_TIMING
+ struct timeval t;
+ #endif
+
+ spin_lock_irqsave(&(smi_info->si_lock), flags);
+
+ spin_lock(&smi_info->count_lock);
+ smi_info->interrupts++;
+ spin_unlock(&smi_info->count_lock);
+
+ if (smi_info->stop_operation)
+ goto out;
+
+ #ifdef DEBUG_TIMING
+ do_gettimeofday(&t);
+ printk("***Interrupt: %d.%9d\n", t.tv_sec, t.tv_usec);
+ #endif
+ smi_event_handler(smi_info, 0);
+ out:
+ spin_unlock_irqrestore(&(smi_info->si_lock), flags);
+ return IRQ_HANDLED;
+ }
+
+ static struct ipmi_smi_handlers handlers =
+ {
+ .owner = THIS_MODULE,
+ .sender = sender,
+ .request_events = request_events,
+ .set_run_to_completion = set_run_to_completion
+ };
+
+ /* There can be 4 IO ports passed in (with or without IRQs), 4 addresses,
+ a default IO port, and 1 ACPI/SPMI address. That sets SI_MAX_DRIVERS */
+
+ #define SI_MAX_PARAMS 4
+ #define SI_MAX_DRIVERS ((SI_MAX_PARAMS * 2) + 2)
+ static struct smi_info *smi_infos[SI_MAX_DRIVERS] =
+ { NULL, NULL, NULL, NULL };
+
+ #define DEVICE_NAME "ipmi_si"
+
+ #define DEFAULT_KCS_IO_PORT 0xca2
+ #define DEFAULT_SMIC_IO_PORT 0xca9
+ #define DEFAULT_BT_IO_PORT 0xe4
+
+ static int si_trydefaults = 1;
+ static char *si_type[SI_MAX_PARAMS] = { NULL, NULL, NULL, NULL };
+ static unsigned long si_addrs[SI_MAX_PARAMS] = { 0, 0, 0, 0 };
+ static unsigned int si_ports[SI_MAX_PARAMS] = { 0, 0, 0, 0 };
+ static int si_irqs[SI_MAX_PARAMS] = { 0, 0, 0, 0 };
+
+

```

```

+
+MODULE_PARM(si_trydefaults, "i");
+MODULE_PARM(si_type, "1-4s");
+MODULE_PARM(si_addrs, "1-4l");
+MODULE_PARM(si_irqs, "1-4i");
+MODULE_PARM(si_ports, "1-4i");
+
+
+
+#if defined(CONFIG_ACPI_INTERPETER) || defined(CONFIG_X86) || defined(CONFIG_PCI)
+#define IPMI_MEM_ADDR_SPACE 1
+#define IPMI_IO_ADDR_SPACE 2
+static int is_new_interface(int intf, u8 addr_space, unsigned long base_addr)
+{
+ int i;
+
+ for (i = 0; i < SI_MAX_PARAMS; ++i) {
+ /* Don't check our address. */
+ if (i == intf)
+ continue;
+ if (si_type[i] != NULL) {
+ if ((addr_space == IPMI_MEM_ADDR_SPACE &&
+ base_addr == si_addrs[i]) ||
+ (addr_space == IPMI_IO_ADDR_SPACE &&
+ base_addr == si_ports[i]))
+ return 0;
+ }
+ else
+ break;
+ }
+
+ return 1;
+}
+#endif
+
+static int std_irq_setup(struct smi_info *info)
+{
+ int rv;
+
+ if (!info->irq)
+ return 0;
+
+ rv = request_irq(info->irq,
+ si_irq_handler,
+ SA_INTERRUPT,
+ DEVICE_NAME,
+ info);
+ if (rv) {
+ printk(KERN_WARNING
+ "ipmi_si: %s unable to claim interrupt %d,"
+ " running polled\n",
+ DEVICE_NAME, info->irq);

```

```

+ info->irq = 0;
+ } else {
+ printk(" Using irq %d\n", info->irq);
+ }
+
+ return rv;
+}
+
+static void std_irq_cleanup(struct smi_info *info)
+{
+ if (!info->irq)
+ return;
+
+ free_irq(info->irq, info);
+}
+
+static unsigned char port_inb(struct si_sm_io *io, unsigned int offset)
+{
+ unsigned int *addr = io->info;
+
+ return inb((*addr)+offset);
+}
+
+static void port_outb(struct si_sm_io *io, unsigned int offset,
+ unsigned char b)
+{
+ unsigned int *addr = io->info;
+
+ outb(b, (*addr)+offset);
+}
+
+static int port_setup(struct smi_info *info)
+{
+ unsigned int *addr = info->io.info;
+
+ if (!addr || (!*addr))
+ return -ENODEV;
+
+ if (request_region(*addr, info->io_size, DEVICE_NAME) == NULL)
+ return -EIO;
+ return 0;
+}
+
+static void port_cleanup(struct smi_info *info)
+{
+ unsigned int *addr = info->io.info;
+
+ if (addr && (*addr))
+ release_region (*addr, info->io_size);
+ kfree(info);
+}

```

```

+
+static int try_init_port(int intf_num, struct smi_info **new_info)
+{
+ struct smi_info *info;
+
+ if (!si_ports[intf_num])
+ return -ENODEV;
+
+ if (!is_new_interface(intf_num, IPMI_IO_ADDR_SPACE,
+ si_ports[intf_num]))
+ return -ENODEV;
+
+ info = kmalloc(sizeof(*info), GFP_KERNEL);
+ if (!info) {
+ printk(KERN_ERR "ipmi_si: Could not allocate SI data (1)\n");
+ return -ENOMEM;
+ }
+ memset(info, 0, sizeof(*info));
+
+ info->io_setup = port_setup;
+ info->io_cleanup = port_cleanup;
+ info->io.inputb = port_inb;
+ info->io.outputb = port_outb;
+ info->io.info = &(si_ports[intf_num]);
+ info->io.addr = NULL;
+ info->irq = 0;
+ info->irq_setup = NULL;
+ *new_info = info;
+
+ printk("ipmi_si: Trying \"%s\" at I/O port 0x%x\n",
+ si_type[intf_num], si_ports[intf_num]);
+ return 0;
+}
+
+static unsigned char mem_inb(struct si_sm_io *io, unsigned int offset)
+{
+ return readb((io->addr)+offset);
+}
+
+static void mem_outb(struct si_sm_io *io, unsigned int offset,
+ unsigned char b)
+{
+ writeb(b, (io->addr)+offset);
+}
+
+static int mem_setup(struct smi_info *info)
+{
+ unsigned long *addr = info->io.info;
+
+ if (!addr || (!*addr))
+ return -ENODEV;

```

```

+
+ if (request_mem_region(*addr, info->io_size, DEVICE_NAME) == NULL)
+ return -EIO;
+
+ info->io.addr = ioremap(*addr, info->io_size);
+ if (info->io.addr == NULL) {
+ release_mem_region(*addr, info->io_size);
+ return -EIO;
+ }
+ return 0;
+}
+
+static void mem_cleanup(struct smi_info *info)
+{
+ unsigned long *addr = info->io.info;
+
+ if (info->io.addr) {
+ iounmap(info->io.addr);
+ release_mem_region(*addr, info->io_size);
+ }
+ kfree(info);
+}
+
+static int try_init_mem(int intf_num, struct smi_info **new_info)
+{
+ struct smi_info *info;
+
+ if (!si_addrs[intf_num])
+ return -ENODEV;
+
+ if (!is_new_interface(intf_num, IPMI_MEM_ADDR_SPACE,
+ si_addrs[intf_num]))
+ return -ENODEV;
+
+ info = kmalloc(sizeof(*info), GFP_KERNEL);
+ if (!info) {
+ printk(KERN_ERR "ipmi_si: Could not allocate SI data (2)\n");
+ return -ENOMEM;
+ }
+ memset(info, 0, sizeof(*info));
+
+ info->io_setup = mem_setup;
+ info->io_cleanup = mem_cleanup;
+ info->io.inputb = mem_inb;
+ info->io.outputb = mem_outb;
+ info->io.info = (void *) si_addrs[intf_num];
+ info->io.addr = NULL;
+ info->irq = 0;
+ info->irq_setup = NULL;
+ *new_info = info;
+}

```



```

+ info->irq,
+ ACPI_EVENT_LEVEL_TRIGGERED,
+ ipmi_acpi_gpe,
+ info);
+ if (status != AE_OK) {
+ printk(KERN_WARNING
+ "ipmi_si: %s unable to claim ACPI GPE %d,"
+ " running polled\n",
+ DEVICE_NAME, info->irq);
+ info->irq = 0;
+ return -EINVAL;
+ } else {
+ printk(" Using ACPI GPE %d\n", info->irq);
+ return 0;
+ }
+
+
+static void acpi_gpe_irq_cleanup(struct smi_info *info)
+{
+ if (!info->irq)
+ return;
+
+ acpi_remove_gpe_handler(NULL, info->irq, ipmi_acpi_gpe);
+}
+
+/*
+ * Defined at
+ * http://h21007.www2.hp.com/dspp/files/unprotected/devresource/Docs/TechPapers/IA64/hpspmi.pdf
+ */
+struct SPMITable {
+ s8 Signature[4];
+ u32 Length;
+ u8 Revision;
+ u8 Checksum;
+ s8 OEMID[6];
+ s8 OEMTableID[8];
+ s8 OEMRevision[4];
+ s8 CreatorID[4];
+ s8 CreatorRevision[4];
+ u8 InterfaceType;
+ u8 IPMIlegacy;
+ s16 SpecificationRevision;
+
+ /*
+ * Bit 0 – SCI interrupt supported
+ * Bit 1 – I/O APIC/SAPIC
+ */
+ u8 InterruptType;
+
+ /* If bit 0 of InterruptType is set, then this is the SCI

```

```

+ interrupt in the GPEx_STS register. */
+ u8 GPE;
+
+ s16 Reserved;
+
+ /* If bit 1 of InterruptType is set, then this is the I/O
+ APIC/SAPIC interrupt. */
+ u32 GlobalSystemInterrupt;
+
+ /* The actual register address. */
+ struct acpi_generic_address addr;
+
+ u8 UID[4];
+
+ s8 spmi_id[1]; /* A '\0' terminated array starts here. */
+};
+
+static int try_init_acpi(int intf_num, struct smi_info **new_info)
+{
+ struct smi_info *info;
+ acpi_status status;
+ struct SPMITable *spmi;
+ char *io_type;
+ u8 addr_space;
+
+ if (acpi_failure)
+ return -ENODEV;
+
+ status = acpi_get_firmware_table("SPMI", intf_num+1,
+ ACPI_LOGICAL_ADDRESSING,
+ (struct acpi_

```