92

```
/*
 * eeh.c
 * Copyright (C) 2001 Dave Engebretsen & Todd Inglett IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

/* Change Activity:
 * 2001/10/27 : engebret : Created.
 * End Change Activity
 */

#include <linux/init.h>
#include <linux/pci.h>
#include <linux/proc_fs.h>
#include <linux/bootmem.h>
#include <asm/paca.h>
#include <asm/processor.h>
#include <asm/naca.h>
#include <asm/io.h>
#include "pci.h"

#define BUID_HI(buid) ((buid) >> 32)
#define BUID_LO(buid) ((buid) & 0xffffffff)
#define CONFIG_ADDR(busno, devfn) (((((busno) & 0xff) << 8) | ((devfn) & 0xf8)) << 8)

unsigned long eeh_total_mmio_ffs;
unsigned long eeh_false_positives;
/* RTAS tokens */
static int ibm_set_eeh_option;
static int ibm_set_slot_reset;
static int ibm_read_slot_reset_state;

int eeh_implemented;
#define EEH_MAX_OPTS 4096
static char *eeh_opts;
static int eeh_opts_last;
static int eeh_check_opts_config(struct pci_dev *dev, int default_state);


unsigned long eeh_token(unsigned long phb, unsigned long bus, unsigned long devfn, unsigned long offset)
{
        if (phb > 0xff)
                panic("eeh_token: phb 0x%lx is too large\n", phb);
        if (offset & 0x0fffffff00000000)
                panic("eeh_token: offset 0x%lx is out of range\n", offset);
        return ((IO_UNMAPPED_REGION_ID << 60) | (phb << 48UL) | ((bus & 0xff) << 40UL) | (devfn << 32UL) | (offse
t & 0xffffffff));
}

int eeh_get_state(unsigned long ea)
{
        return 0;
}

/* Check for an eeh failure at the given token address.
 * The given value has been read and it should be 1's (0xff, 0xffff or
 * 0xffffffff).
 *
 * Probe to determine if an error actually occurred.  If not return val.
 * Otherwise panic.
 */
unsigned long eeh_check_failure(void *token, unsigned long val)
{
        unsigned long config_addr = (unsigned long)token >> 24; /* PPBBDDRR */
        unsigned long phbidx = (config_addr >> 24) & 0xff;
        struct pci_controller *phb;
        unsigned long ret, rets[2];

        config_addr &= 0xffff00;  /* 00BBDD00 */

        if (phbidx >= global_phb_number) {
                panic("EEH: checking token %p phb index of %ld is greater than max of %d\n", token, phbidx, global_phb_number-1);
        }
        phb = phbtab[phbidx];

        ret = rtas_call(ibm_read_slot_reset_state, 3, 3, rets,
                        config_addr, BUID_HI(phb->buid), BUID_LO(phb->buid));
```

```
90          if (ret == 0 && rets[1] == 1 && rets[0] >= 2) {
91                  struct pci_dev *dev;
92                  int bus = ((unsigned long)token >> 40) & 0xffff; /* include PHB# in bus */
93                  int devfn = (config_addr >> 8) & 0xff;
94
95                  dev = pci_find_slot(bus, devfn);
96                  if (dev) {
97                          printk(KERN_ERR "EEH:  MMIO failure (%ld) on device:\n  %s %s\n",
98                                 rets[0], dev->slot_name, dev->name);
99                          PPCDBG_ENTER_DEBUGGER();
100                         panic("EEH:  MMIO failure (%ld) on device:\n  %s %s\n",
101                                rets[0], dev->slot_name, dev->name);
102                 } else {
103                         printk(KERN_ERR "EEH:  MMIO failure (%ld) on device buid %lx, config_addr %lx\n", rets[0], phb->buid, conf
    ig_addr);
104                         PPCDBG_ENTER_DEBUGGER();
105                         panic("EEH:  MMIO failure (%ld) on device buid %lx, config_addr %lx\n", rets[0], phb->buid, config_addr);
106                 }
107         }
108         eeh_false_positives++;
109         return val;     /* good case */
110 }
111
112 struct eeh_early_enable_info {
113         unsigned int buid_hi;
114         unsigned int buid_lo;
115         int adapters_enabled;
116 };
117
118 /* Enable eeh for the given device node. */
119 static void *early_enable_eeh(struct device_node *dn, void *data)
120 {
121         struct eeh_early_enable_info *info = data;
122         long ret;
123
124         /* Try to enable eeh */
125         ret = rtas_call(ibm_set_eeh_option, 4, 1, NULL,
126                         CONFIG_ADDR(dn->busno, dn->devfn),
127                         info->buid_hi, info->buid_lo, EEH_ENABLE);
128         if (ret == 0)
129                 info->adapters_enabled++;
130         return NULL;
131 }
132
133 /*
134  * Initialize eeh by trying to enable it for all of the adapters in the system.
135  * As a side effect we can determine here if eeh is supported at all.
136  * Note that we leave EEH on so failed config cycles won't cause a machine
137  * check.  If a user turns off EEH for a particular adapter they are really
138  * telling Linux to ignore errors.
139  *
140  * We should probably distinguish between "ignore errors" and "turn EEH off"
141  * but for now disabling EEH for adapters is mostly to work around drivers that
142  * directly access mmio space (without using the macros).
143  *
144  * The eeh-force-off/on option does literally what it says, so if Linux must
145  * avoid enabling EEH this must be done.
146  */
147 void eeh_init(void)
148 {
149         struct device_node *phb;
150         struct eeh_early_enable_info info;
151
152         extern char cmd_line[]; /* Very early cmd line parse.  Cheap, but works. */
153         char *eeh_force_off = strstr(cmd_line, "eeh-force-off");
154         char *eeh_force_on = strstr(cmd_line, "eeh-force-on");
155
156         ibm_set_eeh_option = rtas_token("ibm,set-eeh-option");
157         ibm_set_slot_reset = rtas_token("ibm,set-slot-reset");
158         ibm_read_slot_reset_state = rtas_token("ibm,read-slot-reset-state");
159
160         if (ibm_set_eeh_option == RTAS_UNKNOWN_SERVICE)
161                 return;
162
163         if (eeh_force_off > eeh_force_on) {
164                 /* User is forcing EEH off.  Be noisy if it is implemented. */
165                 if (eeh_implemented)
166                         printk(KERN_WARNING "EEH: WARNING: PCI Enhanced I/O Error Handling is user disabled\n");
167                 eeh_implemented = 0;
168                 return;
169         }
170
171         if (eeh_force_on > eeh_force_off)
172                 eeh_implemented = 1;    /* User is forcing it on. */
173
174         /* Enable EEH for all adapters.  Note that eeh requires buid's */
175         info.adapters_enabled = 0;
176         for (phb = find_devices("pci"); phb; phb = phb->next) {
177                 int len;
178                 int *buid_vals = (int *) get_property(phb, "ibm,fw-phb-id", &len);
```

```
179                     if (!buid_vals)
180                             continue;
181                     if (len == sizeof(int)) {
182                             info.buid_lo = buid_vals[0];
183                             info.buid_hi = 0;
184                     } else if (len == sizeof(int)*2) {
185                             info.buid_hi = buid_vals[0];
186                             info.buid_lo = buid_vals[1];
187                     } else {
188                             printk("EEH: odd ibm,fw−phb−id len returned: %d\n", len);
189                             continue;
190                     }
191                     traverse_pci_devices(phb, early_enable_eeh, NULL, &info);
192             }
193             if (info.adapters_enabled) {
194                     printk(KERN_INFO "EEH: PCI Enhanced I/O Error Handling Enabled\n");
195                     eeh_implemented = 1;
196             }
197     }
198
199
200     /* Given a PCI device check if eeh should be configured or not.
201      * This may look at firmware properties and/or kernel cmdline options.
202      */
203     int is_eeh_configured(struct pci_dev *dev)
204     {
205             struct device_node *dn = pci_device_to_OF_node(dev);
206             struct pci_controller *phb = PCI_GET_PHB_PTR(dev);
207             unsigned long ret, rets[2];
208             int eeh_capable;
209             int default_state = 1;  /* default enable EEH if we can. */
210
211             if (dn == NULL || phb == NULL || !eeh_implemented)
212                     return 0;
213
214             /* Hack: turn off eeh for display class devices by default.
215              * This fixes matrox accel framebuffer.
216              */
217             if ((dev−>class >> 16) == PCI_BASE_CLASS_DISPLAY)
218                     default_state = 0;
219
220             /* Ignore known PHBs and EADs bridges */
221             if (dev−>vendor == PCI_VENDOR_ID_IBM &&
222                 (dev−>device == 0x0102 || dev−>device == 0x008b))
223                     default_state = 0;
224
225             if (!eeh_check_opts_config(dev, default_state)) {
226                     if (default_state)
227                             printk(KERN_INFO "EEH: %s %s user requested to run without EEH.\n", dev−>slot_name, dev−>name);
228                     return 0;
229             }
230
231             ret = rtas_call(ibm_read_slot_reset_state, 3, 3, rets,
232                             CONFIG_ADDR(dn−>busno, dn−>devfn),
233                             BUID_HI(phb−>buid), BUID_LO(phb−>buid));
234             eeh_capable = (ret == 0 && rets[1] == 1);
235             return eeh_capable;
236     }
237
238     int eeh_set_option(struct pci_dev *dev, int option)
239     {
240             struct device_node *dn = pci_device_to_OF_node(dev);
241             struct pci_controller *phb = PCI_GET_PHB_PTR(dev);
242
243             if (dn == NULL || phb == NULL || phb−>buid == 0 || !eeh_implemented)
244                     return −2;
245
246             return rtas_call(ibm_set_eeh_option, 4, 1, NULL,
247                              CONFIG_ADDR(dn−>busno, dn−>devfn),
248                              BUID_HI(phb−>buid), BUID_LO(phb−>buid), option);
249     }
250
251
252     static int eeh_proc_falsepositive_read(char *page, char **start, off_t off,
253                             int count, int *eof, void *data)
254     {
255             int len;
256             len = sprintf(page, "eeh_false_positives=%ld\n"
257                           "eeh_total_mmio_ffs=%ld\n",
258                           eeh_false_positives, eeh_total_mmio_ffs);
259             return len;
260     }
261
262     /* Implementation of /proc/ppc64/eeh
263      * For now it is one file showing false positives.
264      */
265     static int __init eeh_init_proc(void)
266     {
267             struct proc_dir_entry *ent = create_proc_entry("ppc64/eeh", S_IRUGO, 0);
268             if (ent) {
```

```
269                     ent->nlink = 1;
270                     ent->data = NULL;
271                     ent->read_proc = (void *)eeh_proc_falseposition_read;
272             }
273             return 0;
274     }
275
276     /*
277      * Test if "dev" should be configured on or off.
278      * This processes the options literally from left to right.
279      * This lets the user specify stupid combinations of options,
280      * but at least the result should be very predictable.
281      */
282     static int eeh_check_opts_config(struct pci_dev *dev, int default_state)
283     {
284             struct device_node *dn = pci_device_to_OF_node(dev);
285             struct pci_controller *phb = PCI_GET_PHB_PTR(dev);
286             char devname[32], classname[32], phbname[32];
287             char *strs[8], *s;
288             int nstrs, i;
289             int ret = default_state;
290
291             if (dn == NULL || phb == NULL)
292                     return 0;
293             /* Build list of strings to match */
294             nstrs = 0;
295             s = (char *)get_property(dn, "ibm,loc−code", 0);
296             if (s)
297                     strs[nstrs++] = s;
298             sprintf(devname, "dev%04x:%04x", dev->vendor, dev->device);
299             strs[nstrs++] = devname;
300             sprintf(classname, "class%04x", dev->class);
301             strs[nstrs++] = classname;
302             sprintf(phbname, "pci@%lx", phb->buid);
303             strs[nstrs++] = phbname;
304             strs[nstrs++] = "";     /* yes, this matches the empty string */
305
306             /* Now see if any string matches the eeh_opts list.
307              * The eeh_opts list entries start with + or −.
308              */
309             for (s = eeh_opts; s && (s < (eeh_opts + eeh_opts_last)); s += strlen(s)+1) {
310                     for (i = 0; i < nstrs; i++) {
311                             if (strcasecmp(strs[i], s+1) == 0) {
312                                     ret = (strs[i][0] == '+') ? 1 : 0;
313                             }
314                     }
315             }
316             return ret;
317     }
318
319     /* Handle kernel eeh−on & eeh−off cmd line options for eeh.
320      *
321      * We support:
322      *      eeh−off=loc1,loc2,loc3...
323      *
324      * and this option can be repeated so
325      *      eeh−off=loc1,loc2 eeh−off=loc3
326      * is the same as eeh−off=loc1,loc2,loc3
327      *
328      * loc is an IBM location code that can be found in a manual or
329      * via openfirmware (or the Hardware Management Console).
330      *
331      * We also support these additional "loc" values:
332      *
333      *      dev#:#    vendor:device id in hex (e.g. dev1022:2000)
334      *      class#    class id in hex (e.g. class0200)
335      *      pci@buid  all devices under phb (e.g. pci@fef00000)
336      *
337      * If no location code is specified all devices are assumed
338      * so eeh−off means eeh by default is off.
339      */
340
341     /* This is implemented as a null separated list of strings.
342      * Each string looks like this:  "+X" or "−X"
343      * where X is a loc code, dev, class or pci string (as shown above)
344      * or empty which is used to indicate all.
345      *
346      * We interpret this option string list during the buswalk
347      * so that it will literally behave left−to−right even if
348      * some combinations don't make sense.  Give the user exactly
349      * what they want! :)
350      */
351
352     static int __init eeh_parm(char *str, int state)
353     {
354             char *s, *cur, *curend;
355             if (!eeh_opts) {
356                     eeh_opts = alloc_bootmem(EEH_MAX_OPTS);
357                     eeh_opts[eeh_opts_last++] = '+'; /* default */
358                     eeh_opts[eeh_opts_last++] = '\0';
```

```
359             }
360             if (*str == '\0') {
361                     eeh_opts[eeh_opts_last++] = state ? '+' : '−';
362                     eeh_opts[eeh_opts_last++] = '\0';
363                     return 1;
364             }
365             if (*str == '=')
366                     str++;
367             for (s = str; s && *s != '\0'; s = curend) {
368                     cur = s;
369                     while (*cur == ',')
370                             cur++;  /* ignore empties.  Don't treat as "all−on" or "all−off" */
371                     curend = strchr(cur, ',');
372                     if (!curend)
373                             curend = cur + strlen(cur);
374                     if (*cur) {
375                             int curlen = curend−cur;
376                             if (eeh_opts_last + curlen > EEH_MAX_OPTS−2) {
377                                     printk(KERN_INFO "EEH: sorry...too many eeh cmd line options\n");
378                                     return 1;
379                             }
380                             eeh_opts[eeh_opts_last++] = state ? '+' : '−';
381                             strncpy(eeh_opts+eeh_opts_last, cur, curlen);
382                             eeh_opts_last += curlen;
383                             eeh_opts[eeh_opts_last++] = '\0';
384                     }
385             }
386             return 1;
387     }
388
389     static int __init eehoff_parm(char *str)
390     {
391             return eeh_parm(str, 0);
392     }
393
394     static int __init eehon_parm(char *str)
395     {
396             return eeh_parm(str, 1);
397     }
398
399     __initcall(eeh_init_proc);
400     __setup("eeh−off", eehoff_parm);
401     __setup("eeh−on", eehon_parm);
```

```
/*****************************************************************************
 * flight_recorder.c
 *****************************************************************************
 * This code supports the a generic flight recorder.                         *
 * Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                         *
 *                                                                           *
 * 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 program is distributed in the hope that it will be useful,           *
 * but WITHOUT ANY WARRANTY; without even the implied warranty of            *
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the             *
 * GNU General Public License for more details.                              *
 *                                                                           *
 * 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.,                                           *
 * 59 Temple Place, Suite 330,                                               *
 * Boston, MA  02111-1307  USA                                               *
 *****************************************************************************
 * This is a simple text based flight recorder.  Useful for logging
 * information the you may want to retreive at a latter time.  Errors or
 * debug inforamtion are good examples.   A good method to dump the
 * information is via the proc file system.
 *
 * To use.
 * 1. Create the flight recorder object.  Passing a NULL pointer will
 *    kmalloc the space for you.  If it is too early for kmalloc, create
 *    space for the object.   Beware, don't lie about the size, you will
 *    pay for that later.
 *              FlightRecorder* TestFr = alloc_Flight_Recorder(NULL,"TestFr",4096);
 *
 * 2. Log any notable events, initialzation, error conditions, etc.
 *              LOGFR(TestFr,"5. Stack Variable(10) %d",StackVariable);
 *
 * 3. Dump the information to a buffer.
 *              fr_Dump(TestFr, proc_file_buffer, proc_file_buffer_size);
 *
 *****************************************************************************/
#include <stdarg.h>
#include <linux/kernel.h>
#include <linux/rtc.h>
#include <linux/slab.h>
#include <asm/string.h>
#include <asm/time.h>
#include <asm/flight_recorder.h>

static  char       LogText[512];
static  int        LogTextIndex;
static  int        LogCount = 0;
static  spinlock_t Fr_Lock;

/*****************************************************************************
 * Build the log time prefix based on Flags.
 *  00 = No time prefix
 *  01 = Date(mmddyy) Time(hhmmss) prefix
 *  02 = Day(dd) Time(hhmmss) prefix
 *  03 = Time(hhmmss) prefix
 *****************************************************************************/
static void fr_Log_Time(FlightRecorder* Fr)
{
        struct  timeval  TimeClock;
        struct  rtc_time LogTime;

        do_gettimeofday(&TimeClock);
        to_tm(TimeClock.tv_sec, &LogTime);

        if (Fr->Flags == 1) {
                LogTextIndex = sprintf(LogText,"%02d%02d%02d %02d%02d%02d ",
                                       LogTime.tm_mon, LogTime.tm_mday, LogTime.tm_year-2000,
                                LogTime.tm_hour,LogTime.tm_min, LogTime.tm_sec);
        }
        else if (Fr->Flags == 2) {
                LogTextIndex = sprintf(LogText,"%02d %02d%02d%02d ",
                                       LogTime.tm_mday,
                                LogTime.tm_hour,LogTime.tm_min, LogTime.tm_sec);
        }

        else if (Fr->Flags == 3) {
                LogTextIndex = sprintf(LogText,"%02d%02d%02d ",
                                LogTime.tm_hour,LogTime.tm_min, LogTime.tm_sec);
        }
        else {
                ++LogCount;
                LogTextIndex = sprintf(LogText,"%04d. ",LogCount);
        }
}

```

```
91  /*********************************************************************/
92  /* Log entry into buffer,                                            */
93  /* ->If entry is going to wrap, log "WRAP" and start at the top.     */
94  /*********************************************************************/
95  static void fr_Log_Data(FlightRecorder* Fr)
96  {
97          int     TextLen  = strlen(LogText);
98          int     Residual = ( Fr->EndPointer − Fr->NextPointer)−15;
99          if (TextLen > Residual) {
100                 strcpy(Fr->NextPointer,"WRAP");
101                 Fr->WrapPointer = Fr->NextPointer + 5;
102                 Fr->NextPointer = Fr->StartPointer;
103         }
104         strcpy(Fr->NextPointer,LogText);
105         Fr->NextPointer += TextLen+1;
106         strcpy(Fr->NextPointer,"<=");
107 }
108 /*********************************************************************
109  * Build the log text, support variable args.
110  *********************************************************************/
111 void fr_Log_Entry(struct flightRecorder* LogFr, const char *fmt, ...)
112 {
113         va_list arg_ptr;
114         spin_lock(&Fr_Lock);
115
116         fr_Log_Time(LogFr);
117         va_start(arg_ptr, fmt);
118         vsprintf(LogText+LogTextIndex, fmt, arg_ptr);
119         va_end(arg_ptr);
120         fr_Log_Data(LogFr);
121
122         spin_unlock(&Fr_Lock);
123
124 }
125 /*********************************************************************
126  * Dump Flight Recorder into buffer.
127  * -> Handles the buffer wrapping.
128  *********************************************************************/
129 int fr_Dump(FlightRecorder* Fr, char *Buffer, int BufferLen)
130 {
131         int   LineLen = 0;
132         char* StartEntry;
133         char* EndEntry;
134         spin_lock(&Fr_Lock);
135         /*****************************************************************
136          * If Buffer has wrapped, find last usable entry to start with.
137          *****************************************************************/
138         if (Fr->WrapPointer != NULL) {
139                 StartEntry  = Fr->NextPointer+3;
140                 StartEntry += strlen(StartEntry)+1;
141                 EndEntry    = Fr->WrapPointer;
142
143                 while (EndEntry > StartEntry && LineLen < BufferLen) {
144                         LineLen    += sprintf(Buffer+LineLen,"%s\n",StartEntry);
145                         StartEntry += strlen(StartEntry) + 1;
146                 }
147         }
148
149         /*****************************************************************
150          * Dump from the beginning to the last logged entry
151          *****************************************************************/
152         StartEntry = Fr->StartPointer;
153         EndEntry   = Fr->NextPointer;
154         while (EndEntry > StartEntry && LineLen < BufferLen) {
155                 LineLen    += sprintf(Buffer+LineLen,"%s\n",StartEntry);
156                 StartEntry += strlen(StartEntry) + 1;
157         }
158         spin_unlock(&Fr_Lock);
159         return LineLen;
160 }
161
162 /*********************************************************************
163  * Allocate and Initialized the Flight Recorder
164  * -> If no FlightRecorder pointer is passed, the space is kmalloc.
165  *********************************************************************/
166 FlightRecorder* alloc_Flight_Recorder(FlightRecorder* FrPtr, char* Signature, int SizeOfFr)
167 {
168         FlightRecorder* Fr     = FrPtr;                        /* Pointer to Object */
169         int             FrSize = (SizeOfFr/16)*16;      /* Could be static   */
170         if (Fr == NULL)
171                 Fr = (FlightRecorder*)kmalloc(SizeOfFr, GFP_KERNEL);
172         memset(Fr,0,SizeOfFr);
173         strcpy(Fr->Signature,Signature);
174         Fr->Size         = FrSize;
175         Fr->Flags        = 0;
176         Fr->StartPointer = (char*)&Fr->Buffer;
177         Fr->EndPointer   = (char*)Fr + Fr->Size;
178         Fr->NextPointer  = Fr->StartPointer;
179
180         fr_Log_Entry(Fr,"Initialized.");
```

```
181             return Fr;
182     }
```

```
/*
 * PowerPC64 port by Mike Corrigan and Dave Engebretsen
 *   {mikejc|engebret}@us.ibm.com
 *
 *    Copyright (c) 2000 Mike Corrigan <mikejc@us.ibm.com>
 *
 * SMP scalability work:
 *    Copyright (C) 2001 Anton Blanchard <anton@au.ibm.com>, IBM
 *
 *    Module name: htab.c
 *
 *    Description:
 *      PowerPC Hashed Page Table functions
 *
 * 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.
 */

#include <linux/config.h>
#include <linux/spinlock.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/proc_fs.h>
#include <linux/stat.h>
#include <linux/sysctl.h>
#include <linux/ctype.h>
#include <linux/cache.h>

#include <asm/ppcdebug.h>
#include <asm/processor.h>
#include <asm/pgtable.h>
#include <asm/mmu.h>
#include <asm/mmu_context.h>
#include <asm/page.h>
#include <asm/types.h>
#include <asm/uaccess.h>
#include <asm/naca.h>
#include <asm/pmc.h>
#include <asm/machdep.h>
#include <asm/lmb.h>
#include <asm/abs_addr.h>
#include <asm/io.h>
#include <asm/eeh.h>
#include <asm/hvcall.h>
#include <asm/iSeries/LparData.h>
#include <asm/iSeries/HvCallHpt.h>

/*
 * Note:  pte   --> Linux PTE
 *        HPTE  --> PowerPC Hashed Page Table Entry
 *
 * Execution context:
 *   htab_initialize is called with the MMU off (of course), but
 *   the kernel has been copied down to zero so it can directly
 *   reference global data.  At this point it is very difficult
 *   to print debug info.
 *
 */

HTAB htab_data = {NULL, 0, 0, 0, 0};

extern unsigned long _SDR1;
extern unsigned long klimit;

void make_pte(HPTE *htab, unsigned long va, unsigned long pa,
              int mode, unsigned long hash_mask, int large);
long plpar_pte_enter(unsigned long flags,
                     unsigned long ptex,
                     unsigned long new_pteh, unsigned long new_ptel,
                     unsigned long *old_pteh_ret, unsigned long *old_ptel_ret);
static long hpte_remove(unsigned long hpte_group);
static long rpa_lpar_hpte_remove(unsigned long hpte_group);
static long iSeries_hpte_remove(unsigned long hpte_group);
inline unsigned long get_lock_slot(unsigned long vpn);

static spinlock_t pSeries_tlbie_lock = SPIN_LOCK_UNLOCKED;
static spinlock_t pSeries_lpar_tlbie_lock = SPIN_LOCK_UNLOCKED;

#define LOCK_SPLIT
#ifdef LOCK_SPLIT
hash_table_lock_t hash_table_lock[128] __cacheline_aligned_in_smp = { [0 ... 31] = {SPIN_LOCK_UNLOCKED}};
#else
hash_table_lock_t hash_table_lock[1] __cacheline_aligned_in_smp = { [0] = {SPIN_LOCK_UNLOCKED}};
#endif

#define KB (1024)
#define MB (1024*KB)

```

```
91  static inline void
92  loop_forever(void)
93  {
94          volatile unsigned long x = 1;
95          for(;x;x|=1)
96                  ;
97  }
98
99  static inline void
100 create_pte_mapping(unsigned long start, unsigned long end,
101                    unsigned long mode, unsigned long mask, int large)
102 {
103         unsigned long addr;
104         HPTE *htab = (HPTE *)__v2a(htab_data.htab);
105         unsigned int step;
106
107         if (large)
108                 step = 16*MB;
109         else
110                 step = 4*KB;
111
112         for (addr = start; addr < end; addr += step) {
113                 unsigned long vsid = get_kernel_vsid(addr);
114                 unsigned long va = (vsid << 28) | (addr & 0xfffffff);
115                 make_pte(htab, va, (unsigned long)__v2a(addr),
116                          mode, mask, large);
117         }
118 }
119
120 void
121 htab_initialize(void)
122 {
123         unsigned long table, htab_size_bytes;
124         unsigned long pteg_count;
125         unsigned long mode_rw, mask, lock_shift;
126
127 #if 0
128         /* Can't really do the call below since it calls the normal RTAS
129          * entry point and we're still relocate off at the moment.
130          * Temporarily diabling until it can call through the relocate off
131          * RTAS entry point.  −Peter
132          */
133         ppc64_boot_msg(0x05, "htab init");
134 #endif
135         /*
136          * Calculate the required size of the htab.  We want the number of
137          * PTEGs to equal one half the number of real pages.
138          */
139         htab_size_bytes = 1UL << naca->pftSize;
140         pteg_count = htab_size_bytes >> 7;
141
142         /* For debug, make the HTAB 1/8 as big as it normally would be. */
143         ifppcdebug(PPCDBG_HTABSIZE) {
144                 pteg_count >>= 3;
145                 htab_size_bytes = pteg_count << 7;
146         }
147
148         htab_data.htab_num_ptegs = pteg_count;
149         htab_data.htab_hash_mask = pteg_count − 1;
150
151         /*
152          * Calculate the number of bits to shift the pteg selector such that we
153          * use the high order 8 bits to select a page table lock.
154          */
155         asm ("cntlzd %0,%1" : "=r" (lock_shift) :
156              "r" (htab_data.htab_hash_mask));
157         htab_data.htab_lock_shift = (64 − lock_shift) − 8;
158
159         if(systemcfg->platform == PLATFORM_PSERIES) {
160                 /* Find storage for the HPT.  Must be contiguous in
161                  * the absolute address space.
162                  */
163                 table = lmb_alloc(htab_size_bytes, htab_size_bytes);
164                 if ( !table ) {
165                         ppc64_terminate_msg(0x20, "hpt space");
166                         loop_forever();
167                 }
168                 htab_data.htab = (HPTE *)__a2v(table);
169
170                 /* htab absolute addr + encoded htabsize */
171                 _SDR1 = table + __ilog2(pteg_count) − 11;
172
173                 /* Initialize the HPT with no entries */
174                 memset((void *)table, 0, htab_size_bytes);
175         } else {
176                 /* Using a hypervisor which owns the htab */
177                 htab_data.htab = NULL;
178                 _SDR1 = 0;
179         }
180
```

```
181         mode_rw = _PAGE_ACCESSED | _PAGE_COHERENT | PP_RWXX;
182         mask = pteg_count−1;
183
184         /* XXX we currently map kernel text rw, should fix this */
185         if ((systemcfg−>platform & PLATFORM_PSERIES) &&
186            cpu_has_largepage() && (systemcfg−>physicalMemorySize > 256*MB)) {
187                 create_pte_mapping((unsigned long)KERNELBASE,
188                                    KERNELBASE + 256*MB, mode_rw, mask, 0);
189                 create_pte_mapping((unsigned long)KERNELBASE + 256*MB,
190                                    KERNELBASE + (systemcfg−>physicalMemorySize),
191                                    mode_rw, mask, 1);
192         } else {
193                 create_pte_mapping((unsigned long)KERNELBASE,
194                                    KERNELBASE+(systemcfg−>physicalMemorySize),
195                                    mode_rw, mask, 0);
196         }
197 #if 0
198         /* Can't really do the call below since it calls the normal RTAS
199          * entry point and we're still relocate off at the moment.
200          * Temporarily diabling until it can call through the relocate off
201          * RTAS entry point.  −Peter
202          */
203         ppc64_boot_msg(0x06, "htab done");
204 #endif
205 }
206 #undef KB
207 #undef MB
208
209 /*
210  * Create a pte. Used during initialization only.
211  * We assume the PTE will fit in the primary PTEG.
212  */
213 void make_pte(HPTE *htab, unsigned long va, unsigned long pa,
214               int mode, unsigned long hash_mask, int large)
215 {
216         HPTE *hptep, local_hpte, rhpte;
217         unsigned long hash, vpn, flags, lpar_rc;
218         unsigned long i, dummy1, dummy2;
219         long slot;
220
221         if (large)
222                 vpn = va >> LARGE_PAGE_SHIFT;
223         else
224                 vpn = va >> PAGE_SHIFT;
225
226         hash = hpt_hash(vpn, large);
227
228         local_hpte.dw1.dword1 = pa | mode;
229         local_hpte.dw0.dword0 = 0;
230         local_hpte.dw0.dw0.avpn = va >> 23;
231         local_hpte.dw0.dw0.bolted = 1;          /* bolted */
232         if (large) {
233                 local_hpte.dw0.dw0.l = 1;       /* large page */
234                 local_hpte.dw0.dw0.avpn &= ~0x1UL;
235         }
236         local_hpte.dw0.dw0.v = 1;
237
238         if (systemcfg−>platform == PLATFORM_PSERIES) {
239                 hptep  = htab + ((hash & hash_mask)*HPTES_PER_GROUP);
240
241                 for (i = 0; i < 8; ++i, ++hptep) {
242                         if (hptep−>dw0.dw0.v == 0) {            /* !valid */
243                                 *hptep = local_hpte;
244                                 return;
245                         }
246                 }
247         } else if (systemcfg−>platform == PLATFORM_PSERIES_LPAR) {
248                 slot = ((hash & hash_mask)*HPTES_PER_GROUP);
249
250                 /* Set CEC cookie to 0                  */
251                 /* Zero page = 0                        */
252                 /* I−cache Invalidate = 0               */
253                 /* I−cache synchronize = 0              */
254                 /* Exact = 0 − modify any entry in group */
255                 flags = 0;
256
257                 lpar_rc =  plpar_pte_enter(flags, slot, local_hpte.dw0.dword0,
258                                            local_hpte.dw1.dword1,
259                                            &dummy1, &dummy2);
260                 if (lpar_rc != H_Success) {
261                         ppc64_terminate_msg(0x21, "hpte enter");
262                         loop_forever();
263                 }
264                 return;
265         } else if (systemcfg−>platform == PLATFORM_ISERIES_LPAR) {
266                 slot = HvCallHpt_findValid(&rhpte, vpn);
267                 if (slot < 0) {
268                         /* Must find space in primary group */
269                         panic("hash_page: hpte already exists\n");
270                 }
```

```
271                     HvCallHpt_addValidate(slot, 0, (HPTE *)&local_hpte );
272                     return;
273             }
274
275             /* We should _never_ get here and too early to call xmon. */
276             ppc64_terminate_msg(0x22, "hpte platform");
277             loop_forever();
278     }
279
280     /*
281      * find_linux_pte returns the address of a linux pte for a given
282      * effective address and directory.  If not found, it returns zero.
283      */
284     pte_t *find_linux_pte(pgd_t *pgdir, unsigned long ea)
285     {
286             pgd_t *pg;
287             pmd_t *pm;
288             pte_t *pt = NULL;
289             pte_t pte;
290
291             pg = pgdir + pgd_index(ea);
292             if (!pgd_none(*pg)) {
293                     pm = pmd_offset(pg, ea);
294                     if (!pmd_none(*pm)) {
295                             pt = pte_offset(pm, ea);
296                             pte = *pt;
297                             if (!pte_present(pte))
298                                     pt = NULL;
299                     }
300             }
301
302             return pt;
303     }
304
305     static inline unsigned long computeHptePP(unsigned long pte)
306     {
307             return (pte & _PAGE_USER) |
308                     (((pte & _PAGE_USER) >> 1) &
309                      ((~((pte >> 2) &       /* _PAGE_RW */
310                          (pte >> 7))) &     /* _PAGE_DIRTY */
311                       1));
312     }
313
314     /*
315      * Handle a fault by adding an HPTE. If the address can't be determined
316      * to be valid via Linux page tables, return 1. If handled return 0
317      */
318     int __hash_page(unsigned long ea, unsigned long access,
319                     unsigned long vsid, pte_t *ptep)
320     {
321             unsigned long va, vpn;
322             unsigned long newpp, prpn;
323             unsigned long hpteflags, lock_slot;
324             long slot;
325             pte_t old_pte, new_pte;
326
327             /* Search the Linux page table for a match with va */
328             va = (vsid << 28) | (ea & 0x0fffffff);
329             vpn = va >> PAGE_SHIFT;
330             lock_slot = get_lock_slot(vpn);
331
332             /* Acquire the hash table lock to guarantee that the linux
333              * pte we fetch will not change
334              */
335             spin_lock(&hash_table_lock[lock_slot].lock);
336
337             /*
338              * Check the user's access rights to the page.  If access should be
339              * prevented then send the problem up to do_page_fault.
340              */
341             access |= _PAGE_PRESENT;
342             if (unlikely(access & ~(pte_val(*ptep)))) {
343                     spin_unlock(&hash_table_lock[lock_slot].lock);
344                     return 1;
345             }
346
347             /*
348              * We have found a pte (which was present).
349              * The spinlocks prevent this status from changing
350              * The hash_table_lock prevents the _PAGE_HASHPTE status
351              * from changing (RPN, DIRTY and ACCESSED too)
352              * The page_table_lock prevents the pte from being
353              * invalidated or modified
354              */
355
356             /*
357              * At this point, we have a pte (old_pte) which can be used to build
358              * or update an HPTE. There are 2 cases:
359              *
360              * 1. There is a valid (present) pte with no associated HPTE (this is
```

```
361             *      the most common case)
362             * 2. There is a valid (present) pte with an associated HPTE. The
363             *      current values of the pp bits in the HPTE prevent access
364             *      because we are doing software DIRTY bit management and the
365             *      page is currently not DIRTY.
366             */
367
368             old_pte = *ptep;
369             new_pte = old_pte;
370
371             /* If the attempted access was a store */
372             if (access & _PAGE_RW)
373                     pte_val(new_pte) |= _PAGE_ACCESSED | _PAGE_DIRTY;
374             else
375                     pte_val(new_pte) |= _PAGE_ACCESSED;
376
377             newpp = computeHptePP(pte_val(new_pte));
378
379             /* Check if pte already has an hpte (case 2) */
380             if (unlikely(pte_val(old_pte) & _PAGE_HASHPTE)) {
381                     /* There MIGHT be an HPTE for this pte */
382                     unsigned long hash, slot, secondary;
383
384                     /* XXX fix large pte flag */
385                     hash = hpt_hash(vpn, 0);
386                     secondary = (pte_val(old_pte) & _PAGE_SECONDARY) >> 15;
387                     if (secondary)
388                             hash = ~hash;
389                     slot = (hash & htab_data.htab_hash_mask) * HPTES_PER_GROUP;
390                     slot += (pte_val(old_pte) & _PAGE_GROUP_IX) >> 12;
391
392                     /* XXX fix large pte flag */
393                     if (ppc_md.hpte_updatepp(slot, secondary,
394                                              newpp, va, 0) == -1) {
395                             pte_val(old_pte) &= ~_PAGE_HPTEFLAGS;
396                     } else {
397                             if (!pte_same(old_pte, new_pte)) {
398                                     *ptep = new_pte;
399                             }
400                     }
401             }
402
403             if (likely(!(pte_val(old_pte) & _PAGE_HASHPTE))) {
404                     /* Update the linux pte with the HPTE slot */
405                     pte_val(new_pte) &= ~_PAGE_HPTEFLAGS;
406                     pte_val(new_pte) |= _PAGE_HASHPTE;
407                     prpn = pte_val(old_pte) >> PTE_SHIFT;
408
409                     /* copy appropriate flags from linux pte */
410                     hpteflags = (pte_val(new_pte) & 0x1f8) | newpp;
411
412                     slot = ppc_md.hpte_insert(vpn, prpn, hpteflags, 0, 0);
413
414                     pte_val(new_pte) |= ((slot<<12) &
415                                          (_PAGE_GROUP_IX | _PAGE_SECONDARY));
416
417                     *ptep = new_pte;
418             }
419
420             spin_unlock(&hash_table_lock[lock_slot].lock);
421
422             return 0;
423     }
424
425     /*
426      * Handle a fault by adding an HPTE. If the address can't be determined
427      * to be valid via Linux page tables, return 1. If handled return 0
428      */
429     int hash_page(unsigned long ea, unsigned long access)
430     {
431             void *pgdir;
432             unsigned long vsid;
433             struct mm_struct *mm;
434             pte_t *ptep;
435             int ret;
436
437             /* Check for invalid addresses. */
438             if (!IS_VALID_EA(ea)) return 1;
439
440             switch (REGION_ID(ea)) {
441             case USER_REGION_ID:
442                     mm = current->mm;
443                     if (mm == NULL) return 1;
444                     vsid = get_vsid(mm->context, ea);
445                     break;
446             case IO_REGION_ID:
447                     mm = &ioremap_mm;
448                     vsid = get_kernel_vsid(ea);
449                     break;
450             case VMALLOC_REGION_ID:
```

```
451                     mm = &init_mm;
452                     vsid = get_kernel_vsid(ea);
453                     break;
454             case IO_UNMAPPED_REGION_ID:
455                     udbg_printf("EEH Error ea = 0x%lx\n", ea);
456                     PPCDBG_ENTER_DEBUGGER();
457                     panic("EEH Error ea = 0x%lx\n", ea);
458                     break;
459             case KERNEL_REGION_ID:
460                     /*
461                      * As htab_initialize is now, we shouldn't ever get here since
462                      * we're bolting the entire 0xC0... region.
463                      */
464                     udbg_printf("Little faulted on kernel address 0x%lx\n", ea);
465                     PPCDBG_ENTER_DEBUGGER();
466                     panic("Little faulted on kernel address 0x%lx\n", ea);
467                     break;
468             default:
469                     /* Not a valid range, send the problem up to do_page_fault */
470                     return 1;
471                     break;
472             }
473
474             pgdir = mm->pgd;
475             if (pgdir == NULL) return 1;
476
477             /*
478              * Lock the Linux page table to prevent mmap and kswapd
479              * from modifying entries while we search and update
480              */
481             spin_lock(&mm->page_table_lock);
482
483             ptep = find_linux_pte(pgdir, ea);
484             /*
485              * If no pte found or not present, send the problem up to
486              * do_page_fault
487              */
488             if (ptep && pte_present(*ptep)) {
489                     ret = __hash_page(ea, access, vsid, ptep);
490             } else {
491                     /* If no pte, send the problem up to do_page_fault */
492                     ret = 1;
493             }
494
495             spin_unlock(&mm->page_table_lock);
496
497             return ret;
498     }
499
500     void flush_hash_page(unsigned long context, unsigned long ea, pte_t *ptep)
501     {
502             unsigned long vsid, vpn, va, hash, secondary, slot, flags, lock_slot;
503             unsigned long large = 0, local = 0;
504             pte_t pte;
505
506             if ((ea >= USER_START) && (ea <= USER_END))
507                     vsid = get_vsid(context, ea);
508             else
509                     vsid = get_kernel_vsid(ea);
510
511             va = (vsid << 28) | (ea & 0x0fffffff);
512             if (large)
513                     vpn = va >> LARGE_PAGE_SHIFT;
514             else
515                     vpn = va >> PAGE_SHIFT;
516
517             lock_slot = get_lock_slot(vpn);
518             hash = hpt_hash(vpn, large);
519
520             spin_lock_irqsave(&hash_table_lock[lock_slot].lock, flags);
521
522             pte = __pte(pte_update(ptep, _PAGE_HPTEFLAGS, 0));
523             secondary = (pte_val(pte) & _PAGE_SECONDARY) >> 15;
524             if (secondary) hash = ~hash;
525             slot = (hash & htab_data.htab_hash_mask) * HPTES_PER_GROUP;
526             slot += (pte_val(pte) & _PAGE_GROUP_IX) >> 12;
527
528             if (pte_val(pte) & _PAGE_HASHPTE) {
529                     ppc_md.hpte_invalidate(slot, secondary, va, large, local);
530             }
531
532             spin_unlock_irqrestore(&hash_table_lock[lock_slot].lock, flags);
533     }
534
535     long plpar_pte_enter(unsigned long flags,
536                          unsigned long ptex,
537                          unsigned long new_pteh, unsigned long new_ptel,
538                          unsigned long *old_pteh_ret, unsigned long *old_ptel_ret)
539     {
540             unsigned long dummy, ret;
```

```
541         ret = plpar_hcall(H_ENTER, flags, ptex, new_pteh, new_ptel,
542                           old_pteh_ret, old_ptel_ret, &dummy);
543         return(ret);
544 }
545
546 long plpar_pte_remove(unsigned long flags,
547                       unsigned long ptex,
548                       unsigned long avpn,
549                       unsigned long *old_pteh_ret, unsigned long *old_ptel_ret)
550 {
551         unsigned long dummy;
552         return plpar_hcall(H_REMOVE, flags, ptex, avpn, 0,
553                           old_pteh_ret, old_ptel_ret, &dummy);
554 }
555
556 long plpar_pte_read(unsigned long flags,
557                     unsigned long ptex,
558                     unsigned long *old_pteh_ret, unsigned long *old_ptel_ret)
559 {
560         unsigned long dummy;
561         return plpar_hcall(H_READ, flags, ptex, 0, 0,
562                           old_pteh_ret, old_ptel_ret, &dummy);
563 }
564
565 long plpar_pte_protect(unsigned long flags,
566                        unsigned long ptex,
567                        unsigned long avpn)
568 {
569         return plpar_hcall_norets(H_PROTECT, flags, ptex, avpn);
570 }
571
572 static __inline__ void set_pp_bit(unsigned long pp, HPTE *addr)
573 {
574         unsigned long old;
575         unsigned long *p = &addr->dw1.dword1;
576
577         __asm__ __volatile__(
578         "1:     ldarx   %0,0,%3\n\
579         rldimi  %0,%2,0,62\n\
580         stdcx.  %0,0,%3\n\
581                 bne     1b"
582         : "=&r" (old), "=m" (*p)
583         : "r" (pp), "r" (p), "m" (*p)
584         : "cc");
585 }
586
587 /*
588  * Calculate which hash_table_lock to use, based on the pteg being used.
589  *
590  * Given a VPN, use the high order 8 bits to select one of 2^7 locks.  The
591  * highest order bit is used to indicate primary vs. secondary group.  If the
592  * secondary is selected, complement the lock select bits.  This results in
593  * both the primary and secondary groups being covered under the same lock.
594  */
595 inline unsigned long get_lock_slot(unsigned long vpn)
596 {
597         unsigned long lock_slot;
598 #ifdef LOCK_SPLIT
599         lock_slot = (hpt_hash(vpn,0) >> htab_data.htab_lock_shift) & 0xff;
600         if(lock_slot & 0x80) lock_slot = (~lock_slot) & 0x7f;
601 #else
602         lock_slot = 0;
603 #endif
604         return(lock_slot);
605 }
606
607 /*
608  * Functions used to retrieve word 0 of a given page table entry.
609  *
610  * Input : slot : PTE index within the page table of the entry to retrieve
611  * Output: Contents of word 0 of the specified entry
612  */
613 static unsigned long rpa_lpar_hpte_getword0(unsigned long slot)
614 {
615         unsigned long dword0;
616         unsigned long lpar_rc;
617         unsigned long dummy_word1;
618         unsigned long flags;
619
620         /* Read 1 pte at a time                        */
621         /* Do not need RPN to logical page translation */
622         /* No cross CEC PFT access                     */
623         flags = 0;
624
625         lpar_rc = plpar_pte_read(flags, slot, &dword0, &dummy_word1);
626
627         if (lpar_rc != H_Success)
628                 panic("Error on pte read in get_hpte0 rc = %lx\n", lpar_rc);
629
630         return dword0;
```

```
631 }
632
633 unsigned long iSeries_hpte_getword0(unsigned long slot)
634 {
635         unsigned long dword0;
636
637         HPTE hpte;
638         HvCallHpt_get(&hpte, slot);
639         dword0 = hpte.dw0.dword0;
640
641         return dword0;
642 }
643
644 /*
645  * Functions used to find the PTE for a particular virtual address.
646  * Only used during boot when bolting pages.
647  *
648  * Input : vpn      : virtual page number
649  * Output: PTE index within the page table of the entry
650  *         −1 on failure
651  */
652 static long hpte_find(unsigned long vpn)
653 {
654         HPTE *hptep;
655         unsigned long hash;
656         unsigned long i, j;
657         long slot;
658         Hpte_dword0 dw0;
659
660         hash = hpt_hash(vpn, 0);
661
662         for (j = 0; j < 2; j++) {
663                 slot = (hash & htab_data.htab_hash_mask) * HPTES_PER_GROUP;
664                 for (i = 0; i < HPTES_PER_GROUP; i++) {
665                         hptep = htab_data.htab + slot;
666                         dw0 = hptep−>dw0.dw0;
667
668                         if ((dw0.avpn == (vpn >> 11)) && dw0.v &&
669                             (dw0.h == j)) {
670                                 /* HPTE matches */
671                                 if (j)
672                                         slot = −slot;
673                                 return slot;
674                         }
675                         ++slot;
676                 }
677                 hash = ~hash;
678         }
679
680         return −1;
681 }
682
683 static long rpa_lpar_hpte_find(unsigned long vpn)
684 {
685         unsigned long hash;
686         unsigned long i, j;
687         long slot;
688         union {
689                 unsigned long dword0;
690                 Hpte_dword0 dw0;
691         } hpte_dw0;
692         Hpte_dword0 dw0;
693
694         hash = hpt_hash(vpn, 0);
695
696         for (j = 0; j < 2; j++) {
697                 slot = (hash & htab_data.htab_hash_mask) * HPTES_PER_GROUP;
698                 for (i = 0; i < HPTES_PER_GROUP; i++) {
699                         hpte_dw0.dword0 = rpa_lpar_hpte_getword0(slot);
700                         dw0 = hpte_dw0.dw0;
701
702                         if ((dw0.avpn == (vpn >> 11)) && dw0.v &&
703                             (dw0.h == j)) {
704                                 /* HPTE matches */
705                                 if (j)
706                                         slot = −slot;
707                                 return slot;
708                         }
709                         ++slot;
710                 }
711                 hash = ~hash;
712         }
713
714         return −1;
715 }
716
717 static long iSeries_hpte_find(unsigned long vpn)
718 {
719         HPTE hpte;
720         long slot;
```

```
721
722         /*
723          * The HvCallHpt_findValid interface is as follows:
724          * 0xffffffffffffffff : No entry found.
725          * 0x00000000xxxxxxxx : Entry found in primary group, slot x
726          * 0x80000000xxxxxxxx : Entry found in secondary group, slot x
727          */
728         slot = HvCallHpt_findValid(&hpte, vpn);
729         if (hpte.dw0.dw0.v) {
730                 if (slot < 0) {
731                         slot &= 0x7fffffffffffffff;
732                         slot = -slot;
733                 }
734         } else {
735                 slot = -1;
736         }
737
738         return slot;
739 }
740
741 /*
742  * Functions used to invalidate a page table entry from the page table
743  * and tlb.
744  *
745  * Input : slot  : PTE index within the page table of the entry to invalidated
746  *         va    : Virtual address of the entry being invalidated
747  *         large : 1 = large page (16M)
748  *         local : 1 = Use tlbiel to only invalidate the local tlb
749  */
750 static void hpte_invalidate(unsigned long slot,
751                             unsigned long secondary,
752                             unsigned long va,
753                             int large, int local)
754 {
755         HPTE *hptep = htab_data.htab + slot;
756         Hpte_dword0 dw0;
757         unsigned long vpn, avpn;
758         unsigned long flags;
759
760         if (large)
761                 vpn = va >> LARGE_PAGE_SHIFT;
762         else
763                 vpn = va >> PAGE_SHIFT;
764
765         avpn = vpn >> 11;
766
767         dw0 = hptep->dw0.dw0;
768
769         /*
770          * Do not remove bolted entries.  Alternatively, we could check
771          * the AVPN, hash group, and valid bits.  By doing it this way,
772          * it is common with the pSeries LPAR optimal path.
773          */
774         if (dw0.bolted) return;
775
776         /* Invalidate the hpte. */
777         hptep->dw0.dword0 = 0;
778
779         /* Invalidate the tlb */
780         spin_lock_irqsave(&pSeries_tlbie_lock, flags);
781         _tlbie(va, large);
782         spin_unlock_irqrestore(&pSeries_tlbie_lock, flags);
783 }
784
785 static void rpa_lpar_hpte_invalidate(unsigned long slot,
786                                      unsigned long secondary,
787                                      unsigned long va,
788                                      int large, int local)
789 {
790         unsigned long lpar_rc;
791         unsigned long dummy1, dummy2;
792
793         /*
794          * Don't remove a bolted entry.  This case can occur when we bolt
795          * pages dynamically after initial boot.
796          */
797         lpar_rc = plpar_pte_remove(H_ANDCOND, slot, (0x1UL << 4),
798                                    &dummy1, &dummy2);
799
800         if (lpar_rc != H_Success)
801                 panic("Bad return code from invalidate rc = %lx\n", lpar_rc);
802 }
803
804 static void iSeries_hpte_invalidate(unsigned long slot,
805                                     unsigned long secondary,
806                                     unsigned long va,
807                                     int large, int local)
808 {
809         HPTE lhpte;
810         unsigned long vpn, avpn;
```

```
811
812         if (large)
813                 vpn = va >> LARGE_PAGE_SHIFT;
814         else
815                 vpn = va >> PAGE_SHIFT;
816
817         avpn = vpn >> 11;
818
819         lhpte.dw0.dword0 = iSeries_hpte_getword0(slot);
820
821         if ((lhpte.dw0.dw0.avpn == avpn) &&
822             (lhpte.dw0.dw0.v) &&
823             (lhpte.dw0.dw0.h == secondary)) {
824                 HvCallHpt_invalidateSetSwBitsGet(slot, 0, 0);
825         }
826 }
827
828 /*
829  * Functions used to update page protection bits.
830  *
831  * Input : slot  : PTE index within the page table of the entry to update
832  *         newpp : new page protection bits
833  *         va    : Virtual address of the entry being updated
834  *         large : 1 = large page (16M)
835  * Output: 0 on success, -1 on failure
836  */
837 static long hpte_updatepp(unsigned long slot,
838                           unsigned long secondary,
839                           unsigned long newpp,
840                           unsigned long va, int large)
841 {
842         HPTE *hptep = htab_data.htab + slot;
843         Hpte_dword0 dw0;
844         Hpte_dword1 dw1;
845         unsigned long vpn, avpn;
846         unsigned long flags;
847
848         if (large)
849                 vpn = va >> LARGE_PAGE_SHIFT;
850         else
851                 vpn = va >> PAGE_SHIFT;
852
853         avpn = vpn >> 11;
854
855         dw0 = hptep->dw0.dw0;
856         if ((dw0.avpn == avpn) &&
857             (dw0.v) && (dw0.h == secondary)) {
858                 /* Turn off valid bit in HPTE */
859                 dw0.v = 0;
860                 hptep->dw0.dw0 = dw0;
861
862                 /* Ensure it is out of the tlb too */
863                 spin_lock_irqsave(&pSeries_tlbie_lock, flags);
864                 _tlbie(va, large);
865                 spin_unlock_irqrestore(&pSeries_tlbie_lock, flags);
866
867                 /* Insert the new pp bits into the HPTE */
868                 dw1 = hptep->dw1.dw1;
869                 dw1.pp = newpp;
870                 hptep->dw1.dw1 = dw1;
871
872                 /* Ensure it is visible before validating */
873                 __asm__ __volatile__ ("eieio" : : : "memory");
874
875                 /* Turn the valid bit back on in HPTE */
876                 dw0.v = 1;
877                 hptep->dw0.dw0 = dw0;
878
879                 __asm__ __volatile__ ("ptesync" : : : "memory");
880
881                 return 0;
882         }
883
884         return -1;
885 }
886
887 static long rpa_lpar_hpte_updatepp(unsigned long slot,
888                                    unsigned long secondary,
889                                    unsigned long newpp,
890                                    unsigned long va, int large)
891 {
892         unsigned long lpar_rc;
893         unsigned long flags = (newpp & 7);
894         unsigned long avpn = va >> 23;
895         HPTE hpte;
896
897         lpar_rc = plpar_pte_read(0, slot, &hpte.dw0.dword0, &hpte.dw1.dword1);
898
899         if ((hpte.dw0.dw0.avpn == avpn) &&
900             (hpte.dw0.dw0.v) &&
```

```
901             (hpte.dw0.dw0.h == secondary)) {
902                 lpar_rc = plpar_pte_protect(flags, slot, 0);
903                 if (lpar_rc != H_Success)
904                         panic("bad return code from pte protect rc = %lx\n",
905                               lpar_rc);
906                 return 0;
907         }
908
909         return −1;
910 }
911
912 static long iSeries_hpte_updatepp(unsigned long slot,
913                                   unsigned long secondary,
914                                   unsigned long newpp,
915                                   unsigned long va, int large)
916 {
917         unsigned long vpn, avpn;
918         HPTE hpte;
919
920         if (large)
921                 vpn = va >> LARGE_PAGE_SHIFT;
922         else
923                 vpn = va >> PAGE_SHIFT;
924
925         avpn = vpn >> 11;
926
927         HvCallHpt_get(&hpte, slot);
928         if ((hpte.dw0.dw0.avpn == avpn) &&
929             (hpte.dw0.dw0.v) &&
930             (hpte.dw0.dw0.h == secondary)) {
931                 HvCallHpt_setPp(slot, newpp);
932                 return 0;
933         }
934         return −1;
935 }
936
937 /*
938  * Functions used to update the page protection bits. Intended to be used
939  * to create guard pages for kernel data structures on pages which are bolted
940  * in the HPT. Assumes pages being operated on will not be stolen.
941  * Does not work on large pages. No need to lock here because we are the
942  * only user.
943  *
944  * Input : newpp : page protection flags
945  *         ea    : effective kernel address to bolt.
946  */
947 static void hpte_updateboltedpp(unsigned long newpp, unsigned long ea)
948 {
949         unsigned long vsid, va, vpn, flags;
950         long slot;
951         HPTE *hptep;
952
953         vsid = get_kernel_vsid(ea);
954         va = (vsid << 28) | (ea & 0x0fffffff);
955         vpn = va >> PAGE_SHIFT;
956
957         slot = hpte_find(vpn);
958         if (slot == −1)
959                 panic("could not find page to bolt\n");
960         hptep = htab_data.htab + slot;
961
962         set_pp_bit(newpp, hptep);
963
964         /* Ensure it is out of the tlb too */
965         spin_lock_irqsave(&pSeries_tlbie_lock, flags);
966         _tlbie(va, 0);
967         spin_unlock_irqrestore(&pSeries_tlbie_lock, flags);
968 }
969
970 static void rpa_lpar_hpte_updateboltedpp(unsigned long newpp, unsigned long ea)
971 {
972         unsigned long lpar_rc;
973         unsigned long vsid, va, vpn, flags;
974         long slot;
975
976         vsid = get_kernel_vsid(ea);
977         va = (vsid << 28) | (ea & 0x0fffffff);
978         vpn = va >> PAGE_SHIFT;
979
980         slot = rpa_lpar_hpte_find(vpn);
981         if (slot == −1)
982                 panic("updateboltedpp: Could not find page to bolt\n");
983
984         flags = newpp & 3;
985         lpar_rc = plpar_pte_protect(flags, slot, 0);
986
987         if (lpar_rc != H_Success)
988                 panic("Bad return code from pte bolted protect rc = %lx\n",
989                       lpar_rc);
990 }
```

```
991
992 void iSeries_hpte_updateboltedpp(unsigned long newpp, unsigned long ea)
993 {
994         unsigned long vsid,va,vpn;
995         long slot;
996
997         vsid = get_kernel_vsid( ea );
998         va = ( vsid << 28 ) | ( ea & 0x0fffffff );
999         vpn = va >> PAGE_SHIFT;
1000
1001        slot = iSeries_hpte_find(vpn);
1002        if (slot == -1)
1003                panic("updateboltedpp: Could not find page to bolt\n");
1004
1005        HvCallHpt_setPp(slot, newpp);
1006 }
1007
1008 /*
1009  * Functions used to insert new hardware page table entries.
1010  * Will castout non-bolted entries as necessary using a random
1011  * algorithm.
1012  *
1013  * Input : vpn      : virtual page number
1014  *         prpn     : real page number in absolute space
1015  *         hpteflags: page protection flags
1016  *         bolted   : 1 = bolt the page
1017  *         large    : 1 = large page (16M)
1018  * Output: hsss, where h = hash group, sss = slot within that group
1019  */
1020 static long hpte_insert(unsigned long vpn, unsigned long prpn,
1021                         unsigned long hpteflags, int bolted, int large)
1022 {
1023         HPTE *hptep;
1024         Hpte_dword0 dw0;
1025         HPTE lhpte;
1026         int i, secondary;
1027         unsigned long hash = hpt_hash(vpn, 0);
1028         unsigned long avpn = vpn >> 11;
1029         unsigned long arpn = physRpn_to_absRpn(prpn);
1030         unsigned long hpte_group;
1031
1032 repeat:
1033         secondary = 0;
1034         hpte_group = ((hash & htab_data.htab_hash_mask) *
1035                       HPTES_PER_GROUP) & ~0x7UL;
1036         hptep = htab_data.htab + hpte_group;
1037
1038         for (i = 0; i < HPTES_PER_GROUP; i++) {
1039                 dw0 = hptep->dw0.dw0;
1040                 if (!dw0.v) {
1041                         /* retry with lock held */
1042                         dw0 = hptep->dw0.dw0;
1043                         if (!dw0.v)
1044                                 break;
1045                 }
1046                 hptep++;
1047         }
1048
1049         if (i == HPTES_PER_GROUP) {
1050                 secondary = 1;
1051                 hpte_group = ((~hash & htab_data.htab_hash_mask) *
1052                               HPTES_PER_GROUP) & ~0x7UL;
1053                 hptep = htab_data.htab + hpte_group;
1054
1055                 for (i = 0; i < HPTES_PER_GROUP; i++) {
1056                         dw0 = hptep->dw0.dw0;
1057                         if (!dw0.v) {
1058                                 /* retry with lock held */
1059                                 dw0 = hptep->dw0.dw0;
1060                                 if (!dw0.v)
1061                                         break;
1062                         }
1063                         hptep++;
1064                 }
1065                 if (i == HPTES_PER_GROUP) {
1066                         if (mftb() & 0x1)
1067                                 hpte_group=((hash & htab_data.htab_hash_mask)*
1068                                             HPTES_PER_GROUP) & ~0x7UL;
1069
1070                         hpte_remove(hpte_group);
1071                         goto repeat;
1072                 }
1073         }
1074
1075         lhpte.dw1.dword1      = 0;
1076         lhpte.dw1.dw1.rpn     = arpn;
1077         lhpte.dw1.flags.flags = hpteflags;
1078
1079         lhpte.dw0.dword0      = 0;
1080         lhpte.dw0.dw0.avpn    = avpn;
```

```
1081          lhpte.dw0.dw0.h       = secondary;
1082          lhpte.dw0.dw0.bolted  = bolted;
1083          lhpte.dw0.dw0.v       = 1;
1084
1085          if (large) lhpte.dw0.dw0.l = 1;
1086
1087          hptep->dw1.dword1 = lhpte.dw1.dword1;
1088
1089          /* Guarantee the second dword is visible before the valid bit */
1090          __asm__ __volatile__ ("eieio" : : : "memory");
1091
1092          /*
1093           * Now set the first dword including the valid bit
1094           * NOTE: this also unlocks the hpte
1095           */
1096          hptep->dw0.dword0 = lhpte.dw0.dword0;
1097
1098          __asm__ __volatile__ ("ptesync" : : : "memory");
1099
1100          return ((secondary << 3) | i);
1101  }
1102
1103  static long rpa_lpar_hpte_insert(unsigned long vpn, unsigned long prpn,
1104                                   unsigned long hpteflags,
1105                                   int bolted, int large)
1106  {
1107          /* XXX fix for large page */
1108          unsigned long lpar_rc;
1109          unsigned long flags;
1110          unsigned long slot;
1111          HPTE lhpte;
1112          int secondary;
1113          unsigned long hash = hpt_hash(vpn, 0);
1114          unsigned long avpn = vpn >> 11;
1115          unsigned long arpn = physRpn_to_absRpn(prpn);
1116          unsigned long hpte_group;
1117
1118          /* Fill in the local HPTE with absolute rpn, avpn and flags */
1119          lhpte.dw1.dword1      = 0;
1120          lhpte.dw1.dw1.rpn     = arpn;
1121          lhpte.dw1.flags.flags = hpteflags;
1122
1123          lhpte.dw0.dword0      = 0;
1124          lhpte.dw0.dw0.avpn    = avpn;
1125          lhpte.dw0.dw0.bolted  = bolted;
1126          lhpte.dw0.dw0.v       = 1;
1127
1128          if (large) lhpte.dw0.dw0.l = 1;
1129
1130          /* Now fill in the actual HPTE */
1131          /* Set CEC cookie to 0         */
1132          /* Large page = 0              */
1133          /* Zero page = 0               */
1134          /* I−cache Invalidate = 0      */
1135          /* I−cache synchronize = 0     */
1136          /* Exact = 0                   */
1137          flags = 0;
1138
1139          /* XXX why is this here? − Anton */
1140          /*   −− Because at one point we hit a case where non cachable
1141           *      pages where marked coherent & this is rejected by the HV.
1142           *      Perhaps it is no longer an issue ... DRENG.
1143           */
1144          if (hpteflags & (_PAGE_GUARDED|_PAGE_NO_CACHE))
1145                  lhpte.dw1.flags.flags &= ~_PAGE_COHERENT;
1146
1147  repeat:
1148          secondary = 0;
1149          lhpte.dw0.dw0.h = secondary;
1150          hpte_group = ((hash & htab_data.htab_hash_mask) *
1151                        HPTES_PER_GROUP) & ~0x7UL;
1152
1153          __asm__ __volatile__ (
1154                  H_ENTER_r3
1155                  "mr   4, %2\n"
1156                  "mr   5, %3\n"
1157                  "mr   6, %4\n"
1158                  "mr   7, %5\n"
1159                  HSC
1160                  "mr   %0, 3\n"
1161                  "mr   %1, 4\n"
1162                  : "=r" (lpar_rc), "=r" (slot)
1163                  : "r" (flags), "r" (hpte_group), "r" (lhpte.dw0.dword0),
1164                  "r" (lhpte.dw1.dword1)
1165                  : "r0", "r3", "r4", "r5", "r6", "r7",
1166                    "r8", "r9", "r10", "r11", "r12", "cc");
1167
1168          if (lpar_rc == H_PTEG_Full) {
1169                  secondary = 1;
1170                  lhpte.dw0.dw0.h = secondary;
```

```
1171                    hpte_group = ((~hash & htab_data.htab_hash_mask) *
1172                                  HPTES_PER_GROUP) & ~0x7UL;
1173
1174                    __asm__ __volatile__ (
1175                                  H_ENTER_r3
1176                                  "mr   4, %2\n"
1177                                  "mr   5, %3\n"
1178                                  "mr   6, %4\n"
1179                                  "mr   7, %5\n"
1180                                  HSC
1181                                  "mr   %0, 3\n"
1182                                  "mr   %1, 4\n"
1183                                  : "=r" (lpar_rc), "=r" (slot)
1184                                  : "r" (flags), "r" (hpte_group), "r" (lhpte.dw0.dword0),
1185                                  "r" (lhpte.dw1.dword1)
1186                                  : "r0", "r3", "r4", "r5", "r6", "r7",
1187                                    "r8", "r9", "r10", "r11", "r12", "cc");
1188                    if (lpar_rc == H_PTEG_Full) {
1189                            if (mftb() & 0x1)
1190                                    hpte_group=((hash & htab_data.htab_hash_mask)*
1191                                                HPTES_PER_GROUP) & ~0x7UL;
1192
1193                            rpa_lpar_hpte_remove(hpte_group);
1194                            goto repeat;
1195                    }
1196            }
1197
1198            if (lpar_rc != H_Success)
1199                    panic("Bad return code from pte enter rc = %lx\n", lpar_rc);
1200
1201            return ((secondary << 3) | (slot & 0x7));
1202    }
1203
1204    static long iSeries_hpte_insert(unsigned long vpn, unsigned long prpn,
1205                                    unsigned long hpteflags,
1206                                    int bolted, int large)
1207    {
1208            HPTE lhpte;
1209            unsigned long hash, hpte_group;
1210            unsigned long avpn = vpn >> 11;
1211            unsigned long arpn = physRpn_to_absRpn( prpn );
1212            int secondary = 0;
1213            long slot;
1214
1215            hash = hpt_hash(vpn, 0);
1216
1217    repeat:
1218            slot = HvCallHpt_findValid(&lhpte, vpn);
1219            if (lhpte.dw0.dw0.v) {
1220                    panic("select_hpte_slot found entry already valid\n");
1221            }
1222
1223            if (slot == -1) { /* No available entry found in either group */
1224                    if (mftb() & 0x1) {
1225                            hpte_group=((hash & htab_data.htab_hash_mask)*
1226                                        HPTES_PER_GROUP) & ~0x7UL;
1227                    } else {
1228                            hpte_group=((~hash & htab_data.htab_hash_mask)*
1229                                        HPTES_PER_GROUP) & ~0x7UL;
1230                    }
1231
1232                    hash = hpt_hash(vpn, 0);
1233                    iSeries_hpte_remove(hpte_group);
1234                    goto repeat;
1235            } else if (slot < 0) {
1236                    slot &= 0x7fffffffffffffff;
1237                    secondary = 1;
1238            }
1239
1240            /* Create the HPTE */
1241            lhpte.dw1.dword1      = 0;
1242            lhpte.dw1.dw1.rpn     = arpn;
1243            lhpte.dw1.flags.flags = hpteflags;
1244
1245            lhpte.dw0.dword0     = 0;
1246            lhpte.dw0.dw0.avpn   = avpn;
1247            lhpte.dw0.dw0.h      = secondary;
1248            lhpte.dw0.dw0.bolted = bolted;
1249            lhpte.dw0.dw0.v      = 1;
1250
1251            /* Now fill in the actual HPTE */
1252            HvCallHpt_addValidate(slot, secondary, (HPTE *)&lhpte);
1253            return ((secondary << 3) | (slot & 0x7));
1254    }
1255
1256    /*
1257     * Functions used to remove hardware page table entries.
1258     *
1259     * Input : hpte_group: PTE index of the first entry in a group
1260     * Output: offset within the group of the entry removed or
```

```
1261  *          −1 on failure
1262  */
1263 static long hpte_remove(unsigned long hpte_group)
1264 {
1265         HPTE *hptep;
1266         Hpte_dword0 dw0;
1267         int i;
1268         int slot_offset;
1269         unsigned long vsid, group, pi, pi_high;
1270         unsigned long slot;
1271         unsigned long flags;
1272         int large;
1273         unsigned long va;
1274
1275         /* pick a random slot to start at */
1276         slot_offset = mftb() & 0x7;
1277
1278         for (i = 0; i < HPTES_PER_GROUP; i++) {
1279                 hptep = htab_data.htab + hpte_group + slot_offset;
1280                 dw0 = hptep−>dw0.dw0;
1281
1282                 if (dw0.v && !dw0.bolted) {
1283                         /* retry with lock held */
1284                         dw0 = hptep−>dw0.dw0;
1285                         if (dw0.v && !dw0.bolted)
1286                                 break;
1287                 }
1288
1289                 slot_offset++;
1290                 slot_offset &= 0x7;
1291         }
1292
1293         if (i == HPTES_PER_GROUP)
1294                 return −1;
1295
1296         large = dw0.l;
1297
1298         /* Invalidate the hpte. NOTE: this also unlocks it */
1299         hptep−>dw0.dword0 = 0;
1300
1301         /* Invalidate the tlb */
1302         vsid = dw0.avpn >> 5;
1303         slot = hptep − htab_data.htab;
1304         group = slot >> 3;
1305         if (dw0.h)
1306                 group = ~group;
1307         pi = (vsid ^ group) & 0x7ff;
1308         pi_high = (dw0.avpn & 0x1f) << 11;
1309         pi |= pi_high;
1310
1311         if (large)
1312                 va = pi << LARGE_PAGE_SHIFT;
1313         else
1314                 va = pi << PAGE_SHIFT;
1315
1316         spin_lock_irqsave(&pSeries_tlbie_lock, flags);
1317         _tlbie(va, large);
1318         spin_unlock_irqrestore(&pSeries_tlbie_lock, flags);
1319
1320         return i;
1321 }
1322
1323 static long rpa_lpar_hpte_remove(unsigned long hpte_group)
1324 {
1325         unsigned long slot_offset;
1326         unsigned long lpar_rc;
1327         int i;
1328         unsigned long dummy1, dummy2;
1329
1330         /* pick a random slot to start at */
1331         slot_offset = mftb() & 0x7;
1332
1333         for (i = 0; i < HPTES_PER_GROUP; i++) {
1334
1335                 /* Don't remove a bolted entry */
1336                 lpar_rc = plpar_pte_remove(H_ANDCOND, hpte_group + slot_offset,
1337                                            (0x1UL << 4), &dummy1, &dummy2);
1338
1339                 if (lpar_rc == H_Success)
1340                         return i;
1341
1342                 if (lpar_rc != H_Not_Found)
1343                         panic("Bad return code from pte remove rc = %lx\n",
1344                               lpar_rc);
1345
1346                 slot_offset++;
1347                 slot_offset &= 0x7;
1348         }
1349
1350         return −1;
```

```
1351 }
1352
1353 static long iSeries_hpte_remove(unsigned long hpte_group)
1354 {
1355         unsigned long slot_offset;
1356         int i;
1357         HPTE lhpte;
1358
1359         /* Pick a random slot to start at */
1360         slot_offset = mftb() & 0x7;
1361
1362         for (i = 0; i < HPTES_PER_GROUP; i++) {
1363                 lhpte.dw0.dword0 =
1364                         iSeries_hpte_getword0(hpte_group + slot_offset);
1365
1366                 if (!lhpte.dw0.dw0.bolted) {
1367                         HvCallHpt_invalidateSetSwBitsGet(hpte_group +
1368                                                          slot_offset, 0, 0);
1369                         return i;
1370                 }
1371
1372                 slot_offset++;
1373                 slot_offset &= 0x7;
1374         }
1375
1376         return -1;
1377 }
1378
1379 void hpte_init_pSeries(void)
1380 {
1381         ppc_md.hpte_invalidate     = hpte_invalidate;
1382         ppc_md.hpte_updatepp       = hpte_updatepp;
1383         ppc_md.hpte_updateboltedpp = hpte_updateboltedpp;
1384         ppc_md.hpte_insert         = hpte_insert;
1385         ppc_md.hpte_remove         = hpte_remove;
1386 }
1387
1388 void pSeries_lpar_mm_init(void)
1389 {
1390         ppc_md.hpte_invalidate     = rpa_lpar_hpte_invalidate;
1391         ppc_md.hpte_updatepp       = rpa_lpar_hpte_updatepp;
1392         ppc_md.hpte_updateboltedpp = rpa_lpar_hpte_updateboltedpp;
1393         ppc_md.hpte_insert         = rpa_lpar_hpte_insert;
1394         ppc_md.hpte_remove         = rpa_lpar_hpte_remove;
1395 }
1396
1397 void hpte_init_iSeries(void)
1398 {
1399         ppc_md.hpte_invalidate     = iSeries_hpte_invalidate;
1400         ppc_md.hpte_updatepp       = iSeries_hpte_updatepp;
1401         ppc_md.hpte_updateboltedpp = iSeries_hpte_updateboltedpp;
1402         ppc_md.hpte_insert         = iSeries_hpte_insert;
1403         ppc_md.hpte_remove         = iSeries_hpte_remove;
1404 }
1405
```

```
/*
 * HvCall.c
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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.
 */

#include <linux/stddef.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <asm/system.h>
#include <asm/page.h>
#include <asm/iSeries/HvCall.h>
#ifndef  _HVCALLSC_H
#include <asm/iSeries/HvCallSc.h>
#endif
#include <asm/iSeries/LparData.h>

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif


/*============================================================================
 * Note that this call takes at MOST one page worth of data
 */
int HvCall_readLogBuffer(HvLpIndex lpIndex, void *buffer, u64 bufLen)
{
        struct HvLpBufferList *bufList;
        u64 bytesLeft = bufLen;
        u64 leftThisPage;
        u64 curPtr = virt_to_absolute( (unsigned long) buffer );
        u64 retVal;
        int npages;
        int i;

        npages = 0;
        while (bytesLeft) {
                npages++;
                leftThisPage = ((curPtr & PAGE_MASK) + PAGE_SIZE) - curPtr;

                if (leftThisPage > bytesLeft)
                        bytesLeft = 0;
                else
                        bytesLeft -= leftThisPage;

                curPtr = (curPtr & PAGE_MASK) + PAGE_SIZE;
        }

        if (npages == 0)
                return 0;

        bufList = (struct HvLpBufferList *)
                kmalloc(npages * sizeof(struct HvLpBufferList), GFP_ATOMIC);
        bytesLeft = bufLen;
        curPtr = virt_to_absolute( (unsigned long) buffer );
        for(i=0; i<npages; i++) {
                bufList[i].addr = curPtr;

                leftThisPage = ((curPtr & PAGE_MASK) + PAGE_SIZE) - curPtr;

                if (leftThisPage > bytesLeft) {
                        bufList[i].len = bytesLeft;
                        bytesLeft = 0;
                } else {
                        bufList[i].len = leftThisPage;
                        bytesLeft -= leftThisPage;
                }

                curPtr = (curPtr & PAGE_MASK) + PAGE_SIZE;
        }


        retVal = HvCall3(HvCallBaseReadLogBuffer, lpIndex,
                         virt_to_absolute((unsigned long)bufList), bufLen);

        kfree(bufList);

        return (int)retVal;
}

/*============================================================================
 */
void HvCall_writeLogBuffer(const void *buffer, u64 bufLen)
{
        struct HvLpBufferList bufList;
```

```
91          u64 bytesLeft = bufLen;
92          u64 leftThisPage;
93          u64 curPtr = virt_to_absolute( (unsigned long) buffer );
94
95          while (bytesLeft) {
96                  bufList.addr = curPtr;
97
98                  leftThisPage = ((curPtr & PAGE_MASK) + PAGE_SIZE) − curPtr;
99
100                 if (leftThisPage > bytesLeft) {
101                         bufList.len = bytesLeft;
102                         bytesLeft = 0;
103                 } else {
104                         bufList.len = leftThisPage;
105                         bytesLeft −= leftThisPage;
106                 }
107
108                 curPtr = (curPtr & PAGE_MASK) + PAGE_SIZE;
109         }
110
111
112         HvCall2(HvCallBaseWriteLogBuffer,
113                 virt_to_absolute((unsigned long)&bufList), bufLen);
114
115 }
```

```
/*
 * HvLpConfig.c
 * Copyright (C) 2001  Kyle A. Lucke, IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#ifndef _HVLPCONFIG_H
#include <asm/iSeries/HvLpConfig.h>
#endif

HvLpIndex HvLpConfig_getLpIndex_outline(void)
{
        return HvLpConfig_getLpIndex();
}

```

```
/*
 * Copyright 2001 Mike Corrigan IBM Corp
 *
 * 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.
 */
#include <linux/stddef.h>
#include <linux/kernel.h>
#include <asm/system.h>
#include <asm/iSeries/HvLpEvent.h>
#include <asm/iSeries/HvCallEvent.h>
#include <asm/iSeries/LparData.h>

/* Array of LpEvent handler functions */
LpEventHandler lpEventHandler[HvLpEvent_Type_NumTypes];
unsigned lpEventHandlerPaths[HvLpEvent_Type_NumTypes];

/* Register a handler for an LpEvent type */

int HvLpEvent_registerHandler( HvLpEvent_Type eventType, LpEventHandler handler )
{
        int rc = 1;
        if ( eventType < HvLpEvent_Type_NumTypes ) {
                lpEventHandler[eventType] = handler;
                rc = 0;
        }
        return rc;

}

int HvLpEvent_unregisterHandler( HvLpEvent_Type eventType )
{
        int rc = 1;
        if ( eventType < HvLpEvent_Type_NumTypes ) {
                if ( !lpEventHandlerPaths[eventType] ) {
                        lpEventHandler[eventType] = NULL;
                        rc = 0;
                }
        }
        return rc;
}

/* (lpIndex is the partition index of the target partition.
 * needed only for VirtualIo, VirtualLan and SessionMgr.  Zero
 * indicates to use our partition index - for the other types)
 */
int HvLpEvent_openPath( HvLpEvent_Type eventType, HvLpIndex lpIndex )
{
        int rc = 1;
        if ( eventType < HvLpEvent_Type_NumTypes &&
             lpEventHandler[eventType] ) {
                if ( lpIndex == 0 )
                        lpIndex = itLpNaca.xLpIndex;
                HvCallEvent_openLpEventPath( lpIndex, eventType );
                ++lpEventHandlerPaths[eventType];
                rc = 0;
        }
        return rc;
}

int HvLpEvent_closePath( HvLpEvent_Type eventType, HvLpIndex lpIndex )
{
        int rc = 1;
        if ( eventType < HvLpEvent_Type_NumTypes &&
             lpEventHandler[eventType] &&
             lpEventHandlerPaths[eventType] ) {
                if ( lpIndex == 0 )
                        lpIndex = itLpNaca.xLpIndex;
                HvCallEvent_closeLpEventPath( lpIndex, eventType );
                --lpEventHandlerPaths[eventType];
                rc = 0;
        }
        return rc;
}

```

```
/************************************************************************/
/* This module supports the iSeries I/O Address translation mapping     */
/* Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                    */
/*                                                                      */
/* 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 program is distributed in the hope that it will be useful,      */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of       */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        */
/* GNU General Public License for more details.                         */
/*                                                                      */
/* 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.,                                      */
/* 59 Temple Place, Suite 330,                                          */
/* Boston, MA  02111-1307  USA                                          */
/************************************************************************/
/* Change Activity:                                                     */
/*   Created, December 14, 2000                                         */
/*   Added Bar table for IoMm performance.                              */
/*   Ported to ppc64                                                    */
/*   Added dynamic table allocation                                     */
/* End Change Activity                                                  */
/************************************************************************/
#include <asm/types.h>
#include <asm/resource.h>
#include <linux/pci.h>
#include <linux/spinlock.h>
#include <asm/ppcdebug.h>
#include <asm/flight_recorder.h>
#include <asm/iSeries/HvCallPci.h>
#include <asm/iSeries/iSeries_pci.h>

#include "iSeries_IoMmTable.h"
#include "pci.h"

/************************************************************************/
/* Table defines                                                        */
/* Each Entry size is 4 MB * 1024 Entries = 4GB I/O address space.      */
/************************************************************************/
#define Max_Entries 1024
unsigned long iSeries_IoMmTable_Entry_Size = 0x0000000000400000;
unsigned long iSeries_Base_Io_Memory       = 0xE000000000000000;
unsigned long iSeries_Max_Io_Memory        = 0xE000000000000000;
static   long iSeries_CurrentIndex         = 0;

/************************************************************************/
/* Lookup Tables.                                                       */
/************************************************************************/
struct iSeries_Device_Node** iSeries_IoMmTable;
u8*                          iSeries_IoBarTable;

/************************************************************************/
/* Static and Global variables                                          */
/************************************************************************/
static char*      iSeriesPciIoText     = "iSeries PCI I/O";
static spinlock_t iSeriesIoMmTableLock = SPIN_LOCK_UNLOCKED;

/************************************************************************/
/* iSeries_IoMmTable_Initialize                                         */
/************************************************************************/
/* Allocates and initalizes the Address Translation Table and Bar       */
/* Tables to get them ready for use.  Must be called before any         */
/* I/O space is handed out to the device BARs.                          */
/* A follow up method,iSeries_IoMmTable_Status can be called to         */
/* adjust the table after the device BARs have been assiged to          */
/* resize the table.                                                    */
/************************************************************************/
void iSeries_IoMmTable_Initialize(void)
{
        spin_lock(&iSeriesIoMmTableLock);
        iSeries_IoMmTable  = kmalloc(sizeof(void*)*Max_Entries,GFP_KERNEL);
        iSeries_IoBarTable = kmalloc(sizeof(u8)*Max_Entries,   GFP_KERNEL);
        spin_unlock(&iSeriesIoMmTableLock);
        PCIFR("IoMmTable Initialized 0x%p",  iSeries_IoMmTable);
        if(iSeries_IoMmTable == NULL || iSeries_IoBarTable == NULL) {
                panic("PCI: I/O tables allocation failed.\n");
        }
}

/************************************************************************/
/* iSeries_IoMmTable_AllocateEntry                                      */
/************************************************************************/
/* Adds pci_dev entry in address translation table                      */
/************************************************************************/
/* - Allocates the number of entries required in table base on BAR */
/*   size.                                                              */
```

```
91  /* − Allocates starting at iSeries_Base_Io_Memory and increases.   */
92  /* − The size is round up to be a multiple of entry size.          */
93  /* − CurrentIndex is incremented to keep track of the last entry.  */
94  /* − Builds the resource entry for allocated BARs.                 */
95  /*****************************************************************/
96  static void iSeries_IoMmTable_AllocateEntry(struct pci_dev* PciDev, int BarNumber)
97  {
98          struct resource* BarResource = &PciDev−>resource[BarNumber];
99          long             BarSize     = pci_resource_len(PciDev,BarNumber);
100         /*************************************************************/
101         /* No space to allocate, quick exit, skip Allocation.        */
102         /*************************************************************/
103         if(BarSize == 0) return;
104         /*************************************************************/
105         /* Set Resource values.                                      */
106         /*************************************************************/
107         spin_lock(&iSeriesIoMmTableLock);
108         BarResource−>name  = iSeriesPciIoText;
109         BarResource−>start = iSeries_IoMmTable_Entry_Size*iSeries_CurrentIndex;
110         BarResource−>start+= iSeries_Base_Io_Memory;
111         BarResource−>end   = BarResource−>start+BarSize−1;
112         /*************************************************************/
113         /* Allocate the number of table entries needed for BAR.      */
114         /*************************************************************/
115         while (BarSize > 0 ) {
116                 *(iSeries_IoMmTable +iSeries_CurrentIndex) = (struct iSeries_Device_Node*)PciDev−>sysdata;
117                 *(iSeries_IoBarTable+iSeries_CurrentIndex) = BarNumber;
118                 BarSize −= iSeries_IoMmTable_Entry_Size;
119                 ++iSeries_CurrentIndex;
120         }
121         iSeries_Max_Io_Memory = (iSeries_IoMmTable_Entry_Size*iSeries_CurrentIndex)+iSeries_Base_Io_Memory;
122         spin_unlock(&iSeriesIoMmTableLock);
123 }
124
125 /*****************************************************************/
126 /* iSeries_allocateDeviceBars                                    */
127 /*****************************************************************/
128 /* − Allocates ALL pci_dev BAR's and updates the resources with the*/
129 /*   BAR value.  BARS with zero length will have the resources     */
130 /*   The HvCallPci_getBarParms is used to get the size of the BAR  */
131 /*   space.  It calls iSeries_IoMmTable_AllocateEntry to allocate  */
132 /*   each entry.                                                   */
133 /* − Loops through The Bar resourses(0 − 5) including the ROM      */
134 /*   is resource(6).                                               */
135 /*****************************************************************/
136 void iSeries_allocateDeviceBars(struct pci_dev* PciDev)
137 {
138         struct resource* BarResource;
139         int              BarNumber;
140         for(BarNumber = 0; BarNumber <= PCI_ROM_RESOURCE; ++BarNumber) {
141                 BarResource = &PciDev−>resource[BarNumber];
142                 iSeries_IoMmTable_AllocateEntry(PciDev, BarNumber);
143         }
144 }
145
146 /*****************************************************************/
147 /* Translates the IoAddress to the device that is mapped to IoSpace.    */
148 /* This code is inlined, see the iSeries_pci.c file for the replacement.*/
149 /*****************************************************************/
150 struct iSeries_Device_Node* iSeries_xlateIoMmAddress(void* IoAddress)
151 {
152         return NULL;
153 }
154
155 /*********************************************************************
156  * Status hook for IoMmTable
157  *********************************************************************/
158 void     iSeries_IoMmTable_Status(void)
159 {
160         PCIFR("IoMmTable.....: 0x%p",iSeries_IoMmTable);
161         PCIFR("IoMmTable Range: 0x%p to 0x%p",iSeries_Base_Io_Memory,iSeries_Max_Io_Memory);
162         return;
163 }
```

```
#ifndef _ISERIES_IOMMTABLE_H
#define _ISERIES_IOMMTABLE_H
/*****************************************************************************/
/* File iSeries_IoMmTable.h created by Allan Trautman on Dec 12 2001.        */
/*****************************************************************************/
/* Interfaces for the write/read Io address translation table.               */
/* Copyright (C) 20yy  Allan H Trautman, IBM Corporation                     */
/*                                                                           */
/* 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 program is distributed in the hope that it will be useful,           */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of            */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the             */
/* GNU General Public License for more details.                              */
/*                                                                           */
/* 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.,                                           */
/* 59 Temple Place, Suite 330,                                               */
/* Boston, MA  02111-1307  USA                                               */
/*****************************************************************************/
/* Change Activity:                                                          */
/*   Created December 12, 2000                                               */
/*   Ported to ppc64, August 30, 2001                                        */
/* End Change Activity                                                       */
/*****************************************************************************/

struct pci_dev;
struct iSeries_Device_Node;

extern struct iSeries_Device_Node** iSeries_IoMmTable;
extern u8*                          iSeries_IoBarTable;
extern unsigned long iSeries_Base_Io_Memory;
extern unsigned long iSeries_Max_Io_Memory;
extern unsigned long iSeries_Base_Io_Memory;
extern unsigned long iSeries_IoMmTable_Entry_Size;
/*****************************************************************************/
/* iSeries_IoMmTable_Initialize                                              */
/*****************************************************************************/
/* - Initalizes the Address Translation Table and get it ready for use.      */
/*   Must be called before any client calls any of the other methods.        */
/*                                                                           */
/* Parameters: None.                                                         */
/*                                                                           */
/* Return: None.                                                             */
/*****************************************************************************/
extern  void iSeries_IoMmTable_Initialize(void);
extern  void iSeries_IoMmTable_Status(void);

/*****************************************************************************/
/* iSeries_allocateDeviceBars                                                */
/*****************************************************************************/
/* - Allocates ALL pci_dev BAR's and updates the resources with the BAR      */
/*   value.  BARS with zero length will not have the resources.  The         */
/*   HvCallPci_getBarParms is used to get the size of the BAR space.         */
/*   It calls iSeries_IoMmTable_AllocateEntry to allocate each entry.        */
/*                                                                           */
/* Parameters:                                                               */
/* pci_dev = Pointer to pci_dev structure that will be mapped to pseudo      */
/*           I/O Address.                                                    */
/*                                                                           */
/* Return:                                                                   */
/*   The pci_dev I/O resources updated with pseudo I/O Addresses.            */
/*****************************************************************************/
extern  void iSeries_allocateDeviceBars(struct pci_dev* );

/*****************************************************************************/
/* iSeries_xlateIoMmAddress                                                  */
/*****************************************************************************/
/* - Translates an I/O Memory address to Device Node that has been the       */
/*   allocated the psuedo I/O Address.                                       */
/*                                                                           */
/* Parameters:                                                               */
/* IoAddress = I/O Memory Address.                                           */
/*                                                                           */
/* Return:                                                                   */
/*   An iSeries_Device_Node to the device mapped to the I/O address. The     */
/*   BarNumber and BarOffset are valid if the Device Node is returned.       */
/*****************************************************************************/
extern struct iSeries_Device_Node* iSeries_xlateIoMmAddress(void* IoAddress);

#endif /* _ISERIES_IOMMTABLE_H */
```

```
/************************************************************************/
/* This module supports the iSeries PCI bus interrupt handling          */
/* Copyright (C) 20yy  <Robert L Holtorf> <IBM Corp>                    */
/*                                                                      */
/* 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 program is distributed in the hope that it will be useful,      */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of       */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        */
/* GNU General Public License for more details.                         */
/*                                                                      */
/* 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.,                                      */
/* 59 Temple Place, Suite 330,                                          */
/* Boston, MA  02111-1307  USA                                          */
/************************************************************************/
/* Change Activity:                                                     */
/*   Created, December 13, 2000 by Wayne Holm                           */
/* End Change Activity                                                  */
/************************************************************************/
#include <linux/pci.h>
#include <linux/init.h>
#include <linux/threads.h>
#include <linux/smp.h>
#include <linux/param.h>
#include <linux/string.h>
#include <linux/bootmem.h>
#include <linux/blk.h>
#include <linux/ide.h>

#include <linux/irq.h>
#include <linux/spinlock.h>
#include <asm/ppcdebug.h>

#include <asm/iSeries/HvCallPci.h>
#include <asm/iSeries/HvCallXm.h>
#include <asm/iSeries/iSeries_irq.h>
#include <asm/iSeries/XmPciLpEvent.h>


hw_irq_controller iSeries_IRQ_handler = {
        "iSeries irq controller",
        iSeries_startup_IRQ,    /* startup */
        iSeries_shutdown_IRQ,   /* shutdown */
        iSeries_enable_IRQ,     /* enable */
        iSeries_disable_IRQ,    /* disable */
        NULL,                   /* ack  */
        iSeries_end_IRQ,        /* end  */
        NULL                    /* set_affinity */
};


struct iSeries_irqEntry {
        u32 dsa;
        struct iSeries_irqEntry* next;
};

struct iSeries_irqAnchor {
        u8  valid : 1;
        u8  reserved : 7;
        u16  entryCount;
        struct iSeries_irqEntry* head;
};

struct iSeries_irqAnchor iSeries_irqMap[NR_IRQS];

void iSeries_init_irqMap(int irq);

/*  This is called by init_IRQ.  set in ppc_md.init_IRQ by iSeries_setup.c */
void __init iSeries_init_IRQ(void)
{
        int i;
        for (i = 0; i < NR_IRQS; i++) {
                irq_desc[i].handler = &iSeries_IRQ_handler;
                irq_desc[i].status = 0;
                irq_desc[i].status |= IRQ_DISABLED;
                irq_desc[i].depth = 1;
                iSeries_init_irqMap(i);
        }
        /* Register PCI event handler and open an event path */
        PPCDBG(PPCDBG_BUSWALK,"Register PCI event handler and open an event path\n");
        XmPciLpEvent_init();
        return;
}

/*********************************************************************
```

```
 *  Called by iSeries_init_IRQ
 * Prevent IRQs 0 and 255 from being used.  IRQ 0 appears in
 * uninitialized devices.  IRQ 255 appears in the PCI interrupt
 * line register if a PCI error occurs,
 ************************************************************************/
void __init iSeries_init_irqMap(int irq)
{
        iSeries_irqMap[irq].valid = (irq == 0 || irq == 255)? 0 : 1;
        iSeries_irqMap[irq].entryCount = 0;
        iSeries_irqMap[irq].head = NULL;
}

/* This is called out of iSeries_scan_slot to allocate an IRQ for an EADS slot */
/* It calculates the irq value for the slot.                                   */
int __init iSeries_allocate_IRQ(HvBusNumber busNumber, HvSubBusNumber subBusNumber, HvAgentId deviceId)
{
        u8 idsel = (deviceId >> 4);
        u8 function = deviceId & 0x0F;
        int irq = ((((busNumber-1)*16 + (idsel-1)*8 + function)*9/8) % 253) + 2;
        return irq;
}

/* This is called out of iSeries_scan_slot to assign the EADS slot to its IRQ number */
int __init iSeries_assign_IRQ(int irq, HvBusNumber busNumber, HvSubBusNumber subBusNumber, HvAgentId deviceId)
{
        int rc;
        u32 dsa = (busNumber << 16) | (subBusNumber << 8) | deviceId;
        struct iSeries_irqEntry* newEntry;
        unsigned long flags;

        if (irq < 0 || irq >= NR_IRQS) {
                return -1;
        }
        newEntry = kmalloc(sizeof(*newEntry), GFP_KERNEL);
        if (newEntry == NULL) {
                return -ENOMEM;
        }
        newEntry->dsa  = dsa;
        newEntry->next = NULL;
        /*********************************************************************
        * Probably not necessary to lock the irq since allocation is only
        * done during buswalk, but it should not hurt anything except a
        * little performance to be smp safe.
        *********************************************************************/
        spin_lock_irqsave(&irq_desc[irq].lock, flags);

        if (iSeries_irqMap[irq].valid) {
                /* Push the new element onto the irq stack */
                newEntry->next = iSeries_irqMap[irq].head;
                iSeries_irqMap[irq].head = newEntry;
                ++iSeries_irqMap[irq].entryCount;
                rc = 0;
                PPCDBG(PPCDBG_BUSWALK,"iSeries_assign_IRQ  0x%04X.%02X.%02X = 0x%04X\n",busNumber, subBusNumber, deviceId
, irq);
        }
        else {
                printk("PCI: Something is wrong with the iSeries_irqMap. \n");
                kfree(newEntry);
                rc = -1;
    }
        spin_unlock_irqrestore(&irq_desc[irq].lock, flags);
        return rc;
}


/* This is called by iSeries_activate_IRQs */
unsigned int iSeries_startup_IRQ(unsigned int irq)
{
        struct iSeries_irqEntry* entry;
        u32 bus, subBus, deviceId, function, mask;
        for(entry=iSeries_irqMap[irq].head; entry!=NULL; entry=entry->next) {
                bus      = (entry->dsa >> 16) & 0xFFFF;
                subBus   = (entry->dsa >> 8) & 0xFF;
                deviceId = entry->dsa & 0xFF;
                function = deviceId & 0x0F;
                /* Link the IRQ number to the bridge */
                HvCallXm_connectBusUnit(bus, subBus, deviceId, irq);
                /* Unmask bridge interrupts in the FISR */
                mask = 0x01010000 << function;
                HvCallPci_unmaskFisr(bus, subBus, deviceId, mask);
                PPCDBG(PPCDBG_BUSWALK,"iSeries_activate_IRQ 0x%02X.%02X.%02X  Irq:0x%02X\n",bus,subBus,deviceId,irq);
        }
        return 0;
}

/* This is called out of iSeries_fixup to activate interrupt
 * generation for usable slots                                 */
void __init iSeries_activate_IRQs()
{
        int irq;
```

```
180         unsigned long flags;
181         for (irq=0; irq < NR_IRQS; irq++) {
182                 spin_lock_irqsave(&irq_desc[irq].lock, flags);
183                 irq_desc[irq].handler->startup(irq);
184                 spin_unlock_irqrestore(&irq_desc[irq].lock, flags);
185         }
186 }
187
188 /*  this is not called anywhere currently */
189 void iSeries_shutdown_IRQ(unsigned int irq) {
190         struct iSeries_irqEntry* entry;
191         u32 bus, subBus, deviceId, function, mask;
192
193         /* irq should be locked by the caller */
194
195         for (entry=iSeries_irqMap[irq].head; entry; entry=entry->next) {
196                 bus = (entry->dsa >> 16) & 0xFFFF;
197                 subBus = (entry->dsa >> 8) & 0xFF;
198                 deviceId = entry->dsa & 0xFF;
199                 function = deviceId & 0x0F;
200                 /* Invalidate the IRQ number in the bridge */
201                 HvCallXm_connectBusUnit(bus, subBus, deviceId, 0);
202                 /* Mask bridge interrupts in the FISR */
203                 mask = 0x01010000 << function;
204                 HvCallPci_maskFisr(bus, subBus, deviceId, mask);
205         }
206
207 }
208
209 /*********************************************************
210  * This will be called by device drivers (via disable_IRQ)
211  * to disable INTA in the bridge interrupt status register.
212  *********************************************************/
213 void iSeries_disable_IRQ(unsigned int irq)
214 {
215         struct iSeries_irqEntry* entry;
216         u32 bus, subBus, deviceId, mask;
217
218         /* The IRQ has already been locked by the caller */
219
220         for (entry=iSeries_irqMap[irq].head; entry; entry=entry->next) {
221                 bus      = (entry->dsa >> 16) & 0xFFFF;
222                 subBus   = (entry->dsa >> 8) & 0xFF;
223                 deviceId = entry->dsa & 0xFF;
224                 /* Mask secondary INTA   */
225                 mask = 0x80000000;
226                 HvCallPci_maskInterrupts(bus, subBus, deviceId, mask);
227                 PPCDBG(PPCDBG_BUSWALK,"iSeries_disable_IRQ 0x%02X.%02X.%02X 0x%04X\n",bus,subBus,deviceId,irq);
228         }
229 }
230
231 /*********************************************************
232  * This will be called by device drivers (via enable_IRQ)
233  * to enable INTA in the bridge interrupt status register.
234  *********************************************************/
235 void iSeries_enable_IRQ(unsigned int irq)
236 {
237         struct iSeries_irqEntry* entry;
238         u32 bus, subBus, deviceId, mask;
239
240         /* The IRQ has already been locked by the caller */
241         for (entry=iSeries_irqMap[irq].head; entry; entry=entry->next) {
242                 bus      = (entry->dsa >> 16) & 0xFFFF;
243                 subBus   = (entry->dsa >> 8) & 0xFF;
244                 deviceId = entry->dsa & 0xFF;
245                 /* Unmask secondary INTA */
246                 mask = 0x80000000;
247                 HvCallPci_unmaskInterrupts(bus, subBus, deviceId, mask);
248                 PPCDBG(PPCDBG_BUSWALK,"iSeries_enable_IRQ 0x%02X.%02X.%02X 0x%04X\n",bus,subBus,deviceId,irq);
249         }
250 }
251
252 /* Need to define this so ppc_irq_dispatch_handler will NOT call
253    enable_IRQ at the end of interrupt handling.  However, this
254    does nothing because there is not enough information provided
255    to do the EOI HvCall.  This is done by XmPciLpEvent.c */
256 void iSeries_end_IRQ(unsigned int irq)
257 {
258 }
259
```

```
/************************************************************************/
/* File iSeries_pci_reset.c created by Allan Trautman on Mar 21 2001.   */
/************************************************************************/
/* This code supports the pci interface on the IBM iSeries systems.     */
/* Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                     */
/*                                                                      */
/* 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 program is distributed in the hope that it will be useful,      */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of       */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        */
/* GNU General Public License for more details.                         */
/*                                                                      */
/* 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.,                                      */
/* 59 Temple Place, Suite 330,                                          */
/* Boston, MA  02111-1307  USA                                          */
/************************************************************************/
/* Change Activity:                                                     */
/*   Created, March 20, 2001                                            */
/*   April 30, 2001, Added return codes on functions.                   */
/*   September 10, 2001, Ported to ppc64.                               */
/* End Change Activity                                                  */
/************************************************************************/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/irq.h>

#include <asm/io.h>
#include <asm/init.h>
#include <asm/iSeries/HvCallPci.h>
#include <asm/iSeries/HvTypes.h>
#include <asm/iSeries/mf.h>
#include <asm/flight_recorder.h>
#include <asm/pci.h>

#include <asm/iSeries/iSeries_pci.h>
#include "pci.h"

/************************************************************************/
/* Interface to toggle the reset line                                   */
/* Time is in .1 seconds, need for seconds.                             */
/************************************************************************/
int  iSeries_Device_ToggleReset(struct pci_dev* PciDev, int AssertTime, int DelayTime)
{
        unsigned long AssertDelay, WaitDelay;
        struct iSeries_Device_Node* DeviceNode = (struct iSeries_Device_Node*)PciDev->sysdata;
        if (DeviceNode == NULL) {
                printk("PCI: Pci Reset Failed, Device Node not found for pci_dev %p\n",PciDev);
                return -1;
        }
        /*****************************************************************
         * Set defaults, Assert is .5 second, Wait is 3 seconds.
         *****************************************************************/
        if (AssertTime == 0) AssertDelay = ( 5 * HZ)/10;
        else                 AssertDelay = (AssertTime*HZ)/10;
        if (WaitDelay == 0)  WaitDelay   = (30 * HZ)/10;
        else                 WaitDelay   = (DelayTime* HZ)/10;

        /*****************************************************************
         * Assert reset
         *****************************************************************/
        DeviceNode->ReturnCode = HvCallPci_setSlotReset(ISERIES_BUS(DeviceNode),0x00,DeviceNode->AgentId,1);
        if (DeviceNode->ReturnCode == 0) {
                set_current_state(TASK_UNINTERRUPTIBLE);
                schedule_timeout(AssertDelay);       /* Sleep for the time     */
                DeviceNode->ReturnCode = HvCallPci_setSlotReset(ISERIES_BUS(DeviceNode),0x00,DeviceNode->AgentId,
 0);

                /*************************************************************
                 * Wait for device to reset
                 *************************************************************/
                set_current_state(TASK_UNINTERRUPTIBLE);
                schedule_timeout(WaitDelay);
        }
        if (DeviceNode->ReturnCode == 0) {
                PCIFR("Slot 0x%04X.%02X Reset\n",ISERIES_BUS(DeviceNode),DeviceNode->AgentId );
        }
        else {
                printk("PCI: Slot 0x%04X.%02X Reset Failed, RCode: %04X\n",ISERIES_BUS(DeviceNode),DeviceNode->AgentId,Devic
eNode->ReturnCode);
                PCIFR(       "Slot 0x%04X.%02X Reset Failed, RCode: %04X\n",ISERIES_BUS(DeviceNode),DeviceNode->AgentId,Dev
iceNode->ReturnCode);
        }
        return DeviceNode->ReturnCode;
```

```
88 }
```

```
/************************************************************************/
/* File iSeries_vpdInfo.c created by Allan Trautman on Fri Feb  2 2001. */
/************************************************************************/
/* This code gets the card location of the hardware                     */
/* Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                    */
/*                                                                      */
/* 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 program is distributed in the hope that it will be useful,      */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of       */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        */
/* GNU General Public License for more details.                         */
/*                                                                      */
/* 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.,                                      */
/* 59 Temple Place, Suite 330,                                          */
/* Boston, MA  02111-1307  USA                                          */
/************************************************************************/
/* Change Activity:                                                     */
/*   Created, Feb 2, 2001                                              */
/*   Ported to ppc64, August 20, 2001                                  */
/* End Change Activity                                                  */
/************************************************************************/
#include <linux/init.h>
#include <linux/pci.h>
#include <asm/types.h>
#include <asm/resource.h>

#include <asm/iSeries/HvCallPci.h>
#include <asm/iSeries/HvTypes.h>
#include <asm/iSeries/mf.h>
#include <asm/iSeries/LparData.h>
#include <asm/iSeries/HvCallPci.h>
//#include <asm/iSeries/iSeries_VpdInfo.h>
#include <asm/iSeries/iSeries_pci.h>
#include "pci.h"

/*********************************************/
/* Size of Bus VPD data                      */
/*********************************************/
#define BUS_VPDSIZE      1024
/*********************************************/
/* Bus Vpd Tags                              */
/*********************************************/
#define  VpdEndOfDataTag   0x78
#define  VpdEndOfAreaTag   0x79
#define  VpdIdStringTag    0x82
#define  VpdVendorAreaTag  0x84
/*********************************************/
/* Mfg Area Tags                             */
/*********************************************/
#define  VpdFruFlag       0x4647     // "FG"
#define  VpdFruFrameId    0x4649     // "FI"
#define  VpdSlotMapFormat 0x4D46     // "MF"
#define  VpdAsmPartNumber 0x504E     // "PN"
#define  VpdFruSerial     0x534E     // "SN"
#define  VpdSlotMap       0x534D     // "SM"

/*********************************************/
/* Structures of the areas                   */
/*********************************************/
struct MfgVpdAreaStruct {
        u16 Tag;
        u8  TagLength;
        u8  AreaData1;
        u8  AreaData2;
};
typedef struct MfgVpdAreaStruct MfgArea;
#define MFG_ENTRY_SIZE   3

struct SlotMapStruct {
        u8   AgentId;
        u8   SecondaryAgentId;
        u8   PhbId;
        char CardLocation[3];
        char Parms[8];
        char Reserved[2];
};
typedef struct SlotMapStruct SlotMap;
#define SLOT_ENTRY_SIZE   16

/*************************************************************
 *                                                           *
 * Bus, Card, Board, FrameId, CardLocation.                  *
 *************************************************************/
LocationData* iSeries_GetLocationData(struct pci_dev* PciDev)
```

```
{
        struct iSeries_Device_Node* DevNode = (struct iSeries_Device_Node*)PciDev->sysdata;
        LocationData* LocationPtr = (LocationData*)kmalloc(LOCATION_DATA_SIZE, GFP_KERNEL);
        if (LocationPtr == NULL) {
                printk("PCI: LocationData area allocation failed!\n");
                return NULL;
        }
        memset(LocationPtr,0,LOCATION_DATA_SIZE);
        LocationPtr->Bus             = ISERIES_BUS(DevNode);
        LocationPtr->Board           = DevNode->Board;
        LocationPtr->FrameId         = DevNode->FrameId;
        LocationPtr->Card            = PCI_SLOT(DevNode->DevFn);
        strcpy(&LocationPtr->CardLocation[0],&DevNode->CardLocation[0]);
        return LocationPtr;
}

/*********************************************************************/
/* Formats the device information.                                   */
/* - Pass in pci_dev* pointer to the device.                         */
/* - Pass in buffer to place the data.  Danger here is the buffer must  */
/*   be as big as the client says it is.   Should be at least 128 bytes.*/
/* Return will the length of the string data put in the buffer.      */
/* Format:                                                           */
/* PCI: Bus  0, Device 26, Vendor 0x12AE  Frame  1, Card  C10  Ethernet */
/* controller                                                        */
/*********************************************************************/
int  iSeries_Device_Information(struct pci_dev* PciDev,char* Buffer, int BufferSize)
{
        struct iSeries_Device_Node* DevNode = (struct iSeries_Device_Node*)PciDev->sysdata;
        char*           BufPtr  = Buffer;
        int             LineLen = 0;

        if (DevNode == NULL) {
                LineLen = sprintf(BufPtr+LineLen, "PCI: iSeries_Device_Information DevNode is NULL");
                return LineLen;
        }

        if (BufferSize >= 128) {
                LineLen =  sprintf(BufPtr+LineLen,"PCI: Bus%3d, Device%3d, Vendor %04X ",
                                ISERIES_BUS(DevNode), PCI_SLOT(PciDev->devfn),PciDev->vendor);

                LineLen += sprintf(BufPtr+LineLen,"Frame%3d, Card %4s ", DevNode->FrameId,DevNode->CardLocation);

                if (pci_class_name(PciDev->class >> 8) == 0) {
                        LineLen += sprintf(BufPtr+LineLen,"0x%04X ",(int)(PciDev->class >> 8));
                }
                else {
                        LineLen += sprintf(BufPtr+LineLen,"%s",pci_class_name(PciDev->class >> 8) );
                }
        }
        return LineLen;
}
/*********************************************************************/
/* Build a character string of the device location, Frame  1, Card  C10 */
/*********************************************************************/
int   device_Location(struct pci_dev* PciDev,char* BufPtr)
{
        struct iSeries_Device_Node* DevNode = (struct iSeries_Device_Node*)PciDev->sysdata;
        return sprintf(BufPtr,"PCI: Bus%3d, AgentId%3d, Vendor %04X, Location %s",
                        DevNode->DsaAddr.busNumber,
                        DevNode->AgentId,
                        DevNode->Vendor,
                        DevNode->Location);
}

/*************************************************************/
/* Parse the Slot Area                                       */
/*************************************************************/
void  iSeries_Parse_SlotArea(SlotMap* MapPtr,int MapLen, struct iSeries_Device_Node* DevNode)
{
        int      SlotMapLen = MapLen;
        SlotMap* SlotMapPtr = MapPtr;
        /*******************************************************/
        /* Parse Slot label until we find the one requrested   */
        /*******************************************************/
        while (SlotMapLen > 0) {
                if (SlotMapPtr->AgentId == DevNode->AgentId ) {
                        /*************************************************/
                        /* If Phb wasn't found, grab the entry first one found.*/
                        /*************************************************/
                        if (DevNode->PhbId == 0xff) {
                                DevNode->PhbId = SlotMapPtr->PhbId;
                        }
                        /*************************************************/
                        /* Found it, extract the data.                   */
                        /*************************************************/
                        if (SlotMapPtr->PhbId == DevNode->PhbId ) {
                                memcpy(&DevNode->CardLocation,&SlotMapPtr->CardLocation,3);
                                DevNode->CardLocation[3]  = 0;
                                break;
```

```
                        }
                }
                /*****************************************************/
                /* Point to the next Slot                            */
                /*****************************************************/
                SlotMapPtr = (SlotMap*)((char*)SlotMapPtr+SLOT_ENTRY_SIZE);
                SlotMapLen −= SLOT_ENTRY_SIZE;
        }
}

/*************************************************************/
/* Parse the Mfg Area                                        */
/*************************************************************/
static void  iSeries_Parse_MfgArea(u8* AreaData,int AreaLen, struct iSeries_Device_Node* DevNode)
{
        MfgArea* MfgAreaPtr = (MfgArea*)AreaData;
        int      MfgAreaLen = AreaLen;
        u16      SlotMapFmt = 0;

        /*****************************************************/
        /* Parse Mfg Data                                    */
        /*****************************************************/
        while (MfgAreaLen > 0) {
                int MfgTagLen  = MfgAreaPtr−>TagLength;
                /*****************************************************/
                /* Frame ID         (FI 4649020310 )                 */
                /*****************************************************/
                if (MfgAreaPtr−>Tag == VpdFruFrameId) {     /* FI  */
                        DevNode−>FrameId = MfgAreaPtr−>AreaData1;
                }
                /*****************************************************/
                /* Slot Map Format  (MF 4D46020004 )                 */
                /*****************************************************/
                else if (MfgAreaPtr−>Tag == VpdSlotMapFormat){   /* MF  */
                        SlotMapFmt = (MfgAreaPtr−>AreaData1*256)+(MfgAreaPtr−>AreaData2);
                }
                /*****************************************************/
                /* Slot Map         (SM 534D90                       */
                /*****************************************************/
                else if (MfgAreaPtr−>Tag == VpdSlotMap){         /* SM  */
                        SlotMap*  SlotMapPtr;
                        if (SlotMapFmt == 0x1004) SlotMapPtr = (SlotMap*)((char*)MfgAreaPtr+MFG_ENTRY_SIZE+1);
                        else                      SlotMapPtr = (SlotMap*)((char*)MfgAreaPtr+MFG_ENTRY_SIZE);
                        iSeries_Parse_SlotArea(SlotMapPtr,MfgTagLen, DevNode);
                }
                /*****************************************************/
                /* Point to the next Mfg Area                        */
                /* Use defined size, sizeof give wrong answer        */
                /*****************************************************/
                MfgAreaPtr = (MfgArea*)((char*)MfgAreaPtr + MfgTagLen + MFG_ENTRY_SIZE);
                MfgAreaLen −= (MfgTagLen + MFG_ENTRY_SIZE);
        }
}

/*************************************************************/
/* Look for "BUS".. Data is not Null terminated.             */
/* PHBID of 0xFF indicates PHB was not found in VPD Data.    */
/*************************************************************/
static int iSeries_Parse_PhbId(u8* AreaPtr,int AreaLength)
{
        u8*  PhbPtr  = AreaPtr;
        int  DataLen = AreaLength;
        char PhbId   = 0xFF;
        while (DataLen > 0) {
                if (*PhbPtr == 'B' && *(PhbPtr+1) == 'U' && *(PhbPtr+2) == 'S') {
                        PhbPtr += 3;
                        while(*PhbPtr == ' ') ++PhbPtr;
                        PhbId = (*PhbPtr & 0x0F);
                        break;
                }
                ++PhbPtr;
                −−DataLen;
        }
        return PhbId;
}

/*************************************************************/
/* Parse out the VPD Areas                                   */
/*************************************************************/
static void  iSeries_Parse_Vpd(u8* VpdData, int VpdDataLen, struct iSeries_Device_Node* DevNode)
{
        u8*  TagPtr  = VpdData;
        int  DataLen = VpdDataLen−3;
        /*****************************************************/
        /* Parse the Areas                                   */
        /*****************************************************/
        while (*TagPtr != VpdEndOfAreaTag && DataLen > 0) {
                int  AreaLen   = *(TagPtr+1) + (*(TagPtr+2)*256);
                u8*  AreaData  = TagPtr+3;

```

```
271                 if (*TagPtr == VpdIdStringTag) {
272                         DevNode->PhbId = iSeries_Parse_PhbId(AreaData,AreaLen);
273                 }
274                 else if (*TagPtr == VpdVendorAreaTag) {
275                         iSeries_Parse_MfgArea(AreaData,AreaLen,DevNode);
276                 }
277                 /*************************************************
278                  * Point to next Area.
279                  *************************************************/
280                 TagPtr  = AreaData + AreaLen;
281                 DataLen −= AreaLen;
282         }
283 }
284
285 /************************************************************
286  *  iSeries_Get_Location_Code(struct iSeries_Device_Node*)      *
287  *
288  ************************************************************/
289 void  iSeries_Get_Location_Code(struct iSeries_Device_Node* DevNode)
290 {
291         int  BusVpdLen = 0;
292         u8*  BusVpdPtr = (u8*)kmalloc(BUS_VPDSIZE, GFP_KERNEL);
293         if (BusVpdPtr == NULL) {
294                 printk("PCI: Bus VPD Buffer allocation failure.\n");
295                 return;
296         }
297         BusVpdLen  = HvCallPci_getBusVpd(ISERIES_BUS(DevNode),REALADDR(BusVpdPtr),BUS_VPDSIZE);
298         if (BusVpdLen == 0) {
299                 kfree(BusVpdPtr);
300                 printk("PCI: Bus VPD Buffer zero length.\n");
301                 return;
302         }
303         //printk("PCI: BusVpdPtr: %p, %d\n",BusVpdPtr, BusVpdLen);
304         /************************************************************/
305         /* Make sure this is what I think it is                     */
306         /************************************************************/
307         if (*BusVpdPtr != VpdIdStringTag) {               /*0x82     */
308                 printk("PCI: Bus VPD Buffer missing starting tag.\n");
309                 kfree(BusVpdPtr);
310                 return;
311         }
312         /************************************************************/
313         /************************************************************/
314         iSeries_Parse_Vpd(BusVpdPtr,BusVpdLen, DevNode);
315         sprintf(DevNode->Location,"Frame%3d, Card %−4s",DevNode->FrameId,DevNode->CardLocation);
316         kfree(BusVpdPtr);
317 }
```

```
/*
 * ItLpQueue.c
 * Copyright (C) 2001 Mike Corrigan  IBM Corporation
 *
 * 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.
 */

#include <linux/stddef.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <asm/system.h>
#include <asm/paca.h>
#include <linux/random.h>
#include <asm/time.h>
#include <asm/iSeries/ItLpQueue.h>
#include <asm/iSeries/HvLpEvent.h>
#include <asm/iSeries/HvCallEvent.h>
#include <asm/iSeries/LparData.h>

static __inline__ int set_inUse( struct ItLpQueue * lpQueue )
{
        int t;
        u32 * inUseP = &(lpQueue->xInUseWord);

        __asm__ __volatile__("\n\
1:      lwarx   %0,0,%2         \n\
        cmpi    0,%0,0          \n\
        li      %0,0            \n\
        bne-    2f              \n\
        addi    %0,%0,1         \n\
        stwcx.  %0,0,%2         \n\
        bne-    1b              \n\
2:      eieio"
        : "=&r" (t), "=m" (lpQueue->xInUseWord)
        : "r" (inUseP), "m" (lpQueue->xInUseWord)
        : "cc");

        return t;
}

static __inline__ void clear_inUse( struct ItLpQueue * lpQueue )
{
        lpQueue->xInUseWord = 0;
}

/* Array of LpEvent handler functions */
extern LpEventHandler lpEventHandler[HvLpEvent_Type_NumTypes];
unsigned long ItLpQueueInProcess = 0;

struct HvLpEvent * ItLpQueue_getNextLpEvent( struct ItLpQueue * lpQueue )
{
        struct HvLpEvent * nextLpEvent =
                (struct HvLpEvent *)lpQueue->xSlicCurEventPtr;
        if ( nextLpEvent->xFlags.xValid ) {
                /* rmb() needed only for weakly consistent machines (regatta) */
                rmb();
                /* Set pointer to next potential event */
                lpQueue->xSlicCurEventPtr += ((nextLpEvent->xSizeMinus1 +
                                        LpEventAlign ) /
                                        LpEventAlign ) *
                                        LpEventAlign;
                /* Wrap to beginning if no room at end */
                if (lpQueue->xSlicCurEventPtr > lpQueue->xSlicLastValidEventPtr)
                        lpQueue->xSlicCurEventPtr = lpQueue->xSlicEventStackPtr;
        }
        else
                nextLpEvent = NULL;

        return nextLpEvent;
}

int ItLpQueue_isLpIntPending( struct ItLpQueue * lpQueue )
{
        int retval = 0;
        struct HvLpEvent * nextLpEvent;
        if ( lpQueue ) {
                nextLpEvent = (struct HvLpEvent *)lpQueue->xSlicCurEventPtr;
                retval = nextLpEvent->xFlags.xValid | lpQueue->xPlicOverflowIntPending;
        }
        return retval;
}

void ItLpQueue_clearValid( struct HvLpEvent * event )
{
        /* Clear the valid bit of the event
         * Also clear bits within this event that might
         * look like valid bits (on 64-byte boundaries)
```

```
91              */
92          unsigned extra = (( event->xSizeMinus1 + LpEventAlign ) /
93                                                  LpEventAlign ) - 1;
94          switch ( extra ) {
95            case 3:
96             ((struct HvLpEvent*)((char*)event+3*LpEventAlign))->xFlags.xValid=0;
97            case 2:
98             ((struct HvLpEvent*)((char*)event+2*LpEventAlign))->xFlags.xValid=0;
99            case 1:
100            ((struct HvLpEvent*)((char*)event+1*LpEventAlign))->xFlags.xValid=0;
101           case 0:
102            ;
103         }
104         mb();
105         event->xFlags.xValid = 0;
106 }
107
108 unsigned ItLpQueue_process( struct ItLpQueue * lpQueue, struct pt_regs *regs )
109 {
110         unsigned numIntsProcessed = 0;
111         struct HvLpEvent * nextLpEvent;
112
113         /* If we have recursed, just return */
114         if ( !set_inUse( lpQueue ) )
115                 return 0;
116
117         if (ItLpQueueInProcess == 0)
118                 ItLpQueueInProcess = 1;
119         else
120                 BUG();
121
122         for (;;) {
123                 nextLpEvent = ItLpQueue_getNextLpEvent( lpQueue );
124                 if ( nextLpEvent ) {
125                         /* Count events to return to caller
126                          * and count processed events in lpQueue
127                          */
128                         ++numIntsProcessed;
129                         lpQueue->xLpIntCount++;
130                         /* Call appropriate handler here, passing
131                          * a pointer to the LpEvent.  The handler
132                          * must make a copy of the LpEvent if it
133                          * needs it in a bottom half. (perhaps for
134                          * an ACK)
135                          *
136                          *  Handlers are responsible for ACK processing
137                          *
138                          * The Hypervisor guarantees that LpEvents will
139                          * only be delivered with types that we have
140                          * registered for, so no type check is necessary
141                          * here!
142                          */
143                         if ( nextLpEvent->xType < HvLpEvent_Type_NumTypes )
144                                 lpQueue->xLpIntCountByType[nextLpEvent->xType]++;
145                         if ( nextLpEvent->xType < HvLpEvent_Type_NumTypes &&
146                              lpEventHandler[nextLpEvent->xType] )
147                                 lpEventHandler[nextLpEvent->xType](nextLpEvent, regs);
148                         else
149                                 printk(KERN_INFO "Unexpected Lp Event type=%d\n", nextLpEvent->xType );
150
151                         ItLpQueue_clearValid( nextLpEvent );
152                 }
153                 else    /* No more valid events
154                          * If overflow events are pending
155                          * process them
156                          */
157                         if ( lpQueue->xPlicOverflowIntPending ) {
158                                 HvCallEvent_getOverflowLpEvents(
159                                                 lpQueue->xIndex);
160                         }
161                         else    /* If nothing left then we are done */
162                                 break;
163         }
164
165         ItLpQueueInProcess = 0;
166         mb();
167         clear_inUse( lpQueue );
168
169         get_paca()->lpEvent_count += numIntsProcessed;
170
171         /* Use LPEvents as a source of randomness.  Since there
172          * Isn't an LPEvent Randomness call, pretend these are
173          * mouse events (which is fair since we don't have mice
174          * on the iSeries)
175          */
176         add_mouse_randomness(get_tb());
177
178         return numIntsProcessed;
179 }
```

```
/*
 *
 * Procedures for interfacing to Open Firmware.
 *
 * Peter Bergner, IBM Corp.     June 2001.
 * Copyright (C) 2001 Peter Bergner.
 *
 *      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.
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <asm/types.h>
#include <asm/page.h>
#include <asm/prom.h>
#include <asm/lmb.h>
#include <asm/abs_addr.h>
#include <asm/bitops.h>
#include <asm/udbg.h>

extern unsigned long klimit;
extern unsigned long reloc_offset(void);


static long lmb_add_region(struct lmb_region *, unsigned long, unsigned long, unsigned long);

struct lmb lmb = {
        0, 0,
        {0,0,0,0,{{0,0,0}}},
        {0,0,0,0,{{0,0,0}}}
};


/* Assumption: base addr of region 1 < base addr of region 2 */
static void
lmb_coalesce_regions(struct lmb_region *rgn, unsigned long r1, unsigned long r2)
{
        unsigned long i;

        rgn->region[r1].size += rgn->region[r2].size;
        for (i=r2; i < rgn->cnt-1 ;i++) {
                rgn->region[i].base = rgn->region[i+1].base;
                rgn->region[i].physbase = rgn->region[i+1].physbase;
                rgn->region[i].size = rgn->region[i+1].size;
                rgn->region[i].type = rgn->region[i+1].type;
        }
        rgn->cnt--;
}


/* This routine called with relocation disabled. */
void
lmb_init(void)
{
        unsigned long offset = reloc_offset();
        struct lmb *_lmb = PTRRELOC(&lmb);

        /* Create a dummy zero size LMB which will get coalesced away later.
         * This simplifies the lmb_add() code below...
         */
        _lmb->memory.region[0].base = 0;
        _lmb->memory.region[0].size = 0;
        _lmb->memory.region[0].type = LMB_MEMORY_AREA;
        _lmb->memory.cnt = 1;

        /* Ditto. */
        _lmb->reserved.region[0].base = 0;
        _lmb->reserved.region[0].size = 0;
        _lmb->reserved.region[0].type = LMB_MEMORY_AREA;
        _lmb->reserved.cnt = 1;
}

/* This routine called with relocation disabled. */
void
lmb_analyze(void)
{
        unsigned long i;
        unsigned long mem_size = 0;
        unsigned long io_size = 0;
        unsigned long size_mask = 0;
        unsigned long offset = reloc_offset();
        struct lmb *_lmb = PTRRELOC(&lmb);
#ifdef CONFIG_MSCHUNKS
        unsigned long physbase = 0;
#endif

        for (i=0; i < _lmb->memory.cnt ;i++) {
```

```
91                  unsigned long lmb_type = _lmb->memory.region[i].type;
92                  unsigned long lmb_size;
93
94                  if ( lmb_type != LMB_MEMORY_AREA )
95                          continue;
96
97                  lmb_size = _lmb->memory.region[i].size;
98
99  #ifdef CONFIG_MSCHUNKS
100                 _lmb->memory.region[i].physbase = physbase;
101                 physbase += lmb_size;
102 #else
103                 _lmb->memory.region[i].physbase = _lmb->memory.region[i].base;
104 #endif
105                 mem_size += lmb_size;
106                 size_mask |= lmb_size;
107         }
108
109 #ifdef CONFIG_MSCHUNKS
110         for (i=0; i < _lmb->memory.cnt ;i++) {
111                 unsigned long lmb_type = _lmb->memory.region[i].type;
112                 unsigned long lmb_size;
113
114                 if ( lmb_type != LMB_IO_AREA )
115                         continue;
116
117                 lmb_size = _lmb->memory.region[i].size;
118
119                 _lmb->memory.region[i].physbase = physbase;
120                 physbase += lmb_size;
121                 io_size += lmb_size;
122                 size_mask |= lmb_size;
123         }
124 #endif /* CONFIG_MSCHUNKS */
125
126         _lmb->memory.size = mem_size;
127         _lmb->memory.iosize = io_size;
128         _lmb->memory.lcd_size = (1UL << cnt_trailing_zeros(size_mask));
129 }
130
131 /* This routine called with relocation disabled. */
132 long
133 lmb_add(unsigned long base, unsigned long size)
134 {
135         unsigned long offset = reloc_offset();
136         struct lmb *_lmb = PTRRELOC(&lmb);
137         struct lmb_region *_rgn = &(_lmb->memory);
138
139         /* On pSeries LPAR systems, the first LMB is our RMO region. */
140         if ( base == 0 )
141                 _lmb->rmo_size = size;
142
143         return lmb_add_region(_rgn, base, size, LMB_MEMORY_AREA);
144
145 }
146
147 #ifdef CONFIG_MSCHUNKS
148 /* This routine called with relocation disabled. */
149 long
150 lmb_add_io(unsigned long base, unsigned long size)
151 {
152         unsigned long offset = reloc_offset();
153         struct lmb *_lmb = PTRRELOC(&lmb);
154         struct lmb_region *_rgn = &(_lmb->memory);
155
156         return lmb_add_region(_rgn, base, size, LMB_IO_AREA);
157
158 }
159 #endif /* CONFIG_MSCHUNKS */
160
161 long
162 lmb_reserve(unsigned long base, unsigned long size)
163 {
164         unsigned long offset = reloc_offset();
165         struct lmb *_lmb = PTRRELOC(&lmb);
166         struct lmb_region *_rgn = &(_lmb->reserved);
167
168         return lmb_add_region(_rgn, base, size, LMB_MEMORY_AREA);
169 }
170
171 /* This routine called with relocation disabled. */
172 static long
173 lmb_add_region(struct lmb_region *rgn, unsigned long base, unsigned long size,
174                 unsigned long type)
175 {
176         unsigned long i, coalesced = 0;
177         long adjacent;
178
179         /* First try and coalesce this LMB with another. */
180         for (i=0; i < rgn->cnt ;i++) {
```

```
181                     unsigned long rgnbase = rgn−>region[i].base;
182                     unsigned long rgnsize = rgn−>region[i].size;
183                     unsigned long rgntype = rgn−>region[i].type;
184
185                     if ( rgntype != type )
186                             continue;
187
188                     adjacent = lmb_addrs_adjacent(base,size,rgnbase,rgnsize);
189                     if ( adjacent > 0 ) {
190                             rgn−>region[i].base −= size;
191                             rgn−>region[i].physbase −= size;
192                             rgn−>region[i].size += size;
193                             coalesced++;
194                             break;
195                     }
196                     else if ( adjacent < 0 ) {
197                             rgn−>region[i].size += size;
198                             coalesced++;
199                             break;
200                     }
201             }
202
203             if ((i < rgn−>cnt−1) && lmb_regions_adjacent(rgn, i, i+1) ) {
204                     lmb_coalesce_regions(rgn, i, i+1);
205                     coalesced++;
206             }
207
208             if ( coalesced ) {
209                     return coalesced;
210             } else if ( rgn−>cnt >= MAX_LMB_REGIONS ) {
211                     return −1;
212             }
213
214             /* Couldn't coalesce the LMB, so add it to the sorted table. */
215             for (i=rgn−>cnt−1; i >= 0 ;i−−) {
216                     if (base < rgn−>region[i].base) {
217                             rgn−>region[i+1].base = rgn−>region[i].base;
218                             rgn−>region[i+1].physbase = rgn−>region[i].physbase;
219                             rgn−>region[i+1].size = rgn−>region[i].size;
220                             rgn−>region[i+1].type = rgn−>region[i].type;
221                     }  else {
222                             rgn−>region[i+1].base = base;
223                             rgn−>region[i+1].physbase = lmb_abs_to_phys(base);
224                             rgn−>region[i+1].size = size;
225                             rgn−>region[i+1].type = type;
226                             break;
227                     }
228             }
229             rgn−>cnt++;
230
231             return 0;
232     }
233
234     long
235     lmb_overlaps_region(struct lmb_region *rgn, unsigned long base, unsigned long size)
236     {
237             unsigned long i;
238
239             for (i=0; i < rgn−>cnt ;i++) {
240                     unsigned long rgnbase = rgn−>region[i].base;
241                     unsigned long rgnsize = rgn−>region[i].size;
242                     if ( lmb_addrs_overlap(base,size,rgnbase,rgnsize) ) {
243                             break;
244                     }
245             }
246
247             return (i < rgn−>cnt) ? i : −1;
248     }
249
250     unsigned long
251     lmb_alloc(unsigned long size, unsigned long align)
252     {
253             return lmb_alloc_base(size, align, LMB_ALLOC_ANYWHERE);
254     }
255
256     unsigned long
257     lmb_alloc_base(unsigned long size, unsigned long align, unsigned long max_addr)
258     {
259             long i, j;
260             unsigned long base = 0;
261             unsigned long offset = reloc_offset();
262             struct lmb *_lmb = PTRRELOC(&lmb);
263             struct lmb_region *_mem = &(_lmb−>memory);
264             struct lmb_region *_rsv = &(_lmb−>reserved);
265
266             for (i=_mem−>cnt−1; i >= 0 ;i−−) {
267                     unsigned long lmbbase = _mem−>region[i].base;
268                     unsigned long lmbsize = _mem−>region[i].size;
269                     unsigned long lmbtype = _mem−>region[i].type;
270
```

```
271                     if ( lmbtype != LMB_MEMORY_AREA )
272                             continue;
273
274                     if ( max_addr == LMB_ALLOC_ANYWHERE )
275                             base = _ALIGN_DOWN(lmbbase+lmbsize-size, align);
276                     else if ( lmbbase < max_addr )
277                             base = _ALIGN_DOWN(min(lmbbase+lmbsize,max_addr)-size, align);
278                     else
279                             continue;
280
281                     while ( (lmbbase <= base) &&
282                             ((j = lmb_overlaps_region(_rsv,base,size)) >= 0) ) {
283                             base = _ALIGN_DOWN(_rsv->region[j].base-size, align);
284                     }
285
286                     if ( (base != 0) && (lmbbase <= base) )
287                             break;
288             }
289
290             if ( i < 0 )
291                     return 0;
292
293             lmb_add_region(_rsv, base, size, LMB_MEMORY_AREA);
294
295             return base;
296     }
297
298     unsigned long
299     lmb_phys_mem_size(void)
300     {
301             unsigned long offset = reloc_offset();
302             struct lmb *_lmb = PTRRELOC(&lmb);
303     #ifdef CONFIG_MSCHUNKS
304             return _lmb->memory.size;
305     #else
306             struct lmb_region *_mem = &(_lmb->memory);
307             unsigned long idx = _mem->cnt-1;
308             unsigned long lastbase = _mem->region[idx].physbase;
309             unsigned long lastsize = _mem->region[idx].size;
310
311             return (lastbase + lastsize);
312     #endif /* CONFIG_MSCHUNKS */
313     }
314
315     unsigned long
316     lmb_end_of_DRAM(void)
317     {
318             unsigned long offset = reloc_offset();
319             struct lmb *_lmb = PTRRELOC(&lmb);
320             struct lmb_region *_mem = &(_lmb->memory);
321             unsigned long idx;
322
323             for(idx=_mem->cnt-1; idx >= 0 ;idx--) {
324                     if ( _mem->region[idx].type != LMB_MEMORY_AREA )
325                             continue;
326     #ifdef CONFIG_MSCHUNKS
327                     return (_mem->region[idx].physbase + _mem->region[idx].size);
328     #else
329                     return (_mem->region[idx].base + _mem->region[idx].size);
330     #endif /* CONFIG_MSCHUNKS */
331             }
332
333             return 0;
334     }
335
336
337     unsigned long
338     lmb_abs_to_phys(unsigned long aa)
339     {
340             unsigned long i, pa = aa;
341             unsigned long offset = reloc_offset();
342             struct lmb *_lmb = PTRRELOC(&lmb);
343             struct lmb_region *_mem = &(_lmb->memory);
344
345             for (i=0; i < _mem->cnt ;i++) {
346                     unsigned long lmbbase = _mem->region[i].base;
347                     unsigned long lmbsize = _mem->region[i].size;
348                     if ( lmb_addrs_overlap(aa,1,lmbbase,lmbsize) ) {
349                             pa = _mem->region[i].physbase + (aa - lmbbase);
350                             break;
351                     }
352             }
353
354             return pa;
355     }
356
357     void
358     lmb_dump(char *str)
359     {
360             unsigned long i;
```

```
361
362             udbg_printf("\nlmb_dump: %s\n", str);
363             udbg_printf("  debug              = %s\n",
364                     (lmb.debug) ? "TRUE" : "FALSE");
365             udbg_printf("  memory.cnt         = %d\n",
366                     lmb.memory.cnt);
367             udbg_printf("  memory.size        = 0x%lx\n",
368                     lmb.memory.size);
369             udbg_printf("  memory.lcd_size    = 0x%lx\n",
370                     lmb.memory.lcd_size);
371             for (i=0; i < lmb.memory.cnt ;i++) {
372                     udbg_printf("  memory.region[%d].base     = 0x%lx\n",
373                             i, lmb.memory.region[i].base);
374                     udbg_printf("          .physbase = 0x%lx\n",
375                             lmb.memory.region[i].physbase);
376                     udbg_printf("          .size     = 0x%lx\n",
377                             lmb.memory.region[i].size);
378                     udbg_printf("          .type     = 0x%lx\n",
379                             lmb.memory.region[i].type);
380             }
381
382             udbg_printf("\n");
383             udbg_printf("  reserved.cnt       = %d\n",
384                     lmb.reserved.cnt);
385             udbg_printf("  reserved.size      = 0x%lx\n",
386                     lmb.reserved.size);
387             udbg_printf("  reserved.lcd_size  = 0x%lx\n",
388                     lmb.reserved.lcd_size);
389             for (i=0; i < lmb.reserved.cnt ;i++) {
390                     udbg_printf("  reserved.region[%d].base   = 0x%lx\n",
391                             i, lmb.reserved.region[i].base);
392                     udbg_printf("          .physbase = 0x%lx\n",
393                             lmb.reserved.region[i].physbase);
394                     udbg_printf("          .size     = 0x%lx\n",
395                             lmb.reserved.region[i].size);
396                     udbg_printf("          .type     = 0x%lx\n",
397                             lmb.reserved.region[i].type);
398             }
399     }
```

```
/*
 * Copyright 2001 Mike Corrigan, IBM Corp
 *
 * 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.
 */
#include <asm/types.h>
#include <asm/page.h>
#include <stddef.h>
#include <linux/threads.h>
#include <asm/processor.h>
#include <asm/ptrace.h>
#include <asm/naca.h>
#include <asm/abs_addr.h>
#include <asm/bitops.h>
#include <asm/iSeries/ItLpNaca.h>
#include <asm/iSeries/ItLpPaca.h>
#include <asm/iSeries/ItLpRegSave.h>
#include <asm/paca.h>
#include <asm/iSeries/HvReleaseData.h>
#include <asm/iSeries/LparMap.h>
#include <asm/iSeries/ItVpdAreas.h>
#include <asm/iSeries/ItIplParmsReal.h>
#include <asm/iSeries/ItExtVpdPanel.h>
#include <asm/iSeries/ItLpQueue.h>
#include <asm/iSeries/IoHriProcessorVpd.h>
#include <asm/iSeries/ItSpCommArea.h>

extern char _start_boltedStacks[];

/* The LparMap data is now located at offset 0x6000 in head.S
 * It was put there so that the HvReleaseData could address it
 * with a 32−bit offset as required by the iSeries hypervisor
 *
 * The Naca has a pointer to the ItVpdAreas.  The hypervisor finds
 * the Naca via the HvReleaseData area.  The HvReleaseData has the
 * offset into the Naca of the pointer to the ItVpdAreas.
 */

extern struct ItVpdAreas itVpdAreas;

/* The LpQueue is used to pass event data from the hypervisor to
 * the partition.  This is where I/O interrupt events are communicated.
 * The ItLpQueue must be initialized (even though only to all zeros)
 * If it were uninitialized (in .bss) it would get zeroed after the
 * kernel gets control.  The hypervisor will have filled in some fields
 * before the kernel gets control.  By initializing it we keep it out
 * of the .bss
 */

struct ItLpQueue xItLpQueue = {};


/* The HvReleaseData is the root of the information shared between
 * the hypervisor and Linux.
 */

struct HvReleaseData hvReleaseData = {
        0xc8a5d9c4,     /* desc = "HvRD" ebcdic */
        sizeof(struct HvReleaseData),
        offsetof(struct naca_struct, xItVpdAreas),
        (struct naca_struct *)(KERNELBASE+0x4000), /* 64−bit Naca address */
        0x6000,         /* offset of LparMap within loadarea (see head.S) */
        0,
        1,              /* tags inactive       */
        0,              /* 64 bit              */
        0,              /* shared processors   */
        0,              /* HMT allowed         */
        6,              /* TEMP: This allows non−GA driver */
        4,              /* We are v5r2m0             */
        3,              /* Min supported PLIC = v5r1m0 */
        3,              /* Min usuable PLIC   = v5r1m0 */
        { 0xd3, 0x89, 0x95, 0xa4, /* "Linux 2.4   "*/
          0xa7, 0x40, 0xf2, 0x4b,
          0xf4, 0x4b, 0xf6, 0xf4 },
        {0}
};

extern void SystemReset_Iseries(void);
extern void MachineCheck_Iseries(void);
extern void DataAccess_Iseries(void);
extern void InstructionAccess_Iseries(void);
extern void HardwareInterrupt_Iseries(void);
extern void Alignment_Iseries(void);
extern void ProgramCheck_Iseries(void);
extern void FPUnavailable_Iseries(void);
extern void Decrementer_Iseries(void);
extern void Trap_0a_Iseries(void);
```

```
91  extern void Trap_0b_Iseries(void);
92  extern void SystemCall_Iseries(void);
93  extern void SingleStep_Iseries(void);
94  extern void Trap_0e_Iseries(void);
95  extern void PerformanceMonitor_Iseries(void);
96  extern void DataAccessSLB_Iseries(void);
97  extern void InstructionAccessSLB_Iseries(void);
98
99  struct ItLpNaca itLpNaca = {
100         0xd397d581,     /* desc = "LpNa" ebcdic */
101         0x0400,         /* size of ItLpNaca     */
102         0x0300, 19,     /* offset to int array, # ents */
103         0, 0, 0,        /* Part # of primary, serv, me */
104         0, 0x100,       /* # of LP queues, offset */
105         0, 0, 0,        /* Piranha stuff */
106         { 0,0,0,0,0 },  /* reserved */
107         0,0,0,0,0,0,0,  /* stuff    */
108         { 0,0,0,0,0 },  /* reserved */
109         0,              /* reserved */
110         0,              /* VRM index of PLIC */
111         0, 0,           /* min supported, compat SLIC */
112         0,              /* 64−bit addr of load area   */
113         0,              /* chunks for load area  */
114         0, 0,           /* PASE mask, seg table  */
115         { 0 },          /* 64 reserved bytes  */
116         { 0 },          /* 128 reserved bytes */
117         { 0 },          /* Old LP Queue       */
118         { 0 },          /* 384 reserved bytes */
119         {
120                 (u64)SystemReset_Iseries,       /* 0x100 System Reset */
121                 (u64)MachineCheck_Iseries,      /* 0x200 Machine Check */
122                 (u64)DataAccess_Iseries,        /* 0x300 Data Access */
123                 (u64)InstructionAccess_Iseries, /* 0x400 Instruction Access */
124                 (u64)HardwareInterrupt_Iseries, /* 0x500 External */
125                 (u64)Alignment_Iseries,         /* 0x600 Alignment */
126                 (u64)ProgramCheck_Iseries,      /* 0x700 Program Check */
127                 (u64)FPUnavailable_Iseries,     /* 0x800 FP Unavailable */
128                 (u64)Decrementer_Iseries,       /* 0x900 Decrementer */
129                 (u64)Trap_0a_Iseries,           /* 0xa00 Trap 0A */
130                 (u64)Trap_0b_Iseries,           /* 0xb00 Trap 0B */
131                 (u64)SystemCall_Iseries,        /* 0xc00 System Call */
132                 (u64)SingleStep_Iseries,        /* 0xd00 Single Step */
133                 (u64)Trap_0e_Iseries,           /* 0xe00 Trap 0E */
134                 (u64)PerformanceMonitor_Iseries,/* 0xf00 Performance Monitor */
135                 0,                              /* int 0x1000 */
136                 0,                              /* int 0x1010 */
137                 0,                              /* int 0x1020 CPU ctls */
138                 (u64)HardwareInterrupt_Iseries, /* SC Ret Hdlr */
139                 (u64)DataAccessSLB_Iseries,     /* 0x380 D−SLB */
140                 (u64)InstructionAccessSLB_Iseries /* 0x480 I−SLB */
141         }
142 };
143
144 struct ItIplParmsReal xItIplParmsReal = {};
145
146 struct ItExtVpdPanel xItExtVpdPanel = {};
147
148 #define maxPhysicalProcessors 32
149
150 struct IoHriProcessorVpd xIoHriProcessorVpd[maxPhysicalProcessors] = {
151         {
152                 xInstCacheOperandSize: 32,
153                 xDataCacheOperandSize: 32,
154                 xProcFreq:     50000000,
155                 xTimeBaseFreq: 50000000,
156                 xPVR: 0x3600
157         }
158 };
159
160
161 u64    xMsVpd[3400] = {};              /* Space for Main Store Vpd 27,200 bytes */
162
163 u64    xRecoveryLogBuffer[32] = {};    /* Space for Recovery Log Buffer */
164
165 struct SpCommArea xSpCommArea = {
166         0xE2D7C3C2,
167         1,
168         {0},
169         0, 0, 0, 0, {0}
170 };
171
172 struct ItVpdAreas itVpdAreas = {
173         0xc9a3e5c1,     /* "ItVA" */
174         sizeof( struct ItVpdAreas ),
175         0, 0,
176         26,             /* # VPD array entries */
177         10,             /* # DMA array entries */
178         MAX_PROCESSORS*2, maxPhysicalProcessors,        /* Max logical, physical procs */
179         offsetof(struct ItVpdAreas,xPlicDmaToks),/* offset to DMA toks */
180         offsetof(struct ItVpdAreas,xSlicVpdAdrs),/* offset to VPD addrs */
```

```
181         offsetof(struct ItVpdAreas,xPlicDmaLens),/* offset to DMA lens */
182         offsetof(struct ItVpdAreas,xSlicVpdLens),/* offset to VPD lens */
183         0,              /* max slot labels */
184         1,              /* max LP queues */
185         {0}, {0},       /* reserved */
186         {0},            /* DMA lengths */
187         {0},            /* DMA tokens */
188         {               /* VPD lengths */
189                 0,0,0,                  /*  0 − 2 */
190                 sizeof(xItExtVpdPanel), /*       3 Extended VPD   */
191                 sizeof(struct paca_struct),     /*       4 length of Paca  */
192                 0,                      /*       5 */
193                 sizeof(struct ItIplParmsReal),/* 6 length of IPL parms */
194                 26992,                  /*       7 length of MS VPD */
195                 0,                      /*       8 */
196                 sizeof(struct ItLpNaca),/*       9 length of LP Naca */
197                 0,                      /*      10 */
198                 256,                    /*      11 length of Recovery Log Buf */
199                 sizeof(struct SpCommArea), /*   12 length of SP Comm Area */
200                 0,0,0,                  /* 13 − 15 */
201                 sizeof(struct IoHriProcessorVpd),/* 16 length of Proc Vpd */
202                 0,0,0,0,0,0,            /* 17 − 22  */
203                 sizeof(struct ItLpQueue),/*     23 length of Lp Queue */
204                 0,0                     /* 24 − 25 */
205                 },
206         {                       /* VPD addresses */
207                 0,0,0,                  /*       0 −  2 */
208                 &xItExtVpdPanel,        /*       3 Extended VPD */
209                 &paca[0],               /*       4 first Paca */
210                 0,                      /*       5 */
211                 &xItIplParmsReal,       /*       6 IPL parms */
212                 &xMsVpd,                /*       7 MS Vpd */
213                 0,                      /*       8 */
214                 &itLpNaca,              /*       9 LpNaca */
215                 0,                      /*      10 */
216                 &xRecoveryLogBuffer,    /*      11 Recovery Log Buffer */
217                 &xSpCommArea,           /*      12 SP Comm Area */
218                 0,0,0,                  /* 13 − 15 */
219                 &xIoHriProcessorVpd,    /*      16 Proc Vpd */
220                 0,0,0,0,0,0,            /* 17 − 22 */
221                 &xItLpQueue,            /*      23 Lp Queue */
222                 0,0
223         }
224 };
225
226 struct msChunks msChunks = {0, 0, 0, 0, NULL};
227
228 /* Depending on whether this is called from iSeries or pSeries setup
229  * code, the location of the msChunks struct may or may not have
230  * to be reloc'd, so we force the caller to do that for us by passing
231  * in a pointer to the structure.
232  */
233 unsigned long
234 msChunks_alloc(unsigned long mem, unsigned long num_chunks, unsigned long chunk_size)
235 {
236         unsigned long offset = reloc_offset();
237         struct msChunks *_msChunks = PTRRELOC(&msChunks);
238
239         _msChunks->num_chunks  = num_chunks;
240         _msChunks->chunk_size  = chunk_size;
241         _msChunks->chunk_shift = __ilog2(chunk_size);
242         _msChunks->chunk_mask  = (1UL<<_msChunks->chunk_shift)−1;
243
244         mem = _ALIGN(mem, sizeof(msChunks_entry));
245         _msChunks->abs = (msChunks_entry *)(mem + offset);
246         mem += num_chunks * sizeof(msChunks_entry);
247
248         return mem;
249 }
```

```
/*
  * mf.c
  * Copyright (C) 2001 Troy D. Armstrong  IBM Corporation
  *
  * This modules exists as an interface between a Linux secondary partition
  * running on an iSeries and the primary partition's Virtual Service
  * Processor (VSP) object.  The VSP has final authority over powering on/off
  * all partitions in the iSeries.  It also provides miscellaneous low−level
  * machine facility type operations.
  *
  *
  * 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 program is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  * GNU General Public License for more details.
  *
  * 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., 59 Temple Place, Suite 330, Boston, MA  02111−1307 USA
  */

#include <asm/iSeries/mf.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/mm.h>
#include <asm/iSeries/HvLpConfig.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <asm/nvram.h>
#include <asm/time.h>
#include <asm/iSeries/ItSpCommArea.h>
#include <asm/iSeries/mf_proc.h>
#include <asm/iSeries/iSeries_proc.h>
#include <asm/uaccess.h>
#include <linux/pci.h>

extern struct pci_dev * iSeries_vio_dev;

/*
 * This is the structure layout for the Machine Facilites LPAR event
 * flows.
 */
struct VspCmdData;
struct CeMsgData;
union SafeCast
{
        u64 ptrAsU64;
        void *ptr;
};


typedef void (*CeMsgCompleteHandler)( void *token, struct CeMsgData *vspCmdRsp );

struct CeMsgCompleteData
{
        CeMsgCompleteHandler xHdlr;
        void *xToken;
};

struct VspRspData
{
        struct semaphore *xSemaphore;
        struct VspCmdData *xResponse;
};

struct IoMFLpEvent
{
        struct HvLpEvent xHvLpEvent;

        u16 xSubtypeRc;
        u16 xRsvd1;
        u32 xRsvd2;

        union
        {

                struct AllocData
                {
                        u16 xSize;
                        u16 xType;
                        u32 xCount;
                        u16 xRsvd3;
                        u8 xRsvd4;
```

```
91                      HvLpIndex xTargetLp;
92              } xAllocData;
93
94              struct CeMsgData
95              {
96                      u8 xCEMsg[12];
97                      char xReserved[4];
98                      struct CeMsgCompleteData *xToken;
99              } xCEMsgData;
100
101             struct VspCmdData
102             {
103                     union SafeCast xTokenUnion;
104                     u16 xCmd;
105                     HvLpIndex xLpIndex;
106                     u8 xRc;
107                     u32 xReserved1;
108
109                     union VspCmdSubData
110                     {
111                             struct
112                             {
113                                     u64 xState;
114                             } xGetStateOut;
115
116                             struct
117                             {
118                                     u64 xIplType;
119                             } xGetIplTypeOut, xFunction02SelectIplTypeIn;
120
121                             struct
122                             {
123                                     u64 xIplMode;
124                             } xGetIplModeOut, xFunction02SelectIplModeIn;
125
126                             struct
127                             {
128                                     u64 xPage[4];
129                             } xGetSrcHistoryIn;
130
131                             struct
132                             {
133                                     u64 xFlag;
134                             } xGetAutoIplWhenPrimaryIplsOut,
135                                     xSetAutoIplWhenPrimaryIplsIn,
136                                     xWhiteButtonPowerOffIn,
137                                     xFunction08FastPowerOffIn,
138                                     xIsSpcnRackPowerIncompleteOut;
139
140                             struct
141                             {
142                                     u64 xToken;
143                                     u64 xAddressType;
144                                     u64 xSide;
145                                     u32 xTransferLength;
146                                     u32 xOffset;
147                             } xSetKernelImageIn,
148                                     xGetKernelImageIn,
149                                     xSetKernelCmdLineIn,
150                                     xGetKernelCmdLineIn;
151
152                             struct
153                             {
154                                     u32 xTransferLength;
155                             } xGetKernelImageOut,xGetKernelCmdLineOut;
156
157
158                             u8 xReserved2[80];
159
160                     } xSubData;
161             } xVspCmd;
162     } xUnion;
163 };
164
165
166 /*
167  * All outgoing event traffic is kept on a FIFO queue.  The first
168  * pointer points to the one that is outstanding, and all new
169  * requests get stuck on the end.  Also, we keep a certain number of
170  * preallocated stack elements so that we can operate very early in
171  * the boot up sequence (before kmalloc is ready).
172  */
173 struct StackElement
174 {
175         struct StackElement * next;
176         struct IoMFLpEvent event;
177         MFCompleteHandler hdlr;
178         char dmaData[72];
179         unsigned dmaDataLength;
180         unsigned remoteAddress;
```

```
181 };
182 static spinlock_t spinlock;
183 static struct StackElement * head = NULL;
184 static struct StackElement * tail = NULL;
185 static struct StackElement * avail = NULL;
186 static struct StackElement prealloc[16];
187
188 /*
189  * Put a stack element onto the available queue, so it can get reused.
190  * Attention! You must have the spinlock before calling!
191  */
192 void free( struct StackElement * element )
193 {
194         if ( element != NULL )
195         {
196                 element->next = avail;
197                 avail = element;
198         }
199 }
200
201 /*
202  * Enqueue the outbound event onto the stack.  If the queue was
203  * empty to begin with, we must also issue it via the Hypervisor
204  * interface.  There is a section of code below that will touch
205  * the first stack pointer without the protection of the spinlock.
206  * This is OK, because we know that nobody else will be modifying
207  * the first pointer when we do this.
208  */
209 static int signalEvent( struct StackElement * newElement )
210 {
211         int rc = 0;
212         unsigned long flags;
213         int go = 1;
214         struct StackElement * element;
215         HvLpEvent_Rc hvRc;
216
217         /* enqueue the event */
218         if ( newElement != NULL )
219         {
220                 spin_lock_irqsave( &spinlock, flags );
221                 if ( head == NULL )
222                         head = newElement;
223                 else {
224                         go = 0;
225                         tail->next = newElement;
226                 }
227                 newElement->next = NULL;
228                 tail = newElement;
229                 spin_unlock_irqrestore( &spinlock, flags );
230         }
231
232         /* send the event */
233         while ( go )
234         {
235                 go = 0;
236
237                 /* any DMA data to send beforehand? */
238                 if ( head->dmaDataLength > 0 )
239                         HvCallEvent_dmaToSp( head->dmaData, head->remoteAddress, head->dmaDataLength, HvLpDma_Dir
    ection_LocalToRemote );
240
241                 hvRc = HvCallEvent_signalLpEvent(&head->event.xHvLpEvent);
242                 if ( hvRc != HvLpEvent_Rc_Good )
243                 {
244                         printk( KERN_ERR "mf.c: HvCallEvent_signalLpEvent() failed with %d\n", (int)hvRc );
245
246                         spin_lock_irqsave( &spinlock, flags );
247                         element = head;
248                         head = head->next;
249                         if ( head != NULL )
250                                 go = 1;
251                         spin_unlock_irqrestore( &spinlock, flags );
252
253                         if ( element == newElement )
254                                 rc = -EIO;
255                         else {
256                                 if ( element->hdlr != NULL )
257                                 {
258                                         union SafeCast mySafeCast;
259                                         mySafeCast.ptrAsU64 = element->event.xHvLpEvent.xCorrelationToken;
260                                         (*element->hdlr)( mySafeCast.ptr, -EIO );
261                                 }
262                         }
263
264                         spin_lock_irqsave( &spinlock, flags );
265                         free( element );
266                         spin_unlock_irqrestore( &spinlock, flags );
267                 }
268         }
269
```

```
270         return rc;
271 }
272
273 /*
274  * Allocate a new StackElement structure, and initialize it.
275  */
276 static struct StackElement * newStackElement( void )
277 {
278         struct StackElement * newElement = NULL;
279         HvLpIndex primaryLp = HvLpConfig_getPrimaryLpIndex();
280         unsigned long flags;
281
282         if ( newElement == NULL )
283         {
284                 spin_lock_irqsave( &spinlock, flags );
285                 if ( avail != NULL )
286                 {
287                         newElement = avail;
288                         avail = avail->next;
289                 }
290                 spin_unlock_irqrestore( &spinlock, flags );
291         }
292
293         if ( newElement == NULL )
294                 newElement = kmalloc(sizeof(struct StackElement),GFP_ATOMIC);
295
296         if ( newElement == NULL )
297         {
298                 printk( KERN_ERR "mf.c: unable to kmalloc %ld bytes\n", sizeof(struct StackElement) );
299                 return NULL;
300         }
301
302         memset( newElement, 0, sizeof(struct StackElement) );
303         newElement->event.xHvLpEvent.xFlags.xValid = 1;
304         newElement->event.xHvLpEvent.xFlags.xAckType = HvLpEvent_AckType_ImmediateAck;
305         newElement->event.xHvLpEvent.xFlags.xAckInd = HvLpEvent_AckInd_DoAck;
306         newElement->event.xHvLpEvent.xFlags.xFunction = HvLpEvent_Function_Int;
307         newElement->event.xHvLpEvent.xType = HvLpEvent_Type_MachineFac;
308         newElement->event.xHvLpEvent.xSourceLp = HvLpConfig_getLpIndex();
309         newElement->event.xHvLpEvent.xTargetLp = primaryLp;
310         newElement->event.xHvLpEvent.xSizeMinus1 = sizeof(newElement->event)-1;
311         newElement->event.xHvLpEvent.xRc = HvLpEvent_Rc_Good;
312         newElement->event.xHvLpEvent.xSourceInstanceId = HvCallEvent_getSourceLpInstanceId(primaryLp,HvLpEvent_Ty
    pe_MachineFac);
313         newElement->event.xHvLpEvent.xTargetInstanceId = HvCallEvent_getTargetLpInstanceId(primaryLp,HvLpEvent_Ty
    pe_MachineFac);
314
315         return newElement;
316 }
317
318 static int signalVspInstruction( struct VspCmdData *vspCmd )
319 {
320         struct StackElement * newElement = newStackElement();
321         int rc = 0;
322         struct VspRspData response;
323         DECLARE_MUTEX_LOCKED(Semaphore);
324         response.xSemaphore = &Semaphore;
325         response.xResponse = vspCmd;
326
327         if ( newElement == NULL )
328                 rc = -ENOMEM;
329         else {
330                 newElement->event.xHvLpEvent.xSubtype = 6;
331                 newElement->event.xHvLpEvent.x.xSubtypeData = ('M'<<24)+('F'<<16)+('V'<<8)+('I'<<0);
332                 newElement->event.xUnion.xVspCmd.xTokenUnion.ptr = &response;
333                 newElement->event.xUnion.xVspCmd.xCmd = vspCmd->xCmd;
334                 newElement->event.xUnion.xVspCmd.xLpIndex = HvLpConfig_getLpIndex();
335                 newElement->event.xUnion.xVspCmd.xRc = 0xFF;
336                 newElement->event.xUnion.xVspCmd.xReserved1 = 0;
337                 memcpy(&(newElement->event.xUnion.xVspCmd.xSubData),&(vspCmd->xSubData), sizeof(vspCmd->xSubData)
    );
338                 mb();
339
340                 rc = signalEvent(newElement);
341         }
342
343         if (rc == 0)
344         {
345                 down(&Semaphore);
346         }
347
348         return rc;
349 }
350
351
352 /*
353  * Send a 12-byte CE message to the primary partition VSP object
354  */
355 static int signalCEMsg( char * ceMsg, void * token )
356 {
```

```
357         struct StackElement * newElement = newStackElement();
358         int rc = 0;
359
360         if ( newElement == NULL )
361                 rc = −ENOMEM;
362         else {
363                 newElement−>event.xHvLpEvent.xSubtype = 0;
364                 newElement−>event.xHvLpEvent.x.xSubtypeData = ('M'<<24)+('F'<<16)+('C'<<8)+('E'<<0);
365                 memcpy( newElement−>event.xUnion.xCEMsgData.xCEMsg, ceMsg, 12 );
366                 newElement−>event.xUnion.xCEMsgData.xToken = token;
367                 rc = signalEvent(newElement);
368         }
369
370         return rc;
371 }
372
373 /*
374  * Send a 12−byte CE message and DMA data to the primary partition VSP object
375  */
376 static int dmaAndSignalCEMsg( char * ceMsg, void * token, void * dmaData, unsigned dmaDataLength, unsigned remote
    Address )
377 {
378         struct StackElement * newElement = newStackElement();
379         int rc = 0;
380
381         if ( newElement == NULL )
382                 rc = −ENOMEM;
383         else {
384                 newElement−>event.xHvLpEvent.xSubtype = 0;
385                 newElement−>event.xHvLpEvent.x.xSubtypeData = ('M'<<24)+('F'<<16)+('C'<<8)+('E'<<0);
386                 memcpy( newElement−>event.xUnion.xCEMsgData.xCEMsg, ceMsg, 12 );
387                 newElement−>event.xUnion.xCEMsgData.xToken = token;
388                 memcpy( newElement−>dmaData, dmaData, dmaDataLength );
389                 newElement−>dmaDataLength = dmaDataLength;
390                 newElement−>remoteAddress = remoteAddress;
391                 rc = signalEvent(newElement);
392         }
393
394         return rc;
395 }
396
397 /*
398  * Initiate a nice (hopefully) shutdown of Linux.  We simply are
399  * going to try and send the init process a SIGINT signal.  If
400  * this fails (why?), we'll simply force it off in a not−so−nice
401  * manner.
402  */
403 static int shutdown( void )
404 {
405         int rc = kill_proc(1,SIGINT,1);
406
407         if ( rc )
408         {
409                 printk( KERN_ALERT "mf.c: SIGINT to init failed (%d), hard shutdown commencing\n", rc );
410                 mf_powerOff();
411         }
412         else
413                 printk( KERN_INFO "mf.c: init has been successfully notified to proceed with shutdown\n" );
414
415         return rc;
416 }
417
418 /*
419  * The primary partition VSP object is sending us a new
420  * event flow.  Handle it...
421  */
422 static void intReceived( struct IoMFLpEvent * event )
423 {
424         int freeIt = 0;
425         struct StackElement * two = NULL;
426         /* ack the interrupt */
427         event−>xHvLpEvent.xRc = HvLpEvent_Rc_Good;
428         HvCallEvent_ackLpEvent( &event−>xHvLpEvent );
429
430     /* process interrupt */
431         switch( event−>xHvLpEvent.xSubtype )
432         {
433         case 0: /* CE message */
434                 switch( event−>xUnion.xCEMsgData.xCEMsg[3] )
435                 {
436                 case 0x5B:      /* power control notification */
437                         if ( (event−>xUnion.xCEMsgData.xCEMsg[5]&0x20) != 0 )
438                         {
439                                 printk( KERN_INFO "mf.c: Commencing partition shutdown\n" );
440                                 if ( shutdown() == 0 )
441                                         signalCEMsg( "\x00\x00\x00\xDB\x00\x00\x00\x00\x00\x00\x00\x00", NULL );
442                         }
443                         break;
444                 case 0xC0:      /* get time */
445                         {
```

```
446                                     if ( (head != NULL) && ( head->event.xUnion.xCEMsgData.xCEMsg[3] == 0x40 ) )
447                                     {
448                                             freeIt = 1;
449                                             if ( head->event.xUnion.xCEMsgData.xToken != 0 )
450                                             {
451                                                     CeMsgCompleteHandler xHdlr = head->event.xUnion.xCEMsgData.xToken
    ->xHdlr;
452                                                     void * token = head->event.xUnion.xCEMsgData.xToken->xToken;
453
454                                                     if (xHdlr != NULL)
455                                                             (*xHdlr)( token, &(event->xUnion.xCEMsgData) );
456                                             }
457                                     }
458                             }
459                             break;
460                     }
461
462                     /* remove from queue */
463                     if ( freeIt == 1 )
464                     {
465                             unsigned long flags;
466                             spin_lock_irqsave( &spinlock, flags );
467                             if ( head != NULL )
468                             {
469                                     struct StackElement *oldHead = head;
470                                     head = head->next;
471                                     two = head;
472                                     free( oldHead );
473                             }
474                             spin_unlock_irqrestore( &spinlock, flags );
475                     }
476
477                     /* send next waiting event */
478                     if ( two != NULL )
479                             signalEvent( NULL );
480                     break;
481             case 1: /* IT sys shutdown */
482                     printk( KERN_INFO "mf.c: Commencing system shutdown\n" );
483                     shutdown();
484                     break;
485             }
486     }
487
488     /*
489      * The primary partition VSP object is acknowledging the receipt
490      * of a flow we sent to them.  If there are other flows queued
491      * up, we must send another one now...
492      */
493     static void ackReceived( struct IoMFLpEvent * event )
494     {
495             unsigned long flags;
496             struct StackElement * two = NULL;
497             unsigned long freeIt = 0;
498
499         /* handle current event */
500             if ( head != NULL )
501             {
502                     switch( event->xHvLpEvent.xSubtype )
503                     {
504                     case 0:     /* CE msg */
505                             if ( event->xUnion.xCEMsgData.xCEMsg[3] == 0x40 )
506                             {
507                                     if ( event->xUnion.xCEMsgData.xCEMsg[2] != 0 )
508                                     {
509                                             freeIt = 1;
510                                             if ( head->event.xUnion.xCEMsgData.xToken != 0 )
511                                             {
512                                                     CeMsgCompleteHandler xHdlr = head->event.xUnion.xCEMsgData.xToken
    ->xHdlr;
513                                                     void * token = head->event.xUnion.xCEMsgData.xToken->xToken;
514
515                                                     if (xHdlr != NULL)
516                                                             (*xHdlr)( token, &(event->xUnion.xCEMsgData) );
517                                             }
518                                     }
519                             } else {
520                                     freeIt = 1;
521                             }
522                             break;
523                     case 4: /* allocate */
524                     case 5: /* deallocate */
525                             if ( head->hdlr != NULL )
526                             {
527                                     union SafeCast mySafeCast;
528                                     mySafeCast.ptrAsU64 = event->xHvLpEvent.xCorrelationToken;
529                                     (*head->hdlr)( mySafeCast.ptr, event->xUnion.xAllocData.xCount );
530                             }
531                             freeIt = 1;
532                             break;
533                     case 6:
```

```
534                             {
535                                     struct VspRspData *rsp = (struct VspRspData *)event->xUnion.xVspCmd.xTokenUnion.p
    tr;
536
537                                     if (rsp != NULL)
538                                     {
539                                             if (rsp->xResponse != NULL)
540                                                     memcpy(rsp->xResponse, &(event->xUnion.xVspCmd), sizeof(event->xU
    nion.xVspCmd));
541                                             if (rsp->xSemaphore != NULL)
542                                                     up(rsp->xSemaphore);
543                                     } else {
544                                             printk( KERN_ERR "mf.c: no rsp\n");
545                                     }
546                                     freeIt = 1;
547                             }
548                             break;
549                     }
550             }
551             else
552                     printk( KERN_ERR "mf.c: stack empty for receiving ack\n" );
553
554         /* remove from queue */
555             spin_lock_irqsave( &spinlock, flags );
556             if (( head != NULL ) && ( freeIt == 1 ))
557             {
558                     struct StackElement *oldHead = head;
559                     head = head->next;
560                     two = head;
561                     free( oldHead );
562             }
563             spin_unlock_irqrestore( &spinlock, flags );
564
565         /* send next waiting event */
566             if ( two != NULL )
567                     signalEvent( NULL );
568     }
569
570     /*
571      * This is the generic event handler we are registering with
572      * the Hypervisor.  Ensure the flows are for us, and then
573      * parse it enough to know if it is an interrupt or an
574      * acknowledge.
575      */
576     static void hvHandler( struct HvLpEvent * event, struct pt_regs * regs )
577     {
578             if ( (event != NULL) && (event->xType == HvLpEvent_Type_MachineFac) )
579             {
580                     switch( event->xFlags.xFunction )
581                     {
582                     case HvLpEvent_Function_Ack:
583                             ackReceived( (struct IoMFLpEvent *)event );
584                             break;
585                     case HvLpEvent_Function_Int:
586                             intReceived( (struct IoMFLpEvent *)event );
587                             break;
588                     default:
589                             printk( KERN_ERR "mf.c: non ack/int event received\n" );
590                             break;
591                     }
592             }
593             else
594                     printk( KERN_ERR "mf.c: alien event received\n" );
595     }
596
597     /*
598      * Global kernel interface to allocate and seed events into the
599      * Hypervisor.
600      */
601     void mf_allocateLpEvents( HvLpIndex targetLp,
602                               HvLpEvent_Type type,
603                               unsigned size,
604                               unsigned count,
605                               MFCompleteHandler hdlr,
606                               void * userToken )
607     {
608             struct StackElement * newElement = newStackElement();
609             int rc = 0;
610
611             if ( newElement == NULL )
612                     rc = -ENOMEM;
613             else {
614                     union SafeCast mine;
615                     mine.ptr = userToken;
616                     newElement->event.xHvLpEvent.xSubtype = 4;
617                     newElement->event.xHvLpEvent.xCorrelationToken = mine.ptrAsU64;
618                     newElement->event.xHvLpEvent.x.xSubtypeData = ('M'<<24)+('F'<<16)+('M'<<8)+('A'<<0);
619                     newElement->event.xUnion.xAllocData.xTargetLp = targetLp;
620                     newElement->event.xUnion.xAllocData.xType = type;
621                     newElement->event.xUnion.xAllocData.xSize = size;
```

```
622                     newElement->event.xUnion.xAllocData.xCount = count;
623                     newElement->hdlr = hdlr;
624                     rc = signalEvent(newElement);
625             }
626
627             if ( (rc != 0) && (hdlr != NULL) )
628                     (*hdlr)( userToken, rc );
629     }
630
631     /*
632      * Global kernel interface to unseed and deallocate events already in
633      * Hypervisor.
634      */
635     void mf_deallocateLpEvents( HvLpIndex targetLp,
636                                 HvLpEvent_Type type,
637                                 unsigned count,
638                                 MFCompleteHandler hdlr,
639                                 void * userToken )
640     {
641             struct StackElement * newElement = newStackElement();
642             int rc = 0;
643
644             if ( newElement == NULL )
645                     rc = -ENOMEM;
646             else {
647                     union SafeCast mine;
648                     mine.ptr = userToken;
649                     newElement->event.xHvLpEvent.xSubtype = 5;
650                     newElement->event.xHvLpEvent.xCorrelationToken = mine.ptrAsU64;
651                     newElement->event.xHvLpEvent.x.xSubtypeData = ('M'<<24)+('F'<<16)+('M'<<8)+('D'<<0);
652                     newElement->event.xUnion.xAllocData.xTargetLp = targetLp;
653                     newElement->event.xUnion.xAllocData.xType = type;
654                     newElement->event.xUnion.xAllocData.xCount = count;
655                     newElement->hdlr = hdlr;
656                     rc = signalEvent(newElement);
657             }
658
659             if ( (rc != 0) && (hdlr != NULL) )
660                     (*hdlr)( userToken, rc );
661     }
662
663     /*
664      * Global kernel interface to tell the VSP object in the primary
665      * partition to power this partition off.
666      */
667     void mf_powerOff( void )
668     {
669             printk( KERN_INFO "mf.c: Down it goes...\n" );
670             signalCEMsg( "\x00\x00\x00\x4D\x00\x00\x00\x00\x00\x00\x00\x00", NULL );
671             for (;;);
672     }
673
674     /*
675      * Global kernel interface to tell the VSP object in the primary
676      * partition to reboot this partition.
677      */
678     void mf_reboot( void )
679     {
680             printk( KERN_INFO "mf.c: Preparing to bounce...\n" );
681             signalCEMsg( "\x00\x00\x00\x4E\x00\x00\x00\x00\x00\x00\x00\x00", NULL );
682             for (;;);
683     }
684
685     /*
686      * Display a single word SRC onto the VSP control panel.
687      */
688     void mf_displaySrc( u32 word )
689     {
690             u8 ce[12];
691
692             memcpy( ce, "\x00\x00\x00\x4A\x00\x00\x00\x01\x00\x00\x00\x00", 12 );
693             ce[8] = word>>24;
694             ce[9] = word>>16;
695             ce[10] = word>>8;
696             ce[11] = word;
697             signalCEMsg( ce, NULL );
698     }
699
700     /*
701      * Display a single word SRC of the form "PROGXXXX" on the VSP control panel.
702      */
703     void mf_displayProgress( u16 value )
704     {
705             u8 ce[12];
706             u8 src[72];
707
708             memcpy( ce, "\x00\x00\x04\x4A\x00\x00\x00\x48\x00\x00\x00\x00", 12 );
709             memcpy( src,
710                     "\x01\x00\x00\x01"
711                     "\x00\x00\x00\x00"
```

```
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "\x00\x00\x00\x00"
                "PROGxxxx"
                "          ",
                72 );
        src[6] = value>>8;
        src[7] = value&255;
        src[44] = "0123456789ABCDEF"[(value>>12)&15];
        src[45] = "0123456789ABCDEF"[(value>>8)&15];
        src[46] = "0123456789ABCDEF"[(value>>4)&15];
        src[47] = "0123456789ABCDEF"[value&15];
        dmaAndSignalCEMsg( ce, NULL, src, sizeof(src), 9*64*1024 );
}

/*
 * Clear the VSP control panel.  Used to "erase" an SRC that was
 * previously displayed.
 */
void mf_clearSrc( void )
{
        signalCEMsg( "\x00\x00\x00\x4B\x00\x00\x00\x00\x00\x00\x00\x00", NULL );
}

/*
 * Initialization code here.
 */
void mf_init( void )
{
        int i;

    /* initialize */
        spin_lock_init( &spinlock );
        for ( i = 0; i < sizeof(prealloc)/sizeof(*prealloc); ++i )
                free( &prealloc[i] );
        HvLpEvent_registerHandler( HvLpEvent_Type_MachineFac, &hvHandler );

        /* virtual continue ack */
        signalCEMsg( "\x00\x00\x00\x57\x00\x00\x00\x00\x00\x00\x00\x00", NULL );

        /* initialization complete */
        printk( KERN_NOTICE "mf.c: iSeries Linux LPAR Machine Facilities initialized\n" );

        iSeries_proc_callback(&mf_proc_init);
}

void mf_setSide(char side)
{
        int rc = 0;
        u64 newSide = 0;
        struct VspCmdData myVspCmd;

        memset(&myVspCmd, 0, sizeof(myVspCmd));
        if (side == 'A')
                newSide = 0;
        else if (side == 'B')
                newSide = 1;
        else if (side == 'C')
                newSide = 2;
        else
                newSide = 3;

        myVspCmd.xSubData.xFunction02SelectIplTypeIn.xIplType = newSide;
        myVspCmd.xCmd = 10;

        rc = signalVspInstruction(&myVspCmd);
}

char mf_getSide(void)
{
        char returnValue = ' ';
        int rc = 0;
        struct VspCmdData myVspCmd;

        memset(&myVspCmd, 0, sizeof(myVspCmd));
        myVspCmd.xCmd = 2;
        myVspCmd.xSubData.xFunction02SelectIplTypeIn.xIplType = 0;
        mb();
        rc = signalVspInstruction(&myVspCmd);

        if (rc != 0)
        {
                return returnValue;
        } else {
                if (myVspCmd.xRc == 0)
```

```
802                 {
803                         if (myVspCmd.xSubData.xGetIplTypeOut.xIplType == 0)
804                                 returnValue = 'A';
805                         else if (myVspCmd.xSubData.xGetIplTypeOut.xIplType == 1)
806                                 returnValue = 'B';
807                         else if (myVspCmd.xSubData.xGetIplTypeOut.xIplType == 2)
808                                 returnValue = 'C';
809                         else
810                                 returnValue = 'D';
811                 }
812         }
813
814         return returnValue;
815 }
816
817 void mf_getSrcHistory(char *buffer, int size)
818 {
819     /*    struct IplTypeReturnStuff returnStuff;
820      struct StackElement * newElement = newStackElement();
821      int rc = 0;
822      char *pages[4];
823
824      pages[0] = kmalloc(4096, GFP_ATOMIC);
825      pages[1] = kmalloc(4096, GFP_ATOMIC);
826      pages[2] = kmalloc(4096, GFP_ATOMIC);
827      pages[3] = kmalloc(4096, GFP_ATOMIC);
828      if (( newElement == NULL ) || (pages[0] == NULL) || (pages[1] == NULL) || (pages[2] == NULL) || (pages[3] ==
     NULL))
829      rc = -ENOMEM;
830      else
831      {
832      returnStuff.xType = 0;
833      returnStuff.xRc = 0;
834      returnStuff.xDone = 0;
835      newElement->event.xHvLpEvent.xSubtype = 6;
836      newElement->event.xHvLpEvent.x.xSubtypeData = ('M'<<24)+('F'<<16)+('V'<<8)+('I'<<0);
837      newElement->event.xUnion.xVspCmd.xEvent = &returnStuff;
838      newElement->event.xUnion.xVspCmd.xCmd = 4;
839      newElement->event.xUnion.xVspCmd.xLpIndex = HvLpConfig_getLpIndex();
840      newElement->event.xUnion.xVspCmd.xRc = 0xFF;
841      newElement->event.xUnion.xVspCmd.xReserved1 = 0;
842      newElement->event.xUnion.xVspCmd.xSubData.xGetSrcHistoryIn.xPage[0] = (0x8000000000000000ULL | virt_to_absol
    ute((unsigned long)pages[0]));
843      newElement->event.xUnion.xVspCmd.xSubData.xGetSrcHistoryIn.xPage[1] = (0x8000000000000000ULL | virt_to_absol
    ute((unsigned long)pages[1]));
844      newElement->event.xUnion.xVspCmd.xSubData.xGetSrcHistoryIn.xPage[2] = (0x8000000000000000ULL | virt_to_absol
    ute((unsigned long)pages[2]));
845      newElement->event.xUnion.xVspCmd.xSubData.xGetSrcHistoryIn.xPage[3] = (0x8000000000000000ULL | virt_to_absol
    ute((unsigned long)pages[3]));
846      mb();
847      rc = signalEvent(newElement);
848      }
849
850      if (rc != 0)
851      {
852      return;
853      }
854      else
855      {
856      while (returnStuff.xDone != 1)
857      {
858      udelay(10);
859      }
860
861      if (returnStuff.xRc == 0)
862      {
863      memcpy(buffer, pages[0], size);
864      }
865      }
866
867      kfree(pages[0]);
868      kfree(pages[1]);
869      kfree(pages[2]);
870      kfree(pages[3]);*/
871 }
872
873 void mf_setCmdLine(const char *cmdline, int size, u64 side)
874 {
875         struct VspCmdData myVspCmd;
876         int rc = 0;
877         dma_addr_t dma_addr = 0;
878         char *page = pci_alloc_consistent(iSeries_vio_dev, size, &dma_addr);
879
880         if (page == NULL) {
881                 printk(KERN_ERR "mf.c: couldn't allocate memory to set command line\n");
882                 return;
883         }
884
885         copy_from_user(page, cmdline, size);
886
```

```
887         memset(&myVspCmd, 0, sizeof(myVspCmd));
888         myVspCmd.xCmd = 31;
889         myVspCmd.xSubData.xSetKernelCmdLineIn.xToken = dma_addr;
890         myVspCmd.xSubData.xSetKernelCmdLineIn.xAddressType = HvLpDma_AddressType_TceIndex;
891         myVspCmd.xSubData.xSetKernelCmdLineIn.xSide = side;
892         myVspCmd.xSubData.xSetKernelCmdLineIn.xTransferLength = size;
893         mb();
894         rc = signalVspInstruction(&myVspCmd);
895
896         pci_free_consistent(iSeries_vio_dev, size, page, dma_addr);
897 }
898
899 int mf_getCmdLine(char *cmdline, int *size, u64 side)
900 {
901         struct VspCmdData myVspCmd;
902         int rc = 0;
903         int len = *size;
904         dma_addr_t dma_addr = pci_map_single(iSeries_vio_dev, cmdline, *size, PCI_DMA_FROMDEVICE);
905
906         memset(cmdline, 0, *size);
907         memset(&myVspCmd, 0, sizeof(myVspCmd));
908         myVspCmd.xCmd = 33;
909         myVspCmd.xSubData.xGetKernelCmdLineIn.xToken = dma_addr;
910         myVspCmd.xSubData.xGetKernelCmdLineIn.xAddressType = HvLpDma_AddressType_TceIndex;
911         myVspCmd.xSubData.xGetKernelCmdLineIn.xSide = side;
912         myVspCmd.xSubData.xGetKernelCmdLineIn.xTransferLength = *size;
913         mb();
914         rc = signalVspInstruction(&myVspCmd);
915
916         if ( ! rc ) {
917
918                 if (myVspCmd.xRc == 0)
919                 {
920                         len = myVspCmd.xSubData.xGetKernelCmdLineOut.xTransferLength;
921                 }
922                 /* else
923                         {
924                         memcpy(cmdline, "Bad cmdline", 11);
925                         }
926                 */
927         }
928
929         pci_unmap_single(iSeries_vio_dev, dma_addr, *size, PCI_DMA_FROMDEVICE);
930
931         return len;
932 }
933
934
935 int mf_setVmlinuxChunk(const char *buffer, int size, int offset, u64 side)
936 {
937         struct VspCmdData myVspCmd;
938         int rc = 0;
939
940         dma_addr_t dma_addr = 0;
941
942         char *page = pci_alloc_consistent(iSeries_vio_dev, size, &dma_addr);
943
944         if (page == NULL) {
945                 printk(KERN_ERR "mf.c: couldn't allocate memory to set vmlinux chunk\n");
946                 return -ENOMEM;
947         }
948
949         copy_from_user(page, buffer, size);
950         memset(&myVspCmd, 0, sizeof(myVspCmd));
951
952         myVspCmd.xCmd = 30;
953         myVspCmd.xSubData.xGetKernelImageIn.xToken = dma_addr;
954         myVspCmd.xSubData.xGetKernelImageIn.xAddressType = HvLpDma_AddressType_TceIndex;
955         myVspCmd.xSubData.xGetKernelImageIn.xSide = side;
956         myVspCmd.xSubData.xGetKernelImageIn.xOffset = offset;
957         myVspCmd.xSubData.xGetKernelImageIn.xTransferLength = size;
958         mb();
959         rc = signalVspInstruction(&myVspCmd);
960
961         if (rc == 0)
962         {
963                 if (myVspCmd.xRc == 0)
964                 {
965                         rc = 0;
966                 } else {
967                         rc = -ENOMEM;
968                 }
969         }
970
971         pci_free_consistent(iSeries_vio_dev, size, page, dma_addr);
972
973         return rc;
974 }
975
976 int mf_getVmlinuxChunk(char *buffer, int *size, int offset, u64 side)
```

```
977 {
978         struct VspCmdData myVspCmd;
979         int rc = 0;
980         int len = *size;
981
982         dma_addr_t dma_addr = pci_map_single(iSeries_vio_dev, buffer, *size, PCI_DMA_FROMDEVICE);
983
984         memset(buffer, 0, len);
985
986         memset(&myVspCmd, 0, sizeof(myVspCmd));
987         myVspCmd.xCmd = 32;
988         myVspCmd.xSubData.xGetKernelImageIn.xToken = dma_addr;
989         myVspCmd.xSubData.xGetKernelImageIn.xAddressType = HvLpDma_AddressType_TceIndex;
990         myVspCmd.xSubData.xGetKernelImageIn.xSide = side;
991         myVspCmd.xSubData.xGetKernelImageIn.xOffset = offset;
992         myVspCmd.xSubData.xGetKernelImageIn.xTransferLength = len;
993         mb();
994         rc = signalVspInstruction(&myVspCmd);
995
996         if (rc == 0)
997         {
998                 if (myVspCmd.xRc == 0)
999                 {
1000                        *size = myVspCmd.xSubData.xGetKernelImageOut.xTransferLength;
1001                } else {
1002                        rc = -ENOMEM;
1003                }
1004        }
1005
1006        pci_unmap_single(iSeries_vio_dev, dma_addr, len, PCI_DMA_FROMDEVICE);
1007
1008        return rc;
1009 }
1010
1011 int mf_setRtcTime(unsigned long time)
1012 {
1013        struct rtc_time tm;
1014
1015        to_tm(time, &tm);
1016
1017        return mf_setRtc( &tm );
1018 }
1019
1020 struct RtcTimeData
1021 {
1022        struct semaphore *xSemaphore;
1023        struct CeMsgData xCeMsg;
1024        int xRc;
1025 };
1026
1027 void getRtcTimeComplete(void * token, struct CeMsgData *ceMsg)
1028 {
1029        struct RtcTimeData *rtc = (struct RtcTimeData *)token;
1030
1031        memcpy(&(rtc->xCeMsg), ceMsg, sizeof(rtc->xCeMsg));
1032
1033        rtc->xRc = 0;
1034        up(rtc->xSemaphore);
1035 }
1036
1037 static unsigned long lastsec = 1;
1038
1039 int mf_getRtcTime(unsigned long *time)
1040 {
1041 /*    unsigned long usec, tsec; */
1042
1043        u32 dataWord1 = *((u32 *)(&xSpCommArea.xBcdTimeAtIplStart));
1044        u32 dataWord2 = *(((u32 *)&(xSpCommArea.xBcdTimeAtIplStart)) + 1);
1045        int year = 1970;
1046        int year1 = ( dataWord1 >> 24 ) & 0x000000FF;
1047        int year2 = ( dataWord1 >> 16 ) & 0x000000FF;
1048        int sec = ( dataWord1 >> 8 ) & 0x000000FF;
1049        int min = dataWord1 & 0x000000FF;
1050        int hour = ( dataWord2 >> 24 ) & 0x000000FF;
1051        int day = ( dataWord2 >> 8 ) & 0x000000FF;
1052        int mon = dataWord2 & 0x000000FF;
1053
1054        BCD_TO_BIN(sec);
1055        BCD_TO_BIN(min);
1056        BCD_TO_BIN(hour);
1057        BCD_TO_BIN(day);
1058        BCD_TO_BIN(mon);
1059        BCD_TO_BIN(year1);
1060        BCD_TO_BIN(year2);
1061        year = year1 * 100 + year2;
1062
1063        *time = mktime(year, mon, day, hour, min, sec);
1064
1065        *time += ( jiffies / HZ );
1066
```

```
1067          /* Now THIS is a nasty hack!
1068           * It ensures that the first two calls to mf_getRtcTime get different
1069           * answers.  That way the loop in init_time (time.c) will not think
1070           * the clock is stuck.
1071           */
1072          if ( lastsec ) {
1073                  *time −= lastsec;
1074                  −−lastsec;
1075          }
1076
1077          return 0;
1078
1079  }
1080
1081  int mf_getRtc( struct rtc_time * tm )
1082  {
1083
1084          struct CeMsgCompleteData ceComplete;
1085          struct RtcTimeData rtcData;
1086          int rc = 0;
1087          DECLARE_MUTEX_LOCKED(Semaphore);
1088
1089          memset(&ceComplete, 0, sizeof(ceComplete));
1090          memset(&rtcData, 0, sizeof(rtcData));
1091
1092          rtcData.xSemaphore = &Semaphore;
1093
1094          ceComplete.xHdlr = &getRtcTimeComplete;
1095          ceComplete.xToken = (void *)&rtcData;
1096
1097          rc = signalCEMsg( "\x00\x00\x00\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00", &ceComplete );
1098
1099          if ( rc == 0 )
1100          {
1101                  down(&Semaphore);
1102
1103                  if ( rtcData.xRc == 0)
1104                  {
1105                          if ( ( rtcData.xCeMsg.xCEMsg[2] == 0xa9 ) ||
1106                               ( rtcData.xCeMsg.xCEMsg[2] == 0xaf ) ) {
1107                                  /* TOD clock is not set */
1108                                  tm->tm_sec = 1;
1109                                  tm->tm_min = 1;
1110                                  tm->tm_hour = 1;
1111                                  tm->tm_mday = 10;
1112                                  tm->tm_mon = 8;
1113                                  tm->tm_year = 71;
1114                                  mf_setRtc( tm );
1115                          }
1116                          {
1117                                  u32 dataWord1 = *((u32 *)(rtcData.xCeMsg.xCEMsg+4));
1118                                  u32 dataWord2 = *((u32 *)(rtcData.xCeMsg.xCEMsg+8));
1119                                  u8 year = (dataWord1 >> 16 ) & 0x000000FF;
1120                                  u8 sec = ( dataWord1 >> 8 ) & 0x000000FF;
1121                                  u8 min = dataWord1 & 0x000000FF;
1122                                  u8 hour = ( dataWord2 >> 24 ) & 0x000000FF;
1123                                  u8 day = ( dataWord2 >> 8 ) & 0x000000FF;
1124                                  u8 mon = dataWord2 & 0x000000FF;
1125
1126                                  BCD_TO_BIN(sec);
1127                                  BCD_TO_BIN(min);
1128                                  BCD_TO_BIN(hour);
1129                                  BCD_TO_BIN(day);
1130                                  BCD_TO_BIN(mon);
1131                                  BCD_TO_BIN(year);
1132
1133                                  if ( year <= 69 )
1134                                          year += 100;
1135
1136                                  tm->tm_sec = sec;
1137                                  tm->tm_min = min;
1138                                  tm->tm_hour = hour;
1139                                  tm->tm_mday = day;
1140                                  tm->tm_mon = mon;
1141                                  tm->tm_year = year;
1142                          }
1143                  } else {
1144                          rc = rtcData.xRc;
1145                          tm->tm_sec = 0;
1146                          tm->tm_min = 0;
1147                          tm->tm_hour = 0;
1148                          tm->tm_mday = 15;
1149                          tm->tm_mon = 5;
1150                          tm->tm_year = 52;
1151
1152                  }
1153                  tm->tm_wday = 0;
1154                  tm->tm_yday = 0;
1155                  tm->tm_isdst = 0;
1156
```

```
1157          }
1158
1159          return rc;
1160
1161  }
1162
1163  int mf_setRtc(struct rtc_time * tm)
1164  {
1165          char ceTime[12] = "\x00\x00\x00\x41\x00\x00\x00\x00\x00\x00\x00\x00";
1166          int rc = 0;
1167          u8 day, mon, hour, min, sec, y1, y2;
1168          unsigned year;
1169
1170          year = 1900 + tm->tm_year;
1171          y1 = year / 100;
1172          y2 = year % 100;
1173
1174          sec = tm->tm_sec;
1175          min = tm->tm_min;
1176          hour = tm->tm_hour;
1177          day = tm->tm_mday;
1178          mon = tm->tm_mon + 1;
1179
1180          BIN_TO_BCD(sec);
1181          BIN_TO_BCD(min);
1182          BIN_TO_BCD(hour);
1183          BIN_TO_BCD(mon);
1184          BIN_TO_BCD(day);
1185          BIN_TO_BCD(y1);
1186          BIN_TO_BCD(y2);
1187
1188          ceTime[4] = y1;
1189          ceTime[5] = y2;
1190          ceTime[6] = sec;
1191          ceTime[7] = min;
1192          ceTime[8] = hour;
1193          ceTime[10] = day;
1194          ceTime[11] = mon;
1195
1196          rc = signalCEMsg( ceTime, NULL );
1197
1198          return rc;
1199  }
1200
1201
1202
```

```
/*
 *  c 2001 PPC 64 Team, IBM Corp
 *
 *      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.
 *
 * /dev/nvram driver for PPC64
 *
 * This perhaps should live in drivers/char
 */

#include <linux/module.h>

#include <linux/types.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/fcntl.h>
#include <linux/nvram.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <asm/nvram.h>
#include <asm/rtas.h>
#include <asm/prom.h>

static unsigned int rtas_nvram_size;
static unsigned int nvram_fetch, nvram_store;
static char nvram_buf[4];       /* assume this is in the first 4GB */

static loff_t nvram_llseek(struct file *file, loff_t offset, int origin)
{
        switch (origin) {
        case 1:
                offset += file->f_pos;
                break;
        case 2:
                offset += rtas_nvram_size;
                break;
        }
        if (offset < 0)
                return -EINVAL;
        file->f_pos = offset;
        return file->f_pos;
}


static ssize_t read_nvram(struct file *file, char *buf,
                          size_t count, loff_t *ppos)
{
        unsigned int i;
        unsigned long len;
        char *p = buf;

        if (verify_area(VERIFY_WRITE, buf, count))
                return -EFAULT;
        if (*ppos >= rtas_nvram_size)
                return 0;
        for (i = *ppos; count > 0 && i < rtas_nvram_size; ++i, ++p, --count) {
                if ((rtas_call(nvram_fetch, 3, 2, &len, i, __pa(nvram_buf), 1) != 0) ||
                    len != 1)
                        return -EIO;
                if (__put_user(nvram_buf[0], p))
                        return -EFAULT;
        }
        *ppos = i;
        return p - buf;
}

static ssize_t write_nvram(struct file *file, const char *buf,
                           size_t count, loff_t *ppos)
{
        unsigned int i;
        unsigned long len;
        const char *p = buf;
        char c;

        if (verify_area(VERIFY_READ, buf, count))
                return -EFAULT;
        if (*ppos >= rtas_nvram_size)
                return 0;
        for (i = *ppos; count > 0 && i < rtas_nvram_size; ++i, ++p, --count) {
                if (__get_user(c, p))
                        return -EFAULT;
                nvram_buf[0] = c;
                if ((rtas_call(nvram_store, 3, 2, &len, i, __pa(nvram_buf), 1) != 0) ||
                    len != 1)
                        return -EIO;
        }
```

```
91          *ppos = i;
92          return p - buf;
93  }
94
95  static int nvram_ioctl(struct inode *inode, struct file *file,
96          unsigned int cmd, unsigned long arg)
97  {
98          return -EINVAL;
99  }
100
101 struct file_operations nvram_fops = {
102         owner:          THIS_MODULE,
103         llseek:         nvram_llseek,
104         read:           read_nvram,
105         write:          write_nvram,
106         ioctl:          nvram_ioctl,
107 };
108
109 static struct miscdevice nvram_dev = {
110         NVRAM_MINOR,
111         "nvram",
112         &nvram_fops
113 };
114
115 int __init nvram_init(void)
116 {
117         struct device_node *nvram;
118         unsigned int *nbytes_p, proplen;
119         if ((nvram = find_type_devices("nvram")) != NULL) {
120                 nbytes_p = (unsigned int *)get_property(nvram, "#bytes", &proplen);
121                 if (nbytes_p && proplen == sizeof(unsigned int)) {
122                         rtas_nvram_size = *nbytes_p;
123                 }
124         }
125         nvram_fetch = rtas_token("nvram-fetch");
126         nvram_store = rtas_token("nvram-store");
127         printk(KERN_INFO "PPC64 nvram contains %d bytes\n", rtas_nvram_size);
128
129         misc_register(&nvram_dev);
130         return 0;
131 }
132
133 void __exit nvram_cleanup(void)
134 {
135         misc_deregister( &nvram_dev );
136 }
137
138 module_init(nvram_init);
139 module_exit(nvram_cleanup);
140 MODULE_LICENSE("GPL");
```

```
/*
 * c 2001 PPC 64 Team, IBM Corp
 *
 *      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.
 */

#include <asm/types.h>
#include <asm/page.h>
#include <stddef.h>
#include <linux/config.h>
#include <linux/threads.h>
#include <asm/page.h>
#include <asm/mmu.h>
#include <asm/processor.h>
#include <asm/ptrace.h>

#include <asm/iSeries/ItLpPaca.h>
#include <asm/naca.h>
#include <asm/paca.h>

struct naca_struct *naca;
struct systemcfg *systemcfg;

/* The Paca is an array with one entry per processor.  Each contains an
 * ItLpPaca, which contains the information shared between the
 * hypervisor and Linux.  Each also contains an ItLpRegSave area which
 * is used by the hypervisor to save registers.
 * On systems with hardware multi-threading, there are two threads
 * per processor.  The Paca array must contain an entry for each thread.
 * The VPD Areas will give a max logical processors = 2 * max physical
 * processors.  The processor VPD array needs one entry per physical
 * processor (not thread).
 */
#define PACAINITDATA(number,start,lpq,asrr,asrv) \
{                                                                       \
        xLpPacaPtr: &paca[number].xLpPaca,                              \
        xLpRegSavePtr: &paca[number].xRegSav,                           \
        xPacaIndex: (number),           /* Paca Index        */         \
        default_decr: 0x00ff0000,       /* Initial Decr      */         \
        xStab_data: {                                                   \
                real: (asrr),           /* Real pointer to segment table */ \
                virt: (asrv),           /* Virt pointer to segment table */ \
                next_round_robin: 1     /* Round robin index */         \
        },                                                              \
        lpQueuePtr: (lpq),              /* &xItLpQueue,                  */ \
        /* xRtas: {                                                        \
                lock: SPIN_LOCK_UNLOCKED                                \
        }, */                                                           \
        xProcStart: (start),            /* Processor start */           \
        xLpPaca: {                                                      \
                xDesc: 0xd397d781,      /* "LpPa"          */           \
                xSize: sizeof(struct ItLpPaca),                         \
                xFPRegsInUse: 1,                                        \
                xDynProcStatus: 2,                                      \
                xDecrVal: 0x00ff0000,                                   \
                xEndOfQuantum: 0xffffffffffffffff                       \
        },                                                              \
        xRegSav: {                                                      \
                xDesc: 0xd397d9e2,      /* "LpRS"          */           \
                xSize: sizeof(struct ItLpRegSave)                       \
        },                                                              \
        exception_sp:                                                   \
                (&paca[number].exception_stack[0]) - EXC_FRAME_SIZE,    \
}

struct paca_struct paca[MAX_PACAS] __page_aligned = {
#ifdef CONFIG_PPC_ISERIES
        PACAINITDATA( 0, 1, &xItLpQueue, 0, STAB0_VIRT_ADDR),
#else
        PACAINITDATA( 0, 1, 0, STAB0_PHYS_ADDR, STAB0_VIRT_ADDR),
#endif
        PACAINITDATA( 1, 0, 0, 0, 0),
        PACAINITDATA( 2, 0, 0, 0, 0),
        PACAINITDATA( 3, 0, 0, 0, 0),
        PACAINITDATA( 4, 0, 0, 0, 0),
        PACAINITDATA( 5, 0, 0, 0, 0),
        PACAINITDATA( 6, 0, 0, 0, 0),
        PACAINITDATA( 7, 0, 0, 0, 0),
        PACAINITDATA( 8, 0, 0, 0, 0),
        PACAINITDATA( 9, 0, 0, 0, 0),
        PACAINITDATA(10, 0, 0, 0, 0),
        PACAINITDATA(11, 0, 0, 0, 0),
        PACAINITDATA(12, 0, 0, 0, 0),
        PACAINITDATA(13, 0, 0, 0, 0),
        PACAINITDATA(14, 0, 0, 0, 0),
        PACAINITDATA(15, 0, 0, 0, 0),
        PACAINITDATA(16, 0, 0, 0, 0),
```

```
91          PACAINITDATA(17, 0, 0, 0, 0),
92          PACAINITDATA(18, 0, 0, 0, 0),
93          PACAINITDATA(19, 0, 0, 0, 0),
94          PACAINITDATA(20, 0, 0, 0, 0),
95          PACAINITDATA(21, 0, 0, 0, 0),
96          PACAINITDATA(22, 0, 0, 0, 0),
97          PACAINITDATA(23, 0, 0, 0, 0),
98          PACAINITDATA(24, 0, 0, 0, 0),
99          PACAINITDATA(25, 0, 0, 0, 0),
100         PACAINITDATA(26, 0, 0, 0, 0),
101         PACAINITDATA(27, 0, 0, 0, 0),
102         PACAINITDATA(28, 0, 0, 0, 0),
103         PACAINITDATA(29, 0, 0, 0, 0),
104         PACAINITDATA(30, 0, 0, 0, 0),
105         PACAINITDATA(31, 0, 0, 0, 0),
106         PACAINITDATA(32, 0, 0, 0, 0),
107         PACAINITDATA(33, 0, 0, 0, 0),
108         PACAINITDATA(34, 0, 0, 0, 0),
109         PACAINITDATA(35, 0, 0, 0, 0),
110         PACAINITDATA(36, 0, 0, 0, 0),
111         PACAINITDATA(37, 0, 0, 0, 0),
112         PACAINITDATA(38, 0, 0, 0, 0),
113         PACAINITDATA(39, 0, 0, 0, 0),
114         PACAINITDATA(40, 0, 0, 0, 0),
115         PACAINITDATA(41, 0, 0, 0, 0),
116         PACAINITDATA(42, 0, 0, 0, 0),
117         PACAINITDATA(43, 0, 0, 0, 0),
118         PACAINITDATA(44, 0, 0, 0, 0),
119         PACAINITDATA(45, 0, 0, 0, 0),
120         PACAINITDATA(46, 0, 0, 0, 0),
121         PACAINITDATA(47, 0, 0, 0, 0)
122 };
```

```
/*
 * pci_dma.c
 * Copyright (C) 2001 Mike Corrigan & Dave Engebretsen, IBM Corporation
 *
 * Dynamic DMA mapping support.
 *
 * Manages the TCE space assigned to this partition.
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#include <linux/init.h>
#include <linux/types.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/spinlock.h>
#include <linux/string.h>
#include <linux/pci.h>
#include <asm/io.h>
#include <asm/prom.h>
#include <asm/rtas.h>
#include <asm/ppcdebug.h>

#include <asm/iSeries/HvCallXm.h>
#include <asm/iSeries/LparData.h>
#include <asm/pci_dma.h>
#include <asm/pci-bridge.h>
#include <asm/iSeries/iSeries_pci.h>

#include <asm/machdep.h>

#include "pci.h"

/* #define DEBUG_TCE 1   */
/* #define MONITOR_TCE 1 */ /* Turn on to sanity check TCE generation. */


/* Initialize so this guy does not end up in the BSS section.
 * Only used to pass OF initialization data set in prom.c into the main
 * kernel code -- data ultimately copied into tceTables[].
 */
extern struct _of_tce_table of_tce_table[];

extern struct pci_controller* hose_head;
extern struct pci_controller** hose_tail;
extern struct list_head iSeries_Global_Device_List;

struct TceTable   virtBusVethTceTable;  /* Tce table for virtual ethernet */
struct TceTable   virtBusVioTceTable;   /* Tce table for virtual I/O */

struct iSeries_Device_Node iSeries_veth_dev_node = { LogicalSlot: 0xFF, DevTceTable: &virtBusVethTceTable };
struct iSeries_Device_Node iSeries_vio_dev_node  = { LogicalSlot: 0xFF, DevTceTable: &virtBusVioTceTable };

struct pci_dev    iSeries_veth_dev_st = { sysdata: &iSeries_veth_dev_node };
struct pci_dev    iSeries_vio_dev_st  = { sysdata: &iSeries_vio_dev_node  };

struct pci_dev  * iSeries_veth_dev = &iSeries_veth_dev_st;
struct pci_dev  * iSeries_vio_dev  = &iSeries_vio_dev_st;

/* Device TceTable is stored in Device Node */
/* struct TceTable * tceTables[256]; */ /* Tce tables for 256 busses
                                         * Bus 255 is the virtual bus
                                         * zero indicates no bus defined
                                         */
/* allocates a contiguous range of tces (power-of-2 size) */
static inline long alloc_tce_range(struct TceTable *,
                                   unsigned order );

/* allocates a contiguous range of tces (power-of-2 size)
 * assumes lock already held
 */
static long alloc_tce_range_nolock(struct TceTable *,
                                   unsigned order );

/* frees a contiguous range of tces (power-of-2 size) */
static inline void free_tce_range(struct TceTable *,
                                  long tcenum,
                                  unsigned order );
```

```
91
92  /* frees a contiguous rnage of tces (power−of−2 size)
93   * assumes lock already held
94   */
95  void free_tce_range_nolock(struct TceTable *,
96                             long tcenum,
97                             unsigned order );
98
99  /* allocates a range of tces and sets them to the pages  */
100 static inline dma_addr_t get_tces( struct TceTable *,
101                                    unsigned order,
102                                    void *page,
103                                    unsigned numPages,
104                                    int direction );
105
106 static long test_tce_range( struct TceTable *,
107                             long tcenum,
108                             unsigned order );
109
110 static unsigned fill_scatterlist_sg(struct scatterlist *sg, int nents,
111                                     dma_addr_t dma_addr,
112                                     unsigned long numTces );
113
114 static unsigned long num_tces_sg( struct scatterlist *sg,
115                                   int nents );
116
117 static dma_addr_t create_tces_sg( struct TceTable *tbl,
118                                   struct scatterlist *sg,
119                                   int nents,
120                                   unsigned numTces,
121                                   int direction );
122
123 static void getTceTableParmsiSeries(struct iSeries_Device_Node* DevNode,
124                                     struct TceTable *tce_table_parms );
125
126 static void getTceTableParmsPSeries( struct pci_controller *phb,
127                                      struct device_node *dn,
128                                      struct TceTable *tce_table_parms );
129
130 static void getTceTableParmsPSeriesLP(struct pci_controller *phb,
131                                       struct device_node *dn,
132                                       struct TceTable *newTceTable );
133
134 static struct TceTable* findHwTceTable(struct TceTable * newTceTable );
135
136 void create_pci_bus_tce_table( unsigned long token );
137
138 u8 iSeries_Get_Bus( struct pci_dev * dv )
139 {
140         return 0;
141 }
142
143 static inline struct TceTable *get_tce_table(struct pci_dev *dev)
144 {
145         if (!dev)
146                 dev = ppc64_isabridge_dev;
147         if (!dev)
148                 return NULL;
149         if (systemcfg->platform == PLATFORM_ISERIES_LPAR) {
150                 return ISERIES_DEVNODE(dev)->DevTceTable;
151         } else {
152                 return PCI_GET_DN(dev)->tce_table;
153         }
154 }
155
156 static unsigned long __inline__ count_leading_zeros64( unsigned long x )
157 {
158         unsigned long lz;
159         asm("cntlzd %0,%1" : "=r"(lz) : "r"(x));
160         return lz;
161 }
162
163 static void tce_build_iSeries(struct TceTable *tbl, long tcenum,
164                               unsigned long uaddr, int direction )
165 {
166         u64 setTceRc;
167         union Tce tce;
168
169         PPCDBG(PPCDBG_TCE, "build_tce: uaddr = 0x%lx\n", uaddr);
170         PPCDBG(PPCDBG_TCE, "\ttcenum = 0x%lx, tbl = 0x%lx, index=%lx\n",
171                tcenum, tbl, tbl->index);
172
173         tce.wholeTce = 0;
174         tce.tceBits.rpn = (virt_to_absolute(uaddr)) >> PAGE_SHIFT;
175
176         /* If for virtual bus */
177         if ( tbl->tceType == TCE_VB ) {
178                 tce.tceBits.valid = 1;
179                 tce.tceBits.allIo = 1;
180                 if ( direction != PCI_DMA_TODEVICE )
```

```
181                         tce.tceBits.readWrite = 1;
182         } else {
183                 /* If for PCI bus */
184                 tce.tceBits.readWrite = 1; // Read allowed
185                 if ( direction != PCI_DMA_TODEVICE )
186                         tce.tceBits.pciWrite = 1;
187         }
188
189         setTceRc = HvCallXm_setTce((u64)tbl->index,
190                                    (u64)tcenum,
191                                    tce.wholeTce );
192         if(setTceRc) {
193                 panic("PCI_DMA: HvCallXm_setTce failed, Rc: 0x%lx\n", setTceRc);
194         }
195 }
196
197 static void tce_build_pSeries(struct TceTable *tbl, long tcenum,
198                               unsigned long uaddr, int direction )
199 {
200         union Tce tce;
201         union Tce *tce_addr;
202
203         PPCDBG(PPCDBG_TCE, "build_tce: uaddr = 0x%lx\n", uaddr);
204         PPCDBG(PPCDBG_TCE, "\ttcenum = 0x%lx, tbl = 0x%lx, index=%lx\n",
205                tcenum, tbl, tbl->index);
206
207         tce.wholeTce = 0;
208         tce.tceBits.rpn = (virt_to_absolute(uaddr)) >> PAGE_SHIFT;
209
210         tce.tceBits.readWrite = 1; // Read allowed
211         if ( direction != PCI_DMA_TODEVICE ) tce.tceBits.pciWrite = 1;
212
213         tce_addr = ((union Tce *)tbl->base) + tcenum;
214         *tce_addr = (union Tce)tce.wholeTce;
215
216 }
217
218 /*
219  * Build a TceTable structure.  This contains a multi-level bit map which
220  * is used to manage allocation of the tce space.
221  */
222 static struct TceTable *build_tce_table( struct TceTable * tbl )
223 {
224         unsigned long bits, bytes, totalBytes;
225         unsigned long numBits[NUM_TCE_LEVELS], numBytes[NUM_TCE_LEVELS];
226         unsigned i, k, m;
227         unsigned char * pos, * p, b;
228
229         PPCDBG(PPCDBG_TCEINIT, "build_tce_table: tbl = 0x%lx\n", tbl);
230         spin_lock_init( &(tbl->lock) );
231
232         tbl->mlbm.maxLevel = 0;
233
234         /* Compute number of bits and bytes for each level of the
235          * multi-level bit map
236          */
237         totalBytes = 0;
238         bits = tbl->size * (PAGE_SIZE / sizeof( union Tce ));
239
240         for ( i=0; i<NUM_TCE_LEVELS; ++i ) {
241                 bytes = ((bits+63)/64) * 8;
242                 PPCDBG(PPCDBG_TCEINIT, "build_tce_table: level %d bits=%ld, bytes=%ld\n", i, bits, bytes );
243                 numBits[i] = bits;
244                 numBytes[i] = bytes;
245                 bits /= 2;
246                 totalBytes += bytes;
247         }
248         PPCDBG(PPCDBG_TCEINIT, "build_tce_table: totalBytes=%ld\n", totalBytes );
249
250         pos = (char *)__get_free_pages( GFP_ATOMIC, get_order( totalBytes ));
251
252         if ( pos == NULL ) {
253                 panic("PCI_DMA: Allocation failed in build_tce_table!\n");
254         }
255
256         /* For each level, fill in the pointer to the bit map,
257          * and turn on the last bit in the bit map (if the
258          * number of bits in the map is odd).  The highest
259          * level will get all of its bits turned on.
260          */
261         memset( pos, 0, totalBytes );
262         for (i=0; i<NUM_TCE_LEVELS; ++i) {
263                 if ( numBytes[i] ) {
264                         tbl->mlbm.level[i].map = pos;
265                         tbl->mlbm.maxLevel = i;
266
267                         if ( numBits[i] & 1 ) {
268                                 p = pos + numBytes[i] - 1;
269                                 m = (( numBits[i] % 8) - 1) & 7;
270                                 *p = 0x80 >> m;
```

```
271                                     PPCDBG(PPCDBG_TCEINIT, "build_tce_table: level %d last bit %x\n", i, 0x80>>m );
272                             }
273                     }
274                     else
275                             tbl->mlbm.level[i].map = 0;
276                     pos += numBytes[i];
277                     tbl->mlbm.level[i].numBits = numBits[i];
278                     tbl->mlbm.level[i].numBytes = numBytes[i];
279             }
280
281             /* For the highest level, turn on all the bits */
282
283             i = tbl->mlbm.maxLevel;
284             p = tbl->mlbm.level[i].map;
285             m = numBits[i];
286             PPCDBG(PPCDBG_TCEINIT, "build_tce_table: highest level (%d) has all bits set\n", i);
287             for (k=0; k<numBytes[i]; ++k) {
288                     if ( m >= 8 ) {
289                             /* handle full bytes */
290                             *p++ = 0xff;
291                             m -= 8;
292                     }
293                     else if(m>0) {
294                             /* handle the last partial byte */
295                             b = 0x80;
296                             *p = 0;
297                             while (m) {
298                                     *p |= b;
299                                     b >>= 1;
300                                     --m;
301                             }
302                     } else {
303                             break;
304                     }
305             }
306
307             return tbl;
308     }
309
310     static inline long alloc_tce_range( struct TceTable *tbl, unsigned order )
311     {
312             long retval;
313             unsigned long flags;
314
315             /* Lock the tce allocation bitmap */
316             spin_lock_irqsave( &(tbl->lock), flags );
317
318             /* Do the actual work */
319             retval = alloc_tce_range_nolock( tbl, order );
320
321             /* Unlock the tce allocation bitmap */
322             spin_unlock_irqrestore( &(tbl->lock), flags );
323
324             return retval;
325     }
326
327     static long alloc_tce_range_nolock( struct TceTable *tbl, unsigned order )
328     {
329             unsigned long numBits, numBytes;
330             unsigned long i, bit, block, mask;
331             long tcenum;
332             u64 * map;
333
334             /* If the order (power of 2 size) requested is larger than our
335              * biggest, indicate failure
336              */
337             if(order >= NUM_TCE_LEVELS) {
338                     /* This can happen if block of TCE's are not found. This code      */
339                     /*  maybe in a recursive loop looking up the bit map for the range.*/
340                     panic("PCI_DMA: alloc_tce_range_nolock: invalid order: %d\n",order);
341             }
342
343             numBits =  tbl->mlbm.level[order].numBits;
344             numBytes = tbl->mlbm.level[order].numBytes;
345             map =      (u64 *)tbl->mlbm.level[order].map;
346
347             /* Initialize return value to -1 (failure) */
348             tcenum = -1;
349
350             /* Loop through the bytes of the bitmap */
351             for (i=0; i<numBytes/8; ++i) {
352                     if ( *map ) {
353                             /* A free block is found, compute the block
354                              * number (of this size)
355                              */
356                             bit = count_leading_zeros64( *map );
357                             block = (i * 64) + bit;    /* Bit count to free entry */
358
359                             /* turn off the bit in the map to indicate
360                              * that the block is now in use
```

```
361                              */
362                             mask = 0x1UL << (63 - bit);
363                             *map &= ~mask;
364
365                             /* compute the index into our tce table for
366                              * the first tce in the block
367                              */
368                             PPCDBG(PPCDBG_TCE, "alloc_tce_range_nolock: allocating block %ld, (byte=%ld, bit=%ld) order %d\n", block, i, bi
    t, order );
369                             tcenum = block << order;
370                             return tcenum;
371                     }
372                     ++map;
373             }
374
375     #ifdef DEBUG_TCE
376             if ( tcenum == -1 ) {
377                     PPCDBG(PPCDBG_TCE, "alloc_tce_range_nolock: no available blocks of order = %d\n", order );
378                     if ( order < tbl->mlbm.maxLevel ) {
379                             PPCDBG(PPCDBG_TCE, "alloc_tce_range_nolock: trying next bigger size\n" );
380                     }
381                     else {
382                             panic("PCI_DMA: alloc_tce_range_nolock: maximum size reached...failing\n");
383                     }
384             }
385     #endif
386
387             /* If no block of the requested size was found, try the next
388              * size bigger.  If one of those is found, return the second
389              * half of the block to freespace and keep the first half
390              */
391             if((tcenum == -1) && (order < (NUM_TCE_LEVELS - 1))) {
392                     tcenum = alloc_tce_range_nolock( tbl, order+1 );
393                     if ( tcenum != -1 ) {
394                             free_tce_range_nolock( tbl, tcenum+(1<<order), order );
395                     }
396             }
397
398             /* Return the index of the first tce in the block
399              * (or -1 if we failed)
400              */
401             return tcenum;
402     }
403
404     static inline void free_tce_range(struct TceTable *tbl,
405                                       long tcenum, unsigned order )
406     {
407             unsigned long flags;
408
409             /* Lock the tce allocation bitmap */
410             spin_lock_irqsave( &(tbl->lock), flags );
411
412             /* Do the actual work */
413             free_tce_range_nolock( tbl, tcenum, order );
414
415             /* Unlock the tce allocation bitmap */
416             spin_unlock_irqrestore( &(tbl->lock), flags );
417
418     }
419
420     void free_tce_range_nolock(struct TceTable *tbl,
421                                long tcenum, unsigned order )
422     {
423             unsigned long block;
424             unsigned byte, bit, mask, b;
425             unsigned char  * map, * bytep;
426
427             if (order >= NUM_TCE_LEVELS) {
428                     panic("PCI_DMA: free_tce_range: invalid order: 0x%x\n",order);
429                     return;
430             }
431
432             block = tcenum >> order;
433
434     #ifdef MONITOR_TCE
435             if ( tcenum != (block << order ) ) {
436                     printk("PCI_DMA: Free_tce_range: tcenum %lx misaligned for order %x\n",tcenum, order);
437                     return;
438             }
439             if ( block >= tbl->mlbm.level[order].numBits ) {
440                     printk("PCI_DMA: Free_tce_range: tcenum %lx is outside the range of this map (order %x, numBits %lx\n",
441                             tcenum, order, tbl->mlbm.level[order].numBits );
442                     return;
443             }
444             if ( test_tce_range( tbl, tcenum, order ) ) {
445                     printk("PCI_DMA: Freeing range not allocated: tTceTable %p, tcenum %lx, order %x\n",tbl, tcenum, order );
446                     return;
447             }
448     #endif
449
```

```
450         map = tbl->mlbm.level[order].map;
451         byte  = block / 8;
452         bit   = block % 8;
453         mask  = 0x80 >> bit;
454         bytep = map + byte;
455
456 #ifdef DEBUG_TCE
457         PPCDBG(PPCDBG_TCE,"free_tce_range_nolock: freeing block %ld (byte=%d, bit=%d) of order %d\n",
458                block, byte, bit, order);
459 #endif
460
461 #ifdef MONITOR_TCE
462         if ( *bytep & mask ) {
463                 panic("PCI_DMA: Tce already free: TceTable %p, tcenum %lx, order %x\n",tbl,tcenum,order);
464         }
465 #endif
466
467         *bytep |= mask;
468
469         /* If there is a higher level in the bit map than this we may be
470          * able to buddy up this block with its partner.
471          *   If this is the highest level we can't buddy up
472          *   If this level has an odd number of bits and
473          *      we are freeing the last block we can't buddy up
474          * Don't buddy up if it's in the first 1/4 of the level
475          */
476         if (( order < tbl->mlbm.maxLevel ) &&
477             ( block > (tbl->mlbm.level[order].numBits/4) ) &&
478             (( block < tbl->mlbm.level[order].numBits-1 ) ||
479               ( 0 == ( tbl->mlbm.level[order].numBits & 1)))) {
480                 /* See if we can buddy up the block we just freed */
481                 bit  &= 6;              /* get to the first of the buddy bits */
482                 mask  = 0xc0 >> bit;    /* build two bit mask */
483                 b     = *bytep & mask;  /* Get the two bits */
484                 if ( 0 == (b ^ mask) ) { /* If both bits are on */
485                         /* both of the buddy blocks are free we can combine them */
486                         *bytep ^= mask; /* turn off the two bits */
487                         block = ( byte * 8 ) + bit; /* block of first of buddies */
488                         tcenum = block << order;
489                         /* free the buddied block */
490                         PPCDBG(PPCDBG_TCE,
491                                "free_tce_range: buddying blocks %ld & %ld\n",
492                                block, block+1);
493                         free_tce_range_nolock( tbl, tcenum, order+1 );
494                 }
495         }
496 }
497
498 static long test_tce_range( struct TceTable *tbl, long tcenum, unsigned order )
499 {
500         unsigned long block;
501         unsigned byte, bit, mask, b;
502         long    retval, retLeft, retRight;
503         unsigned char  * map;
504
505         map = tbl->mlbm.level[order].map;
506         block = tcenum >> order;
507         byte = block / 8;               /* Byte within bitmap */
508         bit  = block % 8;               /* Bit within byte */
509         mask = 0x80 >> bit;
510         b    = (*(map+byte) & mask );   /* 0 if block is allocated, else free */
511         if ( b )
512                 retval = 1;             /* 1 == block is free */
513         else
514                 retval = 0;             /* 0 == block is allocated */
515         /* Test bits at all levels below this to ensure that all agree */
516
517         if (order) {
518                 retLeft  = test_tce_range( tbl, tcenum, order-1 );
519                 retRight = test_tce_range( tbl, tcenum+(1<<(order-1)), order-1 );
520                 if ( retLeft || retRight ) {
521                         retval = 2;
522                 }
523         }
524
525         /* Test bits at all levels above this to ensure that all agree */
526
527         return retval;
528 }
529
530 static inline dma_addr_t get_tces( struct TceTable *tbl, unsigned order, void *page, unsigned numPages, int direc
    tion )
531 {
532         long tcenum;
533         unsigned long uaddr;
534         unsigned i;
535         dma_addr_t retTce = NO_TCE;
536
537         uaddr = (unsigned long)page & PAGE_MASK;
538
```

```
539         /* Allocate a range of tces */
540         tcenum = alloc_tce_range( tbl, order );
541         if ( tcenum != -1 ) {
542                 /* We got the tces we wanted */
543                 tcenum += tbl->startOffset;     /* Offset into real TCE table */
544                 retTce = tcenum << PAGE_SHIFT;  /* Set the return dma address */
545                 /* Setup a tce for each page */
546                 for (i=0; i<numPages; ++i) {
547                         ppc_md.tce_build(tbl, tcenum, uaddr, direction);
548                         ++tcenum;
549                         uaddr += PAGE_SIZE;
550                 }
551                 /* Make sure the update is visible to hardware.
552                    sync required to synchronize the update to
553                    the TCE table with the MMIO that will send
554                    the bus address to the IOA */
555                 __asm__ __volatile__ ("sync" : : : "memory");
556         }
557         else {
558                 panic("PCI_DMA: Tce Allocation failure in get_tces. 0x%p\n",tbl);
559         }
560
561         return retTce;
562 }
563
564 static void tce_free_one_iSeries( struct TceTable *tbl, long tcenum )
565 {
566         u64 set_tce_rc;
567         union Tce tce;
568         tce.wholeTce = 0;
569         set_tce_rc = HvCallXm_setTce((u64)tbl->index,
570                                      (u64)tcenum,
571                                      tce.wholeTce);
572         if ( set_tce_rc )
573                 panic("PCI_DMA: HvCallXm_setTce failed, Rc: 0x%lx\n", set_tce_rc);
574
575 }
576
577 static void tce_free_one_pSeries( struct TceTable *tbl, long tcenum )
578 {
579         union Tce tce;
580         union Tce *tce_addr;
581
582         tce.wholeTce = 0;
583
584         tce_addr  = ((union Tce *)tbl->base) + tcenum;
585         *tce_addr = (union Tce)tce.wholeTce;
586
587 }
588
589 static void tce_free(struct TceTable *tbl, dma_addr_t dma_addr,
590                              unsigned order, unsigned num_pages)
591 {
592         long tcenum, total_tces, free_tce;
593         unsigned i;
594
595         total_tces = (tbl->size * (PAGE_SIZE / sizeof(union Tce)));
596
597         tcenum = dma_addr >> PAGE_SHIFT;
598         free_tce = tcenum - tbl->startOffset;
599
600         if ( ( (free_tce + num_pages) > total_tces ) ||
601              ( tcenum < tbl->startOffset ) ) {
602                 printk("tce_free: invalid tcenum\n");
603                 printk("\ttcenum   = 0x%lx\n", tcenum);
604                 printk("\tTCE Table = 0x%lx\n", (u64)tbl);
605                 printk("\tbus#     = 0x%lx\n", (u64)tbl->busNumber );
606                 printk("\tsize     = 0x%lx\n", (u64)tbl->size);
607                 printk("\tstartOff = 0x%lx\n", (u64)tbl->startOffset );
608                 printk("\tindex    = 0x%lx\n", (u64)tbl->index);
609                 return;
610         }
611
612         for (i=0; i<num_pages; ++i) {
613                 ppc_md.tce_free_one(tbl, tcenum);
614                 ++tcenum;
615         }
616
617         /* No sync (to make TCE change visible) is required here.
618            The lwsync when acquiring the lock in free_tce_range
619            is sufficient to synchronize with the bitmap.
620         */
621
622         free_tce_range( tbl, free_tce, order );
623 }
624
625 void __init create_virtual_bus_tce_table(void)
626 {
627         struct TceTable *t;
628         struct TceTableManagerCB virtBusTceTableParms;
```

```
629         u64 absParmsPtr;
630
631         virtBusTceTableParms.busNumber = 255;   /* Bus 255 is the virtual bus */
632         virtBusTceTableParms.virtualBusFlag = 0xff; /* Ask for virtual bus */
633
634         absParmsPtr = virt_to_absolute( (u64)&virtBusTceTableParms );
635         HvCallXm_getTceTableParms( absParmsPtr );
636
637         virtBusVethTceTable.size = virtBusTceTableParms.size / 2;
638         virtBusVethTceTable.busNumber = virtBusTceTableParms.busNumber;
639         virtBusVethTceTable.startOffset = virtBusTceTableParms.startOffset;
640         virtBusVethTceTable.index = virtBusTceTableParms.index;
641         virtBusVethTceTable.tceType = TCE_VB;
642
643         virtBusVioTceTable.size = virtBusTceTableParms.size − virtBusVethTceTable.size;
644         virtBusVioTceTable.busNumber = virtBusTceTableParms.busNumber;
645         virtBusVioTceTable.startOffset = virtBusTceTableParms.startOffset +
646                         virtBusVethTceTable.size * (PAGE_SIZE/sizeof(union Tce));
647         virtBusVioTceTable.index = virtBusTceTableParms.index;
648         virtBusVioTceTable.tceType = TCE_VB;
649
650         t = build_tce_table( &virtBusVethTceTable );
651         if ( t ) {
652                 /* tceTables[255] = t; */
653                 //VirtBusVethTceTable = t;
654                 printk( "Virtual Bus VETH TCE table built successfully.\n");
655                 printk( " TCE table size = %ld entries\n",
656                                 (unsigned long)t->size*(PAGE_SIZE/sizeof(union Tce)) );
657                 printk( " TCE table token = %d\n",
658                                 (unsigned)t->index );
659                 printk( " TCE table start entry = 0x%lx\n",
660                                 (unsigned long)t->startOffset );
661         }
662         else printk( "Virtual Bus VETH TCE table failed.\n");
663
664         t = build_tce_table( &virtBusVioTceTable );
665         if ( t ) {
666                 //VirtBusVioTceTable = t;
667                 printk( "Virtual Bus VIO TCE table built successfully.\n");
668                 printk( " TCE table size = %ld entries\n",
669                                 (unsigned long)t->size*(PAGE_SIZE/sizeof(union Tce)) );
670                 printk( " TCE table token = %d\n",
671                                 (unsigned)t->index );
672                 printk( " TCE table start entry = 0x%lx\n",
673                                 (unsigned long)t->startOffset );
674         }
675         else printk( "Virtual Bus VIO TCE table failed.\n");
676 }
677
678 void create_tce_tables_for_buses(struct list_head *bus_list)
679 {
680         struct pci_controller* phb;
681         struct device_node *dn, *first_dn;
682         int num_slots, num_slots_ilog2;
683         int first_phb = 1;
684
685         for (phb=hose_head;phb;phb=phb->next) {
686                 first_dn = ((struct device_node *)phb->arch_data)->child;
687                 /* Carve 2GB into the largest dma_window_size possible */
688                 for (dn = first_dn, num_slots = 0; dn != NULL; dn = dn->sibling)
689                         num_slots++;
690                 num_slots_ilog2 = __ilog2(num_slots);
691                 if ((1<<num_slots_ilog2) != num_slots)
692                         num_slots_ilog2++;
693                 phb->dma_window_size = 1 << (22 − num_slots_ilog2);
694                 /* Reserve 16MB of DMA space on the first PHB.
695                  * We should probably be more careful and use firmware props.
696                  * In reality this space is remapped, not lost.  But we don't
697                  * want to get that smart to handle it −− too much work.
698                  */
699                 phb->dma_window_base_cur = first_phb ? (1 << 12) : 0;
700                 first_phb = 0;
701                 for (dn = first_dn, num_slots = 0; dn != NULL; dn = dn->sibling) {
702                         create_pci_bus_tce_table((unsigned long)dn);
703                 }
704         }
705 }
706
707 void create_tce_tables_for_busesLP(struct list_head *bus_list)
708 {
709         struct list_head *ln;
710         struct pci_bus *bus;
711         struct device_node *busdn;
712         u32 *dma_window;
713         for (ln=bus_list->next; ln != bus_list; ln=ln->next) {
714                 bus = pci_bus_b(ln);
715                 busdn = PCI_GET_DN(bus);
716                 /* NOTE: there should never be a window declared on a bus when
717                  * child devices also have a window.  If this should ever be
718                  * architected, we probably want children to have priority.
```

```
719                      * In reality, the PHB containing ISA has the property, but otherwise
720                      * it is the pci-bridges that have the property.
721                      */
722                     dma_window = (u32 *)get_property(busdn, "ibm,dma-window", 0);
723                     if (dma_window) {
724                             /* Busno hasn't been copied yet.
725                              * Do it now because getTceTableParmsPSeriesLP needs it.
726                              */
727                             busdn->busno = bus->number;
728                             create_pci_bus_tce_table((unsigned long)busdn);
729                     } else
730                             create_tce_tables_for_busesLP(&bus->children);
731             }
732     }
733
734     void create_tce_tables(void) {
735             struct pci_dev *dev;
736             struct device_node *dn, *mydn;
737
738             if (systemcfg->platform == PLATFORM_PSERIES_LPAR) {
739                     create_tce_tables_for_busesLP(&pci_root_buses);
740             }
741             else {
742                     create_tce_tables_for_buses(&pci_root_buses);
743             }
744             /* Now copy the tce_table ptr from the bus devices down to every
745              * pci device_node.  This means get_tce_table() won't need to search
746              * up the device tree to find it.
747              */
748             pci_for_each_dev(dev) {
749                     mydn = dn = PCI_GET_DN(dev);
750                     while (dn && dn->tce_table == NULL)
751                             dn = dn->parent;
752                     if (dn) {
753                             mydn->tce_table = dn->tce_table;
754                     }
755             }
756     }
757
758
759     /*
760      * iSeries token = iSeries_device_Node*
761      * pSeries token = pci_controller*
762      *
763      */
764     void create_pci_bus_tce_table( unsigned long token ) {
765             struct TceTable * newTceTable;
766
767             PPCDBG(PPCDBG_TCE, "Entering create_pci_bus_tce_table.\n");
768             PPCDBG(PPCDBG_TCE, "\ttoken = 0x%lx\n", token);
769
770             newTceTable = (struct TceTable *)kmalloc( sizeof(struct TceTable), GFP_KERNEL );
771
772             /*************************************************************/
773             /* For the iSeries machines, the HvTce Table can be one of three */
774             /* flavors,                                                  */
775             /* - Single bus TCE table,                                   */
776             /* - Tce Table Share between buses,                          */
777             /* - Tce Table per logical slot.                             */
778             /*************************************************************/
779             if(systemcfg->platform == PLATFORM_ISERIES_LPAR) {
780
781                     struct iSeries_Device_Node* DevNode = (struct iSeries_Device_Node*)token;
782                     getTceTableParmsiSeries(DevNode,newTceTable);
783
784                     /* Look for existing TCE table for this device.          */
785                     DevNode->DevTceTable = findHwTceTable(newTceTable );
786                     if( DevNode->DevTceTable == NULL) {
787                             DevNode->DevTceTable = build_tce_table( newTceTable );
788                     }
789                     else {
790                         /* We're using a shared table, free this new one.    */
791                         kfree(newTceTable);
792                     }
793                     printk("Pci Device 0x%p TceTable: %p\n",DevNode,DevNode->DevTceTable);
794                     return;
795             }
796             /* pSeries Leg */
797             else {
798                     struct device_node *dn;
799                     struct pci_controller *phb;
800
801                     dn = (struct device_node *)token;
802                     phb = dn->phb;
803                     if (systemcfg->platform == PLATFORM_PSERIES)
804                             getTceTableParmsPSeries(phb, dn, newTceTable);
805                     else
806                             getTceTableParmsPSeriesLP(phb, dn, newTceTable);
807
808                     dn->tce_table  = build_tce_table( newTceTable );
```

```
809         }
810 }
811
812 /*****************************************************************/
813 /* This function compares the known Tce tables to find a TceTable that */
814 /* has already been built for hardware TCEs.                     */
815 /* Search the complete(all devices) for a TCE table assigned.  If the  */
816 /* startOffset, index, and size match, then the TCE for this device has*/
817 /* already been built and it should be shared with this device   */
818 /*****************************************************************/
819 static struct TceTable* findHwTceTable(struct TceTable * newTceTable )
820 {
821         struct list_head* Device_Node_Ptr    = iSeries_Global_Device_List.next;
822         /* Cache the compare values. */
823         u64  startOffset = newTceTable->startOffset;
824         u64  index       = newTceTable->index;
825         u64  size        = newTceTable->size;
826
827         while(Device_Node_Ptr != &iSeries_Global_Device_List) {
828                 struct iSeries_Device_Node* CmprNode = (struct iSeries_Device_Node*)Device_Node_Ptr;
829                 if( CmprNode->DevTceTable != NULL &&
830                     CmprNode->DevTceTable->tceType == TCE_PCI) {
831                         if( CmprNode->DevTceTable->startOffset == startOffset &&
832                             CmprNode->DevTceTable->index       == index       &&
833                             CmprNode->DevTceTable->size        == size        ) {
834                                 printk("PCI TCE table matches 0x%p \n",CmprNode->DevTceTable);
835                                 return CmprNode->DevTceTable;
836                         }
837                 }
838                 /* Get next Device Node in List             */
839                 Device_Node_Ptr = Device_Node_Ptr->next;
840         }
841         return NULL;
842 }
843
844 /*****************************************************************/
845 /* Call Hv with the architected data structure to get TCE table info.  */
846 /* info. Put the returned data into the Linux representation of the    */
847 /* TCE table data.                                               */
848 /* The Hardware Tce table comes in three flavors.                */
849 /* 1. TCE table shared between Buses.                            */
850 /* 2. TCE table per Bus.                                         */
851 /* 3. TCE Table per IOA.                                         */
852 /*****************************************************************/
853 static void getTceTableParmsiSeries(struct iSeries_Device_Node* DevNode,
854                                     struct TceTable* newTceTable )
855 {
856         struct TceTableManagerCB* pciBusTceTableParms = (struct TceTableManagerCB*)kmalloc( sizeof(struct TceTabl
    eManagerCB), GFP_KERNEL );
857         if(pciBusTceTableParms == NULL) panic("PCI_DMA: TCE Table Allocation failed.");
858
859         memset( (void*)pciBusTceTableParms,0,sizeof(struct TceTableManagerCB) );
860         pciBusTceTableParms->busNumber      = ISERIES_BUS(DevNode);
861         pciBusTceTableParms->logicalSlot    = DevNode->LogicalSlot;
862         pciBusTceTableParms->virtualBusFlag = 0;
863
864         HvCallXm_getTceTableParms( REALADDR(pciBusTceTableParms) );
865
866         /* PciTceTableParms Bus:0x18 Slot:0x04 Start:0x000000 Offset:0x04c000 Size:0x0020 */
867         printk("PciTceTableParms Bus:0x%02lx Slot:0x%02x Start:0x%06lx Offset:0x%06lx Size:0x%04lx\n",
868                pciBusTceTableParms->busNumber,
869                pciBusTceTableParms->logicalSlot,
870                pciBusTceTableParms->start,
871                pciBusTceTableParms->startOffset,
872                pciBusTceTableParms->size);
873
874         if(pciBusTceTableParms->size == 0) {
875                 printk("PCI_DMA: Possible Structure mismatch, 0x%p\n",pciBusTceTableParms);
876                 panic( "PCI_DMA: pciBusTceTableParms->size is zero, halt here!");
877         }
878
879         newTceTable->size        = pciBusTceTableParms->size;
880         newTceTable->busNumber   = pciBusTceTableParms->busNumber;
881         newTceTable->startOffset = pciBusTceTableParms->startOffset;
882         newTceTable->index       = pciBusTceTableParms->index;
883         newTceTable->tceType     = TCE_PCI;
884
885         kfree(pciBusTceTableParms);
886 }
887
888 static void getTceTableParmsPSeries(struct pci_controller *phb,
889                                     struct device_node *dn,
890                                     struct TceTable *newTceTable ) {
891         phandle node;
892         unsigned long i;
893
894         node = ((struct device_node *)(phb->arch_data))->node;
895
896         PPCDBG(PPCDBG_TCEINIT, "getTceTableParms: start\n");
897         PPCDBG(PPCDBG_TCEINIT, "\tof_tce_table = 0x%lx\n", of_tce_table);
```

```
898         PPCDBG(PPCDBG_TCEINIT, "\tphb      = 0x%lx\n", phb);
899         PPCDBG(PPCDBG_TCEINIT, "\tdn       = 0x%lx\n", dn);
900         PPCDBG(PPCDBG_TCEINIT, "\tdn->name    = %s\n", dn->name);
901         PPCDBG(PPCDBG_TCEINIT, "\tdn->full_name= %s\n", dn->full_name);
902         PPCDBG(PPCDBG_TCEINIT, "\tnewTceTable  = 0x%lx\n", newTceTable);
903         PPCDBG(PPCDBG_TCEINIT, "\tdma_window_size = 0x%lx\n", phb->dma_window_size);
904
905         i = 0;
906         while(of_tce_table[i].node) {
907                 PPCDBG(PPCDBG_TCEINIT, "\tof_tce_table[%d].node = 0x%lx\n",
908                         i, of_tce_table[i].node);
909                 PPCDBG(PPCDBG_TCEINIT, "\tof_tce_table[%d].base = 0x%lx\n",
910                         i, of_tce_table[i].base);
911                 PPCDBG(PPCDBG_TCEINIT, "\tof_tce_table[%d].size = 0x%lx\n",
912                         i, of_tce_table[i].size >> PAGE_SHIFT);
913                 PPCDBG(PPCDBG_TCEINIT, "\tphb->arch_data->node = 0x%lx\n",
914                        node);
915
916                 if(of_tce_table[i].node == node) {
917                         memset((void *)of_tce_table[i].base,
918                                 0, of_tce_table[i].size);
919                         newTceTable->busNumber = phb->bus->number;
920
921                         /* Units of tce entries.                        */
922                         newTceTable->startOffset = phb->dma_window_base_cur;
923
924                         /* Adjust the current table offset to the next  */
925                         /* region.  Measured in TCE entries. Force an   */
926                         /* alignment to the size alloted per IOA. This  */
927                         /* makes it easier to remove the 1st 16MB.      */
928                         phb->dma_window_base_cur += (phb->dma_window_size>>3);
929                         phb->dma_window_base_cur &=
930                                 ~((phb->dma_window_size>>3)-1);
931
932                         /* Set the tce table size - measured in units   */
933                         /* of pages of tce table.                       */
934                         newTceTable->size = ((phb->dma_window_base_cur -
935                                               newTceTable->startOffset) << 3)
936                                             >> PAGE_SHIFT;
937
938                         /* Test if we are going over 2GB of DMA space.  */
939                         if(phb->dma_window_base_cur > (1 << 19)) {
940                                 panic("PCI_DMA: Unexpected number of IOAs under this PHB.\n");
941                         }
942
943                         newTceTable->base = of_tce_table[i].base;
944                         newTceTable->index = 0;
945
946                         PPCDBG(PPCDBG_TCEINIT,
947                                 "\tnewTceTable->base     = 0x%lx\n",
948                                 newTceTable->base);
949                         PPCDBG(PPCDBG_TCEINIT,
950                                 "\tnewTceTable->startOffset = 0x%lx"
951                                 "(# tce entries)\n",
952                                 newTceTable->startOffset);
953                         PPCDBG(PPCDBG_TCEINIT,
954                                 "\tnewTceTable->size      = 0x%lx"
955                                 "(# pages of tce table)\n",
956                                 newTceTable->size);
957                 }
958                 i++;
959         }
960 }
961
962 /*
963  * getTceTableParmsPSeriesLP
964  *
965  * Function: On pSeries LPAR systems, return TCE table info, given a pci bus.
966  *
967  * ToDo: properly interpret the ibm,dma-window property.  The definition is:
968  *      logical-bus-number      (1 word)
969  *      phys-address            (#address-cells words)
970  *      size                    (#cell-size words)
971  *
972  * Currently we hard code these sizes (more or less).
973  */
974 static void getTceTableParmsPSeriesLP(struct pci_controller *phb,
975                                       struct device_node *dn,
976                                       struct TceTable *newTceTable ) {
977         u32 *dma_window = (u32 *)get_property(dn, "ibm,dma-window", 0);
978         if (!dma_window) {
979                 panic("PCI_DMA: getTceTableParmsPSeriesLP: device %s has no ibm,dma-window property!\n", dn->full_name);
980         }
981
982         newTceTable->busNumber = dn->busno;
983         newTceTable->size = (((((unsigned long)dma_window[4] << 32) | (unsigned long)dma_window[5]) >> PAGE_SHIFT
    ) << 3) >> PAGE_SHIFT;
984         newTceTable->startOffset = ((((unsigned long)dma_window[2] << 32) | (unsigned long)dma_window[3]) >> 12);
985         newTceTable->base = 0;
986         newTceTable->index = dma_window[0];
```

```
987         PPCDBG(PPCDBG_TCEINIT, "getTceTableParmsPSeriesLP for bus 0x%lx:\n", dn->busno);
988         PPCDBG(PPCDBG_TCEINIT, "\tDevice = %s\n", dn->full_name);
989         PPCDBG(PPCDBG_TCEINIT, "\tnewTceTable->index     = 0x%lx\n", newTceTable->index);
990         PPCDBG(PPCDBG_TCEINIT, "\tnewTceTable->startOffset = 0x%lx\n", newTceTable->startOffset);
991         PPCDBG(PPCDBG_TCEINIT, "\tnewTceTable->size      = 0x%lx\n", newTceTable->size);
992 }
993
994 /* Allocates a contiguous real buffer and creates TCEs over it.
995  * Returns the virtual address of the buffer and sets dma_handle
996  * to the dma address (tce) of the first page.
997  */
998 void *pci_alloc_consistent(struct pci_dev *hwdev, size_t size,
999                            dma_addr_t *dma_handle)
1000 {
1001         struct TceTable * tbl;
1002         void *ret = NULL;
1003         unsigned order, nPages;
1004         dma_addr_t tce;
1005
1006         PPCDBG(PPCDBG_TCE, "pci_alloc_consistent:\n");
1007         PPCDBG(PPCDBG_TCE, "\thwdev    = 0x%16.16lx\n", hwdev);
1008         PPCDBG(PPCDBG_TCE, "\tsize     = 0x%16.16lx\n", size);
1009         PPCDBG(PPCDBG_TCE, "\tdma_handle = 0x%16.16lx\n", dma_handle);
1010
1011         size = PAGE_ALIGN(size);
1012         order = get_order(size);
1013         nPages = 1 << order;
1014
1015         /* Client asked for way to much space.  This is checked later anyway */
1016         /* It is easier to debug here for the drivers than in the tce tables.*/
1017         if(order >= NUM_TCE_LEVELS) {
1018                 printk("PCI_DMA: pci_alloc_consistent size to large: 0x%lx \n",size);
1019                 return (void *)NO_TCE;
1020         }
1021
1022         tbl = get_tce_table(hwdev);
1023
1024         if ( tbl ) {
1025                 /* Alloc enough pages (and possibly more) */
1026                 ret = (void *)__get_free_pages( GFP_ATOMIC, order );
1027                 if ( ret ) {
1028                         /* Page allocation succeeded */
1029                         memset(ret, 0, nPages << PAGE_SHIFT);
1030                         /* Set up tces to cover the allocated range */
1031                         tce = get_tces( tbl, order, ret, nPages, PCI_DMA_BIDIRECTIONAL );
1032                         if ( tce == NO_TCE ) {
1033                                 PPCDBG(PPCDBG_TCE, "pci_alloc_consistent: get_tces failed\n" );
1034                                 free_pages( (unsigned long)ret, order );
1035                                 ret = NULL;
1036                         }
1037                         else
1038                         {
1039                                 *dma_handle = tce;
1040                         }
1041                 }
1042                 else PPCDBG(PPCDBG_TCE, "pci_alloc_consistent: __get_free_pages failed for order = %d\n", order);
1043         }
1044         else PPCDBG(PPCDBG_TCE, "pci_alloc_consistent: get_tce_table failed for 0x%016lx\n", hwdev);
1045
1046         PPCDBG(PPCDBG_TCE, "\tpci_alloc_consistent: dma_handle = 0x%16.16lx\n", *dma_handle);
1047         PPCDBG(PPCDBG_TCE, "\tpci_alloc_consistent: return     = 0x%16.16lx\n", ret);
1048         return ret;
1049 }
1050
1051 void pci_free_consistent(struct pci_dev *hwdev, size_t size,
1052                          void *vaddr, dma_addr_t dma_handle)
1053 {
1054         struct TceTable * tbl;
1055         unsigned order, nPages;
1056
1057         PPCDBG(PPCDBG_TCE, "pci_free_consistent:\n");
1058         PPCDBG(PPCDBG_TCE, "\thwdev = 0x%16.16lx, size = 0x%16.16lx, dma_handle = 0x%16.16lx, vaddr = 0x%16.16lx\n", hwdev, size, dma_h
    andle, vaddr);
1059
1060         size = PAGE_ALIGN(size);
1061         order = get_order(size);
1062         nPages = 1 << order;
1063
1064         /* Client asked for way to much space.  This is checked later anyway */
1065         /* It is easier to debug here for the drivers than in the tce tables.*/
1066         if(order >= NUM_TCE_LEVELS) {
1067                 printk("PCI_DMA: pci_free_consistent size to large: 0x%lx \n",size);
1068                 return;
1069         }
1070
1071         tbl = get_tce_table(hwdev);
1072
1073         if ( tbl ) {
1074                 tce_free(tbl, dma_handle, order, nPages);
1075                 free_pages( (unsigned long)vaddr, order );
```

```
1076            }
1077    }
1078
1079    /* Creates TCEs for a user provided buffer.  The user buffer must be
1080     * contiguous real kernel storage (not vmalloc).  The address of the buffer
1081     * passed here is the kernel (virtual) address of the buffer.  The buffer
1082     * need not be page aligned, the dma_addr_t returned will point to the same
1083     * byte within the page as vaddr.
1084     */
1085    dma_addr_t pci_map_single(struct pci_dev *hwdev, void *vaddr,
1086                              size_t size, int direction )
1087    {
1088            struct TceTable * tbl;
1089            dma_addr_t dma_handle = NO_TCE;
1090            unsigned long uaddr;
1091            unsigned order, nPages;
1092
1093            PPCDBG(PPCDBG_TCE, "pci_map_single:\n");
1094            PPCDBG(PPCDBG_TCE, "\thwdev = 0x%16.16lx, size = 0x%16.16lx, direction = 0x%16.16lx, vaddr = 0x%16.16lx\n", hwdev, size, directio
    n, vaddr);
1095            if ( direction == PCI_DMA_NONE )
1096                    BUG();
1097
1098            uaddr = (unsigned long)vaddr;
1099            nPages = PAGE_ALIGN( uaddr + size ) − ( uaddr & PAGE_MASK );
1100            order = get_order( nPages & PAGE_MASK );
1101            nPages >>= PAGE_SHIFT;
1102
1103            /* Client asked for way to much space.  This is checked later anyway */
1104            /* It is easier to debug here for the drivers than in the tce tables.*/
1105            if(order >= NUM_TCE_LEVELS) {
1106                    printk("PCI_DMA: pci_map_single size to large: 0x%lx \n",size);
1107                    return NO_TCE;
1108            }
1109
1110            tbl = get_tce_table(hwdev);
1111
1112            if ( tbl ) {
1113                    dma_handle = get_tces( tbl, order, vaddr, nPages, direction );
1114                    dma_handle |= ( uaddr & ~PAGE_MASK );
1115            }
1116
1117            return dma_handle;
1118    }
1119
1120    void pci_unmap_single( struct pci_dev *hwdev, dma_addr_t dma_handle, size_t size, int direction )
1121    {
1122            struct TceTable * tbl;
1123            unsigned order, nPages;
1124
1125            PPCDBG(PPCDBG_TCE, "pci_unmap_single:\n");
1126            PPCDBG(PPCDBG_TCE, "\thwdev = 0x%16.16lx, size = 0x%16.16lx, direction = 0x%16.16lx, dma_handle = 0x%16.16lx\n", hwdev, size, dir
    ection, dma_handle);
1127            if ( direction == PCI_DMA_NONE )
1128                    BUG();
1129
1130            nPages = PAGE_ALIGN( dma_handle + size ) − ( dma_handle & PAGE_MASK );
1131            order = get_order( nPages & PAGE_MASK );
1132            nPages >>= PAGE_SHIFT;
1133
1134            /* Client asked for way to much space.  This is checked later anyway */
1135            /* It is easier to debug here for the drivers than in the tce tables.*/
1136            if(order >= NUM_TCE_LEVELS) {
1137                    printk("PCI_DMA: pci_unmap_single size to large: 0x%lx \n",size);
1138                    return;
1139            }
1140
1141            tbl = get_tce_table(hwdev);
1142
1143            if ( tbl )
1144                    tce_free(tbl, dma_handle, order, nPages);
1145
1146    }
1147
1148    /* Figure out how many TCEs are actually going to be required
1149     * to map this scatterlist.  This code is not optimal.  It
1150     * takes into account the case where entry n ends in the same
1151     * page in which entry n+1 starts.  It does not handle the
1152     * general case of entry n ending in the same page in which
1153     * entry m starts.
1154     */
1155    static unsigned long num_tces_sg( struct scatterlist *sg, int nents )
1156    {
1157            unsigned long nTces, numPages, startPage, endPage, prevEndPage;
1158            unsigned i;
1159
1160            prevEndPage = 0;
1161            nTces = 0;
1162
1163            for (i=0; i<nents; ++i) {
```

```
1164                    /* Compute the starting page number and
1165                     * the ending page number for this entry
1166                     */
1167                    startPage = (unsigned long)sg->address >> PAGE_SHIFT;
1168                    endPage = ((unsigned long)sg->address + sg->length - 1) >> PAGE_SHIFT;
1169                    numPages = endPage - startPage + 1;
1170                    /* Simple optimization: if the previous entry ended
1171                     * in the same page in which this entry starts
1172                     * then we can reduce the required pages by one.
1173                     * This matches assumptions in fill_scatterlist_sg and
1174                     * create_tces_sg
1175                     */
1176                    if ( startPage == prevEndPage )
1177                            --numPages;
1178                    nTces += numPages;
1179                    prevEndPage = endPage;
1180                    sg++;
1181            }
1182            return nTces;
1183    }
1184
1185    /* Fill in the dma data in the scatterlist
1186     * return the number of dma sg entries created
1187     */
1188    static unsigned fill_scatterlist_sg( struct scatterlist *sg, int nents,
1189                                    dma_addr_t dma_addr , unsigned long numTces)
1190    {
1191            struct scatterlist *dma_sg;
1192            u32 cur_start_dma;
1193            unsigned long cur_len_dma, cur_end_virt, uaddr;
1194            unsigned num_dma_ents;
1195
1196            dma_sg = sg;
1197            num_dma_ents = 1;
1198
1199            /* Process the first sg entry */
1200            cur_start_dma = dma_addr + ((unsigned long)sg->address & (~PAGE_MASK));
1201            cur_len_dma = sg->length;
1202            /* cur_end_virt holds the address of the byte immediately after the
1203             * end of the current buffer.
1204             */
1205            cur_end_virt = (unsigned long)sg->address + cur_len_dma;
1206            /* Later code assumes that unused sg->dma_address and sg->dma_length
1207             * fields will be zero.  Other archs seem to assume that the user
1208             * (device driver) guarantees that...I don't want to depend on that
1209             */
1210            sg->dma_address = sg->dma_length = 0;
1211
1212            /* Process the rest of the sg entries */
1213            while (--nents) {
1214                    ++sg;
1215                    /* Clear possibly unused fields. Note: sg >= dma_sg so
1216                     * this can't be clearing a field we've already set
1217                     */
1218                    sg->dma_address = sg->dma_length = 0;
1219
1220                    /* Check if it is possible to make this next entry
1221                     * contiguous (in dma space) with the previous entry.
1222                     */
1223
1224                    /* The entries can be contiguous in dma space if
1225                     * the previous entry ends immediately before the
1226                     * start of the current entry (in virtual space)
1227                     * or if the previous entry ends at a page boundary
1228                     * and the current entry starts at a page boundary.
1229                     */
1230                    uaddr = (unsigned long)sg->address;
1231                    if ( ( uaddr != cur_end_virt ) &&
1232                         ( ( ( uaddr | cur_end_virt ) & (~PAGE_MASK) ) ||
1233                           ( ( uaddr & PAGE_MASK ) == ( ( cur_end_virt-1 ) & PAGE_MASK ) ) ) ) {
1234                            /* This entry can not be contiguous in dma space.
1235                             * save the previous dma entry and start a new one
1236                             */
1237                            dma_sg->dma_address = cur_start_dma;
1238                            dma_sg->dma_length  = cur_len_dma;
1239
1240                            ++dma_sg;
1241                            ++num_dma_ents;
1242
1243                            cur_start_dma += cur_len_dma-1;
1244                            /* If the previous entry ends and this entry starts
1245                             * in the same page then they share a tce.  In that
1246                             * case don't bump cur_start_dma to the next page
1247                             * in dma space.  This matches assumptions made in
1248                             * num_tces_sg and create_tces_sg.
1249                             */
1250                            if ((uaddr & PAGE_MASK) == ((cur_end_virt-1) & PAGE_MASK))
1251                                    cur_start_dma &= PAGE_MASK;
1252                            else
1253                                    cur_start_dma = PAGE_ALIGN(cur_start_dma+1);
```

```
1254                            cur_start_dma += ( uaddr & (~PAGE_MASK) );
1255                            cur_len_dma = 0;
1256                    }
1257                    /* Accumulate the length of this entry for the next
1258                     * dma entry
1259                     */
1260                    cur_len_dma += sg->length;
1261                    cur_end_virt = uaddr + sg->length;
1262            }
1263            /* Fill in the last dma entry */
1264            dma_sg->dma_address = cur_start_dma;
1265            dma_sg->dma_length  = cur_len_dma;
1266
1267            if ((((cur_start_dma +cur_len_dma - 1)>> PAGE_SHIFT) - (dma_addr >> PAGE_SHIFT) + 1) != numTces)
1268              {
1269                PPCDBG(PPCDBG_TCE, "fill_scatterlist_sg: numTces %ld, used tces %d\n",
1270                       numTces,
1271                       (unsigned)(((cur_start_dma + cur_len_dma - 1) >> PAGE_SHIFT) - (dma_addr >> PAGE_SHIFT) + 1));
1272              }
1273
1274
1275            return num_dma_ents;
1276    }
1277
1278    /* Call the hypervisor to create the TCE entries.
1279     * return the number of TCEs created
1280     */
1281    static dma_addr_t create_tces_sg( struct TceTable *tbl, struct scatterlist *sg,
1282                        int nents, unsigned numTces, int direction )
1283    {
1284            unsigned order, i, j;
1285            unsigned long startPage, endPage, prevEndPage, numPages, uaddr;
1286            long tcenum, starttcenum;
1287            dma_addr_t dmaAddr;
1288
1289            dmaAddr = NO_TCE;
1290
1291            order = get_order( numTces << PAGE_SHIFT );
1292            /* Client asked for way to much space.  This is checked later anyway */
1293            /* It is easier to debug here for the drivers than in the tce tables.*/
1294            if(order >= NUM_TCE_LEVELS) {
1295                    printk("PCI_DMA: create_tces_sg size to large: 0x%x \n",(numTces << PAGE_SHIFT));
1296                    return NO_TCE;
1297            }
1298
1299            /* allocate a block of tces */
1300            tcenum = alloc_tce_range( tbl, order );
1301            if ( tcenum != -1 ) {
1302                    tcenum += tbl->startOffset;
1303                    starttcenum = tcenum;
1304                    dmaAddr = tcenum << PAGE_SHIFT;
1305                    prevEndPage = 0;
1306                    for (j=0; j<nents; ++j) {
1307                            startPage = (unsigned long)sg->address >> PAGE_SHIFT;
1308                            endPage = ((unsigned long)sg->address + sg->length - 1) >> PAGE_SHIFT;
1309                            numPages = endPage - startPage + 1;
1310
1311                            uaddr = (unsigned long)sg->address;
1312
1313                            /* If the previous entry ended in the same page that
1314                             * the current page starts then they share that
1315                             * tce and we reduce the number of tces we need
1316                             * by one.  This matches assumptions made in
1317                             * num_tces_sg and fill_scatterlist_sg
1318                             */
1319                            if ( startPage == prevEndPage ) {
1320                                    --numPages;
1321                                    uaddr += PAGE_SIZE;
1322                            }
1323
1324                            for (i=0; i<numPages; ++i) {
1325                              ppc_md.tce_build(tbl, tcenum, uaddr, direction);
1326                              ++tcenum;
1327                              uaddr += PAGE_SIZE;
1328                            }
1329
1330                            prevEndPage = endPage;
1331                            sg++;
1332                    }
1333                    /* Make sure the update is visible to hardware.
1334                       sync required to synchronize the update to
1335                       the TCE table with the MMIO that will send
1336                       the bus address to the IOA */
1337                    __asm__ __volatile__ ("sync" : : : "memory");
1338
1339                    if ((tcenum - starttcenum) != numTces)
1340                            PPCDBG(PPCDBG_TCE, "create_tces_sg: numTces %d, tces used %d\n",
1341                                    numTces, (unsigned)(tcenum - starttcenum));
1342
1343            }
```

```
1344
1345            return dmaAddr;
1346    }
1347
1348    int pci_map_sg( struct pci_dev *hwdev, struct scatterlist *sg, int nents, int direction )
1349    {
1350            struct TceTable * tbl;
1351            unsigned numTces;
1352            int num_dma;
1353            dma_addr_t dma_handle;
1354
1355            PPCDBG(PPCDBG_TCE, "pci_map_sg:\n");
1356            PPCDBG(PPCDBG_TCE, "\thwdev = 0x%16.16lx, sg = 0x%16.16lx, direction = 0x%16.16lx, nents = 0x%16.16lx\n", hwdev, sg, direction,
        nents);
1357            /* Fast path for a single entry scatterlist */
1358            if ( nents == 1 ) {
1359                    sg->dma_address = pci_map_single( hwdev, sg->address,
1360                                                      sg->length, direction );
1361                    sg->dma_length = sg->length;
1362                    return 1;
1363            }
1364
1365            if ( direction == PCI_DMA_NONE )
1366                    BUG();
1367
1368            tbl = get_tce_table(hwdev);
1369
1370            if ( tbl ) {
1371                    /* Compute the number of tces required */
1372                    numTces = num_tces_sg( sg, nents );
1373                    /* Create the tces and get the dma address */
1374                    dma_handle = create_tces_sg( tbl, sg, nents, numTces, direction );
1375
1376                    /* Fill in the dma scatterlist */
1377                    num_dma = fill_scatterlist_sg( sg, nents, dma_handle, numTces );
1378            }
1379
1380            return num_dma;
1381    }
1382
1383    void pci_unmap_sg( struct pci_dev *hwdev, struct scatterlist *sg, int nelms, int direction )
1384    {
1385            struct TceTable * tbl;
1386            unsigned order, numTces, i;
1387            dma_addr_t dma_end_page, dma_start_page;
1388
1389            PPCDBG(PPCDBG_TCE, "pci_unmap_sg:\n");
1390            PPCDBG(PPCDBG_TCE, "\thwdev = 0x%16.16lx, sg = 0x%16.16lx, direction = 0x%16.16lx, nelms = 0x%16.16lx\n", hwdev, sg, direction,
         nelms);
1391
1392            if ( direction == PCI_DMA_NONE || nelms == 0 )
1393                    BUG();
1394
1395            dma_start_page = sg->dma_address & PAGE_MASK;
1396            dma_end_page   = 0;
1397            for ( i=nelms; i>0; --i ) {
1398                    unsigned k = i - 1;
1399                    if ( sg[k].dma_length ) {
1400                            dma_end_page = ( sg[k].dma_address +
1401                                             sg[k].dma_length - 1 ) & PAGE_MASK;
1402                            break;
1403                    }
1404            }
1405
1406            numTces = ((dma_end_page - dma_start_page ) >> PAGE_SHIFT) + 1;
1407            order = get_order( numTces << PAGE_SHIFT );
1408
1409            /* Client asked for way to much space.  This is checked later anyway */
1410            /* It is easier to debug here for the drivers than in the tce tables.*/
1411            if(order >= NUM_TCE_LEVELS) {
1412                    printk("PCI_DMA: dma_start_page:0x%lx  dma_end_page:0x%lx\n",dma_start_page,dma_end_page);
1413                    printk("PCI_DMA: pci_unmap_sg size to large: 0x%x \n",(numTces << PAGE_SHIFT));
1414                    return;
1415            }
1416
1417            tbl = get_tce_table(hwdev);
1418
1419            if ( tbl )
1420                    tce_free( tbl, dma_start_page, order, numTces );
1421
1422    }
1423
1424    /*
1425     * phb_tce_table_init
1426     *
1427     * Function: Display TCE config registers.  Could be easily changed
1428     *           to initialize the hardware to use TCEs.
1429     */
1430    unsigned long phb_tce_table_init(struct pci_controller *phb) {
1431            unsigned int r, cfg_rw, i;
```

```
1432          unsigned long r64;
1433          phandle node;
1434
1435          PPCDBG(PPCDBG_TCE, "phb_tce_table_init: start.\n");
1436
1437          node = ((struct device_node *)(phb->arch_data))->node;
1438
1439          PPCDBG(PPCDBG_TCEINIT, "\tphb      = 0x%lx\n", phb);
1440          PPCDBG(PPCDBG_TCEINIT, "\tphb->type    = 0x%lx\n", phb->type);
1441          PPCDBG(PPCDBG_TCEINIT, "\tphb->phb_regs  = 0x%lx\n", phb->phb_regs);
1442          PPCDBG(PPCDBG_TCEINIT, "\tphb->chip_regs = 0x%lx\n", phb->chip_regs);
1443          PPCDBG(PPCDBG_TCEINIT, "\tphb: node    = 0x%lx\n", node);
1444          PPCDBG(PPCDBG_TCEINIT, "\tphb->arch_data = 0x%lx\n", phb->arch_data);
1445
1446          i = 0;
1447          while(of_tce_table[i].node) {
1448                  if(of_tce_table[i].node == node) {
1449                          if(phb->type == phb_type_python) {
1450                                  r = *(((unsigned int *)phb->phb_regs) + (0xf10>>2));
1451                                  PPCDBG(PPCDBG_TCEINIT, "\tTAR(low)  = 0x%x\n", r);
1452                                  r = *(((unsigned int *)phb->phb_regs) + (0xf00>>2));
1453                                  PPCDBG(PPCDBG_TCEINIT, "\tTAR(high)  = 0x%x\n", r);
1454                                  r = *(((unsigned int *)phb->phb_regs) + (0xfd0>>2));
1455                                  PPCDBG(PPCDBG_TCEINIT, "\tPHB cfg(rw) = 0x%x\n", r);
1456                                  break;
1457                          } else if(phb->type == phb_type_speedwagon) {
1458                                  r64 = *(((unsigned long *)phb->chip_regs) +
1459                                          (0x800>>3));
1460                                  PPCDBG(PPCDBG_TCEINIT, "\tNCFG  = 0x%lx\n", r64);
1461                                  r64 = *(((unsigned long *)phb->chip_regs) +
1462                                          (0x580>>3));
1463                                  PPCDBG(PPCDBG_TCEINIT, "\tTAR0  = 0x%lx\n", r64);
1464                                  r64 = *(((unsigned long *)phb->chip_regs) +
1465                                          (0x588>>3));
1466                                  PPCDBG(PPCDBG_TCEINIT, "\tTAR1  = 0x%lx\n", r64);
1467                                  r64 = *(((unsigned long *)phb->chip_regs) +
1468                                          (0x590>>3));
1469                                  PPCDBG(PPCDBG_TCEINIT, "\tTAR2  = 0x%lx\n", r64);
1470                                  r64 = *(((unsigned long *)phb->chip_regs) +
1471                                          (0x598>>3));
1472                                  PPCDBG(PPCDBG_TCEINIT, "\tTAR3  = 0x%lx\n", r64);
1473                                  cfg_rw = *(((unsigned int *)phb->chip_regs) +
1474                                             ((0x160 +
1475                                               (((phb->local_number)+8)<<12))>>2));
1476                                  PPCDBG(PPCDBG_TCEINIT, "\tcfg_rw = 0x%x\n", cfg_rw);
1477                          }
1478                  }
1479                  i++;
1480          }
1481
1482          PPCDBG(PPCDBG_TCEINIT, "phb_tce_table_init: done\n");
1483
1484          return(0);
1485  }
1486
1487  /* These are called very early. */
1488  void tce_init_pSeries(void)
1489  {
1490          ppc_md.tce_build = tce_build_pSeries;
1491          ppc_md.tce_free_one = tce_free_one_pSeries;
1492  }
1493
1494  void tce_init_iSeries(void)
1495  {
1496          ppc_md.tce_build = tce_build_iSeries;
1497          ppc_md.tce_free_one = tce_free_one_iSeries;
1498  }
```

```
/*
 * pci_dn.c
 *
 * Copyright (C) 2001 Todd Inglett, IBM Corporation
 *
 * PCI manipulation via device_nodes.
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/pci.h>
#include <linux/delay.h>
#include <linux/string.h>
#include <linux/init.h>
#include <linux/bootmem.h>

#include <asm/io.h>
#include <asm/pgtable.h>
#include <asm/irq.h>
#include <asm/prom.h>
#include <asm/machdep.h>
#include <asm/init.h>
#include <asm/pci-bridge.h>
#include <asm/ppcdebug.h>
#include <asm/naca.h>
#include <asm/pci_dma.h>

#include "pci.h"

/* Traverse_func that inits the PCI fields of the device node.
 * NOTE: this *must* be done before read/write config to the device.
 */
static void * __init
update_dn_pci_info(struct device_node *dn, void *data)
{
        struct pci_controller *phb = (struct pci_controller *)data;
        u32 *regs;
        char *device_type = get_property(dn, "device_type", 0);
        char *status = get_property(dn, "status", 0);

        dn->phb = phb;
        if (device_type && strcmp(device_type, "pci") == 0 && get_property(dn, "class-code", 0) == 0) {
                /* special case for PHB's.  Sigh. */
                regs = (u32 *)get_property(dn, "bus-range", 0);
                dn->busno = regs[0];
                dn->devfn = 0;  /* assumption */
        } else {
                regs = (u32 *)get_property(dn, "reg", 0);
                if (regs) {
                        /* First register entry is addr (00BBSS00)  */
                        dn->busno = (regs[0] >> 16) & 0xff;
                        dn->devfn = (regs[0] >> 8) & 0xff;
                }
        }
        if (status && strcmp(status, "ok") != 0) {
                char *name = get_property(dn, "name", 0);
                printk(KERN_ERR "PCI: %04x:%02x.%x %s (%s) has bad status from firmware! (%s)", dn->busno, PCI_SLOT(dn->devfn),
PCI_FUNC(dn->devfn), name ? name : "<no name>", device_type ? device_type : "<unknown type>", status);
                dn->status = 1;
        }
        return NULL;
}

/*
 * Hit all the BARs of all the devices with values from OF.
 * This is unnecessary on most systems, but also harmless.
 */
static void * __init
write_OF_bars(struct device_node *dn, void *data)
{
        int i;
        u32 oldbar, newbar, newbartest;
        u8  config_offset;
        char *name = get_property(dn, "name", 0);
        char *device_type = get_property(dn, "device_type", 0);
        char devname[128];
```

```
90          sprintf(devname, "%04x:%02x.%x %s (%s)", dn->busno, PCI_SLOT(dn->devfn), PCI_FUNC(dn->devfn), name ? name :
    "<no name>", device_type ? device_type : "<unknown type>");
91
92          if (device_type && strcmp(device_type, "pci") == 0 &&
93              get_property(dn, "class-code", 0) == 0)
94                  return NULL;    /* This is probably a phb.  Skip it. */
95
96          if (dn->n_addrs == 0)
97                  return NULL;    /* This is normal for some adapters or bridges */
98
99          if (dn->addrs == NULL) {
100                 /* This shouldn't happen. */
101                 printk(KERN_WARNING "write_OF_bars %s: device has %d BARs, but no addrs recorded\n", devname, dn->n_addrs);
102                 return NULL;
103         }
104
105 #ifndef CONFIG_PPC_ISERIES
106         for (i = 0; i < dn->n_addrs; i++) {
107                 newbar = dn->addrs[i].address;
108                 config_offset = dn->addrs[i].space & 0xff;
109                 if (ppc_md.pcibios_read_config_dword(dn, config_offset, &oldbar) != PCIBIOS_SUCCESSFUL) {
110                         printk(KERN_WARNING "write_OF_bars %s: read BAR%d failed\n", devname, i);
111                         continue;
112                 }
113                 /* Need to update this BAR. */
114                 if (ppc_md.pcibios_write_config_dword(dn, config_offset, newbar) != PCIBIOS_SUCCESSFUL) {
115                         printk(KERN_WARNING "write_OF_bars %s: write BAR%d with 0x%08x failed (old was 0x%08x)\n", devname, i, ne
    wbar, oldbar);
116                         continue;
117                 }
118                 /* sanity check */
119                 if (ppc_md.pcibios_read_config_dword(dn, config_offset, &newbartest) != PCIBIOS_SUCCESSFUL) {
120                         printk(KERN_WARNING "write_OF_bars %s: sanity test read BAR%d failed?\n", devname, i);
121                         continue;
122                 }
123                 if ((newbar & PCI_BASE_ADDRESS_MEM_MASK) != (newbartest & PCI_BASE_ADDRESS_MEM_MASK)) {
124                         printk(KERN_WARNING "write_OF_bars %s: oops...BAR%d read back as 0x%08x%s!\n", devname, i, newbartest,
     (oldbar & PCI_BASE_ADDRESS_MEM_MASK) == (newbartest & PCI_BASE_ADDRESS_MEM_MASK) ? " (original value)" : "");
125                         continue;
126                 }
127         }
128 #endif
129         return NULL;
130 }
131
132 #if 0
133 /* Traverse_func that starts the BIST (self test) */
134 static void * __init
135 startBIST(struct device_node *dn, void *data)
136 {
137         struct pci_controller *phb = (struct pci_controller *)data;
138         u8 bist;
139
140         char *name = get_property(dn, "name", 0);
141         udbg_printf("startBIST: %s phb=%p, device=%p\n", name ? name : "<unknown>", phb, dn);
142
143         if (ppc_md.pcibios_read_config_byte(dn, PCI_BIST, &bist) == PCIBIOS_SUCCESSFUL) {
144                 if (bist & PCI_BIST_CAPABLE) {
145                         udbg_printf(" -> is BIST capable!\n", phb, dn);
146                         /* Start bist here */
147                 }
148         }
149         return NULL;
150 }
151 #endif
152
153
154 /*****************************************************************
155  * Traverse a device tree stopping each PCI device in the tree.
156  * This is done depth first.  As each node is processed, a "pre"
157  * function is called, the children are processed recursively, and
158  * then a "post" function is called.
159  *
160  * The "pre" and "post" funcs return a value.  If non-zero
161  * is returned from the "pre" func, the traversal stops and this
162  * value is returned.  The return value from "post" is not used.
163  * This return value is useful when using traverse as
164  * a method of finding a device.
165  *
166  * NOTE: we do not run the funcs for devices that do not appear to
167  * be PCI except for the start node which we assume (this is good
168  * because the start node is often a phb which may be missing PCI
169  * properties).
170  * We use the class-code as an indicator. If we run into
171  * one of these nodes we also assume its siblings are non-pci for
172  * performance.
173  *
174  *****************************************************************/
175 void *traverse_pci_devices(struct device_node *start, traverse_func pre, traverse_func post, void *data)
176 {
```

```
177         struct device_node *dn, *nextdn;
178         void *ret;
179
180         if (pre && (ret = pre(start, data)) != NULL)
181                 return ret;
182         for (dn = start->child; dn; dn = nextdn) {
183                 nextdn = NULL;
184                 if (get_property(dn, "class-code", 0)) {
185                         if (pre && (ret = pre(dn, data)) != NULL)
186                                 return ret;
187                         if (dn->child) {
188                                 /* Depth first...do children */
189                                 nextdn = dn->child;
190                         } else if (dn->sibling) {
191                                 /* ok, try next sibling instead. */
192                                 nextdn = dn->sibling;
193                         } else {
194                                 /* no more children or siblings...call "post" */
195                                 if (post)
196                                         post(dn, data);
197                         }
198                 }
199                 if (!nextdn) {
200                         /* Walk up to next valid sibling. */
201                         do {
202                                 dn = dn->parent;
203                                 if (dn == start)
204                                         return NULL;
205                         } while (dn->sibling == NULL);
206                         nextdn = dn->sibling;
207                 }
208         }
209         return NULL;
210 }
211
212 /* Same as traverse_pci_devices except this does it for all phbs.
213  */
214 void *traverse_all_pci_devices(traverse_func pre)
215 {
216         struct pci_controller* phb;
217         void *ret;
218         for (phb=hose_head;phb;phb=phb->next)
219                 if ((ret = traverse_pci_devices((struct device_node *)phb->arch_data, pre, NULL, phb)) != NULL)
220                         return ret;
221         return NULL;
222 }
223
224
225 /* Traversal func that looks for a <busno,devfcn> value.
226  * If found, the device_node is returned (thus terminating the traversal).
227  */
228 static void *
229 is_devfn_node(struct device_node *dn, void *data)
230 {
231         int busno = ((unsigned long)data >> 8) & 0xff;
232         int devfn = ((unsigned long)data) & 0xff;
233         return (devfn == dn->devfn && busno == dn->busno) ? dn : NULL;
234 }
235
236 /* Same as is_devfn_node except ignore the "fn" part of the "devfn".
237  */
238 static void *
239 is_devfn_sub_node(struct device_node *dn, void *data)
240 {
241         int busno = ((unsigned long)data >> 8) & 0xff;
242         int devfn = ((unsigned long)data) & 0xf8;
243         return (devfn == (dn->devfn & 0xf8) && busno == dn->busno) ? dn : NULL;
244 }
245
246 /* Given an existing EADs (pci bridge) device node create a fake one
247  * that will simulate function zero.  Make it a sibling of other_eads.
248  */
249 static struct device_node *
250 create_eads_node(struct device_node *other_eads)
251 {
252         struct device_node *eads = (struct device_node *)kmalloc(sizeof(struct device_node), GFP_KERNEL);
253
254         if (!eads) return NULL; /* huh? */
255         *eads = *other_eads;
256         eads->devfn &= ~7;      /* make it function zero */
257         eads->tce_table = NULL;
258         /*
259          * NOTE: share properties.  We could copy but for now this should
260          * suffice.  The full_name is also incorrect...but seems harmless.
261          */
262         eads->child = NULL;
263         eads->next = NULL;
264         other_eads->allnext = eads;
265         other_eads->sibling = eads;
266         return eads;
```

```
267 }
268
269 /* This is the "slow" path for looking up a device_node from a
270  * pci_dev.  It will hunt for the device under it's parent's
271  * phb and then update sysdata for a future fastpath.
272  *
273  * It may also do fixups on the actual device since this happens
274  * on the first read/write.
275  *
276  * Note that it also must deal with devices that don't exist.
277  * In this case it may probe for real hardware ("just in case")
278  * and add a device_node to the device tree if necessary.
279  *
280  */
281 struct device_node *fetch_dev_dn(struct pci_dev *dev)
282 {
283         struct device_node *orig_dn = (struct device_node *)dev->sysdata;
284         struct pci_controller *phb = orig_dn->phb; /* assume same phb as orig_dn */
285         struct device_node *phb_dn;
286         struct device_node *dn;
287         unsigned long searchval = (dev->bus->number << 8) | dev->devfn;
288
289         phb_dn = (struct device_node *)(phb->arch_data);
290         dn = (struct device_node *)traverse_pci_devices(phb_dn, is_devfn_node, NULL, (void *)searchval);
291         if (dn) {
292                 dev->sysdata = dn;
293                 /* ToDo: call some device init hook here */
294         } else {
295                 /* Now it is very possible that we can't find the device
296                  * because it is not the zero'th device of a mutifunction
297                  * device and we don't have permission to read the zero'th
298                  * device.  If this is the case, Linux would ordinarily skip
299                  * all the other functions.
300                  */
301                 if ((searchval & 0x7) == 0) {
302                         struct device_node *thisdevdn;
303                         /* Ok, we are looking for fn == 0.  Let's check for other functions. */
304                         thisdevdn = (struct device_node *)traverse_pci_devices(phb_dn, is_devfn_sub_node, NULL, (
    void *)searchval);
305                         if (thisdevdn) {
306                                 /* Ah ha!  There does exist a sub function.
307                                  * Now this isn't an exact match for
308                                  * searchval, but in order to get Linux to
309                                  * believe the sub functions exist we will
310                                  * need to manufacture a fake device_node for
311                                  * this zero'th function.  To keept this
312                                  * simple for now we only handle pci bridges
313                                  * and we just hand back the found node which
314                                  * isn't correct, but Linux won't care.
315                                  */
316                                 char *device_type = (char *)get_property(thisdevdn, "device_type", 0);
317                                 if (device_type && strcmp(device_type, "pci") == 0) {
318                                         return create_eads_node(thisdevdn);
319                                 }
320                         }
321                 }
322                 /* ToDo: device not found...probe for it anyway with a fake dn?
323                 struct device_node fake_dn;
324                 memset(&fake_dn, 0, sizeof(fake_dn));
325                 fake_dn.phb = phb;
326                 fake_dn.busno = dev->bus->number;
327                 fake_dn.devfn = dev->devfn;
328                 ... now do ppc_md.pcibios_read_config_dword(&fake_dn.....)
329                  ... if ok, alloc a real device_node and dn = real_dn;
330                  */
331         }
332         return dn;
333 }
334
335
336 /*****************************************************************
337  * Actually initialize the phbs.
338  * The buswalk on this phb has not happened yet.
339  *****************************************************************/
340 void __init
341 pci_devs_phb_init(void)
342 {
343         /* This must be done first so the device nodes have valid pci info! */
344         traverse_all_pci_devices(update_dn_pci_info);
345
346         /* Hack for regatta which does not init the bars correctly */
347         traverse_all_pci_devices(write_OF_bars);
348 #if 0
349         traverse_all_pci_devices(startBIST);
350         mdelay(5000);
351         traverse_all_pci_devices(checkBIST);
352 #endif
353 }
354
355
```

```
356 static void __init
357 pci_fixup_bus_sysdata_list(struct list_head *bus_list)
358 {
359         struct list_head *ln;
360         struct pci_bus *bus;
361         struct pci_controller *phb;
362         int newnum;
363
364         for (ln=bus_list->next; ln != bus_list; ln=ln->next) {
365                 bus = pci_bus_b(ln);
366                 if (bus->self) {
367                         bus->sysdata = bus->self->sysdata;
368                         /* Also fixup the bus number on large bus systems to
369                          * include the PHB# in the next byte
370                          */
371                         phb = PCI_GET_DN(bus)->phb;
372                         if (phb && phb->buid) {
373                                 newnum = (phb->global_number << 8) | bus->number;
374                                 bus->number = newnum;
375                                 sprintf(bus->name, "PCI Bus #%x", bus->number);
376                         }
377                 }
378                 pci_fixup_bus_sysdata_list(&bus->children);
379         }
380 }
381
382 /*****************************************************************
383  * Fixup the bus->sysdata ptrs to point to the bus' device_node.
384  * This is done late in pcibios_init().  We do this mostly for
385  * sanity, but pci_dma.c uses these at DMA time so they must be
386  * correct.
387  * To do this we recurse down the bus hierarchy.  Note that PHB's
388  * have bus->self == NULL, but fortunately bus->sysdata is already
389  * correct in this case.
390  *****************************************************************/
391 void __init
392 pci_fix_bus_sysdata(void)
393 {
394         pci_fixup_bus_sysdata_list(&pci_root_buses);
395 }
```

```
/*
 * pmc.c
 * Copyright (C) 2001 Dave Engebretsen & Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

/* Change Activity:
 * 2001/06/05 : engebret : Created.
 * 2002/04/11 : engebret : Add btmalloc code.
 * End Change Activity
 */

#include <asm/proc_fs.h>
#include <asm/paca.h>
#include <asm/iSeries/ItLpPaca.h>
#include <asm/iSeries/ItLpQueue.h>
#include <asm/processor.h>

#include <linux/proc_fs.h>
#include <linux/spinlock.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <asm/pmc.h>
#include <asm/uaccess.h>
#include <asm/naca.h>
#include <asm/pgalloc.h>
#include <asm/pgtable.h>
#include <asm/mmu_context.h>
#include <asm/page.h>
#include <asm/machdep.h>
#include <asm/lmb.h>
#include <asm/abs_addr.h>
#include <asm/ppcdebug.h>

struct _pmc_sw pmc_sw_system = {
        0
};

struct _pmc_sw pmc_sw_cpu[NR_CPUS] = {
        {0 },
};

/*
 * Provide enough storage for either system level counters or
 * one cpu's counters.
 */
struct _pmc_sw_text pmc_sw_text;
struct _pmc_hw_text pmc_hw_text;

extern pte_t *find_linux_pte( pgd_t * pgdir, unsigned long ea );
extern pgd_t *bolted_pgd;

static struct vm_struct *get_btm_area(unsigned long size, unsigned long flags);
static int local_free_bolted_pages(unsigned long ea, unsigned long num);

extern pgd_t bolted_dir[];
pgd_t *bolted_pgd  = (pgd_t *)&bolted_dir;

struct vm_struct *btmlist = NULL;
struct mm_struct btmalloc_mm = {pgd             : bolted_dir,
                                page_table_lock : SPIN_LOCK_UNLOCKED};

extern spinlock_t hash_table_lock;

char *
ppc64_pmc_stab(int file)
{
        int  n;
        unsigned long stab_faults, stab_capacity_castouts, stab_invalidations;
        unsigned long i;

        stab_faults = stab_capacity_castouts = stab_invalidations = n = 0;

        if (file == -1) {
                for (i = 0;  i < smp_num_cpus; i++) {
                        stab_faults += pmc_sw_cpu[i].stab_faults;
                        stab_capacity_castouts += pmc_sw_cpu[i].stab_capacity_castouts;
```

```
91                              stab_invalidations += pmc_sw_cpu[i].stab_invalidations;
92                      }
93                      n += sprintf(pmc_sw_text.buffer + n,
94                              "Faults     0x%lx\n", stab_faults);
95                      n += sprintf(pmc_sw_text.buffer + n,
96                              "Castouts    0x%lx\n", stab_capacity_castouts);
97                      n += sprintf(pmc_sw_text.buffer + n,
98                              "Invalidations 0x%lx\n", stab_invalidations);
99              } else {
100                     n += sprintf(pmc_sw_text.buffer + n,
101                             "Faults     0x%lx\n",
102                             pmc_sw_cpu[file].stab_faults);
103
104                     n += sprintf(pmc_sw_text.buffer + n,
105                             "Castouts    0x%lx\n",
106                             pmc_sw_cpu[file].stab_capacity_castouts);
107
108                     n += sprintf(pmc_sw_text.buffer + n,
109                             "Invalidations 0x%lx\n",
110                             pmc_sw_cpu[file].stab_invalidations);
111
112                     for (i = 0; i < STAB_ENTRY_MAX; i++) {
113                             if (pmc_sw_cpu[file].stab_entry_use[i]) {
114                                     n += sprintf(pmc_sw_text.buffer + n,
115                                                     "Entry %02ld    0x%lx\n", i,
116                                                     pmc_sw_cpu[file].stab_entry_use[i]);
117                             }
118                     }
119
120             }
121
122             return(pmc_sw_text.buffer);
123     }
124
125     char *
126     ppc64_pmc_htab(int file)
127     {
128             int  n;
129             unsigned long htab_primary_overflows, htab_capacity_castouts;
130             unsigned long htab_read_to_write_faults;
131
132             htab_primary_overflows = htab_capacity_castouts = 0;
133             htab_read_to_write_faults = n = 0;
134
135             if (file == -1) {
136                     n += sprintf(pmc_sw_text.buffer + n,
137                             "Primary Overflows 0x%lx\n",
138                             pmc_sw_system.htab_primary_overflows);
139                     n += sprintf(pmc_sw_text.buffer + n,
140                             "Castouts      0x%lx\n",
141                             pmc_sw_system.htab_capacity_castouts);
142             } else {
143                     n += sprintf(pmc_sw_text.buffer + n,
144                             "Primary Overflows N/A\n");
145
146                     n += sprintf(pmc_sw_text.buffer + n,
147                             "Castouts      N/A\n\n");
148
149             }
150
151             return(pmc_sw_text.buffer);
152     }
153
154     char *
155     ppc64_pmc_hw(int file)
156     {
157             int  n;
158
159             n = 0;
160             if (file == -1) {
161                     n += sprintf(pmc_hw_text.buffer + n, "Not Implemented\n");
162             } else {
163                     n += sprintf(pmc_hw_text.buffer + n,
164                             "MMCR0 0x%lx\n", mfspr(MMCR0));
165                     n += sprintf(pmc_hw_text.buffer + n,
166                             "MMCR1 0x%lx\n", mfspr(MMCR1));
167     #if 0
168                     n += sprintf(pmc_hw_text.buffer + n,
169                             "MMCRA 0x%lx\n", mfspr(MMCRA));
170     #endif
171
172                     n += sprintf(pmc_hw_text.buffer + n,
173                             "PMC1  0x%lx\n", mfspr(PMC1));
174                     n += sprintf(pmc_hw_text.buffer + n,
175                             "PMC2  0x%lx\n", mfspr(PMC2));
176                     n += sprintf(pmc_hw_text.buffer + n,
177                             "PMC3  0x%lx\n", mfspr(PMC3));
178                     n += sprintf(pmc_hw_text.buffer + n,
179                             "PMC4  0x%lx\n", mfspr(PMC4));
180                     n += sprintf(pmc_hw_text.buffer + n,
```

```
                                "PMC5  0x%lx\n", mfspr(PMC5));
                n += sprintf(pmc_hw_text.buffer + n,
                                "PMC6  0x%lx\n", mfspr(PMC6));
                n += sprintf(pmc_hw_text.buffer + n,
                                "PMC7  0x%lx\n", mfspr(PMC7));
                n += sprintf(pmc_hw_text.buffer + n,
                                "PMC8  0x%lx\n", mfspr(PMC8));
        }

        return(pmc_hw_text.buffer);
}

/*
 * Manage allocations of storage which is bolted in the HPT and low fault
 * overhead in the segment tables. Intended to be used for buffers used
 * to collect performance data.
 *
 * Remaining Issues:
 *   - Power4 is not tested at all, 0xB regions will always be castout of slb
 *   - On Power3, 0xB00000000 esid is left in the stab for all time,
 *     other 0xB segments are castout, but not explicitly removed.
 *   - Error path checking is weak at best, wrong at worst.
 *
 * btmalloc - Allocate a buffer which is bolted in the HPT and (eventually)
 *            the segment table.
 *
 * Input : unsigned long size: bytes of storage to allocate.
 * Return: void * : pointer to the kernal address of the buffer.
 */
void* btmalloc (unsigned long size) {
        pgd_t *pgdp;
        pmd_t *pmdp;
        pte_t *ptep, pte;
        unsigned long ea_base, ea, hpteflags;
        struct vm_struct *area;
        unsigned long pa, pg_count, page, vsid, slot, va, arpn, vpn;

        size = PAGE_ALIGN(size);
        if (!size || (size >> PAGE_SHIFT) > num_physpages) return NULL;

        spin_lock(&btmalloc_mm.page_table_lock);
        spin_lock(&hash_table_lock);

        /* Get a virtual address region in the bolted space */
        area = get_btm_area(size, 0);
        if (!area) {
                spin_unlock(&btmalloc_mm.page_table_lock);
                return NULL;
        }

        ea_base = (unsigned long) area->addr;
        pg_count = (size >> PAGE_SHIFT);

        /* Create a Linux page table entry and an HPTE for each page */
        for(page = 0; page < pg_count; page++) {
                pa = get_free_page(GFP_KERNEL) - PAGE_OFFSET;
                ea = ea_base + (page * PAGE_SIZE);
                vsid = get_kernel_vsid(ea);
                va = ( vsid << 28 ) | ( pa & 0xffffff );
                vpn = va >> PAGE_SHIFT;
                arpn = ((unsigned long)__v2a(ea)) >> PAGE_SHIFT;

                /* Get a pointer to the linux page table entry for this page
                 * allocating pmd or pte pages along the way as needed.  Note
                 * that the pmd & pte pages are not themselfs bolted.
                 */
                pgdp = pgd_offset_b(ea);
                pmdp = pmd_alloc(&btmalloc_mm, pgdp, ea);
                ptep = pte_alloc(&btmalloc_mm, pmdp, ea);
                pte = *ptep;

                /* Clear any old hpte and set the new linux pte */
                set_pte(ptep, mk_pte_phys(pa & PAGE_MASK, PAGE_KERNEL));

                hpteflags = _PAGE_ACCESSED|_PAGE_COHERENT|PP_RWXX;

                pte_val(pte) &= ~_PAGE_HPTEFLAGS;
                pte_val(pte) |= _PAGE_HASHPTE;

                slot = ppc_md.hpte_insert(vpn, arpn, hpteflags, 1, 0);

                pte_val(pte) |= ((slot<<12) &
                                 (_PAGE_GROUP_IX | _PAGE_SECONDARY));
        }

        spin_unlock(&hash_table_lock);
        spin_unlock(&btmalloc_mm.page_table_lock);
        return (void*)ea_base;
}
```

```
/*
 * Free a range of bolted pages that were allocated with btmalloc
 */
void btfree(void *ea) {
        struct vm_struct **p, *tmp;
        unsigned long size = 0;

        if ((!ea) || ((PAGE_SIZE−1) & (unsigned long)ea)) {
                printk(KERN_ERR "Trying to btfree() bad address (%p)\n", ea);
                return;
        }

        spin_lock(&btmalloc_mm.page_table_lock);

        /* Scan the bolted memory list for an entry matching
         * the address to be freed, get the size (in bytes)
         * and free the entry.  The list lock is not dropped
         * until the page table entries are removed.
         */
        for(p = &btmlist; (tmp = *p); p = &tmp−>next ) {
                if ( tmp−>addr == ea ) {
                        size = tmp−>size;
                        break;
                }
        }

        /* If no entry found, it is an error */
        if ( !size ) {
                printk(KERN_ERR "Trying to btfree() bad address (%p)\n", ea);
                spin_unlock(&btmalloc_mm.page_table_lock);
                return;
        }

        /* Free up the bolted pages and remove the page table entries */
        if(local_free_bolted_pages((unsigned long)ea, size >> PAGE_SHIFT)) {
                *p = tmp−>next;
                kfree(tmp);
        }

        spin_unlock(&btmalloc_mm.page_table_lock);
}

static int local_free_bolted_pages(unsigned long ea, unsigned long num) {
        int i;
        pte_t pte;

        for(i=0; i<num; i++) {
                pte_t *ptep = find_linux_pte(bolted_pgd, ea);
                if(!ptep) {
                        panic("free_bolted_pages – page being freed "
                              "(0x%lx) is not bolted", ea );
                }
                pte = *ptep;
                pte_clear(ptep);
                __free_pages(pte_page(pte), 0);
                flush_hash_page(0, ea, ptep);
                ea += PAGE_SIZE;
        }
        return 1;
}

/*
 * get_btm_area
 *
 * Get a virtual region in the bolted space
 */
static struct vm_struct *get_btm_area(unsigned long size,
                                      unsigned long flags) {
        unsigned long addr;
        struct vm_struct **p, *tmp, *area;

        area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
        if (!area) return NULL;

        addr = BTMALLOC_START;
        for (p = &btmlist; (tmp = *p) ; p = &tmp−>next) {
                if (size + addr < (unsigned long) tmp−>addr)
                        break;
                addr = tmp−>size + (unsigned long) tmp−>addr;
                if (addr + size > BTMALLOC_END) {
                        kfree(area);
                        return NULL;
                }
        }

        if (addr + size > BTMALLOC_END) {
                kfree(area);
                return NULL;
        }
        area−>flags = flags;
```

```
361             area->addr = (void *)addr;
362             area->size = size;
363             area->next = *p;
364             *p = area;
365             return area;
366     }
```

```
/************************************************************************/
/* File pcifr_proc.c created by Allan Trautman on Thu Aug  2 2001.      */
/************************************************************************/
/* Supports the ../proc/ppc64/pcifr for the pci flight recorder.        */
/* Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                    */
/*                                                                      */
/* 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 program is distributed in the hope that it will be useful,      */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of       */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        */
/* GNU General Public License for more details.                         */
/*                                                                      */
/* 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.,                                      */
/* 59 Temple Place, Suite 330,                                          */
/* Boston, MA  02111-1307  USA                                          */
/************************************************************************/
#include <stdarg.h>
#include <linux/kernel.h>

#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <asm/time.h>

#include <linux/pci.h>
#include <asm/pci-bridge.h>
#include <linux/netdevice.h>

#include <asm/flight_recorder.h>
#include <asm/iSeries/iSeries_pci.h>
#include "pci.h"

void pci_Fr_TestCode(void);

static spinlock_t proc_pcifr_lock;
struct flightRecorder* PciFr = NULL;

extern long Pci_Interrupt_Count;
extern long Pci_Event_Count;
extern long Pci_Io_Read_Count;
extern long Pci_Io_Write_Count;
extern long Pci_Cfg_Read_Count;
extern long Pci_Cfg_Write_Count;
extern long Pci_Error_Count;

/************************************************************************/
/* Forward declares.                                                    */
/************************************************************************/
static struct proc_dir_entry *pciFr_proc_root = NULL;
int proc_pciFr_read_proc(char *page, char **start, off_t off, int count, int *eof, void *data);
int proc_pciFr_write_proc(struct file *file, const char *buffer, unsigned long count, void *data);

static struct proc_dir_entry *pciDev_proc_root = NULL;
int proc_pciDev_read_proc(char *page, char **start, off_t off, int count, int *eof, void *data);
int proc_pciDev_write_proc(struct file *file, const char *buffer, unsigned long count, void *data);

/************************************************************************/
/* Create entry ../proc/ppc64/pcifr                                     */
/************************************************************************/
void proc_pciFr_init(struct proc_dir_entry *proc_ppc64_root)
{
        if (proc_ppc64_root == NULL) return;

        /* Read = User,Group,Other, Write User */
        printk("PCI: Creating ../proc/ppc64/pcifr \n");
        spin_lock(&proc_pcifr_lock);
        pciFr_proc_root = create_proc_entry("pcifr", S_IFREG | S_IRUGO | S_IWUSR, proc_ppc64_root);
        spin_unlock(&proc_pcifr_lock);

        if (pciFr_proc_root == NULL) return;

        pciFr_proc_root->nlink = 1;
        pciFr_proc_root->data = (void *)0;
        pciFr_proc_root->read_proc  = proc_pciFr_read_proc;
        pciFr_proc_root->write_proc = proc_pciFr_write_proc;

        PciFr = alloc_Flight_Recorder(NULL,"PciFr", 4096);

        printk("PCI: Creating ../proc/ppc64/pci \n");
        spin_lock(&proc_pcifr_lock);
        pciDev_proc_root = create_proc_entry("pci", S_IFREG | S_IRUGO | S_IWUSR, proc_ppc64_root);
        spin_unlock(&proc_pcifr_lock);

        if (pciDev_proc_root == NULL) return;

```

```
91          pciDev_proc_root->nlink = 1;
92          pciDev_proc_root->data = (void *)0;
93          pciDev_proc_root->read_proc  = proc_pciDev_read_proc;
94          pciDev_proc_root->write_proc = proc_pciDev_write_proc;
95  }
96
97  static  char* PciFrBuffer = NULL;
98  static  int   PciFrBufLen = 0;
99  static  char* PciFrBufPtr = NULL;
100 static  int   PciFileSize = 0;
101
102 /*****************************************************************************/
103 /* Read function for ../proc/ppc64/pcifr.                                    */
104 /*  -> Function grabs a copy of the pcifr(could change) and writes the data to */
105 /*     the caller.  Note, it may not all fit in the buffer.  The function     */
106 /*     handles the repeated calls until all the data has been read.           */
107 /* Tip:                                                                      */
108 /* ./fs/proc/generic.c::proc_file_read is the caller of this routine.        */
109 /*****************************************************************************/
110 int proc_pciFr_read_proc(char *page, char **start, off_t off, int count, int *eof, void *data)
111 {
112         /* First call will have offset 0, get snapshot the pcifr          */
113         if( off == 0) {
114                 spin_lock(&proc_pcifr_lock);
115                 PciFrBuffer = (char*)kmalloc(PciFr->Size, GFP_KERNEL);
116                 PciFrBufLen = fr_Dump(PciFr,PciFrBuffer, PciFr->Size);
117                 PciFrBufPtr = PciFrBuffer;
118                 PciFileSize = 0;
119         }
120         /* For the persistant folks, set eof and return zero length.      */
121         else if( PciFrBuffer == NULL) {
122                 *eof = 1;
123                 return 0;
124         }
125         /* - If there is more data than will fit, move what will fit.     */
126         /* - The rest will get moved on the next call.                    */
127         int MoveSize = PciFrBufLen;
128         if( MoveSize > count) MoveSize = count;
129
130         /* Move the data info the FileSystem buffer.                      */
131         memcpy(page+off,PciFrBufPtr,MoveSize);
132         PciFrBufPtr += MoveSize;
133         PciFileSize += MoveSize;
134         PciFrBufLen -= MoveSize;
135
136         /* If all the data has been moved, free the buffer and set EOF.   */
137         if( PciFrBufLen == 0) {
138                 kfree(PciFrBuffer);
139                 PciFrBuffer = NULL;
140                 spin_unlock(&proc_pcifr_lock);
141                 *eof = 1;
142         }
143         return PciFileSize;
144 }
145 /*****************************************************************************/
146 /* Gets called when client writes to ../proc/ppc64/pcifr                     */
147 /*****************************************************************************/
148 int proc_pciFr_write_proc(struct file *file, const char *buffer, unsigned long count, void *data)
149 {
150         return count;
151 }
152 static  spinlock_t ProcBufferLock;
153 static  char* ProcBuffer   = NULL;
154 static  int   ProcBufSize  = 0;
155 static  char* ProcBufPtr   = NULL;
156 static  int   ProcFileSize = 0;
157
158 /*****************************************************************************/
159 /* Build Device Buffer for /proc/ppc64/pci                                   */
160 /*****************************************************************************/
161 static int build_PciDev_Buffer(int BufferSize)
162 {
163         ProcBuffer  = (char*)kmalloc(BufferSize, GFP_KERNEL);
164         ProcBufPtr  = ProcBuffer;
165
166         int BufLen  = 0;
167
168         BufLen += sprintf(ProcBuffer+BufLen,"Pci I/O Reads. %8ld ",Pci_Io_Read_Count);
169         BufLen += sprintf(ProcBuffer+BufLen,"Pci I/O Writes %8ld\n",Pci_Io_Write_Count);
170
171         BufLen += sprintf(ProcBuffer+BufLen,"Pci Cfg Reads. %8ld ",Pci_Cfg_Read_Count);
172         BufLen += sprintf(ProcBuffer+BufLen,"Pci Cfg Writes %8ld\n",Pci_Cfg_Write_Count);
173
174         BufLen += sprintf(ProcBuffer+BufLen,"Pci I/O Errors %8ld\n",Pci_Error_Count);
175         BufLen += sprintf(ProcBuffer+BufLen,"\n");
176
177         /*************************************************************************/
178         /* List the devices                                                      */
179         /*************************************************************************/
180         struct pci_dev*    PciDev;                /* Device pointer             */
```

```
181         struct net_device* dev;                     /* net_device pointer          */
182         int    DeviceCount  = 0;
183         pci_for_each_dev(PciDev) {
184                 if ( BufLen > BufferSize−128) {    /* Room for another line?       */
185                         BufLen +=sprintf(ProcBuffer+BufLen,"Buffer Full\n");
186                         break;
187                 }
188                 if( PCI_SLOT(PciDev−>devfn) != 0) {
189                         ++DeviceCount;
190                         BufLen += sprintf(ProcBuffer+BufLen,"%3d. ",DeviceCount);
191                         if ( PciDev−>sysdata != NULL ) {
192                                 BufLen += format_device_location(PciDev,ProcBuffer+BufLen,128);
193                         }
194                         else {
195                                 BufLen += sprintf(ProcBuffer+BufLen,"No Device Node!\n");
196                         }
197                         BufLen += sprintf(ProcBuffer+BufLen,"\n");
198
199                         /* look for the net devices out */
200                         for (dev = dev_base; dev != NULL; dev = dev−>next)      {
201                                 if (dev−>base_addr == PciDev−>resource[0].start ) {
202                                         BufLen += sprintf(ProcBuffer+BufLen, "  − Net device: %s\n", dev−>name);
203                                         break;
204                                 } /* if */
205                         } /* for */
206                 } /* if(PCI_SLOT(PciDev−>devfn) != 0)  */
207         }
208         return BufLen;
209 }
210 /*****************************************************************************/
211 /* Get called when client reads the ../proc/ppc64/pcifr.                     */
212 /*****************************************************************************/
213 int proc_pciDev_read_proc(char *page, char **start, off_t off, int count, int *eof, void *data)
214 {
215         /* First call will have offset 0                                     */
216         if( off == 0) {
217                 spin_lock(&ProcBufferLock);
218                 ProcBufSize  = build_PciDev_Buffer(4096);
219                 ProcFileSize = 0;
220         }
221         /* For the persistant folks, set eof and return zero length.         */
222         else if( ProcBuffer == NULL) {
223                 *eof = 1;
224                 return 0;
225         }
226         /* How much data can be moved                                        */
227         int MoveSize = ProcBufSize;
228         if( MoveSize > count) MoveSize = count;
229
230         /* Move the data info the FileSystem buffer.                         */
231         memcpy(page+off,ProcBufPtr,MoveSize);
232         ProcBufPtr   += MoveSize;
233         ProcBufSize  −= MoveSize;
234         ProcFileSize += MoveSize;
235
236         /* If all the data has been moved, free the buffer and set EOF.      */
237         if( ProcBufSize == 0) {
238                 kfree(ProcBuffer );
239                 ProcBuffer  = NULL;
240                 spin_unlock(&ProcBufferLock);
241                 *eof = 1;
242         }
243         return ProcFileSize;
244 }
245 /*****************************************************************************/
246 /* Gets called when client writes to ../proc/ppc64/pcifr                     */
247 /*****************************************************************************/
248 int proc_pciDev_write_proc(struct file *file, const char *buffer, unsigned long count, void *data)
249 {
250         return count;
251 }
```

```
/*
 * pSeries_lpar.c
 * Copyright (C) 2001 Todd Inglett, IBM Corporation
 *
 * pSeries LPAR support.
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/processor.h>
#include <asm/semaphore.h>
#include <asm/mmu.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#include <asm/machdep.h>
#include <asm/abs_addr.h>
#include <asm/mmu_context.h>
#include <asm/ppcdebug.h>
#include <asm/pci_dma.h>
#include <linux/pci.h>
#include <asm/naca.h>
#include <asm/hvcall.h>

long plpar_tce_get(unsigned long liobn,
                   unsigned long ioba,
                   unsigned long *tce_ret)
{
        unsigned long dummy;
        return plpar_hcall(H_GET_TCE, liobn, ioba, 0, 0,
                           tce_ret, &dummy, &dummy);
}


long plpar_tce_put(unsigned long liobn,
                   unsigned long ioba,
                   unsigned long tceval)
{
        return plpar_hcall_norets(H_PUT_TCE, liobn, ioba, tceval);
}

long plpar_get_term_char(unsigned long termno,
                         unsigned long *len_ret,
                         char *buf_ret)
{
        unsigned long *lbuf = (unsigned long *)buf_ret;  /* ToDo: alignment? */
        return plpar_hcall(H_GET_TERM_CHAR, termno, 0, 0, 0,
                           len_ret, lbuf+0, lbuf+1);
}

long plpar_put_term_char(unsigned long termno,
                         unsigned long len,
                         const char *buffer)
{
        unsigned long dummy;
        unsigned long *lbuf = (unsigned long *)buffer;  /* ToDo: alignment? */
        return plpar_hcall(H_PUT_TERM_CHAR, termno, len,
                           lbuf[0], lbuf[1], &dummy, &dummy, &dummy);
}

long plpar_eoi(unsigned long xirr)
{
        return plpar_hcall_norets(H_EOI, xirr);
}

long plpar_cppr(unsigned long cppr)
{
        return plpar_hcall_norets(H_CPPR, cppr);
}

long plpar_ipi(unsigned long servernum,
               unsigned long mfrr)
{
        return plpar_hcall_norets(H_IPI, servernum, mfrr);
}

```

```
91  long plpar_xirr(unsigned long *xirr_ret)
92  {
93          unsigned long dummy;
94          return plpar_hcall(H_XIRR, 0, 0, 0, 0,
95                             xirr_ret, &dummy, &dummy);
96  }
97
98  long plpar_ipoll(unsigned long servernum, unsigned long* xirr_ret, unsigned long* mfrr_ret)
99  {
100         unsigned long dummy;
101         return plpar_hcall(H_IPOLL, servernum, 0, 0, 0,
102                            xirr_ret, mfrr_ret, &dummy);
103 }
104
105
106 static void tce_build_pSeriesLP(struct TceTable *tbl, long tcenum,
107                                 unsigned long uaddr, int direction )
108 {
109         u64 set_tce_rc;
110         union Tce tce;
111
112         PPCDBG(PPCDBG_TCE, "build_tce: uaddr = 0x%lx\n", uaddr);
113         PPCDBG(PPCDBG_TCE, "\ttcenum = 0x%lx, tbl = 0x%lx, index=%lx\n",
114                tcenum, tbl, tbl->index);
115
116         tce.wholeTce = 0;
117         tce.tceBits.rpn = (virt_to_absolute(uaddr)) >> PAGE_SHIFT;
118
119         tce.tceBits.readWrite = 1;
120         if ( direction != PCI_DMA_TODEVICE ) tce.tceBits.pciWrite = 1;
121
122         set_tce_rc = plpar_tce_put((u64)tbl->index,
123                                    (u64)tcenum << 12,
124                                    tce.wholeTce );
125
126         if(set_tce_rc) {
127                 printk("tce_build_pSeriesLP: plpar_tce_put failed. rc=%ld\n", set_tce_rc);
128                 printk("\tindex   = 0x%lx\n", (u64)tbl->index);
129                 printk("\ttcenum  = 0x%lx\n", (u64)tcenum);
130                 printk("\ttce val = 0x%lx\n", tce.wholeTce );
131         }
132 }
133
134 static void tce_free_one_pSeriesLP(struct TceTable *tbl, long tcenum)
135 {
136         u64 set_tce_rc;
137         union Tce tce;
138
139         tce.wholeTce = 0;
140         set_tce_rc = plpar_tce_put((u64)tbl->index,
141                                    (u64)tcenum << 12,
142                                    tce.wholeTce );
143         if ( set_tce_rc ) {
144                 printk("tce_free_one_pSeriesLP: plpar_tce_put failed\n");
145                 printk("\trc     = %ld\n", set_tce_rc);
146                 printk("\tindex  = 0x%lx\n", (u64)tbl->index);
147                 printk("\ttcenum = 0x%lx\n", (u64)tcenum);
148                 printk("\ttce val = 0x%lx\n", tce.wholeTce );
149         }
150
151 }
152
153 /* PowerPC Interrupts for lpar. */
154 /* NOTE: this typedef is duplicated (for now) from xics.c! */
155 typedef struct {
156         int (*xirr_info_get)(int cpu);
157         void (*xirr_info_set)(int cpu, int val);
158         void (*cppr_info)(int cpu, u8 val);
159         void (*qirr_info)(int cpu, u8 val);
160 } xics_ops;
161 static int pSeriesLP_xirr_info_get(int n_cpu)
162 {
163         unsigned long lpar_rc;
164         unsigned long return_value;
165
166         lpar_rc = plpar_xirr(&return_value);
167         if (lpar_rc != H_Success) {
168                 panic(" bad return code xirr – rc = %lx \n", lpar_rc);
169         }
170         return ((int)(return_value));
171 }
172
173 static void pSeriesLP_xirr_info_set(int n_cpu, int value)
174 {
175         unsigned long lpar_rc;
176         unsigned long val64 = value & 0xffffffff;
177
178         lpar_rc = plpar_eoi(val64);
179         if (lpar_rc != H_Success) {
180                 panic(" bad return code EOI – rc = %ld, value=%lx \n", lpar_rc, val64);
```

```
181         }
182 }
183
184 static void pSeriesLP_cppr_info(int n_cpu, u8 value)
185 {
186         unsigned long lpar_rc;
187
188         lpar_rc = plpar_cppr(value);
189         if (lpar_rc != H_Success) {
190                 panic(" bad return code cppr – rc = %lx \n", lpar_rc);
191         }
192 }
193
194 static void pSeriesLP_qirr_info(int n_cpu , u8 value)
195 {
196         unsigned long lpar_rc;
197
198         lpar_rc = plpar_ipi(get_hard_smp_processor_id(n_cpu),value);
199         if (lpar_rc != H_Success) {
200     udbg_printf("pSeriesLP_qirr_info – plpar_ipi failed!!!!!!!! \n");
201                 panic(" bad return code qirr –ipi  – rc = %lx \n", lpar_rc);
202         }
203 }
204
205 xics_ops pSeriesLP_ops = {
206         pSeriesLP_xirr_info_get,
207         pSeriesLP_xirr_info_set,
208         pSeriesLP_cppr_info,
209         pSeriesLP_qirr_info
210 };
211 /* end TAI−LPAR */
212
213
214 int vtermno;    /* virtual terminal# for udbg  */
215
216 static void udbg_putcLP(unsigned char c)
217 {
218         char buf[16];
219         unsigned long rc;
220
221         if (c == '\n')
222                 udbg_putcLP('\r');
223
224         buf[0] = c;
225         do {
226                 rc = plpar_put_term_char(vtermno, 1, buf);
227         } while(rc == H_Busy);
228 }
229
230 /* Buffered chars getc */
231 static long inbuflen;
232 static long inbuf[2];   /* must be 2 longs */
233
234 static int udbg_getc_pollLP(void)
235 {
236         /* The interface is tricky because it may return up to 16 chars.
237          * We save them statically for future calls to udbg_getc().
238          */
239         char ch, *buf = (char *)inbuf;
240         int i;
241         long rc;
242         if (inbuflen == 0) {
243                 /* get some more chars. */
244                 inbuflen = 0;
245                 rc = plpar_get_term_char(vtermno, &inbuflen, buf);
246                 if (rc != H_Success)
247                         inbuflen = 0;   /* otherwise inbuflen is garbage */
248         }
249         if (inbuflen <= 0 || inbuflen > 16) {
250                 /* Catch error case as well as other oddities (corruption) */
251                 inbuflen = 0;
252                 return −1;
253         }
254         ch = buf[0];
255         for (i = 1; i < inbuflen; i++)  /* shuffle them down. */
256                 buf[i−1] = buf[i];
257         inbuflen−−;
258         return ch;
259 }
260
261 static unsigned char udbg_getcLP(void)
262 {
263         int ch;
264         for (;;) {
265                 ch = udbg_getc_pollLP();
266                 if (ch == −1) {
267                         /* This shouldn't be needed...but... */
268                         volatile unsigned long delay;
269                         for (delay=0; delay < 2000000; delay++)
270                                 ;
```

```
271                     } else {
272                             return ch;
273                     }
274             }
275     }
276
277
278
279     /* Code for hvc_console.  Should move it back eventually. */
280
281     int hvc_get_chars(int index, char *buf, int count)
282     {
283             unsigned long got;
284
285             if (plpar_hcall(H_GET_TERM_CHAR, index, 0, 0, 0, &got,
286                     (unsigned long *)buf, (unsigned long *)buf+1) == H_Success) {
287                     /*
288                      * Work around a HV bug where it gives us a null
289                      * after every \r.  -- paulus
290                      */
291                     if (got > 0) {
292                             int i;
293                             for (i = 1; i < got; ++i) {
294                                     if (buf[i] == 0 && buf[i−1] == '\r') {
295                                             --got;
296                                             if (i < got)
297                                                     memmove(&buf[i], &buf[i+1],
298                                                             got − i);
299                                     }
300                             }
301                     }
302                     return got;
303             }
304             return 0;
305     }
306
307     int hvc_put_chars(int index, const char *buf, int count)
308     {
309             unsigned long dummy;
310             unsigned long *lbuf = (unsigned long *) buf;
311             long ret;
312
313             ret = plpar_hcall(H_PUT_TERM_CHAR, index, count, lbuf[0], lbuf[1],
314                               &dummy, &dummy, &dummy);
315             if (ret == H_Success)
316                     return count;
317             if (ret == H_Busy)
318                     return 0;
319             return −1;
320     }
321
322     int hvc_count(int *start_termno)
323     {
324             u32 *termno;
325             struct device_node *dn;
326
327             if ((dn = find_path_device("/rtas")) != NULL) {
328                     if ((termno = (u32 *)get_property(dn, "ibm,termno", 0)) != NULL) {
329                             if (start_termno)
330                                     *start_termno = termno[0];
331                             return termno[1];
332                     }
333             }
334             return 0;
335     }
336
337     #ifndef CONFIG_PPC_ISERIES
338     void pSeries_lpar_mm_init(void);
339
340     /* This is called early in setup.c.
341      * Use it to setup page table ppc_md stuff as well as udbg.
342      */
343     void pSeriesLP_init_early(void)
344     {
345             pSeries_lpar_mm_init();
346
347             ppc_md.tce_build         = tce_build_pSeriesLP;
348             ppc_md.tce_free_one      = tce_free_one_pSeriesLP;
349
350     #ifdef CONFIG_SMP
351             smp_init_pSeries();
352     #endif
353             pSeries_pcibios_init_early();
354
355             /* The keyboard is not useful in the LPAR environment.
356              * Leave all the interfaces NULL.
357              */
358
359             /* lookup the first virtual terminal number in case we don't have a com port.
360              * Zero is probably correct in case someone calls udbg before the init.
```

```
361             * The property is a pair of numbers.  The first is the starting termno (the
362             * one we use) and the second is the number of terminals.
363             */
364            u32 *termno;
365            struct device_node *np = find_path_device("/rtas");
366            if (np) {
367                    termno = (u32 *)get_property(np, "ibm,termno", 0);
368                    if (termno)
369                            vtermno = termno[0];
370            }
371            ppc_md.udbg_putc = udbg_putcLP;
372            ppc_md.udbg_getc = udbg_getcLP;
373            ppc_md.udbg_getc_poll = udbg_getc_pollLP;
374    }
375    #endif
```

```
1
2   /*
3    * ras.c
4    * Copyright (C) 2001 Dave Engebretsen IBM Corporation
5    *
6    * This program is free software; you can redistribute it and/or modify
7    * it under the terms of the GNU General Public License as published by
8    * the Free Software Foundation; either version 2 of the License, or
9    * (at your option) any later version.
10   *
11   * This program is distributed in the hope that it will be useful,
12   * but WITHOUT ANY WARRANTY; without even the implied warranty of
13   * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14   * GNU General Public License for more details.
15   *
16   * You should have received a copy of the GNU General Public License
17   * along with this program; if not, write to the Free Software
18   * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111−1307 USA
19   */
20
21  /* Change Activity:
22   * 2001/09/21 : engebret : Created with minimal EPOW and HW exception support.
23   * End Change Activity
24   */
25
26  #include <linux/ptrace.h>
27  #include <linux/errno.h>
28  #include <linux/threads.h>
29  #include <linux/kernel_stat.h>
30  #include <linux/signal.h>
31  #include <linux/sched.h>
32  #include <linux/ioport.h>
33  #include <linux/interrupt.h>
34  #include <linux/timex.h>
35  #include <linux/init.h>
36  #include <linux/slab.h>
37  #include <linux/pci.h>
38  #include <linux/delay.h>
39  #include <linux/irq.h>
40  #include <linux/proc_fs.h>
41  #include <linux/random.h>
42  #include <linux/sysrq.h>
43
44  #include <asm/uaccess.h>
45  #include <asm/bitops.h>
46  #include <asm/system.h>
47  #include <asm/io.h>
48  #include <asm/pgtable.h>
49  #include <asm/irq.h>
50  #include <asm/cache.h>
51  #include <asm/prom.h>
52  #include <asm/ptrace.h>
53  #include <asm/iSeries/LparData.h>
54  #include <asm/machdep.h>
55  #include <asm/rtas.h>
56  #include <asm/ppcdebug.h>
57
58  static void ras_epow_interrupt(int irq, void *dev_id, struct pt_regs * regs);
59  static void ras_error_interrupt(int irq, void *dev_id, struct pt_regs * regs);
60  void init_ras_IRQ(void);
61
62  /* #define DEBUG */
63
64  /*
65   * Initialize handlers for the set of interrupts caused by hardware errors
66   * and power system events.
67   */
68  void init_ras_IRQ(void) {
69          struct device_node *np;
70          unsigned int *ireg, len, i;
71
72          if((np = find_path_device("/event−sources/internal−errors")) &&
73             (ireg = (unsigned int *)get_property(np, "open−pic−interrupt",
74                                                  &len))) {
75                  for(i=0; i<(len / sizeof(*ireg)); i++) {
76                          request_irq(virt_irq_create_mapping(*(ireg)) + NUM_8259_INTERRUPTS,
77                                      &ras_error_interrupt, 0,
78                                      "RAS_ERROR", NULL);
79                          ireg++;
80                  }
81          }
82
83          if((np = find_path_device("/event−sources/epow−events")) &&
84             (ireg = (unsigned int *)get_property(np, "open−pic−interrupt",
85                                                  &len))) {
86                  for(i=0; i<(len / sizeof(*ireg)); i++) {
87                          request_irq(virt_irq_create_mapping(*(ireg)) + NUM_8259_INTERRUPTS,
88                                      &ras_epow_interrupt, 0,
89                                      "RAS_EPOW", NULL);
90                          ireg++;
```

```
                }
        }
}

/*
 * Handle power subsystem events (EPOW).
 *
 * Presently we just log the event has occured.  This should be fixed
 * to examine the type of power failure and take appropriate action where
 * the time horizon permits something useful to be done.
 */
static void
ras_epow_interrupt(int irq, void *dev_id, struct pt_regs * regs)
{
        struct rtas_error_log log_entry;
        unsigned int size = sizeof(log_entry);
        long status = 0xdeadbeef;

        status = rtas_call(rtas_token("check-exception"), 6, 1, NULL,
                           0x500, irq,
                           EPOW_WARNING | POWERMGM_EVENTS,
                           1,  /* Time Critical */
                           __pa(&log_entry), size);

        udbg_printf("EPOW <0x%lx 0x%lx>\n",
                    *((unsigned long *)&log_entry), status);
        printk(KERN_WARNING
               "EPOW <0x%lx 0x%lx>\n",*((unsigned long *)&log_entry), status);
}

/*
 * Handle hardware error interrupts.
 *
 * RTAS check-exception is called to collect data on the exception.  If
 * the error is deemed recoverable, we log a warning and return.
 * For nonrecoverable errors, an error is logged and we stop all processing
 * as quickly as possible in order to prevent propagation of the failure.
 */
static void
ras_error_interrupt(int irq, void *dev_id, struct pt_regs * regs)
{
        struct rtas_error_log log_entry;
        unsigned int size = sizeof(log_entry);
        long status = 0xdeadbeef;

        status = rtas_call(rtas_token("check-exception"), 6, 1, NULL,
                           0x500, irq,
                           INTERNAL_ERROR,
                           1, /* Time Critical */
                           __pa(&log_entry), size);

        if((status != 1) &&
           (log_entry.severity >= SEVERITY_ERROR_SYNC)) {
                udbg_printf("HW Error <0x%lx 0x%lx>\n",
                            *((unsigned long *)&log_entry), status);
                printk(KERN_EMERG
                       "Error: Fatal hardware error <0x%lx 0x%lx>\n",
                       *((unsigned long *)&log_entry), status);

#ifndef DEBUG
                /* Don't actually power off when debugging so we can test
                 * without actually failing while injecting errors.
                 */
                ppc_md.power_off();
#endif
        } else {
                udbg_printf("Recoverable HW Error <0x%lx 0x%lx>\n",
                            *((unsigned long *)&log_entry), status);
                printk(KERN_WARNING
                       "Warning: Recoverable hardware error <0x%lx 0x%lx>\n",
                       *((unsigned long *)&log_entry), status);

                return;
        }
}
```

```
/*
 *  c 2001 PPC 64 Team, IBM Corp
 *
 *      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.
 *
 * /proc/ppc64/rtas/firmware_flash interface
 *
 * This file implements a firmware_flash interface to pump a firmware
 * image into the kernel.  At reboot time rtas_restart() will see the
 * firmware image and flash it as it reboots (see rtas.c).
 */

#include <linux/module.h>

#include <linux/config.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <asm/rtas.h>

#define MODULE_VERSION "1.0"
#define MODULE_NAME "rtas_flash"

#define FIRMWARE_FLASH_NAME "firmware_flash"

/* Local copy of the flash block list.
 * We only allow one open of the flash proc file and create this
 * list as we go.  This list will be put in the kernel's
 * rtas_firmware_flash_list global var once it is fully read.
 *
 * For convenience as we build the list we use virtual addrs,
 * we do not fill in the version number, and the length field
 * is treated as the number of entries currently in the block
 * (i.e. not a byte count).  This is all fixed on release.
 */
static struct flash_block_list *flist;
static char *flash_msg;
static int flash_possible;

static int rtas_flash_open(struct inode *inode, struct file *file)
{
        if ((file->f_mode & FMODE_WRITE) && flash_possible) {
                if (flist)
                        return -EBUSY;
                flist = (struct flash_block_list *)get_free_page(GFP_KERNEL);
                if (!flist)
                        return -ENOMEM;
        }
        return 0;
}

/* Do simple sanity checks on the flash image. */
static int flash_list_valid(struct flash_block_list *flist)
{
        struct flash_block_list *f;
        int i;
        unsigned long block_size, image_size;

        flash_msg = NULL;
        /* Paranoid self test here.  We also collect the image size. */
        image_size = 0;
        for (f = flist; f; f = f->next) {
                for (i = 0; i < f->num_blocks; i++) {
                        if (f->blocks[i].data == NULL) {
                                flash_msg = "error: internal error null data\n";
                                return 0;
                        }
                        block_size = f->blocks[i].length;
                        if (block_size <= 0 || block_size > PAGE_SIZE) {
                                flash_msg = "error: internal error bad length\n";
                                return 0;
                        }
                        image_size += block_size;
                }
        }
        if (image_size < (256 << 10)) {
                if (image_size < 2)
                        flash_msg = NULL;       /* allow "clear" of image */
                else
                        flash_msg = "error: flash image short\n";
                return 0;
        }
        printk(KERN_INFO "FLASH: flash image with %ld bytes stored for hardware flash on reboot\n", image_size);
        return 1;
}

static void free_flash_list(struct flash_block_list *f)
```

```
91  {
92          struct flash_block_list *next;
93          int i;
94
95          while (f) {
96                  for (i = 0; i < f->num_blocks; i++)
97                          free_page((unsigned long)(f->blocks[i].data));
98                  next = f->next;
99                  free_page((unsigned long)f);
100                 f = next;
101         }
102 }
103
104 static int rtas_flash_release(struct inode *inode, struct file *file)
105 {
106         if (flist) {
107                 /* Always clear saved list on a new attempt. */
108                 if (rtas_firmware_flash_list.next) {
109                         free_flash_list(rtas_firmware_flash_list.next);
110                         rtas_firmware_flash_list.next = NULL;
111                 }
112
113                 if (flash_list_valid(flist))
114                         rtas_firmware_flash_list.next = flist;
115                 else
116                         free_flash_list(flist);
117                 flist = NULL;
118         }
119         return 0;
120 }
121
122 /* Reading the proc file will show status (not the firmware contents) */
123 static ssize_t rtas_flash_read(struct file *file, char *buf,
124                                size_t count, loff_t *ppos)
125 {
126         int error;
127         char *msg;
128         int msglen;
129
130         if (!flash_possible) {
131                 msg = "error: this partition does not have service authority\n";
132         } else if (flist) {
133                 msg = "info: this file is busy for write by some process\n";
134         } else if (flash_msg) {
135                 msg = flash_msg;        /* message from last flash attempt */
136         } else if (rtas_firmware_flash_list.next) {
137                 msg = "ready: firmware image ready for flash on reboot\n";
138         } else {
139                 msg = "info: no firmware image for flash\n";
140         }
141         msglen = strlen(msg);
142         if (msglen > count)
143                 msglen = count;
144
145         if (ppos && *ppos != 0)
146                 return 0;       /* be cheap */
147
148         error = verify_area(VERIFY_WRITE, buf, msglen);
149         if (error)
150                 return -EINVAL;
151
152         copy_to_user(buf, msg, msglen);
153
154         if (ppos)
155                 *ppos = msglen;
156         return msglen;
157 }
158
159 /* We could be much more efficient here.  But to keep this function
160  * simple we allocate a page to the block list no matter how small the
161  * count is.  If the system is low on memory it will be just as well
162  * that we fail....
163  */
164 static ssize_t rtas_flash_write(struct file *file, const char *buffer,
165                                 size_t count, loff_t *off)
166 {
167         size_t len = count;
168         char *p;
169         int next_free;
170         struct flash_block_list *fl = flist;
171
172         if (!flash_possible || len == 0)
173                 return len;     /* discard data */
174
175         while (fl->next)
176                 fl = fl->next; /* seek to last block_list for append */
177         next_free = fl->num_blocks;
178         if (next_free == FLASH_BLOCKS_PER_NODE) {
179                 /* Need to allocate another block_list */
180                 fl->next = (struct flash_block_list *)get_free_page(GFP_KERNEL);
```

```
181                     if (!fl->next)
182                             return -ENOMEM;
183                     fl = fl->next;
184                     next_free = 0;
185             }
186
187             if (len > PAGE_SIZE)
188                     len = PAGE_SIZE;
189             p = (char *)get_free_page(GFP_KERNEL);
190             if (!p)
191                     return -ENOMEM;
192             if(copy_from_user(p, buffer, len)) {
193                     free_page((unsigned long)p);
194                     return -EFAULT;
195             }
196             fl->blocks[next_free].data = p;
197             fl->blocks[next_free].length = len;
198             fl->num_blocks++;
199
200             return len;
201     }
202
203     static struct file_operations rtas_flash_operations = {
204             read:           rtas_flash_read,
205             write:          rtas_flash_write,
206             open:           rtas_flash_open,
207             release:        rtas_flash_release,
208     };
209
210
211     int __init rtas_flash_init(void)
212     {
213             struct proc_dir_entry *ent = NULL;
214
215             if (!rtas_proc_dir) {
216                     printk(KERN_WARNING "rtas proc dir does not already exist");
217                     return -ENOENT;
218             }
219
220             if (rtas_token("ibm,update-flash-64-and-reboot") != RTAS_UNKNOWN_SERVICE)
221                     flash_possible = 1;
222
223             if ((ent = create_proc_entry(FIRMWARE_FLASH_NAME, S_IRUSR | S_IWUSR, rtas_proc_dir)) != NULL) {
224                     ent->nlink = 1;
225                     ent->proc_fops = &rtas_flash_operations;
226                     ent->owner = THIS_MODULE;
227             }
228             return 0;
229     }
230
231     void __exit rtas_flash_cleanup(void)
232     {
233             if (!rtas_proc_dir)
234                     return;
235             remove_proc_entry(FIRMWARE_FLASH_NAME, rtas_proc_dir);
236     }
237
238     module_init(rtas_flash_init);
239     module_exit(rtas_flash_cleanup);
240     MODULE_LICENSE("GPL");
```

```
/*
 *  c 2001 PPC 64 Team, IBM Corp
 *
 *      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.
 *
 * scan−log−data driver for PPC64  Todd Inglett <tinglett@vnet.ibm.com>
 *
 * When ppc64 hardware fails the service processor dumps internal state
 * of the system.  After a reboot the operating system can access a dump
 * of this data using this driver.  A dump exists if the device−tree
 * /chosen/ibm,scan−log−data property exists.
 *
 * This driver exports /proc/ppc64/scan−log−dump which can be read.
 * The driver supports only sequential reads.
 *
 * The driver looks at a write to the driver for the single word "reset".
 * If given, the driver will reset the scanlog so the platform can free it.
 */

#include <linux/module.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <asm/rtas.h>
#include <asm/prom.h>

#define MODULE_VERSION "1.0"
#define MODULE_NAME "scanlog"

/* Status returns from ibm,scan−log−dump */
#define SCANLOG_COMPLETE 0
#define SCANLOG_HWERROR −1
#define SCANLOG_CONTINUE 1

#define DEBUG(A...) do { if (scanlog_debug) printk(KERN_ERR "scanlog: " A); } while (0)

static int scanlog_debug;
static unsigned int ibm_scan_log_dump;                  /* RTAS token */
static struct proc_dir_entry *proc_ppc64_scan_log_dump; /* The proc file */


static ssize_t scanlog_read(struct file *file, char *buf,
                            size_t count, loff_t *ppos)
{
        struct proc_dir_entry *dp = file->f_dentry->d_inode->u.generic_ip;
        unsigned int *data = (unsigned int *)dp->data;
        unsigned long status;
        unsigned long len, off;
        unsigned int wait_time;

        if (!data) {
                printk(KERN_ERR "scanlog: read failed no data\n");
                return −EIO;
        }

        if (count > RTAS_DATA_BUF_SIZE)
                count = RTAS_DATA_BUF_SIZE;

        if (count < 1024) {
                /* This is the min supported by this RTAS call.  Rather
                 * than do all the buffering we insist the user code handle
                 * larger reads.  As long as cp works... :)
                 */
                printk(KERN_ERR "scanlog: cannot perform a small read (%ld)\n", count);
                return −EINVAL;
        }

        if (verify_area(VERIFY_WRITE, buf, count))
                return −EFAULT;

        for (;;) {
                wait_time = HZ/2;       /* default wait if no data */
                spin_lock(&rtas_data_buf_lock);
                memcpy(rtas_data_buf, data, RTAS_DATA_BUF_SIZE);
                status = rtas_call(ibm_scan_log_dump, 2, 1, NULL,
                                   __pa(rtas_data_buf), count);
                memcpy(data, rtas_data_buf, RTAS_DATA_BUF_SIZE);
                spin_unlock(&rtas_data_buf_lock);

                DEBUG("status=%ld, data[0]=%x, data[1]=%x, data[2]=%x\n",
                      status, data[0], data[1], data[2]);
                switch (status) {
                    case SCANLOG_COMPLETE:
                        DEBUG("hit eof\n");
                        return 0;
```

```
91                      case SCANLOG_HWERROR:
92                          DEBUG("hardware error reading scan log data\n");
93                          return -EIO;
94                      case SCANLOG_CONTINUE:
95                          /* We may or may not have data yet */
96                          len = data[1];
97                          off = data[2];
98                          if (len > 0) {
99                                  if (copy_to_user(buf, ((char *)data)+off, len))
100                                         return -EFAULT;
101                                 return len;
102                         }
103                         /* Break to sleep default time */
104                         break;
105                     default:
106                         if (status > 9900 && status <= 9905) {
107                                 /* No data.  RTAS is hinting at a delay required
108                                  * between 1-100000 milliseconds
109                                  */
110                                 int ms = 1;
111                                 for (; status > 9900; status--)
112                                         ms = ms * 10;
113                                 /* Use microseconds for reasonable accuracy */
114                                 ms *= 1000;
115                                 wait_time = ms / (1000000/HZ); /* round down is fine */
116                                 /* Fall through to sleep */
117                         } else {
118                                 printk(KERN_ERR "scanlog: unknown error from rtas: %ld\n", status);
119                                 return -EIO;
120                         }
121                 }
122                 /* Apparently no data yet.  Wait and try again. */
123                 set_current_state(TASK_INTERRUPTIBLE);
124                 schedule_timeout(wait_time);
125         }
126         /*NOTREACHED*/
127 }
128
129 static ssize_t scanlog_write(struct file * file, const char * buf,
130                              size_t count, loff_t *ppos)
131 {
132         unsigned long status;
133
134         if (buf) {
135                 if (strncmp(buf, "reset", 5) == 0) {
136                         DEBUG("reset scanlog\n");
137                         status = rtas_call(ibm_scan_log_dump, 2, 1, NULL, NULL, 0);
138                         DEBUG("rtas returns %ld\n", status);
139                 } else if (strncmp(buf, "debugon", 7) == 0) {
140                         printk(KERN_ERR "scanlog: debug on\n");
141                         scanlog_debug = 1;
142                 } else if (strncmp(buf, "debugoff", 8) == 0) {
143                         printk(KERN_ERR "scanlog: debug off\n");
144                         scanlog_debug = 0;
145                 }
146         }
147         return count;
148 }
149
150 static int scanlog_open(struct inode * inode, struct file * file)
151 {
152         struct proc_dir_entry *dp = file->f_dentry->d_inode->u.generic_ip;
153         unsigned int *data = (unsigned int *)dp->data;
154
155         if (!data) {
156                 printk(KERN_ERR "scanlog: open failed no data\n");
157                 return -EIO;
158         }
159         if (data[0] != 0) {
160                 /* This imperfect test stops a second copy of the
161                  * data (or a reset while data is being copied)
162                  */
163                 return -EBUSY;
164         }
165
166         data[0] = 0;    /* re-init so we restart the scan */
167
168         return 0;
169 }
170
171 static int scanlog_release(struct inode * inode, struct file * file)
172 {
173         struct proc_dir_entry *dp = file->f_dentry->d_inode->u.generic_ip;
174         unsigned int *data = (unsigned int *)dp->data;
175
176         if (!data) {
177                 printk(KERN_ERR "scanlog: release failed no data\n");
178                 return -EIO;
179         }
180         data[0] = 0;
```

```
181
182         return 0;
183 }
184
185 struct file_operations scanlog_fops = {
186         owner:          THIS_MODULE,
187         read:           scanlog_read,
188         write:          scanlog_write,
189         open:           scanlog_open,
190         release:        scanlog_release,
191 };
192
193 int __init scanlog_init(void)
194 {
195         struct proc_dir_entry *ent;
196
197         ibm_scan_log_dump = rtas_token("ibm,scan-log-dump");
198         if (ibm_scan_log_dump == RTAS_UNKNOWN_SERVICE) {
199                 printk(KERN_ERR "scan-log-dump not implemented on this system\n");
200                 return -EIO;
201         }
202
203         ent = create_proc_entry("ppc64/scan-log-dump", S_IRUSR, NULL);
204         if (ent) {
205                 ent->proc_fops = &scanlog_fops;
206                 /* Ideally we could allocate a buffer < 4G */
207                 ent->data = kmalloc(RTAS_DATA_BUF_SIZE, GFP_KERNEL);
208                 if (!ent->data) {
209                         printk(KERN_ERR "Failed to allocate a buffer\n");
210                         remove_proc_entry("scan-log-dump", ent->parent);
211                         return -ENOMEM;
212                 }
213         } else {
214                 printk(KERN_ERR "Failed to create ppc64/scan-log-dump proc entry\n");
215                 return -EIO;
216         }
217         proc_ppc64_scan_log_dump = ent;
218
219         return 0;
220 }
221
222 void __exit scanlog_cleanup(void)
223 {
224         if (proc_ppc64_scan_log_dump) {
225                 if (proc_ppc64_scan_log_dump->data)
226                     kfree(proc_ppc64_scan_log_dump->data);
227                 remove_proc_entry("scan-log-dump", proc_ppc64_scan_log_dump->parent);
228         }
229 }
230
231 module_init(scanlog_init);
232 module_exit(scanlog_cleanup);
233 MODULE_LICENSE("GPL");
```

```
/*
 * PowerPC64 Segment Translation Support.
 *
 * Dave Engebretsen and Mike Corrigan {engebret|mikejc}@us.ibm.com
 *    Copyright (c) 2001 Dave Engebretsen
 *
 *      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.
 */

#include <linux/config.h>
#include <asm/pgtable.h>
#include <asm/mmu.h>
#include <asm/mmu_context.h>
#include <asm/paca.h>
#include <asm/naca.h>
#include <asm/pmc.h>

inline int make_ste(unsigned long stab,
                    unsigned long esid, unsigned long vsid);
inline void make_slbe(unsigned long esid, unsigned long vsid,
                      int large);

/*
 * Build an entry for the base kernel segment and put it into
 * the segment table or SLB.  All other segment table or SLB
 * entries are faulted in.
 */
void stab_initialize(unsigned long stab)
{
        unsigned long esid, vsid;

        esid = GET_ESID(KERNELBASE);
        vsid = get_kernel_vsid(esid << SID_SHIFT);

        if (!__is_processor(PV_POWER4) && !__is_processor(PV_POWER4p)) {
                __asm__ __volatile__("isync; slbia; isync":::"memory");
                make_ste(stab, esid, vsid);
        } else {
                /* Invalidate the entire SLB & all the ERATS */
                __asm__ __volatile__("isync" : : : "memory");
#ifndef CONFIG_PPC_ISERIES
                __asm__ __volatile__("slbmte  %0,%0"
                                     : : "r" (0) : "memory");
                __asm__ __volatile__("isync; slbia; isync":::"memory");
                make_slbe(esid, vsid, 0);
#else
                __asm__ __volatile__("isync; slbia; isync":::"memory");
#endif
        }
}

/*
 * Create a segment table entry for the given esid/vsid pair.
 */
inline int
make_ste(unsigned long stab, unsigned long esid, unsigned long vsid)
{
        unsigned long entry, group, old_esid, castout_entry, i;
        unsigned int global_entry;
        STE *ste, *castout_ste;

        /* Search the primary group first. */
        global_entry = (esid & 0x1f) << 3;
        ste = (STE *)(stab | ((esid & 0x1f) << 7));

        /*
         * Find an empty entry, if one exists.
         */
        for(group = 0; group < 2; group++) {
                for(entry = 0; entry < 8; entry++, ste++) {
                        if(!(ste->dw0.dw0.v)) {
                                ste->dw1.dw1.vsid = vsid;
                                /* Order VSID updte */
                                __asm__ __volatile__ ("eieio" : : : "memory");
                                ste->dw0.dw0.esid = esid;
                                ste->dw0.dw0.v  = 1;
                                ste->dw0.dw0.kp = 1;
                                /* Order update     */
                                __asm__ __volatile__ ("sync" : : : "memory");

                                return(global_entry | entry);
                        }
                }
                /* Now search the secondary group. */
                global_entry = ((~esid) & 0x1f) << 3;
                ste = (STE *)(stab | (((~esid) & 0x1f) << 7));
        }
```

```
91
92          /*
93           * Could not find empty entry, pick one with a round robin selection.
94           * Search all entries in the two groups.  Note that the first time
95           * we get here, we start with entry 1 so the initializer
96           * can be common with the SLB castout code.
97           */
98
99          /* This assumes we never castout when initializing the stab. */
100         PMC_SW_PROCESSOR(stab_capacity_castouts);
101
102         castout_entry = get_paca()->xStab_data.next_round_robin;
103         for(i = 0; i < 16; i++) {
104                 if(castout_entry < 8) {
105                         global_entry = (esid & 0x1f) << 3;
106                         ste = (STE *)(stab | ((esid & 0x1f) << 7));
107                         castout_ste = ste + castout_entry;
108                 } else {
109                         global_entry = ((~esid) & 0x1f) << 3;
110                         ste = (STE *)(stab | (((~esid) & 0x1f) << 7));
111                         castout_ste = ste + (castout_entry - 8);
112                 }
113
114                 if((((castout_ste->dw0.dw0.esid) >> 32) == 0) ||
115                    (((castout_ste->dw0.dw0.esid) & 0xffffffff) > 0)) {
116                         /* Found an entry to castout.  It is either a user */
117                         /* region, or a secondary kernel segment.          */
118                         break;
119                 }
120
121                 castout_entry = (castout_entry + 1) & 0xf;
122         }
123
124         get_paca()->xStab_data.next_round_robin = (castout_entry + 1) & 0xf;
125
126         /* Modify the old entry to the new value. */
127
128         /* Force previous translations to complete. DRENG */
129         __asm__ __volatile__ ("isync" : : : "memory" );
130
131         castout_ste->dw0.dw0.v = 0;
132         __asm__ __volatile__ ("sync" : : : "memory" );    /* Order update */
133         castout_ste->dw1.dw1.vsid = vsid;
134         __asm__ __volatile__ ("eieio" : : : "memory" );   /* Order update */
135         old_esid = castout_ste->dw0.dw0.esid;
136         castout_ste->dw0.dw0.esid = esid;
137         castout_ste->dw0.dw0.v  = 1;
138         castout_ste->dw0.dw0.kp = 1;
139         __asm__ __volatile__ ("slbie %0" : : "r" (old_esid << SID_SHIFT));
140         /* Ensure completion of slbie */
141         __asm__ __volatile__ ("sync" : : : "memory" );
142
143         return(global_entry | (castout_entry & 0x7));
144 }
145
146 /*
147  * Create a segment buffer entry for the given esid/vsid pair.
148  */
149 inline void make_slbe(unsigned long esid, unsigned long vsid, int large)
150 {
151         unsigned long entry, castout_entry;
152         slb_dword0 castout_esid_data;
153         union {
154                 unsigned long word0;
155                 slb_dword0    data;
156         } esid_data;
157         union {
158                 unsigned long word0;
159                 slb_dword1    data;
160         } vsid_data;
161
162         /*
163          * Find an empty entry, if one exists.
164          */
165         for(entry = 0; entry < naca->slb_size; entry++) {
166                 __asm__ __volatile__("slbmfee %0,%1"
167                                      : "=r" (esid_data) : "r" (entry));
168                 if(!esid_data.data.v) {
169                         /*
170                          * Write the new SLB entry.
171                          */
172                         vsid_data.word0 = 0;
173                         vsid_data.data.vsid = vsid;
174                         vsid_data.data.kp = 1;
175                         if (large)
176                                 vsid_data.data.l = 1;
177
178                         esid_data.word0 = 0;
179                         esid_data.data.esid = esid;
180                         esid_data.data.v = 1;
```

```
181                             esid_data.data.index = entry;
182
183                             /* slbie not needed as no previous mapping existed. */
184                             /* Order update  */
185                             __asm__ __volatile__ ("isync" : : : "memory");
186                             __asm__ __volatile__ ("slbmte  %0,%1"
187                                                   : : "r" (vsid_data),
188                                                   "r" (esid_data));
189                             /* Order update  */
190                             __asm__ __volatile__ ("isync" : : : "memory");
191                             return;
192                     }
193             }
194
195             /*
196              * Could not find empty entry, pick one with a round robin selection.
197              */
198
199             PMC_SW_PROCESSOR(stab_capacity_castouts);
200
201             castout_entry = get_paca()->xStab_data.next_round_robin;
202             __asm__ __volatile__("slbmfee  %0,%1"
203                                  : "=r" (castout_esid_data)
204                                  : "r" (castout_entry));
205
206             entry = castout_entry;
207             castout_entry++;
208             if(castout_entry >= naca->slb_size) {
209                     castout_entry = 1;
210             }
211             get_paca()->xStab_data.next_round_robin = castout_entry;
212
213             /* Invalidate the old entry. */
214             castout_esid_data.v = 0; /* Set the class to 0 */
215             /* slbie not needed as the previous mapping is still valid. */
216             __asm__ __volatile__("slbie %0" : : "r" (castout_esid_data));
217
218             /*
219              * Write the new SLB entry.
220              */
221             vsid_data.word0 = 0;
222             vsid_data.data.vsid = vsid;
223             vsid_data.data.kp = 1;
224             if (large)
225                     vsid_data.data.l = 1;
226
227             esid_data.word0 = 0;
228             esid_data.data.esid = esid;
229             esid_data.data.v = 1;
230             esid_data.data.index = entry;
231
232             __asm__ __volatile__ ("isync" : : : "memory");   /* Order update */
233             __asm__ __volatile__ ("slbmte  %0,%1"
234                                   : : "r" (vsid_data), "r" (esid_data));
235             __asm__ __volatile__ ("isync" : : : "memory" );   /* Order update */
236     }
237
238     /*
239      * Allocate a segment table entry for the given ea.
240      */
241     int ste_allocate ( unsigned long ea,
242                        unsigned long trap)
243     {
244             unsigned long vsid, esid;
245             int kernel_segment = 0;
246
247             PMC_SW_PROCESSOR(stab_faults);
248
249             /* Check for invalid effective addresses. */
250             if (!IS_VALID_EA(ea)) {
251                     return 1;
252             }
253
254             /* Kernel or user address? */
255             if (REGION_ID(ea)) {
256                     kernel_segment = 1;
257                     vsid = get_kernel_vsid( ea );
258             } else {
259                     struct mm_struct *mm = current->mm;
260                     if ( mm ) {
261                             vsid = get_vsid(mm->context, ea );
262                     } else {
263                             return 1;
264                     }
265             }
266
267             esid = GET_ESID(ea);
268             if (trap == 0x380 || trap == 0x480) {
269     #ifndef CONFIG_PPC_ISERIES
270                     if (REGION_ID(ea) == KERNEL_REGION_ID)
```

```
271                             make_slbe(esid, vsid, 1);
272                     else
273     #endif
274                             make_slbe(esid, vsid, 0);
275             } else {
276                     unsigned char top_entry, stab_entry, *segments;
277
278                     stab_entry = make_ste(get_paca()->xStab_data.virt, esid, vsid);
279                     PMC_SW_PROCESSOR_A(stab_entry_use, stab_entry & 0xf);
280
281                     segments = get_paca()->xSegments;
282                     top_entry = segments[0];
283                     if(!kernel_segment && top_entry < (STAB_CACHE_SIZE - 1)) {
284                             top_entry++;
285                             segments[top_entry] = stab_entry;
286                             if(top_entry == STAB_CACHE_SIZE - 1) top_entry = 0xff;
287                             segments[0] = top_entry;
288                     }
289             }
290
291             return(0);
292     }
293
294     /*
295      * Flush all entries from the segment table of the current processor.
296      * Kernel and Bolted entries are not removed as we cannot tolerate
297      * faults on those addresses.
298      */
299
300     #define STAB_PRESSURE 0
301
302     void flush_stab(void)
303     {
304             STE *stab = (STE *) get_paca()->xStab_data.virt;
305             unsigned char *segments = get_paca()->xSegments;
306             unsigned long flags, i;
307
308             if (!__is_processor(PV_POWER4) && !__is_processor(PV_POWER4p)) {
309                     unsigned long entry;
310                     STE *ste;
311
312                     /* Force previous translations to complete. DRENG */
313                     __asm__ __volatile__ ("isync" : : : "memory");
314
315                     __save_and_cli(flags);
316                     if(segments[0] != 0xff && !STAB_PRESSURE) {
317                             for(i = 1; i <= segments[0]; i++) {
318                                     ste = stab + segments[i];
319                                     ste->dw0.dw0.v = 0;
320                                     PMC_SW_PROCESSOR(stab_invalidations);
321                             }
322                     } else {
323                             /* Invalidate all entries. */
324                             ste = stab;
325
326                             /* Never flush the first entry. */
327                             ste += 1;
328                             for(entry = 1;
329                                 entry < (PAGE_SIZE / sizeof(STE));
330                                 entry++, ste++) {
331                                     unsigned long ea;
332                                     ea = ste->dw0.dw0.esid << SID_SHIFT;
333                                     if (STAB_PRESSURE || (!REGION_ID(ea))) {
334                                             ste->dw0.dw0.v = 0;
335                                             PMC_SW_PROCESSOR(stab_invalidations);
336                                     }
337                             }
338                     }
339
340                     *((unsigned long *)segments) = 0;
341                     __restore_flags(flags);
342
343                     /* Invalidate the SLB. */
344                     /* Force invals to complete. */
345                     __asm__ __volatile__ ("sync" : : : "memory");
346                     /* Flush the SLB.              */
347                     __asm__ __volatile__ ("slbia" : : : "memory");
348                     /* Force flush to complete.  */
349                     __asm__ __volatile__ ("sync" : : : "memory");
350             } else {
351                     unsigned long flags;
352
353                     PMC_SW_PROCESSOR(stab_invalidations);
354
355                     __save_and_cli(flags);
356                     __asm__ __volatile__("isync; slbia; isync":::"memory");
357                     __restore_flags(flags);
358             }
359     }
```

```
/*
 * NS16550 Serial Port (uart) debugging stuff.
 *
 * c 2001 PPC 64 Team, IBM Corp
 *
 * NOTE: I am trying to make this code avoid any static data references to
 *  simplify debugging early boot.  We'll see how that goes...
 *
 * To use this call udbg_init() first.  It will init the uart to 9600 8N1.
 * You may need to update the COM1 define if your uart is at a different addr.
 *
 *      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.
 */

#include <stdarg.h>
#define WANT_PPCDBG_TAB /* Only defined here */
#include <asm/ppcdebug.h>
#include <asm/processor.h>
#include <asm/naca.h>
#include <asm/uaccess.h>
#include <asm/machdep.h>

struct NS16550 {
        /* this struct must be packed */
        unsigned char rbr;  /* 0 */
        unsigned char ier;  /* 1 */
        unsigned char fcr;  /* 2 */
        unsigned char lcr;  /* 3 */
        unsigned char mcr;  /* 4 */
        unsigned char lsr;  /* 5 */
        unsigned char msr;  /* 6 */
        unsigned char scr;  /* 7 */
};

#define thr rbr
#define iir fcr
#define dll rbr
#define dlm ier
#define dlab lcr

#define LSR_DR   0x01  /* Data ready */
#define LSR_OE   0x02  /* Overrun */
#define LSR_PE   0x04  /* Parity error */
#define LSR_FE   0x08  /* Framing error */
#define LSR_BI   0x10  /* Break */
#define LSR_THRE 0x20  /* Xmit holding register empty */
#define LSR_TEMT 0x40  /* Xmitter empty */
#define LSR_ERR  0x80  /* Error */

volatile struct NS16550 *udbg_comport;

void
udbg_init_uart(void *comport)
{
        if (comport) {
                udbg_comport = (struct NS16550 *)comport;
                udbg_comport->lcr = 0x00; eieio();
                udbg_comport->ier = 0xFF; eieio();
                udbg_comport->ier = 0x00; eieio();
                udbg_comport->lcr = 0x80; eieio();      /* Access baud rate */
                udbg_comport->dll = 12;   eieio();      /* 1 = 115200,  2 = 57600, 3 = 38400, 12 = 9600 baud */
                udbg_comport->dlm = 0;    eieio();      /* dll >> 8 which should be zero for fast rates; */
                udbg_comport->lcr = 0x03; eieio();      /* 8 data, 1 stop, no parity */
                udbg_comport->mcr = 0x03; eieio();      /* RTS/DTR */
                udbg_comport->fcr = 0x07; eieio();      /* Clear & enable FIFOs */
        }
}

void
udbg_putc(unsigned char c)
{
        if ( udbg_comport ) {
                while ((udbg_comport->lsr & LSR_THRE) == 0)
                        /* wait for idle */;
                udbg_comport->thr = c; eieio();
                if (c == '\n') {
                        /* Also put a CR.  This is for convenience. */
                        while ((udbg_comport->lsr & LSR_THRE) == 0)
                                /* wait for idle */;
                        udbg_comport->thr = '\r'; eieio();
                }
        } else if (systemcfg->platform == PLATFORM_ISERIES_LPAR) {
                /* ToDo: switch this via ppc_md */
                printk("%c", c);
        }
}

```

```
91  int udbg_getc_poll(void)
92  {
93          if (udbg_comport) {
94                  if ((udbg_comport->lsr & LSR_DR) != 0)
95                          return udbg_comport->rbr;
96                  else
97                          return -1;
98          }
99          return -1;
100 }
101
102 unsigned char
103 udbg_getc(void)
104 {
105         if ( udbg_comport ) {
106                 while ((udbg_comport->lsr & LSR_DR) == 0)
107                         /* wait for char */;
108                 return udbg_comport->rbr;
109         }
110         return 0;
111 }
112
113 void
114 udbg_puts(const char *s)
115 {
116         if (ppc_md.udbg_putc) {
117                 char c;
118
119                 if (s && *s != '\0') {
120                         while ((c = *s++) != '\0')
121                                 ppc_md.udbg_putc(c);
122                 }
123         } else {
124                 printk("%s", s);
125         }
126 }
127
128 int
129 udbg_write(const char *s, int n)
130 {
131         int remain = n;
132         char c;
133         if (!ppc_md.udbg_putc)
134                 for (;;);       /* stop here for cpuctl */
135         if ( s && *s != '\0' ) {
136                 while ( (( c = *s++ ) != '\0') && (remain-- > 0)) {
137                         ppc_md.udbg_putc(c);
138                 }
139         }
140         return n - remain;
141 }
142
143 int
144 udbg_read(char *buf, int buflen) {
145         char c, *p = buf;
146         int i;
147         if (!ppc_md.udbg_putc)
148                 for (;;);       /* stop here for cpuctl */
149         for (i = 0; i < buflen; ++i) {
150                 do {
151                         c = ppc_md.udbg_getc();
152                 } while (c == 0x11 || c == 0x13);
153                 *p++ = c;
154         }
155         return i;
156 }
157
158 void
159 udbg_console_write(struct console *con, const char *s, unsigned int n)
160 {
161         udbg_write(s, n);
162 }
163
164 void
165 udbg_puthex(unsigned long val)
166 {
167         int i, nibbles = sizeof(val)*2;
168         unsigned char buf[sizeof(val)*2+1];
169         for (i = nibbles-1;  i >= 0;  i--) {
170                 buf[i] = (val & 0xf) + '0';
171                 if (buf[i] > '9')
172                     buf[i] += ('a'-'0'-10);
173                 val >>= 4;
174         }
175         buf[nibbles] = '\0';
176         udbg_puts(buf);
177 }
178
179 void
180 udbg_printSP(const char *s)
```

```
{
        if (systemcfg->platform == PLATFORM_PSERIES) {
                unsigned long sp;
                asm("mr %0,1" : "=r" (sp) :);
                if (s)
                        udbg_puts(s);
                udbg_puthex(sp);
        }
}

void
udbg_printf(const char *fmt, ...)
{
        unsigned char buf[256];

        va_list args;
        va_start(args, fmt);

        vsprintf(buf, fmt, args);
        udbg_puts(buf);

        va_end(args);
}

/* Special print used by PPCDBG() macro */
void
udbg_ppcdbg(unsigned long debug_flags, const char *fmt, ...)
{
        unsigned long active_debugs = debug_flags & naca->debug_switch;

        if ( active_debugs ) {
                va_list ap;
                unsigned char buf[256];
                unsigned long i, len = 0;

                for(i=0; i < PPCDBG_NUM_FLAGS ;i++) {
                        if (((1U << i) & active_debugs) &&
                            trace_names[i]) {
                                len += strlen(trace_names[i]);
                                udbg_puts(trace_names[i]);
                                break;
                        }
                }
                sprintf(buf, " [%s]: ", current->comm);
                len += strlen(buf);
                udbg_puts(buf);

                while(len < 18) {
                        udbg_puts(" ");
                        len++;
                }

                va_start(ap, fmt);
                vsprintf(buf, fmt, ap);
                udbg_puts(buf);

                va_end(ap);
        }
}

unsigned long
udbg_ifdebug(unsigned long flags)
{
        return (flags & naca->debug_switch);
}
```

```
/* −*− linux−c −*−
 *  drivers/cdrom/viocd.c
 *
 ************************************************************************
 *  iSeries Virtual CD Rom
 *
 *  Authors: Dave Boutcher <boutcher@us.ibm.com>
 *           Ryan Arnold <ryanarn@us.ibm.com>
 *           Colin Devilbiss <devilbis@us.ibm.com>
 *
 * (C) Copyright 2000 IBM Corporation
 *
 * 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) anyu later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 ************************************************************************
 * This routine provides access to CD ROM drives owned and managed by an
 * OS/400 partition running on the same box as this Linux partition.
 *
 * All operations are performed by sending messages back and forth to
 * the OS/400 partition.
 *
 *
 * This device driver can either use it's own major number, or it can
 * pretend to be an AZTECH drive. This is controlled with a
 * CONFIG option.  You can either call this an elegant solution to the
 * fact that a lot of software doesn't recognize a new CD major number...
 * or you can call this a really ugly hack.  Your choice.
 *
 */

#include <linux/major.h>
#include <linux/config.h>

/* Decide on the proper naming convention to use for our device */
#ifdef CONFIG_DEVFS_FS
#define VIOCD_DEVICE "cdroms/cdrom%d"
#define VIOCD_DEVICE_OFFSET 0
#else
#ifdef CONFIG_VIOCD_AZTECH
#define VIOCD_DEVICE "aztcd"
#define VIOCD_DEVICE_OFFSET 0
#else
#define VIOCD_DEVICE "iseries/vcd%c"
#define VIOCD_DEVICE_OFFSET 'a'
#endif
#endif

/************************************************************************
 * Decide if we are using our own major or pretending to be an AZTECH drive
 ************************************************************************/
#ifdef CONFIG_VIOCD_AZTECH
#define MAJOR_NR AZTECH_CDROM_MAJOR
#define do_viocd_request do_aztcd_request
#else
#define MAJOR_NR VIOCD_MAJOR
#endif

#define VIOCD_VERS "1.04"

#include <linux/blk.h>
#include <linux/cdrom.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/proc_fs.h>
#include <linux/module.h>

#include <asm/iSeries/HvTypes.h>
#include <asm/iSeries/HvLpEvent.h>
#include "vio.h"
#include <asm/iSeries/iSeries_proc.h>

extern struct pci_dev * iSeries_vio_dev;

#define signalLpEventFast HvCallEvent_signalLpEventFast

struct viocdlpevent {
        struct HvLpEvent event;
        u32 mReserved1;
```

```
91          u16 mVersion;
92          u16 mSubTypeRc;
93          u16 mDisk;
94          u16 mFlags;
95          u32 mToken;
96          u64 mOffset;            // On open, the max number of disks
97          u64 mLen;               // On open, the size of the disk
98          u32 mBlockSize;         // Only set on open
99          u32 mMediaSize;         // Only set on open
100 };
101
102 enum viocdsubtype {
103         viocdopen = 0x0001,
104         viocdclose = 0x0002,
105         viocdread = 0x0003,
106         viocdwrite = 0x0004,
107         viocdlockdoor = 0x0005,
108         viocdgetinfo = 0x0006,
109         viocdcheck = 0x0007
110 };
111
112 /* Should probably make this a module parameter....sigh
113  */
114 #define VIOCD_MAX_CD 8
115 int viocd_blocksizes[VIOCD_MAX_CD];
116 static u64 viocd_size_in_bytes[VIOCD_MAX_CD];
117
118 static const struct vio_error_entry viocd_err_table[] = {
119         {0x0201, EINVAL, "Invalid Range"},
120         {0x0202, EINVAL, "Invalid Token"},
121         {0x0203, EIO, "DMA Error"},
122         {0x0204, EIO, "Use Error"},
123         {0x0205, EIO, "Release Error"},
124         {0x0206, EINVAL, "Invalid CD"},
125         {0x020C, EROFS, "Read Only Device"},
126         {0x020D, EIO, "Changed or Missing Volume (or Varied Off?)"},
127         {0x020E, EIO, "Optical System Error (Varied Off?)"},
128         {0x02FF, EIO, "Internal Error"},
129         {0x3010, EIO, "Changed Volume"},
130         {0xC100, EIO, "Optical System Error"},
131         {0x0000, 0, NULL},
132 };
133
134 /* This is the structure we use to exchange info between driver and interrupt
135  * handler
136  */
137 struct viocd_waitevent {
138         struct semaphore *sem;
139         int rc;
140         u16 subtypeRc;
141         int changed;
142 };
143
144 /* this is a lookup table for the true capabilities of a device */
145 struct capability_entry {
146         char *type;
147         int capability;
148 };
149
150 static struct capability_entry capability_table[] = {
151         { "6330", CDC_LOCK | CDC_DVD_RAM },
152         { "6321", CDC_LOCK },
153         { "632B", 0 },
154         { NULL  , CDC_LOCK },
155 };
156
157 struct block_device_operations viocd_fops =
158 {
159         owner:                  THIS_MODULE,
160         open:                   cdrom_open,
161         release:                cdrom_release,
162         ioctl:                  cdrom_ioctl,
163         check_media_change:     cdrom_media_changed,
164 };
165
166 /* These are our internal structures for keeping track of devices
167  */
168 static int viocd_numdev;
169
170 struct cdrom_info {
171         char rsrcname[10];
172         char type[4];
173         char model[3];
174 };
175 static struct cdrom_info *viocd_unitinfo = NULL;
176
177 struct disk_info{
178         u32 useCount;
179         u32 blocksize;
180         u32 mediasize;
```

```
};
static struct disk_info viocd_diskinfo[VIOCD_MAX_CD];

static struct cdrom_device_info viocd_info[VIOCD_MAX_CD];

static spinlock_t viocd_lock = SPIN_LOCK_UNLOCKED;

#define MAX_CD_REQ 1
static LIST_HEAD(reqlist);

/* End a request
 */
static int viocd_end_request(struct request *req, int uptodate)
{
        if (end_that_request_first(req, uptodate, DEVICE_NAME))
                return 0;
        end_that_request_last(req);
        return 1;
}


/* Get info on CD devices from OS/400
 */
static void get_viocd_info(void)
{
        dma_addr_t dmaaddr;
        HvLpEvent_Rc hvrc;
        int i;
        DECLARE_MUTEX_LOCKED(Semaphore);
        struct viocd_waitevent we;

        // If we don't have a host, bail out
        if (viopath_hostLp == HvLpIndexInvalid)
                return;

        if (viocd_unitinfo == NULL)
                viocd_unitinfo =
                    kmalloc(sizeof(struct cdrom_info) * VIOCD_MAX_CD,
                            GFP_KERNEL);

        memset(viocd_unitinfo, 0x00,
               sizeof(struct cdrom_info) * VIOCD_MAX_CD);

        dmaaddr = pci_map_single(iSeries_vio_dev, viocd_unitinfo,
                                 sizeof(struct cdrom_info) * VIOCD_MAX_CD,
                                 PCI_DMA_FROMDEVICE);
        if (dmaaddr == 0xFFFFFFFF) {
                printk(KERN_WARNING_VIO "error allocating tce\n");
                return;
        }

        we.sem = &Semaphore;

        hvrc = signalLpEventFast(viopath_hostLp,
                                 HvLpEvent_Type_VirtualIo,
                                 viomajorsubtype_cdio | viocdgetinfo,
                                 HvLpEvent_AckInd_DoAck,
                                 HvLpEvent_AckType_ImmediateAck,
                                 viopath_sourceinst(viopath_hostLp),
                                 viopath_targetinst(viopath_hostLp),
                                 (u64) (unsigned long) &we,
                                 VIOVERSION << 16,
                                 dmaaddr,
                                 0,
                                 sizeof(struct cdrom_info) * VIOCD_MAX_CD,
                                 0);
        if (hvrc != HvLpEvent_Rc_Good) {
                printk(KERN_WARNING_VIO "cdrom error sending event. rc %d\n", (int) hvrc);
                return;
        }

        down(&Semaphore);

        if (we.rc) {
                const struct vio_error_entry *err = vio_lookup_rc(viocd_err_table, we.subtypeRc);
                printk(KERN_WARNING_VIO "bad rc %d:0x%04X on getinfo: %s\n", we.rc, we.subtypeRc, err->msg);
                return;
        }


        for (i = 0; (i < VIOCD_MAX_CD) && (viocd_unitinfo[i].rsrcname[0]); i++) {
                viocd_numdev++;
        }
}

/* Open a device
 */
static int viocd_open(struct cdrom_device_info *cdi, int purpose)
{
        DECLARE_MUTEX_LOCKED(Semaphore);
```

```
271         int device_no = MINOR(cdi->dev);
272         HvLpEvent_Rc hvrc;
273         struct viocd_waitevent we;
274         struct disk_info *diskinfo = &viocd_diskinfo[device_no];
275
276         // If we don't have a host, bail out
277         if (viopath_hostLp == HvLpIndexInvalid || device_no >= viocd_numdev)
278                 return -ENODEV;
279
280         we.sem = &Semaphore;
281         hvrc = signalLpEventFast(viopath_hostLp,
282                                  HvLpEvent_Type_VirtualIo,
283                                  viomajorsubtype_cdio | viocdopen,
284                                  HvLpEvent_AckInd_DoAck,
285                                  HvLpEvent_AckType_ImmediateAck,
286                                  viopath_sourceinst(viopath_hostLp),
287                                  viopath_targetinst(viopath_hostLp),
288                                  (u64) (unsigned long) &we,
289                                  VIOVERSION << 16,
290                                  ((u64) device_no << 48),
291                                  0, 0, 0);
292         if (hvrc != 0) {
293                 printk(KERN_WARNING_VIO "bad rc on signalLpEventFast %d\n",
294                        (int) hvrc);
295                 return -EIO;
296         }
297
298         down(&Semaphore);
299
300         if (we.rc) {
301                 const struct vio_error_entry *err = vio_lookup_rc(viocd_err_table, we.subtypeRc);
302                 printk(KERN_WARNING_VIO "bad rc %d:0x%04X on open: %s\n", we.rc, we.subtypeRc, err->msg);
303                 return -err->errno;
304         }
305
306         if (diskinfo->useCount == 0) {
307                 if(diskinfo->blocksize > 0) {
308                         viocd_blocksizes[device_no] = diskinfo->blocksize;
309                         viocd_size_in_bytes[device_no] = diskinfo->blocksize * diskinfo->mediasize;
310                 } else {
311                         viocd_size_in_bytes[device_no] = 0xFFFFFFFFFFFFFFFF;
312                 }
313         }
314         MOD_INC_USE_COUNT;
315         return 0;
316 }
317
318 /* Release a device
319  */
320 static void viocd_release(struct cdrom_device_info *cdi)
321 {
322         int device_no = MINOR(cdi->dev);
323         HvLpEvent_Rc hvrc;
324
325         /* If we don't have a host, bail out */
326         if (viopath_hostLp == HvLpIndexInvalid
327             || device_no >= viocd_numdev)
328                 return;
329
330         hvrc = signalLpEventFast(viopath_hostLp,
331                                  HvLpEvent_Type_VirtualIo,
332                                  viomajorsubtype_cdio | viocdclose,
333                                  HvLpEvent_AckInd_NoAck,
334                                  HvLpEvent_AckType_ImmediateAck,
335                                  viopath_sourceinst(viopath_hostLp),
336                                  viopath_targetinst(viopath_hostLp),
337                                  0,
338                                  VIOVERSION << 16,
339                                  ((u64) device_no << 48),
340                                  0, 0, 0);
341         if (hvrc != 0) {
342                 printk(KERN_WARNING_VIO "bad rc on signalLpEventFast %d\n", (int) hvrc);
343                 return;
344         }
345
346         MOD_DEC_USE_COUNT;
347 }
348
349 /* Send a read or write request to OS/400
350  */
351 static int send_request(struct request *req)
352 {
353         HvLpEvent_Rc hvrc;
354         dma_addr_t dmaaddr;
355         int device_no = DEVICE_NR(req->rq_dev);
356         u64 start = req->sector * 512,
357             len = req->current_nr_sectors * 512;
358         char reading = req->cmd == READ;
359         u16 command = reading ? viocdread : viocdwrite;
360
```

```
361
362         if(start + len > viocd_size_in_bytes[device_no]) {
363                 printk(KERN_WARNING_VIO "viocd%d; access position %lx, past size %lx\n",
364                        device_no, start + len, viocd_size_in_bytes[device_no]);
365                 return -1;
366         }
367
368         dmaaddr = pci_map_single(iSeries_vio_dev, req->buffer, len,
369                                  reading ? PCI_DMA_FROMDEVICE : PCI_DMA_TODEVICE);
370         if (dmaaddr == 0xFFFFFFFF) {
371                 printk(KERN_WARNING_VIO "error allocating tce for address %p len %ld\n",
372                        req->buffer, len);
373                 return -1;
374         }
375
376         hvrc = signalLpEventFast(viopath_hostLp,
377                                  HvLpEvent_Type_VirtualIo,
378                                  viomajorsubtype_cdio | command,
379                                  HvLpEvent_AckInd_DoAck,
380                                  HvLpEvent_AckType_ImmediateAck,
381                                  viopath_sourceinst(viopath_hostLp),
382                                  viopath_targetinst(viopath_hostLp),
383                                  (u64) (unsigned long) req->buffer,
384                                  VIOVERSION << 16,
385                                  ((u64) device_no << 48) | dmaaddr,
386                                  start, len, 0);
387         if (hvrc != HvLpEvent_Rc_Good) {
388                 printk(KERN_WARNING_VIO "hv error on op %d\n", (int) hvrc);
389                 return -1;
390         }
391
392         return 0;
393 }
394
395
396 /* Do a request
397  */
398 static int rwreq;
399 static void do_viocd_request(request_queue_t * q)
400 {
401         for (;;) {
402                 struct request *req;
403                 char err_str[80] = "";
404                 int device_no;
405
406                 INIT_REQUEST;
407                 if (rwreq >= MAX_CD_REQ) {
408                         return;
409                 }
410
411                 device_no = CURRENT_DEV;
412
413                 /* remove the current request from the queue */
414                 req = CURRENT;
415                 blkdev_dequeue_request(req);
416
417                 /* check for any kind of error */
418                 if (device_no > viocd_numdev)
419                         sprintf(err_str, "Invalid device number %d", device_no);
420                 else if (send_request(req) < 0)
421                         strcpy(err_str, "unable to send message to OS/400!");
422
423                 /* if we had any sort of error, log it and cancel the request */
424                 if (*err_str) {
425                         printk(KERN_WARNING_VIO "%s\n", err_str);
426                         viocd_end_request(req, 0);
427                 } else {
428                         spin_lock(&viocd_lock);
429                         list_add_tail(&req->queue, &reqlist);
430                         ++rwreq;
431                         spin_unlock(&viocd_lock);
432                 }
433         }
434 }
435
436 /* Check if the CD changed
437  */
438 static int viocd_media_changed(struct cdrom_device_info *cdi, int disc_nr)
439 {
440         struct viocd_waitevent we;
441         HvLpEvent_Rc hvrc;
442         int device_no = MINOR(cdi->dev);
443
444         /* This semaphore is raised in the interrupt handler                    */
445         DECLARE_MUTEX_LOCKED(Semaphore);
446
447         /* Check that we are dealing with a valid hosting partition             */
448         if (viopath_hostLp == HvLpIndexInvalid) {
449                 printk(KERN_WARNING_VIO "Invalid hosting partition\n");
450                 return -EIO;
```

```
451         }
452
453         we.sem = &Semaphore;
454
455         /* Send the open event to OS/400                                        */
456         hvrc = signalLpEventFast(viopath_hostLp,
457                                  HvLpEvent_Type_VirtualIo,
458                                  viomajorsubtype_cdio | viocdcheck,
459                                  HvLpEvent_AckInd_DoAck,
460                                  HvLpEvent_AckType_ImmediateAck,
461                                  viopath_sourceinst(viopath_hostLp),
462                                  viopath_targetinst(viopath_hostLp),
463                                  (u64) (unsigned long) &we,
464                                  VIOVERSION << 16,
465                                  ((u64) device_no << 48),
466                                  0, 0, 0);
467
468         if (hvrc != 0) {
469                 printk(KERN_WARNING_VIO "bad rc on signalLpEventFast %d\n", (int) hvrc);
470                 return -EIO;
471         }
472
473         /* Wait for the interrupt handler to get the response                   */
474         down(&Semaphore);
475
476         /* Check the return code.  If bad, assume no change                     */
477         if (we.rc) {
478                 const struct vio_error_entry *err = vio_lookup_rc(viocd_err_table, we.subtypeRc);
479                 printk(KERN_WARNING_VIO "bad rc %d:0x%04X on check_change: %s; Assuming no change\n", we.rc, we.subtypeRc, err
    ->msg);
480                 return 0;
481         }
482
483         return we.changed;
484 }
485
486 static int viocd_lock_door(struct cdrom_device_info *cdi, int locking)
487 {
488         HvLpEvent_Rc hvrc;
489         u64 device_no = MINOR(cdi->dev);
490         /* NOTE: flags is 1 or 0 so it won't overwrite the device_no            */
491         u64 flags = !!locking;
492         /* This semaphore is raised in the interrupt handler                    */
493         DECLARE_MUTEX_LOCKED(Semaphore);
494         struct viocd_waitevent we = { sem:&Semaphore };
495
496         /* Check that we are dealing with a valid hosting partition             */
497         if (viopath_hostLp == HvLpIndexInvalid) {
498                 printk(KERN_WARNING_VIO "Invalid hosting partition\n");
499                 return -EIO;
500         }
501
502         we.sem = &Semaphore;
503
504         /* Send the lockdoor event to OS/400                                    */
505         hvrc = signalLpEventFast(viopath_hostLp,
506                                  HvLpEvent_Type_VirtualIo,
507                                  viomajorsubtype_cdio | viocdlockdoor,
508                                  HvLpEvent_AckInd_DoAck,
509                                  HvLpEvent_AckType_ImmediateAck,
510                                  viopath_sourceinst(viopath_hostLp),
511                                  viopath_targetinst(viopath_hostLp),
512                                  (u64) (unsigned long) &we,
513                                  VIOVERSION << 16,
514                                  (device_no << 48) | (flags << 32),
515                                  0, 0, 0);
516
517         if (hvrc != 0) {
518                 printk(KERN_WARNING_VIO "bad rc on signalLpEventFast %d\n", (int) hvrc);
519                 return -EIO;
520         }
521
522         /* Wait for the interrupt handler to get the response                   */
523         down(&Semaphore);
524
525         /* Check the return code.  If bad, assume no change                     */
526         if (we.rc != 0) {
527                 return -EIO;
528         }
529
530         return 0;
531 }
532
533 /* This routine handles incoming CD LP events
534  */
535 static void vioHandleCDEvent(struct HvLpEvent *event)
536 {
537         struct viocdlpevent *bevent = (struct viocdlpevent *) event;
538         struct viocd_waitevent *pwe;
539
```

```
540         if (event == NULL) {
541                 /* Notification that a partition went away! */
542                 return;
543         }
544         /* First, we should NEVER get an int here...only acks */
545         if (event->xFlags.xFunction == HvLpEvent_Function_Int) {
546                 printk(KERN_WARNING_VIO "Yikes! got an int in viocd event handler!\n");
547                 if (event->xFlags.xAckInd == HvLpEvent_AckInd_DoAck) {
548                         event->xRc = HvLpEvent_Rc_InvalidSubtype;
549                         HvCallEvent_ackLpEvent(event);
550                 }
551         }
552
553         switch (event->xSubtype & VIOMINOR_SUBTYPE_MASK) {
554         case viocdopen:
555                 viocd_diskinfo[bevent->mDisk].blocksize = bevent->mBlockSize;
556                 viocd_diskinfo[bevent->mDisk].mediasize = bevent->mMediaSize;
557                 /* FALLTHROUGH !! */
558         case viocdgetinfo:
559         case viocdlockdoor:
560                 pwe = (struct viocd_waitevent *) (unsigned long) event->xCorrelationToken;
561                 pwe->rc = event->xRc;
562                 pwe->subtypeRc = bevent->mSubTypeRc;
563                 up(pwe->sem);
564                 break;
565
566         case viocdclose:
567                 break;
568
569         case viocdwrite:
570         case viocdread:{
571                 unsigned long flags;
572                 int reading = ((event->xSubtype & VIOMINOR_SUBTYPE_MASK) == viocdread);
573                 struct request *req = blkdev_entry_to_request(reqlist.next);
574                 /* Since this is running in interrupt mode, we need to make sure we're not
575                  * stepping on any global I/O operations
576                  */
577                 spin_lock_irqsave(&io_request_lock, flags);
578
579                 pci_unmap_single(iSeries_vio_dev,
580                                  bevent->mToken,
581                                  bevent->mLen,
582                                  reading ? PCI_DMA_FROMDEVICE : PCI_DMA_TODEVICE);
583
584                 /* find the event to which this is a response */
585                 while ((&req->queue != &reqlist) &&
586                        ((u64) (unsigned long) req->buffer != bevent->event.xCorrelationToken))
587                         req = blkdev_entry_to_request(req->queue.next);
588
589                 /* if the event was not there, then what are we responding to?? */
590                 if (&req->queue == &reqlist) {
591                         printk(KERN_WARNING_VIO "Yikes! we never enqueued this guy!\n");
592                         spin_unlock_irqrestore(&io_request_lock,
593                                                flags);
594                         break;
595                 }
596
597                 /* we don't need to keep it around anymore... */
598                 spin_lock(&viocd_lock);
599                 list_del(&req->queue);
600                 --rwreq;
601                 spin_unlock(&viocd_lock);
602                 {
603                         char stat = event->xRc == HvLpEvent_Rc_Good;
604                         int nsect = bevent->mLen >> 9;
605
606                         if (!stat) {
607                                 const struct vio_error_entry *err =
608                                     vio_lookup_rc(viocd_err_table, bevent->mSubTypeRc);
609                                 printk(KERN_WARNING_VIO "request %p failed with rc %d:0x%04X: %s\n",
610                                        req->buffer, event->xRc, bevent->mSubTypeRc, err->msg);
611                         }
612                         while ((nsect > 0) && (req->bh)) {
613                                 nsect -= req->current_nr_sectors;
614                                 viocd_end_request(req, stat);
615                         }
616                         /* we weren't done yet */
617                         if (req->bh) {
618                                 if (send_request(req) < 0) {
619                                         printk(KERN_WARNING_VIO
620                                             "couldn't re-submit req %p\n", req->buffer);
621                                         viocd_end_request(req, 0);
622                                 } else {
623                                         spin_lock(&viocd_lock);
624                                         list_add_tail(&req->queue, &reqlist);
625                                         ++rwreq;
626                                         spin_unlock(&viocd_lock);
627                                 }
628                         }
629                 }
```

```
630
631                 /* restart handling of incoming requests */
632                 do_viocd_request(NULL);
633                 spin_unlock_irqrestore(&io_request_lock, flags);
634                 break;
635         }
636         case viocdcheck:
637                 pwe = (struct viocd_waitevent *) (unsigned long) event->xCorrelationToken;
638                 pwe->rc = event->xRc;
639                 pwe->subtypeRc = bevent->mSubTypeRc;
640                 pwe->changed = bevent->mFlags;
641                 up(pwe->sem);
642                 break;
643
644         default:
645                 printk(KERN_WARNING_VIO "message with invalid subtype %0x04X!\n", event->xSubtype & VIOMINOR_SUBTYPE_MASK);
646                 if (event->xFlags.xAckInd == HvLpEvent_AckInd_DoAck) {
647                         event->xRc = HvLpEvent_Rc_InvalidSubtype;
648                         HvCallEvent_ackLpEvent(event);
649                 }
650         }
651 }
652
653 /* Our file operations table
654  */
655 static struct cdrom_device_ops viocd_dops = {
656         open:viocd_open,
657         release:viocd_release,
658         media_changed:viocd_media_changed,
659         lock_door:viocd_lock_door,
660         capability:CDC_CLOSE_TRAY | CDC_OPEN_TRAY | CDC_LOCK | CDC_SELECT_SPEED | CDC_SELECT_DISC | CDC_MULTI_SES
    SION | CDC_MCN | CDC_MEDIA_CHANGED | CDC_PLAY_AUDIO | CDC_RESET | CDC_IOCTLS | CDC_DRIVE_STATUS | CDC_GENERIC_PAC
    KET | CDC_CD_R | CDC_CD_RW | CDC_DVD | CDC_DVD_R | CDC_DVD_RAM
661 };
662
663 /* Handle reads from the proc file system
664  */
665 static int proc_read(char *buf, char **start, off_t offset,
666                      int blen, int *eof, void *data)
667 {
668         int len = 0;
669         int i;
670
671         for (i = 0; i < viocd_numdev; i++) {
672                 len +=
673                     sprintf(buf + len,
674                             "viocd device %d is iSeries resource %10.10s type %4.4s, model %3.3s\n",
675                             i, viocd_unitinfo[i].rsrcname,
676                             viocd_unitinfo[i].type,
677                             viocd_unitinfo[i].model);
678         }
679         *eof = 1;
680         return len;
681 }
682
683
684 /* setup our proc file system entries
685  */
686 void viocd_proc_init(struct proc_dir_entry *iSeries_proc)
687 {
688         struct proc_dir_entry *ent;
689         ent = create_proc_entry("viocd", S_IFREG | S_IRUSR, iSeries_proc);
690         if (!ent)
691                 return;
692         ent->nlink = 1;
693         ent->data = NULL;
694         ent->read_proc = proc_read;
695 }
696
697 /* clean up our proc file system entries
698  */
699 void viocd_proc_delete(struct proc_dir_entry *iSeries_proc)
700 {
701         remove_proc_entry("viocd", iSeries_proc);
702 }
703
704 static int find_capability(const char *type)
705 {
706         struct capability_entry *entry;
707         for(entry = capability_table; entry->type; ++entry)
708                 if(!strncmp(entry->type, type, 4))
709                         break;
710         return entry->capability;
711 }
712
713 /* Initialize the whole device driver.  Handle module and non-module
714  * versions
715  */
716 __init int viocd_init(void)
717 {
```

```
718         int i, rc;
719
720         if (viopath_hostLp == HvLpIndexInvalid)
721                 vio_set_hostlp();
722
723         /* If we don't have a host, bail out */
724         if (viopath_hostLp == HvLpIndexInvalid)
725                 return -ENODEV;
726
727         rc = viopath_open(viopath_hostLp, viomajorsubtype_cdio, MAX_CD_REQ+2);
728         if (rc) {
729                 printk(KERN_WARNING_VIO "error opening path to host partition %d\n",
730                                 viopath_hostLp);
731                 return rc;
732         }
733
734         /* Initialize our request handler
735          */
736         rwreq = 0;
737         vio_setHandler(viomajorsubtype_cdio, vioHandleCDEvent);
738
739         memset(&viocd_diskinfo, 0x00, sizeof(viocd_diskinfo));
740
741         get_viocd_info();
742
743         if (viocd_numdev == 0) {
744                 vio_clearHandler(viomajorsubtype_cdio);
745                 viopath_close(viopath_hostLp, viomajorsubtype_cdio, MAX_CD_REQ+2);
746                 return 0;
747         }
748
749         printk(KERN_INFO_VIO
750                 "%s: iSeries Virtual CD vers %s, major %d, max disks %d, hosting partition %d\n",
751                DEVICE_NAME, VIOCD_VERS, MAJOR_NR, VIOCD_MAX_CD, viopath_hostLp);
752
753         if (devfs_register_blkdev(MAJOR_NR, "viocd", &viocd_fops) != 0) {
754                 printk(KERN_WARNING_VIO "Unable to get major %d for viocd CD-ROM\n", MAJOR_NR);
755                 return -EIO;
756         }
757
758         blksize_size[MAJOR_NR] = viocd_blocksizes;
759         blk_init_queue(BLK_DEFAULT_QUEUE(MAJOR_NR), DEVICE_REQUEST);
760         read_ahead[MAJOR_NR] = 4;
761
762         memset(&viocd_info, 0x00, sizeof(viocd_info));
763         for (i = 0; i < viocd_numdev; i++) {
764                 viocd_info[i].dev = MKDEV(MAJOR_NR, i);
765                 viocd_info[i].ops = &viocd_dops;
766                 viocd_info[i].speed = 4;
767                 viocd_info[i].capacity = 1;
768                 viocd_info[i].mask = ~find_capability(viocd_unitinfo[i].type);
769                 sprintf(viocd_info[i].name, VIOCD_DEVICE, VIOCD_DEVICE_OFFSET + i);
770                 if (register_cdrom(&viocd_info[i]) != 0) {
771                         printk(KERN_WARNING_VIO "Cannot register viocd CD-ROM %s!\n", viocd_info[i].name);
772                 } else {
773                         printk(KERN_INFO_VIO
774                                 "cd %s is iSeries resource %10.10s type %4.4s, model %3.3s\n",
775                                 viocd_info[i].name,
776                                 viocd_unitinfo[i].rsrcname,
777                                 viocd_unitinfo[i].type,
778                                 viocd_unitinfo[i].model);
779                 }
780         }
781
782         /*
783          * Create the proc entry
784          */
785         iSeries_proc_callback(&viocd_proc_init);
786
787         return 0;
788 }
789
790 #ifdef MODULE
791 void viocd_exit(void)
792 {
793         int i;
794         for (i = 0; i < viocd_numdev; i++) {
795                 if (unregister_cdrom(&viocd_info[i]) != 0) {
796                         printk(KERN_WARNING_VIO "Cannot unregister viocd CD-ROM %s!\n", viocd_info[i].name);
797                 }
798         }
799         if ((devfs_unregister_blkdev(MAJOR_NR, "viocd") == -EINVAL)) {
800                 printk(KERN_WARNING_VIO "can't unregister viocd\n");
801                 return;
802         }
803         blk_cleanup_queue(BLK_DEFAULT_QUEUE(MAJOR_NR));
804         if (viocd_unitinfo)
805                 kfree(viocd_unitinfo);
806
807         iSeries_proc_callback(&viocd_proc_delete);
```

```
808
809             viopath_close(viopath_hostLp, viomajorsubtype_cdio, MAX_CD_REQ+2);
810             vio_clearHandler(viomajorsubtype_cdio);
811     }
812     #endif
813
814     #ifdef MODULE
815     module_init(viocd_init);
816     module_exit(viocd_exit);
817     MODULE_LICENSE("GPL");
818     #endif
```

```
/* −*− linux−c −*−
 *  drivers/char/viocons.c
 *
 *  iSeries Virtual Terminal
 *
 *  Authors: Dave Boutcher <boutcher@us.ibm.com>
 *           Ryan Arnold <ryanarn@us.ibm.com>
 *           Colin Devilbiss <devilbis@us.ibm.com>
 *
 * (C) Copyright 2000 IBM Corporation
 *
 * 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) anyu later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 *
 */
#include <linux/config.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/errno.h>
#include <linux/vmalloc.h>
#include <linux/mm.h>
#include <linux/console.h>
#include <linux/module.h>
#include <asm/uaccess.h>
#include <linux/init.h>
#include <linux/wait.h>
#include <linux/spinlock.h>
#include <asm/ioctls.h>
#include <linux/kd.h>
#include <linux/tty.h>

#include "vio.h"

#include <asm/iSeries/HvLpEvent.h>
#include "asm/iSeries/HvCallEvent.h"
#include "asm/iSeries/HvLpConfig.h"
#include "asm/iSeries/HvCall.h"
#include <asm/iSeries/iSeries_proc.h>

/* Check that the tty_driver_data actually points to our stuff
 */
#define VIOTTY_PARANOIA_CHECK 1
#define VIOTTY_MAGIC (0x0DCB)

static int debug;

static DECLARE_WAIT_QUEUE_HEAD(viocons_wait_queue);

#define VTTY_PORTS 10
#define VIOTTY_SERIAL_START 65

static u64 sndMsgSeq[VTTY_PORTS];
static u64 sndMsgAck[VTTY_PORTS];

static spinlock_t consolelock = SPIN_LOCK_UNLOCKED;

/* THe structure of the events that flow between us and OS/400.  You can't
 * mess with this unless the OS/400 side changes too
 */
struct viocharlpevent {
        struct HvLpEvent event;
        u32 mReserved1;
        u16 mVersion;
        u16 mSubTypeRc;
        u8 virtualDevice;
        u8 immediateDataLen;
        u8 immediateData[VIOCHAR_MAX_DATA];
};

#define viochar_window (10)
#define viochar_highwatermark (3)

enum viocharsubtype {
        viocharopen = 0x0001,
        viocharclose = 0x0002,
        viochardata = 0x0003,
        viocharack = 0x0004,
        viocharconfig = 0x0005
```

```
};

enum viochar_rc {
        viochar_rc_ebusy = 1
};

/* When we get writes faster than we can send it to the partition,
 * buffer the data here.  There is one set of buffers for each virtual
 * port.
 * Note that bufferUsed is a bit map of used buffers.
 * It had better have enough bits to hold NUM_BUF
 * the bitops assume it is a multiple of unsigned long
 */
#define NUM_BUF (8)
#define OVERFLOW_SIZE VIOCHAR_MAX_DATA

static struct overflowBuffers {
        unsigned long bufferUsed;
        u8 *buffer[NUM_BUF];
        int bufferBytes[NUM_BUF];
        int curbuf;
        int bufferOverflow;
        int overflowMessage;
} overflow[VTTY_PORTS];

static void initDataEvent(struct viocharlpevent *viochar, HvLpIndex lp);

static struct tty_driver viotty_driver;
static struct tty_driver viottyS_driver;
static int viotty_refcount;

static struct tty_struct *viotty_table[VTTY_PORTS];
static struct tty_struct *viottyS_table[VTTY_PORTS];
static struct termios *viotty_termios[VTTY_PORTS];
static struct termios *viottyS_termios[VTTY_PORTS];
static struct termios *viotty_termios_locked[VTTY_PORTS];
static struct termios *viottyS_termios_locked[VTTY_PORTS];

void hvlog(char *fmt, ...)
{
        int i;
        static char buf[256];
        va_list args;
        va_start(args, fmt);
        i = vsprintf(buf, fmt, args);
        va_end(args);
        HvCall_writeLogBuffer(buf, i);
        HvCall_writeLogBuffer("\r", 1);

}

/* Our port information.  We store a pointer to one entry in the
 * tty_driver_data
 */
static struct port_info_tag {
        int magic;
        struct tty_struct *tty;
        HvLpIndex lp;
        u8 vcons;
        u8 port;
} port_info[VTTY_PORTS];

/* Make sure we're pointing to a valid port_info structure.  Shamelessly
 * plagerized from serial.c
 */
static inline int viotty_paranoia_check(struct port_info_tag *pi,
                                        kdev_t device, const char *routine)
{
#ifdef VIOTTY_PARANOIA_CHECK
        static const char *badmagic =
            "%s Warning: bad magic number for port_info struct (%s) in %s\n";
        static const char *badinfo =
            "%s Warning: null port_info for (%s) in %s\n";

        if (!pi) {
                printk(badinfo, KERN_WARNING_VIO, kdevname(device),
                       routine);
                return 1;
        }
        if (pi->magic != VIOTTY_MAGIC) {
                printk(badmagic, KERN_WARNING_VIO, kdevname(device),
                       routine);
                return 1;
        }
#endif
        return 0;
}

/*
 * Handle reads from the proc file system.  Right now we just dump the
```

```
181   * state of the first TTY
182   */
183  static int proc_read(char *buf, char **start, off_t offset,
184                       int blen, int *eof, void *data)
185  {
186          int len = 0;
187          struct tty_struct *tty = viotty_table[0];
188          struct termios *termios;
189          if (tty == NULL) {
190                  len += sprintf(buf + len, "no tty\n");
191                  *eof = 1;
192                  return len;
193          }
194
195          len +=
196              sprintf(buf + len,
197                      "tty info: COOK_OUT %ld COOK_IN %ld, NO_WRITE_SPLIT %ld\n",
198                      tty->flags & TTY_HW_COOK_OUT,
199                      tty->flags & TTY_HW_COOK_IN,
200                      tty->flags & TTY_NO_WRITE_SPLIT);
201
202          termios = tty->termios;
203          if (termios == NULL) {
204                  len += sprintf(buf + len, "no termios\n");
205                  *eof = 1;
206                  return len;
207          }
208          len += sprintf(buf + len, "INTR_CHAR   %2.2x\n", INTR_CHAR(tty));
209          len += sprintf(buf + len, "QUIT_CHAR   %2.2x\n", QUIT_CHAR(tty));
210          len +=
211              sprintf(buf + len, "ERASE_CHAR  %2.2x\n", ERASE_CHAR(tty));
212          len += sprintf(buf + len, "KILL_CHAR   %2.2x\n", KILL_CHAR(tty));
213          len += sprintf(buf + len, "EOF_CHAR    %2.2x\n", EOF_CHAR(tty));
214          len += sprintf(buf + len, "TIME_CHAR   %2.2x\n", TIME_CHAR(tty));
215          len += sprintf(buf + len, "MIN_CHAR    %2.2x\n", MIN_CHAR(tty));
216          len += sprintf(buf + len, "SWTC_CHAR   %2.2x\n", SWTC_CHAR(tty));
217          len +=
218              sprintf(buf + len, "START_CHAR  %2.2x\n", START_CHAR(tty));
219          len += sprintf(buf + len, "STOP_CHAR   %2.2x\n", STOP_CHAR(tty));
220          len += sprintf(buf + len, "SUSP_CHAR   %2.2x\n", SUSP_CHAR(tty));
221          len += sprintf(buf + len, "EOL_CHAR    %2.2x\n", EOL_CHAR(tty));
222          len +=
223              sprintf(buf + len, "REPRINT_CHAR %2.2x\n", REPRINT_CHAR(tty));
224          len +=
225              sprintf(buf + len, "DISCARD_CHAR %2.2x\n", DISCARD_CHAR(tty));
226          len +=
227              sprintf(buf + len, "WERASE_CHAR  %2.2x\n", WERASE_CHAR(tty));
228          len +=
229              sprintf(buf + len, "LNEXT_CHAR   %2.2x\n", LNEXT_CHAR(tty));
230          len += sprintf(buf + len, "EOL2_CHAR   %2.2x\n", EOL2_CHAR(tty));
231
232          len += sprintf(buf + len, "I_IGNBRK    %4.4x\n", I_IGNBRK(tty));
233          len += sprintf(buf + len, "I_BRKINT    %4.4x\n", I_BRKINT(tty));
234          len += sprintf(buf + len, "I_IGNPAR    %4.4x\n", I_IGNPAR(tty));
235          len += sprintf(buf + len, "I_PARMRK    %4.4x\n", I_PARMRK(tty));
236          len += sprintf(buf + len, "I_INPCK     %4.4x\n", I_INPCK(tty));
237          len += sprintf(buf + len, "I_ISTRIP    %4.4x\n", I_ISTRIP(tty));
238          len += sprintf(buf + len, "I_INLCR     %4.4x\n", I_INLCR(tty));
239          len += sprintf(buf + len, "I_IGNCR     %4.4x\n", I_IGNCR(tty));
240          len += sprintf(buf + len, "I_ICRNL     %4.4x\n", I_ICRNL(tty));
241          len += sprintf(buf + len, "I_IUCLC     %4.4x\n", I_IUCLC(tty));
242          len += sprintf(buf + len, "I_IXON      %4.4x\n", I_IXON(tty));
243          len += sprintf(buf + len, "I_IXANY     %4.4x\n", I_IXANY(tty));
244          len += sprintf(buf + len, "I_IXOFF     %4.4x\n", I_IXOFF(tty));
245          len += sprintf(buf + len, "I_IMAXBEL   %4.4x\n", I_IMAXBEL(tty));
246
247          len += sprintf(buf + len, "O_OPOST     %4.4x\n", O_OPOST(tty));
248          len += sprintf(buf + len, "O_OLCUC     %4.4x\n", O_OLCUC(tty));
249          len += sprintf(buf + len, "O_ONLCR     %4.4x\n", O_ONLCR(tty));
250          len += sprintf(buf + len, "O_OCRNL     %4.4x\n", O_OCRNL(tty));
251          len += sprintf(buf + len, "O_ONOCR     %4.4x\n", O_ONOCR(tty));
252          len += sprintf(buf + len, "O_ONLRET    %4.4x\n", O_ONLRET(tty));
253          len += sprintf(buf + len, "O_OFILL     %4.4x\n", O_OFILL(tty));
254          len += sprintf(buf + len, "O_OFDEL     %4.4x\n", O_OFDEL(tty));
255          len += sprintf(buf + len, "O_NLDLY     %4.4x\n", O_NLDLY(tty));
256          len += sprintf(buf + len, "O_CRDLY     %4.4x\n", O_CRDLY(tty));
257          len += sprintf(buf + len, "O_TABDLY    %4.4x\n", O_TABDLY(tty));
258          len += sprintf(buf + len, "O_BSDLY     %4.4x\n", O_BSDLY(tty));
259          len += sprintf(buf + len, "O_VTDLY     %4.4x\n", O_VTDLY(tty));
260          len += sprintf(buf + len, "O_FFDLY     %4.4x\n", O_FFDLY(tty));
261
262          len += sprintf(buf + len, "C_BAUD      %4.4x\n", C_BAUD(tty));
263          len += sprintf(buf + len, "C_CSIZE     %4.4x\n", C_CSIZE(tty));
264          len += sprintf(buf + len, "C_CSTOPB    %4.4x\n", C_CSTOPB(tty));
265          len += sprintf(buf + len, "C_CREAD     %4.4x\n", C_CREAD(tty));
266          len += sprintf(buf + len, "C_PARENB    %4.4x\n", C_PARENB(tty));
267          len += sprintf(buf + len, "C_PARODD    %4.4x\n", C_PARODD(tty));
268          len += sprintf(buf + len, "C_HUPCL     %4.4x\n", C_HUPCL(tty));
269          len += sprintf(buf + len, "C_CLOCAL    %4.4x\n", C_CLOCAL(tty));
270          len += sprintf(buf + len, "C_CRTSCTS   %4.4x\n", C_CRTSCTS(tty));
```

```
271
272         len += sprintf(buf + len, "L_ISIG    %4.4x\n", L_ISIG(tty));
273         len += sprintf(buf + len, "L_ICANON   %4.4x\n", L_ICANON(tty));
274         len += sprintf(buf + len, "L_XCASE    %4.4x\n", L_XCASE(tty));
275         len += sprintf(buf + len, "L_ECHO     %4.4x\n", L_ECHO(tty));
276         len += sprintf(buf + len, "L_ECHOE    %4.4x\n", L_ECHOE(tty));
277         len += sprintf(buf + len, "L_ECHOK    %4.4x\n", L_ECHOK(tty));
278         len += sprintf(buf + len, "L_ECHONL   %4.4x\n", L_ECHONL(tty));
279         len += sprintf(buf + len, "L_NOFLSH   %4.4x\n", L_NOFLSH(tty));
280         len += sprintf(buf + len, "L_TOSTOP   %4.4x\n", L_TOSTOP(tty));
281         len += sprintf(buf + len, "L_ECHOCTL  %4.4x\n", L_ECHOCTL(tty));
282         len += sprintf(buf + len, "L_ECHOPRT  %4.4x\n", L_ECHOPRT(tty));
283         len += sprintf(buf + len, "L_ECHOKE   %4.4x\n", L_ECHOKE(tty));
284         len += sprintf(buf + len, "L_FLUSHO   %4.4x\n", L_FLUSHO(tty));
285         len += sprintf(buf + len, "L_PENDIN   %4.4x\n", L_PENDIN(tty));
286         len += sprintf(buf + len, "L_IEXTEN   %4.4x\n", L_IEXTEN(tty));
287
288         *eof = 1;
289         return len;
290 }
291
292 /*
293  * Handle writes to our proc file system.  Right now just turns on and off
294  * our debug flag
295  */
296 static int proc_write(struct file *file, const char *buffer,
297                       unsigned long count, void *data)
298 {
299         if (count) {
300                 if (buffer[0] == '1') {
301                         printk("viocons: debugging on\n");
302                         debug = 1;
303                 } else {
304                         printk("viocons: debugging off\n");
305                         debug = 0;
306                 }
307         }
308         return count;
309 }
310
311 /*
312  * setup our proc file system entries
313  */
314 void viocons_proc_init(struct proc_dir_entry *iSeries_proc)
315 {
316         struct proc_dir_entry *ent;
317         ent =
318             create_proc_entry("viocons", S_IFREG | S_IRUSR, iSeries_proc);
319         if (!ent)
320                 return;
321         ent->nlink = 1;
322         ent->data = NULL;
323         ent->read_proc = proc_read;
324         ent->write_proc = proc_write;
325 }
326
327 /*
328  * clean up our proc file system entries
329  */
330 void viocons_proc_delete(struct proc_dir_entry *iSeries_proc)
331 {
332         remove_proc_entry("viocons", iSeries_proc);
333 }
334
335 /*
336  * Add data to our pending-send buffers.
337  *
338  * NOTE: Don't use printk in here because it gets nastily recursive.  hvlog can be
339  * used to log to the hypervisor buffer
340  */
341 static int bufferAdd(u8 port, const char *buf, size_t len, int userFlag)
342 {
343         size_t bleft = len;
344         size_t curlen;
345         char *cbuf = (char *) buf;
346         int nextbuf;
347         struct overflowBuffers *pov = &overflow[port];
348         while (bleft > 0) {
349                 /* If there is no space left in the current buffer, we have
350                  * filled everything up, so return.  If we filled the previous
351                  * buffer we would already have moved to the next one.
352                  */
353                 if (pov->bufferBytes[pov->curbuf] == OVERFLOW_SIZE) {
354                         hvlog("buffer %d full.  no more space\n",
355                               pov->curbuf);
356                         pov->bufferOverflow++;
357                         pov->overflowMessage = 1;
358                         return len - bleft;
359                 }
360
```

```
361                     /* Turn on the "used" bit for this buffer.  If it's already on, that's
362                      * fine.
363                      */
364                     set_bit(pov->curbuf, &pov->bufferUsed);
365
366                     /*
367                      * See if this buffer has been allocated.  If not, allocate it
368                      */
369                     if (pov->buffer[pov->curbuf] == NULL)
370                             pov->buffer[pov->curbuf] =
371                                 kmalloc(OVERFLOW_SIZE, GFP_ATOMIC);
372
373                     /*
374                      * Figure out how much we can copy into this buffer
375                      */
376                     if (bleft <
377                         (OVERFLOW_SIZE - pov->bufferBytes[pov->curbuf]))
378                             curlen = bleft;
379                     else
380                             curlen =
381                                 OVERFLOW_SIZE - pov->bufferBytes[pov->curbuf];
382
383                     /*
384                      * Copy the data into the buffer
385                      */
386                     if (userFlag)
387                             copy_from_user(pov->buffer[pov->curbuf] +
388                                            pov->bufferBytes[pov->curbuf], cbuf,
389                                            curlen);
390                     else
391                             memcpy(pov->buffer[pov->curbuf] +
392                                    pov->bufferBytes[pov->curbuf], cbuf,
393                                    curlen);
394
395                     pov->bufferBytes[pov->curbuf] += curlen;
396                     cbuf += curlen;
397                     bleft -= curlen;
398
399                     /*
400                      * Now see if we've filled this buffer
401                      */
402                     if (pov->bufferBytes[pov->curbuf] == OVERFLOW_SIZE) {
403                             nextbuf = (pov->curbuf + 1) % NUM_BUF;
404
405                             /*
406                              * Move to the next buffer if it hasn't been used yet
407                              */
408                             if (test_bit(nextbuf, &pov->bufferUsed) == 0) {
409                                     pov->curbuf = nextbuf;
410                             }
411                     }
412             }
413             return len;
414     }
415
416     /* Send pending data
417      *
418      * NOTE: Don't use printk in here because it gets nastily recursive.  hvlog can be
419      * used to log to the hypervisor buffer
420      */
421     void sendBuffers(u8 port, HvLpIndex lp)
422     {
423             HvLpEvent_Rc hvrc;
424             int nextbuf;
425             struct viocharlpevent *viochar;
426             unsigned long flags;
427             struct overflowBuffers *pov = &overflow[port];
428
429             spin_lock_irqsave(&consolelock, flags);
430
431             viochar = (struct viocharlpevent *)
432                 vio_get_event_buffer(viomajorsubtype_chario);
433
434             /* Make sure we got a buffer
435              */
436             if (viochar == NULL) {
437                     hvlog("Yikes...can't get viochar buffer");
438                     spin_unlock_irqrestore(&consolelock, flags);
439                     return;
440             }
441
442             if (pov->bufferUsed == 0) {
443                     hvlog("in sendbuffers, but no buffers used\n");
444                     vio_free_event_buffer(viomajorsubtype_chario, viochar);
445                     spin_unlock_irqrestore(&consolelock, flags);
446                     return;
447             }
448
449             /*
450              * curbuf points to the buffer we're filling.  We want to start sending AFTER
```

```
451             * this one.
452             */
453            nextbuf = (pov->curbuf + 1) % NUM_BUF;
454
455            /*
456             * Loop until we find a buffer with the bufferUsed bit on
457             */
458            while (test_bit(nextbuf, &pov->bufferUsed) == 0)
459                    nextbuf = (nextbuf + 1) % NUM_BUF;
460
461            initDataEvent(viochar, lp);
462
463            /*
464             * While we have buffers with data, and our send window is open, send them
465             */
466            while ((test_bit(nextbuf, &pov->bufferUsed)) &&
467                   ((sndMsgSeq[port] - sndMsgAck[port]) < viochar_window)) {
468                    viochar->immediateDataLen = pov->bufferBytes[nextbuf];
469                    viochar->event.xCorrelationToken = sndMsgSeq[port]++;
470                    viochar->event.xSizeMinus1 =
471                        offsetof(struct viocharlpevent,
472                                 immediateData) + viochar->immediateDataLen;
473
474                    memcpy(viochar->immediateData, pov->buffer[nextbuf],
475                           viochar->immediateDataLen);
476
477                    hvrc = HvCallEvent_signalLpEvent(&viochar->event);
478                    if (hvrc) {
479                            /*
480                             * MUST unlock the spinlock before doing a printk
481                             */
482                            vio_free_event_buffer(viomajorsubtype_chario,
483                                                  viochar);
484                            spin_unlock_irqrestore(&consolelock, flags);
485
486                            printk(KERN_WARNING_VIO
487                                   "console error sending event! return code %d\n",
488                                   (int) hvrc);
489                            return;
490                    }
491
492                    /*
493                     * clear the bufferUsed bit, zero the number of bytes in this buffer,
494                     * and move to the next buffer
495                     */
496                    clear_bit(nextbuf, &pov->bufferUsed);
497                    pov->bufferBytes[nextbuf] = 0;
498                    nextbuf = (nextbuf + 1) % NUM_BUF;
499            }
500
501
502            /*
503             * If we have emptied all the buffers, start at 0 again.
504             * this will re-use any allocated buffers
505             */
506            if (pov->bufferUsed == 0) {
507                    pov->curbuf = 0;
508
509                    if (pov->overflowMessage)
510                            pov->overflowMessage = 0;
511
512                    if (port_info[port].tty) {
513                            if ((port_info[port].tty->
514                                 flags & (1 << TTY_DO_WRITE_WAKEUP))
515                                && (port_info[port].tty->ldisc.write_wakeup))
516                                    (port_info[port].tty->ldisc.
517                                     write_wakeup) (port_info[port].tty);
518                            wake_up_interruptible(&port_info[port].tty->
519                                                  write_wait);
520                    }
521            }
522
523            vio_free_event_buffer(viomajorsubtype_chario, viochar);
524            spin_unlock_irqrestore(&consolelock, flags);
525
526    }
527
528    /* Our internal writer.  Gets called both from the console device and
529     * the tty device.  the tty pointer will be NULL if called from the console.
530     *
531     * NOTE: Don't use printk in here because it gets nastily recursive.  hvlog can be
532     * used to log to the hypervisor buffer
533     */
534    static int internal_write(struct tty_struct *tty, const char *buf,
535                              size_t len, int userFlag)
536    {
537            HvLpEvent_Rc hvrc;
538            size_t bleft = len;
539            size_t curlen;
540            const char *curbuf = buf;
```

```
541         struct viocharlpevent *viochar;
542         unsigned long flags;
543         struct port_info_tag *pi = NULL;
544         HvLpIndex lp;
545         u8 port;
546
547         if (tty) {
548                 pi = (struct port_info_tag *) tty->driver_data;
549
550                 if (!pi
551                     || viotty_paranoia_check(pi, tty->device,
552                                              "viotty_internal_write"))
553                         return -ENODEV;
554
555                 lp = pi->lp;
556                 port = pi->port;
557         } else {
558                 /* If this is the console device, use the lp from the first port entry
559                  */
560                 port = 0;
561                 lp = port_info[0].lp;
562         }
563
564         /* Always put console output in the hypervisor console log
565          */
566         if (port == 0)
567                 HvCall_writeLogBuffer(buf, len);
568
569         /* If the path to this LP is closed, don't bother doing anything more.
570          * just dump the data on the floor
571          */
572         if (!viopath_isactive(lp))
573                 return len;
574
575         /*
576          * If there is already data queued for this port, send it
577          */
578         if (overflow[port].bufferUsed)
579                 sendBuffers(port, lp);
580
581         spin_lock_irqsave(&consolelock, flags);
582
583         viochar = (struct viocharlpevent *)
584             vio_get_event_buffer(viomajorsubtype_chario);
585         /* Make sure we got a buffer
586          */
587         if (viochar == NULL) {
588                 hvlog("Yikes...can't get viochar buffer");
589                 spin_unlock_irqrestore(&consolelock, flags);
590                 return -1;
591         }
592
593         initDataEvent(viochar, lp);
594
595         /* Got the lock, don't cause console output */
596         while ((bleft > 0) &&
597                (overflow[port].bufferUsed == 0) &&
598                ((sndMsgSeq[port] - sndMsgAck[port]) < viochar_window)) {
599                 if (bleft > VIOCHAR_MAX_DATA)
600                         curlen = VIOCHAR_MAX_DATA;
601                 else
602                         curlen = bleft;
603
604                 viochar->immediateDataLen = curlen;
605                 viochar->event.xCorrelationToken = sndMsgSeq[port]++;
606
607                 if (userFlag)
608                         copy_from_user(viochar->immediateData, curbuf,
609                                        curlen);
610                 else
611                         memcpy(viochar->immediateData, curbuf, curlen);
612
613                 viochar->event.xSizeMinus1 =
614                     offsetof(struct viocharlpevent,
615                              immediateData) + curlen;
616
617                 hvrc = HvCallEvent_signalLpEvent(&viochar->event);
618                 if (hvrc) {
619                         /*
620                          * MUST unlock the spinlock before doing a printk
621                          */
622                         vio_free_event_buffer(viomajorsubtype_chario,
623                                               viochar);
624                         spin_unlock_irqrestore(&consolelock, flags);
625
626                         hvlog("viocons: error sending event! %d\n",
627                               (int) hvrc);
628                         return len - bleft;
629                 }
630
```

```
631                     curbuf += curlen;
632                     bleft −= curlen;
633             }
634
635             /*
636              * If we didn't send it all, buffer it
637              */
638             if (bleft > 0) {
639                     bleft −= bufferAdd(port, curbuf, bleft, userFlag);
640             }
641             vio_free_event_buffer(viomajorsubtype_chario, viochar);
642             spin_unlock_irqrestore(&consolelock, flags);
643
644             return len − bleft;
645     }
646
647     /* Initialize the common fields in a charLpEvent
648      */
649     static void initDataEvent(struct viocharlpevent *viochar, HvLpIndex lp)
650     {
651             memset(viochar, 0x00, sizeof(struct viocharlpevent));
652
653             viochar−>event.xFlags.xValid = 1;
654             viochar−>event.xFlags.xFunction = HvLpEvent_Function_Int;
655             viochar−>event.xFlags.xAckInd = HvLpEvent_AckInd_NoAck;
656             viochar−>event.xFlags.xAckType = HvLpEvent_AckType_DeferredAck;
657             viochar−>event.xType = HvLpEvent_Type_VirtualIo;
658             viochar−>event.xSubtype = viomajorsubtype_chario | viochardata;
659             viochar−>event.xSourceLp = HvLpConfig_getLpIndex();
660             viochar−>event.xTargetLp = lp;
661             viochar−>event.xSizeMinus1 = sizeof(struct viocharlpevent);
662             viochar−>event.xSourceInstanceId = viopath_sourceinst(lp);
663             viochar−>event.xTargetInstanceId = viopath_targetinst(lp);
664     }
665
666
667     /* console device write
668      */
669     static void viocons_write(struct console *co, const char *s,
670                               unsigned count)
671     {
672             /* This parser will ensure that all single instances of either \n or \r are
673              * matched into carriage return/line feed combinations.  It also allows for
674              * instances where there already exist \n\r combinations as well as the
675              * reverse, \r\n combinations.
676              */
677
678             int index;
679             char charptr[1];
680             int foundcr;
681             int slicebegin;
682             int sliceend;
683
684             foundcr = 0;
685             slicebegin = 0;
686             sliceend = 0;
687
688             for (index = 0; index < count; index++) {
689                     if (!foundcr && s[index] == 0x0a) {
690                             if ((slicebegin − sliceend > 0)
691                                 && sliceend < count) {
692                                     internal_write(NULL, &s[slicebegin],
693                                                    sliceend − slicebegin, 0);
694                                     slicebegin = sliceend;
695                             }
696                             charptr[0] = '\r';
697                             internal_write(NULL, charptr, 1, 0);
698                     }
699                     if (foundcr && s[index] != 0x0a) {
700                             if ((index − 2) >= 0) {
701                                     if (s[index − 2] != 0x0a) {
702                                             internal_write(NULL,
703                                                            &s[slicebegin],
704                                                            sliceend −
705                                                            slicebegin, 0);
706                                             slicebegin = sliceend;
707                                             charptr[0] = '\n';
708                                             internal_write(NULL, charptr, 1,
709                                                            0);
710                                     }
711                             }
712                     }
713                     sliceend++;
714
715                     if (s[index] == 0x0d)
716                             foundcr = 1;
717                     else
718                             foundcr = 0;
719             }
720
```

```
721         internal_write(NULL, &s[slicebegin], sliceend − slicebegin, 0);
722
723         if (count > 1) {
724                 if (foundcr == 1 && s[count − 1] != 0x0a) {
725                         charptr[0] = '\n';
726                         internal_write(NULL, charptr, 1, 0);
727                 } else if (s[count − 1] == 0x0a && s[count − 2] != 0x0d) {
728
729                         charptr[0] = '\r';
730                         internal_write(NULL, charptr, 1, 0);
731                 }
732         }
733 }
734
735 /* Work out a the device associate with this console
736  */
737 static kdev_t viocons_device(struct console *c)
738 {
739         return MKDEV(TTY_MAJOR, c−>index + viotty_driver.minor_start);
740 }
741
742 /* console device read method
743  */
744 static int viocons_read(struct console *co, const char *s, unsigned count)
745 {
746         printk(KERN_DEBUG_VIO "viocons_read\n");
747         // Implement me
748         interruptible_sleep_on(&viocons_wait_queue);
749         return 0;
750 }
751
752 /* Do console device setup
753  */
754 static int __init viocons_setup(struct console *co, char *options)
755 {
756         return 0;
757 }
758
759 /* console device I/O methods
760  */
761 static struct console viocons = {
762         name:"ttyS",
763         write:viocons_write,
764         read:viocons_read,
765         device:viocons_device,
766         setup:viocons_setup,
767         flags:CON_PRINTBUFFER,
768 };
769
770
771 /* TTY Open method
772  */
773 static int viotty_open(struct tty_struct *tty, struct file *filp)
774 {
775         int port;
776         unsigned long flags;
777         MOD_INC_USE_COUNT;
778         port = MINOR(tty−>device) − tty−>driver.minor_start;
779
780         if (port >= VIOTTY_SERIAL_START)
781                 port −= VIOTTY_SERIAL_START;
782
783         if ((port < 0) || (port >= VTTY_PORTS)) {
784                 MOD_DEC_USE_COUNT;
785                 return −ENODEV;
786         }
787
788         spin_lock_irqsave(&consolelock, flags);
789
790         /*
791          * If some other TTY is already connected here, reject the open
792          */
793         if ((port_info[port].tty) && (port_info[port].tty != tty)) {
794                 spin_unlock_irqrestore(&consolelock, flags);
795                 MOD_DEC_USE_COUNT;
796                 printk(KERN_WARNING_VIO
797                         "console attempt to open device twice from different ttys\n");
798                 return −EBUSY;
799         }
800         tty−>driver_data = &port_info[port];
801         port_info[port].tty = tty;
802         spin_unlock_irqrestore(&consolelock, flags);
803
804         return 0;
805 }
806
807 /* TTY Close method
808  */
809 static void viotty_close(struct tty_struct *tty, struct file *filp)
810 {
```

```
811         unsigned long flags;
812         struct port_info_tag *pi =
813             (struct port_info_tag *) tty->driver_data;
814
815         if (!pi || viotty_paranoia_check(pi, tty->device, "viotty_close"))
816                 return;
817
818         spin_lock_irqsave(&consolelock, flags);
819         if (tty->count == 1) {
820                 pi->tty = NULL;
821         }
822
823         spin_unlock_irqrestore(&consolelock, flags);
824
825         MOD_DEC_USE_COUNT;
826 }
827
828 /* TTY Write method
829  */
830 static int viotty_write(struct tty_struct *tty, int from_user,
831                         const unsigned char *buf, int count)
832 {
833         return internal_write(tty, buf, count, from_user);
834 }
835
836 /* TTY put_char method
837  */
838 static void viotty_put_char(struct tty_struct *tty, unsigned char ch)
839 {
840         internal_write(tty, &ch, 1, 0);
841 }
842
843 /* TTY flush_chars method
844  */
845 static void viotty_flush_chars(struct tty_struct *tty)
846 {
847 }
848
849 /* TTY write_room method
850  */
851 static int viotty_write_room(struct tty_struct *tty)
852 {
853         int i;
854         int room = 0;
855         struct port_info_tag *pi =
856             (struct port_info_tag *) tty->driver_data;
857
858         if (!pi
859             || viotty_paranoia_check(pi, tty->device,
860                                      "viotty_sendbuffers"))
861                 return 0;
862
863         // If no buffers are used, return the max size
864         if (overflow[pi->port].bufferUsed == 0)
865                 return VIOCHAR_MAX_DATA * NUM_BUF;
866
867         for (i = 0; ((i < NUM_BUF) && (room < VIOCHAR_MAX_DATA)); i++) {
868                 room +=
869                     (OVERFLOW_SIZE - overflow[pi->port].bufferBytes[i]);
870         }
871
872         if (room > VIOCHAR_MAX_DATA)
873                 return VIOCHAR_MAX_DATA;
874         else
875                 return room;
876 }
877
878 /* TTY chars_in_buffer_room method
879  */
880 static int viotty_chars_in_buffer(struct tty_struct *tty)
881 {
882         return 0;
883 }
884
885 static void viotty_flush_buffer(struct tty_struct *tty)
886 {
887 }
888
889 static int viotty_ioctl(struct tty_struct *tty, struct file *file,
890                         unsigned int cmd, unsigned long arg)
891 {
892         switch (cmd) {
893                 /* the ioctls below read/set the flags usually shown in the leds */
894                 /* don't use them - they will go away without warning */
895         case KDGETLED:
896         case KDGKBLED:
897                 return put_user(0, (char *) arg);
898
899         case KDSKBLED:
900                 return 0;
```

```
901         }
902
903         return n_tty_ioctl(tty, file, cmd, arg);
904 }
905
906 static void viotty_throttle(struct tty_struct *tty)
907 {
908 }
909
910 static void viotty_unthrottle(struct tty_struct *tty)
911 {
912 }
913
914 static void viotty_set_termios(struct tty_struct *tty,
915                                struct termios *old_termios)
916 {
917 }
918
919 static void viotty_stop(struct tty_struct *tty)
920 {
921 }
922
923 static void viotty_start(struct tty_struct *tty)
924 {
925 }
926
927 static void viotty_hangup(struct tty_struct *tty)
928 {
929 }
930
931 static void viotty_break(struct tty_struct *tty, int break_state)
932 {
933 }
934
935 static void viotty_send_xchar(struct tty_struct *tty, char ch)
936 {
937 }
938
939 static void viotty_wait_until_sent(struct tty_struct *tty, int timeout)
940 {
941 }
942
943 /* Handle an open charLpEvent.  Could be either interrupt or ack
944  */
945 static void vioHandleOpenEvent(struct HvLpEvent *event)
946 {
947         unsigned long flags;
948         u8 eventRc;
949         u16 eventSubtypeRc;
950         struct viocharlpevent *cevent = (struct viocharlpevent *) event;
951         u8 port = cevent->virtualDevice;
952
953         if (event->xFlags.xFunction == HvLpEvent_Function_Ack) {
954                 if (port >= VTTY_PORTS)
955                         return;
956
957                 spin_lock_irqsave(&consolelock, flags);
958                 /* Got the lock, don't cause console output */
959
960                 if (event->xRc == HvLpEvent_Rc_Good) {
961                         sndMsgSeq[port] = sndMsgAck[port] = 0;
962                 }
963
964                 port_info[port].lp = event->xTargetLp;
965
966                 spin_unlock_irqrestore(&consolelock, flags);
967
968                 if (event->xCorrelationToken != 0) {
969                         unsigned long semptr = event->xCorrelationToken;
970                         up((struct semaphore *) semptr);
971                 } else
972                         printk(KERN_WARNING_VIO
973                                "console: wierd...got open ack without semaphore\n");
974         } else {
975                 /* This had better require an ack, otherwise complain
976                  */
977                 if (event->xFlags.xAckInd != HvLpEvent_AckInd_DoAck) {
978                         printk(KERN_WARNING_VIO
979                                "console: viocharopen without ack bit!\n");
980                         return;
981                 }
982
983                 spin_lock_irqsave(&consolelock, flags);
984                 /* Got the lock, don't cause console output */
985
986                 /* Make sure this is a good virtual tty */
987                 if (port >= VTTY_PORTS) {
988                         eventRc = HvLpEvent_Rc_SubtypeError;
989                         eventSubtypeRc = viorc_openRejected;
990                 }
```

```
991
992                 /* If this is tty is already connected to a different
993                    partition, fail */
994                 else if ((port_info[port].lp != HvLpIndexInvalid) &&
995                          (port_info[port].lp != event->xSourceLp)) {
996                         eventRc = HvLpEvent_Rc_SubtypeError;
997                         eventSubtypeRc = viorc_openRejected;
998                 } else {
999                         port_info[port].lp = event->xSourceLp;
1000                        eventRc = HvLpEvent_Rc_Good;
1001                        eventSubtypeRc = viorc_good;
1002                        sndMsgSeq[port] = sndMsgAck[port] = 0;
1003                }
1004
1005                spin_unlock_irqrestore(&consolelock, flags);
1006
1007                /* Return the acknowledgement */
1008                HvCallEvent_ackLpEvent(event);
1009        }
1010 }
1011
1012 /* Handle a close open charLpEvent.  Could be either interrupt or ack
1013  */
1014 static void vioHandleCloseEvent(struct HvLpEvent *event)
1015 {
1016        unsigned long flags;
1017        struct viocharlpevent *cevent = (struct viocharlpevent *) event;
1018        u8 port = cevent->virtualDevice;
1019
1020        if (event->xFlags.xFunction == HvLpEvent_Function_Int) {
1021                if (port >= VTTY_PORTS)
1022                        return;
1023
1024                /* For closes, just mark the console partition invalid */
1025                spin_lock_irqsave(&consolelock, flags);
1026                /* Got the lock, don't cause console output */
1027
1028                if (port_info[port].lp == event->xSourceLp)
1029                        port_info[port].lp = HvLpIndexInvalid;
1030
1031                spin_unlock_irqrestore(&consolelock, flags);
1032                printk(KERN_INFO_VIO
1033                       "console close from %d\n", event->xSourceLp);
1034        } else {
1035                printk(KERN_WARNING_VIO
1036                       "console got unexpected close acknowlegement\n");
1037        }
1038 }
1039
1040 /* Handle a config charLpEvent.  Could be either interrupt or ack
1041  */
1042 static void vioHandleConfig(struct HvLpEvent *event)
1043 {
1044        struct viocharlpevent *cevent = (struct viocharlpevent *) event;
1045        int len;
1046
1047        len = cevent->immediateDataLen;
1048        HvCall_writeLogBuffer(cevent->immediateData,
1049                              cevent->immediateDataLen);
1050
1051        if (cevent->immediateData[0] == 0x01) {
1052                printk(KERN_INFO_VIO
1053                       "console window resized to %d: %d: %d: %d\n",
1054                       cevent->immediateData[1],
1055                       cevent->immediateData[2],
1056                       cevent->immediateData[3], cevent->immediateData[4]);
1057        } else {
1058                printk(KERN_WARNING_VIO "console unknown config event\n");
1059        }
1060        return;
1061 }
1062
1063 /* Handle a data charLpEvent.
1064  */
1065 static void vioHandleData(struct HvLpEvent *event)
1066 {
1067        struct tty_struct *tty;
1068        struct viocharlpevent *cevent = (struct viocharlpevent *) event;
1069        struct port_info_tag *pi;
1070        int len;
1071        u8 port = cevent->virtualDevice;
1072
1073        if (port >= VTTY_PORTS) {
1074                printk(KERN_WARNING_VIO
1075                       "console data on invalid virtual device %d\n",
1076                       port);
1077                return;
1078        }
1079
1080        tty = port_info[port].tty;
```

```
1081
1082            if (tty == NULL) {
1083                    printk(KERN_WARNING_VIO
1084                           "no tty for virtual device %d\n", port);
1085                    return;
1086            }
1087
1088            if (tty->magic != TTY_MAGIC) {
1089                    printk(KERN_WARNING_VIO "tty bad magic\n");
1090                    return;
1091            }
1092
1093            /*
1094             * Just to be paranoid, make sure the tty points back to this port
1095             */
1096            pi = (struct port_info_tag *) tty->driver_data;
1097
1098            if (!pi || viotty_paranoia_check(pi, tty->device, "vioHandleData"))
1099                    return;
1100
1101            len = cevent->immediateDataLen;
1102
1103            if (len == 0)
1104                    return;
1105
1106            /* Don't log the user's input to the hypervisor log or their password
1107             * will appear on a hypervisor log display.
1108             */
1109
1110            /* Don't copy more bytes than there is room for in the buffer */
1111            if (tty->flip.count + len > TTY_FLIPBUF_SIZE) {
1112                    len = TTY_FLIPBUF_SIZE - tty->flip.count;
1113                    printk(KERN_WARNING_VIO
1114                           "console input buffer overflow!\n");
1115            }
1116
1117            memcpy(tty->flip.char_buf_ptr, cevent->immediateData, len);
1118            memset(tty->flip.flag_buf_ptr, TTY_NORMAL, len);
1119
1120            /* Update the kernel buffer end */
1121            tty->flip.count += len;
1122            tty->flip.char_buf_ptr += len;
1123
1124            tty->flip.flag_buf_ptr += len;
1125
1126            tty_flip_buffer_push(tty);
1127    }
1128
1129    /* Handle an ack charLpEvent.
1130     */
1131    static void vioHandleAck(struct HvLpEvent *event)
1132    {
1133            struct viocharlpevent *cevent = (struct viocharlpevent *) event;
1134            unsigned long flags;
1135            u8 port = cevent->virtualDevice;
1136
1137            if (port >= VTTY_PORTS) {
1138                    printk(KERN_WARNING_VIO
1139                           "viocons: data on invalid virtual device\n");
1140                    return;
1141            }
1142
1143            spin_lock_irqsave(&consolelock, flags);
1144            sndMsgAck[port] = event->xCorrelationToken;
1145            spin_unlock_irqrestore(&consolelock, flags);
1146
1147            if (overflow[port].bufferUsed)
1148                    sendBuffers(port, port_info[port].lp);
1149    }
1150
1151    /* Handle charLpEvents and route to the appropriate routine
1152     */
1153    static void vioHandleCharEvent(struct HvLpEvent *event)
1154    {
1155            int charminor;
1156
1157            if (event == NULL) {
1158                    return;
1159            }
1160            charminor = event->xSubtype & VIOMINOR_SUBTYPE_MASK;
1161            switch (charminor) {
1162            case viocharopen:
1163                    vioHandleOpenEvent(event);
1164                    break;
1165            case viocharclose:
1166                    vioHandleCloseEvent(event);
1167                    break;
1168            case viochardata:
1169                    vioHandleData(event);
1170                    break;
```

```
1171          case viocharack:
1172                  vioHandleAck(event);
1173                  break;
1174          case viocharconfig:
1175                  vioHandleConfig(event);
1176                  break;
1177          default:
1178                  if ((event->xFlags.xFunction == HvLpEvent_Function_Int) &&
1179                      (event->xFlags.xAckInd == HvLpEvent_AckInd_DoAck)) {
1180                          event->xRc = HvLpEvent_Rc_InvalidSubtype;
1181                          HvCallEvent_ackLpEvent(event);
1182                  }
1183          }
1184  }
1185
1186  /* Send an open event
1187   */
1188  static int viocons_sendOpen(HvLpIndex remoteLp, u8 port, void *sem)
1189  {
1190          return HvCallEvent_signalLpEventFast(remoteLp,
1191                                               HvLpEvent_Type_VirtualIo,
1192                                               viomajorsubtype_chario
1193                                               | viocharopen,
1194                                               HvLpEvent_AckInd_DoAck,
1195                                               HvLpEvent_AckType_ImmediateAck,
1196                                               viopath_sourceinst
1197                                               (remoteLp),
1198                                               viopath_targetinst
1199                                               (remoteLp),
1200                                               (u64) (unsigned long)
1201                                               sem, VIOVERSION << 16,
1202                                               ((u64) port << 48), 0, 0, 0);
1203
1204  }
1205
1206  int __init viocons_init2(void)
1207  {
1208          DECLARE_MUTEX_LOCKED(Semaphore);
1209          int rc;
1210
1211          /*
1212           * Now open to the primary LP
1213           */
1214          printk(KERN_INFO_VIO "console open path to primary\n");
1215          rc = viopath_open(HvLpConfig_getPrimaryLpIndex(), viomajorsubtype_chario, viochar_window + 2);  /* +2 for
      fudge */
1216          if (rc) {
1217                  printk(KERN_WARNING_VIO
1218                         "console error opening to primary %d\n", rc);
1219          }
1220
1221          if (viopath_hostLp == HvLpIndexInvalid) {
1222                  vio_set_hostlp();
1223          }
1224
1225          /*
1226           * And if the primary is not the same as the hosting LP, open to the
1227           * hosting lp
1228           */
1229          if ((viopath_hostLp != HvLpIndexInvalid) &&
1230              (viopath_hostLp != HvLpConfig_getPrimaryLpIndex())) {
1231                  printk(KERN_INFO_VIO
1232                         "console open path to hosting (%d)\n",
1233                         viopath_hostLp);
1234                  rc = viopath_open(viopath_hostLp, viomajorsubtype_chario, viochar_window + 2);  /* +2 for fudge *
     /
1235                  if (rc) {
1236                          printk(KERN_WARNING_VIO
1237                                 "console error opening to partition %d: %d\n",
1238                                 viopath_hostLp, rc);
1239                  }
1240          }
1241
1242          if (vio_setHandler(viomajorsubtype_chario, vioHandleCharEvent) < 0) {
1243                  printk(KERN_WARNING_VIO
1244                         "Error seting handler for console events!\n");
1245          }
1246
1247          printk(KERN_INFO_VIO "console major number is %d\n", TTY_MAJOR);
1248
1249          /* First, try to open the console to the hosting lp.
1250           * Wait on a semaphore for the response.
1251           */
1252          if ((viopath_isactive(viopath_hostLp)) &&
1253              (viocons_sendOpen(viopath_hostLp, 0, &Semaphore) == 0)) {
1254                  printk(KERN_INFO_VIO
1255                         "opening console to hosting partition %d\n",
1256                         viopath_hostLp);
1257                  down(&Semaphore);
1258          }
```

```
1259
1260            /*
1261             * If we don't have an active console, try the primary
1262             */
1263            if ((!viopath_isactive(port_info[0].lp)) &&
1264                (viopath_isactive(HvLpConfig_getPrimaryLpIndex())) &&
1265                (viocons_sendOpen
1266                 (HvLpConfig_getPrimaryLpIndex(), 0, &Semaphore) == 0)) {
1267                    printk(KERN_INFO_VIO
1268                           "opening console to primary partition\n");
1269                    down(&Semaphore);
1270            }
1271
1272            /* Initialize the tty_driver structure */
1273            memset(&viotty_driver, 0, sizeof(struct tty_driver));
1274            viotty_driver.magic = TTY_DRIVER_MAGIC;
1275            viotty_driver.driver_name = "vioconsole";
1276    #if defined(CONFIG_DEVFS_FS)
1277            viotty_driver.name = "tty%d";
1278    #else
1279            viotty_driver.name = "tty";
1280    #endif
1281            viotty_driver.major = TTY_MAJOR;
1282            viotty_driver.minor_start = 1;
1283            viotty_driver.name_base = 1;
1284            viotty_driver.num = VTTY_PORTS;
1285            viotty_driver.type = TTY_DRIVER_TYPE_CONSOLE;
1286            viotty_driver.subtype = 1;
1287            viotty_driver.init_termios = tty_std_termios;
1288            viotty_driver.flags =
1289                TTY_DRIVER_REAL_RAW | TTY_DRIVER_RESET_TERMIOS;
1290            viotty_driver.refcount = &viotty_refcount;
1291            viotty_driver.table = viotty_table;
1292            viotty_driver.termios = viotty_termios;
1293            viotty_driver.termios_locked = viotty_termios_locked;
1294
1295            viotty_driver.open = viotty_open;
1296            viotty_driver.close = viotty_close;
1297            viotty_driver.write = viotty_write;
1298            viotty_driver.put_char = viotty_put_char;
1299            viotty_driver.flush_chars = viotty_flush_chars;
1300            viotty_driver.write_room = viotty_write_room;
1301            viotty_driver.chars_in_buffer = viotty_chars_in_buffer;
1302            viotty_driver.flush_buffer = viotty_flush_buffer;
1303            viotty_driver.ioctl = viotty_ioctl;
1304            viotty_driver.throttle = viotty_throttle;
1305            viotty_driver.unthrottle = viotty_unthrottle;
1306            viotty_driver.set_termios = viotty_set_termios;
1307            viotty_driver.stop = viotty_stop;
1308            viotty_driver.start = viotty_start;
1309            viotty_driver.hangup = viotty_hangup;
1310            viotty_driver.break_ctl = viotty_break;
1311            viotty_driver.send_xchar = viotty_send_xchar;
1312            viotty_driver.wait_until_sent = viotty_wait_until_sent;
1313
1314            viottyS_driver = viotty_driver;
1315    #if defined(CONFIG_DEVFS_FS)
1316            viottyS_driver.name = "ttyS%d";
1317    #else
1318            viottyS_driver.name = "ttyS";
1319    #endif
1320            viottyS_driver.major = TTY_MAJOR;
1321            viottyS_driver.minor_start = VIOTTY_SERIAL_START;
1322            viottyS_driver.type = TTY_DRIVER_TYPE_SERIAL;
1323            viottyS_driver.table = viottyS_table;
1324            viottyS_driver.termios = viottyS_termios;
1325            viottyS_driver.termios_locked = viottyS_termios_locked;
1326
1327            if (tty_register_driver(&viotty_driver)) {
1328                    printk(KERN_WARNING_VIO
1329                           "Couldn't register console driver\n");
1330            }
1331
1332            if (tty_register_driver(&viottyS_driver)) {
1333                    printk(KERN_WARNING_VIO
1334                           "Couldn't register console S driver\n");
1335            }
1336            /* Now create the vcs and vcsa devfs entries so mingetty works */
1337    #if defined(CONFIG_DEVFS_FS)
1338            {
1339                    struct tty_driver temp_driver = viotty_driver;
1340                    int i;
1341
1342                    temp_driver.name = "vcs%d";
1343                    for (i = 0; i < VTTY_PORTS; i++)
1344                            tty_register_devfs(&temp_driver,
1345                                               0, i + temp_driver.minor_start);
1346
1347                    temp_driver.name = "vcsa%d";
1348                    for (i = 0; i < VTTY_PORTS; i++)
```

```
1349                            tty_register_devfs(&temp_driver,
1350                                               0, i + temp_driver.minor_start);
1351
1352                    // For compatibility with some earlier code only!
1353                    // This will go away!!!
1354                    temp_driver.name = "viocons/%d";
1355                    temp_driver.name_base = 0;
1356                    for (i = 0; i < VTTY_PORTS; i++)
1357                            tty_register_devfs(&temp_driver,
1358                                               0, i + temp_driver.minor_start);
1359            }
1360    #endif
1361
1362            /*
1363             * Create the proc entry
1364             */
1365            iSeries_proc_callback(&viocons_proc_init);
1366
1367            return 0;
1368    }
1369
1370    void __init viocons_init(void)
1371    {
1372            int i;
1373            printk(KERN_INFO_VIO "registering console\n");
1374
1375            memset(&port_info, 0x00, sizeof(port_info));
1376            for (i = 0; i < VTTY_PORTS; i++) {
1377                    sndMsgSeq[i] = sndMsgAck[i] = 0;
1378                    port_info[i].port = i;
1379                    port_info[i].lp = HvLpIndexInvalid;
1380                    port_info[i].magic = VIOTTY_MAGIC;
1381            }
1382
1383            register_console(&viocons);
1384            memset(overflow, 0x00, sizeof(overflow));
1385            debug = 0;
1386
1387            HvCall_setLogBufferFormatAndCodepage(HvCall_LogBuffer_ASCII, 437);
1388    }
```

```
/* −*− linux−c −*−
 * viodasd.c
 *  Authors: Dave Boutcher <boutcher@us.ibm.com>
 *           Ryan Arnold <ryanarn@us.ibm.com>
 *           Colin Devilbiss <devilbis@us.ibm.com>
 *
 * (C) Copyright 2000 IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 ************************************************************************
 * This routine provides access to disk space (termed "DASD" in historical
 * IBM terms) owned and managed by an OS/400 partition running on the
 * same box as this Linux partition.
 *
 * All disk operations are performed by sending messages back and forth to
 * the OS/400 partition.
 *
 * This device driver can either use its own major number, or it can
 * pretend to be an IDE drive (grep 'IDE[0−9]_MAJOR' ../../include/linux/major.h).
 * This is controlled with a CONFIG option.  You can either call this an
 * elegant solution to the fact that a lot of software doesn't recognize
 * a new disk major number...or you can call this a really ugly hack.
 * Your choice.
 */

#include <linux/major.h>
#include <linux/config.h>

#include <linux/fs.h>
#include <linux/blkpg.h>

/* Changelog:
        2001−11−27      devilbis        Added first pass at complete IDE emulation
        2002−07−07      boutcher        Added randomness
 */

/* Decide if we are using our own major or pretending to be an IDE drive
 *
 * If we are using our own major, we only support 7 partitions per physical
 * disk....so with minor numbers 0−255 we get a maximum of 32 disks.  If we
 * are emulating IDE, we get 63 partitions per disk, with a maximum of 4
 * disks per major, but common practice is to place only 2 devices in /dev
 * for each IDE major, for a total of 20 (since there are 10 IDE majors).
 */

#ifdef CONFIG_VIODASD_IDE
static const int major_table[] = {
        IDE0_MAJOR,
        IDE1_MAJOR,
        IDE2_MAJOR,
        IDE3_MAJOR,
        IDE4_MAJOR,
        IDE5_MAJOR,
        IDE6_MAJOR,
        IDE7_MAJOR,
        IDE8_MAJOR,
        IDE9_MAJOR,
};
enum {
        DEV_PER_MAJOR = 2,
        PARTITION_SHIFT = 6,
};
static int major_to_index(int major)
{
        switch(major) {
        case IDE0_MAJOR: return 0;
        case IDE1_MAJOR: return 1;
        case IDE2_MAJOR: return 2;
        case IDE3_MAJOR: return 3;
        case IDE4_MAJOR: return 4;
        case IDE5_MAJOR: return 5;
        case IDE6_MAJOR: return 6;
        case IDE7_MAJOR: return 7;
        case IDE8_MAJOR: return 8;
        case IDE9_MAJOR: return 9;
        default:
                return −1;
        }
```

```
}
#define do_viodasd_request do_hd_request
#define VIOD_DEVICE_NAME "ide"
#define VIOD_GENHD_NAME "hd"
#else                           /* !CONFIG_VIODASD_IDE */
static const int major_table[] = {
        VIODASD_MAJOR,
};
enum {
        DEV_PER_MAJOR = 32,
        PARTITION_SHIFT = 3,
};
static int major_to_index(int major)
{
        if(major != VIODASD_MAJOR)
                return −1;
        return 0;
}
#define VIOD_DEVICE_NAME "viod"
#ifdef CONFIG_DEVFS_FS
#define VIOD_GENHD_NAME "viod"
#else
#define VIOD_GENHD_NAME "iseries/vd"
#endif
#endif                          /* CONFIG_VIODASD_IDE */

#define DEVICE_NR(dev) (devt_to_diskno(dev))
#define LOCAL_END_REQUEST

#include <linux/sched.h>
#include <linux/timer.h>
#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/blk.h>
#include <linux/genhd.h>
#include <linux/hdreg.h>
#include <linux/fd.h>
#include <linux/proc_fs.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/string.h>
#include <linux/pci.h>

#include <asm/iSeries/HvTypes.h>
#include <asm/iSeries/HvLpEvent.h>
#include <asm/iSeries/HvLpConfig.h>
#include "vio.h"
#include <asm/iSeries/iSeries_proc.h>

MODULE_DESCRIPTION("iSeries Virtual DASD");
MODULE_AUTHOR("Dave Boutcher");
MODULE_LICENSE("GPL");

#define VIODASD_VERS "1.50"

enum {
        NUM_MAJORS = sizeof(major_table) / sizeof(major_table[0]),
        MAX_DISKNO = DEV_PER_MAJOR * NUM_MAJORS,
        MAX_MAJOR_NAME = 16 + 1, /* maximum length of a gendisk->name */
};

static volatile int viodasd_max_disk = MAX_DISKNO − 1;

static int diskno_to_major(int diskno)
{
        if (diskno >= MAX_DISKNO)
                return −1;
        return major_table[diskno / DEV_PER_MAJOR];
}
static int devt_to_diskno(kdev_t dev)
{
        return major_to_index(MAJOR(dev)) * DEV_PER_MAJOR +
            (MINOR(dev) >> PARTITION_SHIFT);
}
static int diskno_to_devt(int diskno, int partition)
{
        return MKDEV(diskno_to_major(diskno),
                     ((diskno % DEV_PER_MAJOR) << PARTITION_SHIFT) +
                     partition);
}

#define VIOMAXREQ 16
#define VIOMAXBLOCKDMA        12

extern struct pci_dev *iSeries_vio_dev;

struct openData {
        u64 mDiskLen;
```

```
181         u16 mMaxDisks;
182         u16 mCylinders;
183         u16 mTracks;
184         u16 mSectors;
185         u16 mBytesPerSector;
186 };
187
188 struct rwData {                 // Used during rw
189         u64 mOffset;
190         struct {
191                 u32 mToken;
192                 u32 reserved;
193                 u64 mLen;
194         } dmaInfo[VIOMAXBLOCKDMA];
195 };
196
197 struct vioblocklpevent {
198         struct HvLpEvent event;
199         u32 mReserved1;
200         u16 mVersion;
201         u16 mSubTypeRc;
202         u16 mDisk;
203         u16 mFlags;
204         union {
205                 struct openData openData;
206                 struct rwData rwData;
207                 struct {
208                         u64 changed;
209                 } check;
210         } u;
211 };
212
213 #define vioblockflags_ro   0x0001
214
215 enum vioblocksubtype {
216         vioblockopen = 0x0001,
217         vioblockclose = 0x0002,
218         vioblockread = 0x0003,
219         vioblockwrite = 0x0004,
220         vioblockflush = 0x0005,
221         vioblockcheck = 0x0007
222 };
223
224 /* In a perfect world we will perform better if we get page-aligned I/O
225  * requests, in multiples of pages.  At least peg our block size to the
226  * actual page size.
227  */
228 static int blksize = HVPAGESIZE;        /* in bytes */
229
230 static DECLARE_WAIT_QUEUE_HEAD(viodasd_wait);
231 struct viodasd_waitevent {
232         struct semaphore *sem;
233         int rc;
234         union {
235                 int changed;    /* Used only for check_change */
236                 u16 subRC;
237         } data;
238 };
239
240 static const struct vio_error_entry viodasd_err_table[] = {
241         {0x0201, EINVAL, "Invalid Range"},
242         {0x0202, EINVAL, "Invalid Token"},
243         {0x0203, EIO, "DMA Error"},
244         {0x0204, EIO, "Use Error"},
245         {0x0205, EIO, "Release Error"},
246         {0x0206, EINVAL, "Invalid Disk"},
247         {0x0207, EBUSY, "Cant Lock"},
248         {0x0208, EIO, "Already Locked"},
249         {0x0209, EIO, "Already Unlocked"},
250         {0x020A, EIO, "Invalid Arg"},
251         {0x020B, EIO, "Bad IFS File"},
252         {0x020C, EROFS, "Read Only Device"},
253         {0x02FF, EIO, "Internal Error"},
254         {0x0000, 0, NULL},
255 };
256
257 /* Our gendisk table
258  */
259 static struct gendisk viodasd_gendisk[NUM_MAJORS];
260
261 static struct gendisk *major_to_gendisk(int major)
262 {
263         int index = major_to_index(major);
264         return index < 0 ? NULL : &viodasd_gendisk[index];
265 }
266 static struct hd_struct *devt_to_partition(kdev_t dev)
267 {
268         return &major_to_gendisk(MAJOR(dev))->part[MINOR(dev)];
269 }
270
```

```
271 /* Figure out the biggest I/O request (in sectors) we can accept
272  */
273 #define VIODASD_MAXSECTORS (4096 / 512 * VIOMAXBLOCKDMA)
274
275 /* Keep some statistics on what's happening for the PROC file system
276  */
277 static struct {
278         long tot;
279         long nobh;
280         long ntce[VIOMAXBLOCKDMA];
281 } viod_stats[MAX_DISKNO][2];
282
283 /* Number of disk I/O requests we've sent to OS/400
284  */
285 static int num_req_outstanding;
286
287 /* This is our internal structure for keeping track of disk devices
288  */
289 struct viodasd_device {
290         int useCount;
291         u16 cylinders;
292         u16 tracks;
293         u16 sectors;
294         u16 bytesPerSector;
295         u64 size;
296         int readOnly;
297 } *viodasd_devices;
298
299 /* When we get a disk I/O request we take it off the general request queue
300  * and put it here.
301  */
302 static LIST_HEAD(reqlist);
303
304 /* Handle reads from the proc file system
305  */
306 static int proc_read(char *buf, char **start, off_t offset,
307                      int blen, int *eof, void *data)
308 {
309         int len = 0;
310         int i;
311         int j;
312
313 #if defined(MODULE)
314         len +=
315             sprintf(buf + len,
316                     "viod Module opened %d times.  Major number %d\n",
317                     MOD_IN_USE, major_table[0]);
318 #endif
319         len +=
320             sprintf(buf + len, "viod %d possible devices\n", MAX_DISKNO);
321
322         for (i = 0; i < 16; i++) {
323                 if (viod_stats[i][0].tot || viod_stats[i][1].tot) {
324                         len +=
325                             sprintf(buf + len,
326                                     "DISK %2.2d: rd %−10.10ld wr %−10.10ld (no buffer list rd %−10.10ld wr %−10.10ld\n",
327                                     i, viod_stats[i][0].tot,
328                                     viod_stats[i][1].tot,
329                                     viod_stats[i][0].nobh,
330                                     viod_stats[i][1].nobh);
331
332                         len += sprintf(buf + len, "rd DMA: ");
333
334                         for (j = 0; j < VIOMAXBLOCKDMA; j++)
335                                 len += sprintf(buf + len, " [%2.2d] %ld",
336                                                j,
337                                                viod_stats[i][0].ntce[j]);
338
339                         len += sprintf(buf + len, "\nwr DMA: ");
340
341                         for (j = 0; j < VIOMAXBLOCKDMA; j++)
342                                 len += sprintf(buf + len, " [%2.2d] %ld",
343                                                j,
344                                                viod_stats[i][1].ntce[j]);
345                         len += sprintf(buf + len, "\n");
346                 }
347         }
348
349         *eof = 1;
350         return len;
351 }
352
353 /* Handle writes to our proc file system
354  */
355 static int proc_write(struct file *file, const char *buffer,
356                       unsigned long count, void *data)
357 {
358         return count;
359 }
360
```

```
361 /* setup our proc file system entries
362  */
363 void viodasd_proc_init(struct proc_dir_entry *iSeries_proc)
364 {
365         struct proc_dir_entry *ent;
366         ent =
367             create_proc_entry("viodasd", S_IFREG | S_IRUSR, iSeries_proc);
368         if (!ent)
369                 return;
370         ent->nlink = 1;
371         ent->data = NULL;
372         ent->read_proc = proc_read;
373         ent->write_proc = proc_write;
374 }
375
376 /* clean up our proc file system entries
377  */
378 void viodasd_proc_delete(struct proc_dir_entry *iSeries_proc)
379 {
380         remove_proc_entry("viodasd", iSeries_proc);
381 }
382
383 /* End a request
384  */
385 static void viodasd_end_request(struct request *req, int uptodate)
386 {
387         if (end_that_request_first(req, uptodate, VIOD_DEVICE_NAME))
388                 return;
389
390         add_blkdev_randomness(MAJOR(req->rq_dev));
391
392         end_that_request_last(req);
393 }
394
395 /* This rebuilds the partition information for a single disk device
396  */
397 static int viodasd_revalidate(kdev_t dev)
398 {
399         int i;
400         int device_no = DEVICE_NR(dev);
401         int dev_within_major = device_no % DEV_PER_MAJOR;
402         int part0 = (dev_within_major << PARTITION_SHIFT);
403         int npart = (1 << PARTITION_SHIFT);
404         int major = MAJOR(dev);
405         struct gendisk *gendisk = major_to_gendisk(major);
406
407         if (viodasd_devices[device_no].size == 0)
408                 return 0;
409
410         for (i = npart - 1; i >= 0; i--) {
411                 int minor = part0 + i;
412                 struct hd_struct *partition = &gendisk->part[minor];
413
414                 if (partition->nr_sects != 0) {
415                         kdev_t devp = MKDEV(major, minor);
416                         struct super_block *sb;
417                         fsync_dev(devp);
418
419                         sb = get_super(devp);
420                         if (sb)
421                                 invalidate_inodes(sb);
422
423                         invalidate_buffers(devp);
424                 }
425
426                 partition->start_sect = 0;
427                 partition->nr_sects = 0;
428         }
429
430         grok_partitions(gendisk, dev_within_major, npart,
431                         viodasd_devices[device_no].size >> 9);
432
433         return 0;
434 }
435
436
437 static u16 access_flags(mode_t mode)
438 {
439         u16 flags = 0;
440         if (!(mode & FMODE_WRITE))
441                 flags |= vioblockflags_ro;
442         return flags;
443 }
444
445 static void internal_register_disk(int diskno);
446
447 /* This is the actual open code.  It gets called from the external
448  * open entry point, as well as from the init code when we're figuring
449  * out what disks we have
450  */
```

```
451 static int internal_open(int device_no, u16 flags)
452 {
453         int i;
454         const int dev_within_major = device_no % DEV_PER_MAJOR;
455         struct gendisk *gendisk =
456             major_to_gendisk(diskno_to_major(device_no));
457         HvLpEvent_Rc hvrc;
458         /* This semaphore is raised in the interrupt handler                    */
459         DECLARE_MUTEX_LOCKED(Semaphore);
460         struct viodasd_waitevent we = { sem:&Semaphore };
461
462         /* Check that we are dealing with a valid hosting partition             */
463         if (viopath_hostLp == HvLpIndexInvalid) {
464                 printk(KERN_WARNING_VIO "Invalid hosting partition\n");
465                 return -EIO;
466         }
467
468         /* Send the open event to OS/400                                        */
469         hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
470                                              HvLpEvent_Type_VirtualIo,
471                                              viomajorsubtype_blockio |
472                                              vioblockopen,
473                                              HvLpEvent_AckInd_DoAck,
474                                              HvLpEvent_AckType_ImmediateAck,
475                                              viopath_sourceinst
476                                              (viopath_hostLp),
477                                              viopath_targetinst
478                                              (viopath_hostLp),
479                                              (u64) (unsigned long) &we,
480                                              VIOVERSION << 16,
481                                              ((u64) device_no << 48) |
482                                              ((u64) flags << 32), 0, 0, 0);
483
484         if (hvrc != 0) {
485                 printk(KERN_WARNING_VIO "bad rc on signalLpEvent %d\n",
486                        (int) hvrc);
487                 return -EIO;
488         }
489
490         /* Wait for the interrupt handler to get the response                   */
491         down(&Semaphore);
492
493         /* Check the return code                                                */
494         if (we.rc != 0) {
495                 const struct vio_error_entry *err =
496                     vio_lookup_rc(viodasd_err_table, we.data.subRC);
497                 printk(KERN_WARNING_VIO
498                        "bad rc opening disk: %d:0x%04x (%s)\n",
499                        (int) we.rc, we.data.subRC, err->msg);
500                 return -err->errno;
501         }
502
503         /* If this is the first open of this device, update the device information */
504         /* If this is NOT the first open, assume that it isn't changing         */
505         if (viodasd_devices[device_no].useCount == 0) {
506                 if (viodasd_devices[device_no].size > 0) {
507                         /* divide by 512 */
508                         u64 tmpint = viodasd_devices[device_no].size >> 9;
509                         gendisk->part[dev_within_major << PARTITION_SHIFT].nr_sects = tmpint;
510                         /* Now the value divided by 1024 */
511                         tmpint = tmpint >> 1;
512                         gendisk->sizes[dev_within_major << PARTITION_SHIFT] = tmpint;
513
514                         for (i = dev_within_major << PARTITION_SHIFT;
515                              i < ((dev_within_major + 1) << PARTITION_SHIFT);
516                              i++)
517                         {
518                                 hardsect_size[diskno_to_major(device_no)][i] =
519                                     viodasd_devices[device_no].bytesPerSector;
520                         }
521                 }
522         } else {
523                 /* If the size of the device changed, weird things are happening!     */
524                 if (gendisk->sizes[dev_within_major << PARTITION_SHIFT] !=
525                     viodasd_devices[device_no].size >> 10) {
526                         printk(KERN_WARNING_VIO
527                                "disk size change (%dK to %dK) for device %d\n",
528                                gendisk->sizes[dev_within_major << PARTITION_SHIFT],
529                                (int) viodasd_devices[device_no].size >> 10, device_no);
530                 }
531         }
532
533         internal_register_disk(device_no);
534
535         /* Bump the use count                                                   */
536         viodasd_devices[device_no].useCount++;
537         return 0;
538 }
539
540 /* This is the actual release code.  It gets called from the external
```

```
541  * release entry point, as well as from the init code when we're figuring
542  * out what disks we have
543  */
544 static int internal_release(int device_no, u16 flags)
545 {
546         /* Send the event to OS/400.  We DON'T expect a response               */
547         HvLpEvent_Rc hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
548                                                           HvLpEvent_Type_VirtualIo,
549                                                           viomajorsubtype_blockio
550                                                           | vioblockclose,
551                                                           HvLpEvent_AckInd_NoAck,
552                                                           HvLpEvent_AckType_ImmediateAck,
553                                                           viopath_sourceinst
554                                                           (viopath_hostLp),
555                                                           viopath_targetinst
556                                                           (viopath_hostLp),
557                                                           0,
558                                                           VIOVERSION << 16,
559                                                           ((u64) device_no
560                                                            << 48) | ((u64)
561                                                                      flags
562                                                                      <<
563                                                                      32),
564                                                           0, 0, 0);
565
566         viodasd_devices[device_no].useCount--;
567
568         if (hvrc != 0) {
569                 printk(KERN_WARNING_VIO
570                        "bad rc sending event to OS/400 %d\n", (int) hvrc);
571                 return -EIO;
572         }
573         return 0;
574 }
575
576
577 /* External open entry point.
578  */
579 static int viodasd_open(struct inode *ino, struct file *fil)
580 {
581         int device_no;
582         int old_max_disk = viodasd_max_disk;
583
584         /* Do a bunch of sanity checks                                        */
585         if (!ino) {
586                 printk(KERN_WARNING_VIO "no inode provided in open\n");
587                 return -ENODEV;
588         }
589
590         if (major_to_index(MAJOR(ino->i_rdev)) < 0) {
591                 printk(KERN_WARNING_VIO
592                        "Weird error...wrong major number on open\n");
593                 return -ENODEV;
594         }
595
596         device_no = DEVICE_NR(ino->i_rdev);
597         if (device_no > MAX_DISKNO || device_no < 0) {
598                 printk(KERN_WARNING_VIO
599                        "Invalid device number %d in open\n", device_no);
600                 return -ENODEV;
601         }
602
603         /* Call the actual open code                                          */
604         if (internal_open(device_no, access_flags(fil ? fil->f_mode : 0)) == 0) {
605                 int i;
606                 MOD_INC_USE_COUNT;
607                 /* For each new disk: */
608                 /* update the disk's geometry via internal_open and register it */
609                 for (i = old_max_disk + 1; i <= viodasd_max_disk; ++i) {
610                         internal_open(i, vioblockflags_ro);
611                         internal_release(i, vioblockflags_ro);
612                 }
613                 return 0;
614         } else {
615                 return -EIO;
616         }
617 }
618
619 /* External release entry point.
620  */
621 static int viodasd_release(struct inode *ino, struct file *fil)
622 {
623         int device_no;
624
625         /* Do a bunch of sanity checks                                        */
626         if (!ino) {
627                 printk(KERN_WARNING_VIO "no inode provided in release\n");
628                 return -ENODEV;
629         }
630
```

```
631         if (major_to_index(MAJOR(ino->i_rdev)) < 0) {
632                 printk(KERN_WARNING_VIO
633                        "Weird error...wrong major number on release\n");
634                 return -ENODEV;
635         }
636
637         device_no = DEVICE_NR(ino->i_rdev);
638
639         if (device_no > MAX_DISKNO || device_no < 0) {
640                 printk("Tried to release invalid disk number %d\n",
641                        device_no);
642                 return -ENODEV;
643         }
644
645         /* Call the actual release code                                    */
646         internal_release(device_no, access_flags(fil ? fil->f_mode : 0));
647
648         MOD_DEC_USE_COUNT;
649         return 0;
650 }
651
652 /* External ioctl entry point.
653  */
654 static int viodasd_ioctl(struct inode *ino, struct file *fil,
655                          unsigned int cmd, unsigned long arg)
656 {
657         int device_no;
658         int err;
659         HvLpEvent_Rc hvrc;
660         struct hd_struct *partition;
661         DECLARE_MUTEX_LOCKED(Semaphore);
662
663         /* Sanity checks                                                   */
664         if (!ino) {
665                 printk(KERN_WARNING_VIO "no inode provided in ioctl\n");
666                 return -ENODEV;
667         }
668
669         if (major_to_index(MAJOR(ino->i_rdev)) < 0) {
670                 printk(KERN_WARNING_VIO
671                        "Weird error...wrong major number on ioctl\n");
672                 return -ENODEV;
673         }
674
675         partition = devt_to_partition(ino->i_rdev);
676
677         device_no = DEVICE_NR(ino->i_rdev);
678         if (device_no > viodasd_max_disk) {
679                 printk(KERN_WARNING_VIO
680                        "Invalid device number %d in ioctl\n", device_no);
681                 return -ENODEV;
682         }
683
684         switch (cmd) {
685         case BLKPG:
686                 return blk_ioctl(ino->i_rdev, cmd, arg);
687         case BLKGETSIZE:
688                 /* return the device size in sectors */
689                 if (!arg)
690                         return -EINVAL;
691                 err =
692                     verify_area(VERIFY_WRITE, (long *) arg, sizeof(long));
693                 if (err)
694                         return err;
695
696                 put_user(partition->nr_sects, (long *) arg);
697                 return 0;
698
699         case FDFLUSH:
700         case BLKFLSBUF:
701                 if (!suser())
702                         return -EACCES;
703                 fsync_dev(ino->i_rdev);
704                 invalidate_buffers(ino->i_rdev);
705                 hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
706                                                      HvLpEvent_Type_VirtualIo,
707                                                      viomajorsubtype_blockio
708                                                      | vioblockflush,
709                                                      HvLpEvent_AckInd_DoAck,
710                                                      HvLpEvent_AckType_ImmediateAck,
711                                                      viopath_sourceinst
712                                                      (viopath_hostLp),
713                                                      viopath_targetinst
714                                                      (viopath_hostLp),
715                                                      (u64) (unsigned long)
716                                                      &Semaphore,
717                                                      VIOVERSION << 16,
718                                                      ((u64) device_no <<
719                                                       48), 0, 0, 0);
720
```

```
721
722                 if (hvrc != 0) {
723                         printk(KERN_WARNING_VIO
724                                "bad rc on sync signalLpEvent %d\n",
725                                (int) hvrc);
726                         return -EIO;
727                 }
728
729                 down(&Semaphore);
730
731                 return 0;
732
733         case BLKRAGET:
734                 if (!arg)
735                         return -EINVAL;
736                 err =
737                     verify_area(VERIFY_WRITE, (long *) arg, sizeof(long));
738                 if (err)
739                         return err;
740                 put_user(read_ahead[MAJOR(ino->i_rdev)], (long *) arg);
741                 return 0;
742
743         case BLKRASET:
744                 if (!suser())
745                         return -EACCES;
746                 if (arg > 0x00ff)
747                         return -EINVAL;
748                 read_ahead[MAJOR(ino->i_rdev)] = arg;
749                 return 0;
750
751         case BLKRRPART:
752                 viodasd_revalidate(ino->i_rdev);
753                 return 0;
754
755         case HDIO_GETGEO:
756                 {
757                         unsigned char sectors;
758                         unsigned char heads;
759                         unsigned short cylinders;
760
761                         struct hd_geometry *geo =
762                             (struct hd_geometry *) arg;
763                         if (geo == NULL)
764                                 return -EINVAL;
765
766                         err = verify_area(VERIFY_WRITE, geo, sizeof(*geo));
767                         if (err)
768                                 return err;
769
770                         sectors = viodasd_devices[device_no].sectors;
771                         if (sectors == 0)
772                                 sectors = 32;
773
774                         heads = viodasd_devices[device_no].tracks;
775                         if (heads == 0)
776                                 heads = 64;
777
778                         cylinders = viodasd_devices[device_no].cylinders;
779                         if (cylinders == 0)
780                                 cylinders =
781                                     partition->nr_secta / (sectors *
782                                                            heads);
783
784                         put_user(sectors, &geo->sectors);
785                         put_user(heads, &geo->heads);
786                         put_user(cylinders, &geo->cylinders);
787
788                         put_user(partition->start_sect,
789                                  (long *) &geo->start);
790
791                         return 0;
792                 }
793
794         case HDIO_GETGEO_BIG:
795                 {
796                         unsigned char sectors;
797                         unsigned char heads;
798                         unsigned int cylinders;
799
800                         struct hd_big_geometry *geo =
801                             (struct hd_big_geometry *) arg;
802                         if (geo == NULL)
803                                 return -EINVAL;
804
805                         err = verify_area(VERIFY_WRITE, geo, sizeof(*geo));
806                         if (err)
807                                 return err;
808
809                         sectors = viodasd_devices[device_no].sectors;
810                         if (sectors == 0)
```

```
811                                     sectors = 32;
812
813                             heads = viodasd_devices[device_no].tracks;
814                             if (heads == 0)
815                                     heads = 64;
816
817                             cylinders = viodasd_devices[device_no].cylinders;
818                             if (cylinders == 0)
819                                     cylinders =
820                                         partition->nr_sects / (sectors *
821                                                                heads);
822
823                             put_user(sectors, &geo->sectors);
824                             put_user(heads, &geo->heads);
825                             put_user(cylinders, &geo->cylinders);
826
827                             put_user(partition->start_sect,
828                                      (long *) &geo->start);
829
830                             return 0;
831                     }
832
833     #define PRTIOC(x) case x: printk(KERN_WARNING_VIO "got unsupported FD ioctl " #x "\n"); \
834                               return -EINVAL;
835
836                     PRTIOC(FDCLRPRM);
837                     PRTIOC(FDSETPRM);
838                     PRTIOC(FDDEFPRM);
839                     PRTIOC(FDGETPRM);
840                     PRTIOC(FDMSGON);
841                     PRTIOC(FDMSGOFF);
842                     PRTIOC(FDFMTBEG);
843                     PRTIOC(FDFMTTRK);
844                     PRTIOC(FDFMTEND);
845                     PRTIOC(FDSETEMSGTRESH);
846                     PRTIOC(FDSETMAXERRS);
847                     PRTIOC(FDGETMAXERRS);
848                     PRTIOC(FDGETDRVTYP);
849                     PRTIOC(FDSETDRVPRM);
850                     PRTIOC(FDGETDRVPRM);
851                     PRTIOC(FDGETDRVSTAT);
852                     PRTIOC(FDPOLLDRVSTAT);
853                     PRTIOC(FDRESET);
854                     PRTIOC(FDGETFDCSTAT);
855                     PRTIOC(FDWERRORCLR);
856                     PRTIOC(FDWERRORGET);
857                     PRTIOC(FDRAWCMD);
858                     PRTIOC(FDEJECT);
859                     PRTIOC(FDTWADDLE);
860
861             }
862
863             return -EINVAL;
864     }
865
866     /* Send an actual I/O request to OS/400
867      */
868     static int send_request(struct request *req)
869     {
870             u64 sect_size;
871             u64 start;
872             u64 len;
873             int direction;
874             int nsg;
875             u16 viocmd;
876             HvLpEvent_Rc hvrc;
877             struct vioblocklpevent *bevent;
878             struct scatterlist sg[VIOMAXBLOCKDMA];
879             struct buffer_head *bh;
880             int sgindex;
881             int device_no = DEVICE_NR(req->rq_dev);
882             int dev_within_major = device_no % DEV_PER_MAJOR;
883             int statindex;
884             struct hd_struct *partition = devt_to_partition(req->rq_dev);
885
886             if (device_no > viodasd_max_disk || device_no < 0) {
887                     printk
888                         ("yikes! sending a request to device %d of %d possible?\n",
889                          device_no, viodasd_max_disk + 1);
890             }
891
892             /* Note that this SHOULD always be 512...but lets be architecturally correct */
893             sect_size = hardsect_size[MAJOR(req->rq_dev)][dev_within_major];
894
895             /* Figure out the starting sector and length                          */
896             start = (req->sector + partition->start_sect) * sect_size;
897             len = req->nr_sectors * sect_size;
898
899             /* More paranoia checks                                               */
900             if ((req->sector + req->nr_sectors) >
```

```
901             (partition->start_sect + partition->nr_sects)) {
902                 printk(KERN_WARNING_VIO
903                        "Invalid request offset & length\n");
904                 printk(KERN_WARNING_VIO
905                        "req->sector: %ld, req->nr_sectors: %ld\n",
906                        req->sector, req->nr_sectors);
907                 printk(KERN_WARNING_VIO "major: %d, minor: %d\n",
908                        MAJOR(req->rq_dev), MINOR(req->rq_dev));
909                 return -1;
910         }
911
912         if (req->cmd == READ) {
913                 direction = PCI_DMA_FROMDEVICE;
914                 viocmd = viomajorsubtype_blockio | vioblockread;
915                 statindex = 0;
916         } else {
917                 direction = PCI_DMA_TODEVICE;
918                 viocmd = viomajorsubtype_blockio | vioblockwrite;
919                 statindex = 1;
920         }
921
922         /* Update totals */
923         viod_stats[device_no][statindex].tot++;
924
925         /* Now build the scatter-gather list                                    */
926         memset(&sg, 0x00, sizeof(sg));
927         sgindex = 0;
928
929         /* See if this is a swap I/O (without a bh pointer) or a regular I/O    */
930         if (req->bh) {
931                 /* OK...this loop takes buffers from the request and adds them to the SG
932                    until we're done, or until we hit a maximum.  If we hit a maximum we'll
933                    just finish this request later                                */
934                 bh = req->bh;
935                 while ((bh) && (sgindex < VIOMAXBLOCKDMA)) {
936                         sg[sgindex].address = bh->b_data;
937                         sg[sgindex].length = bh->b_size;
938
939                         sgindex++;
940                         bh = bh->b_reqnext;
941                 }
942                 nsg = pci_map_sg(iSeries_vio_dev, sg, sgindex, direction);
943                 if ((nsg == 0) || (sg[0].dma_length == 0)
944                     || (sg[0].dma_address == 0xFFFFFFFF)) {
945                         printk(KERN_WARNING_VIO "error getting sg tces\n");
946                         return -1;
947                 }
948
949         } else {
950                 /* Update stats */
951                 viod_stats[device_no][statindex].nobh++;
952
953                 sg[0].dma_address =
954                     pci_map_single(iSeries_vio_dev, req->buffer, len,
955                                    direction);
956                 if (sg[0].dma_address == 0xFFFFFFFF) {
957                         printk(KERN_WARNING_VIO
958                                "error allocating tce for address %p len %ld\n",
959                                req->buffer, (long) len);
960                         return -1;
961                 }
962                 sg[0].dma_length = len;
963                 nsg = 1;
964         }
965
966         /* Update stats */
967         viod_stats[device_no][statindex].ntce[sgindex]++;
968
969         /* This optimization handles a single DMA block                         */
970         if (sgindex == 1) {
971                 /* Send the open event to OS/400                                */
972                 hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
973                                                      HvLpEvent_Type_VirtualIo,
974                                                      viomajorsubtype_blockio
975                                                      | viocmd,
976                                                      HvLpEvent_AckInd_DoAck,
977                                                      HvLpEvent_AckType_ImmediateAck,
978                                                      viopath_sourceinst
979                                                      (viopath_hostLp),
980                                                      viopath_targetinst
981                                                      (viopath_hostLp),
982                                                      (u64) (unsigned long)
983                                                      req->buffer,
984                                                      VIOVERSION << 16,
985                                                      ((u64) device_no <<
986                                                       48), start,
987                                                      ((u64) sg[0].
988                                                       dma_address) << 32,
989                                                      sg[0].dma_length);
990         } else {
```

```
991                     bevent =
992                         (struct vioblocklpevent *)
993                         vio_get_event_buffer(viomajorsubtype_blockio);
994                     if (bevent == NULL) {
995                             printk(KERN_WARNING_VIO
996                                    "error allocating disk event buffer\n");
997                             return −1;
998                     }
999
1000                    /* Now build up the actual request.  Note that we store the pointer      */
1001                    /* to the request buffer in the correlation token so we can match        */
1002                    /* this response up later                                                */
1003                    memset(bevent, 0x00, sizeof(struct vioblocklpevent));
1004                    bevent−>event.xFlags.xValid = 1;
1005                    bevent−>event.xFlags.xFunction = HvLpEvent_Function_Int;
1006                    bevent−>event.xFlags.xAckInd = HvLpEvent_AckInd_DoAck;
1007                    bevent−>event.xFlags.xAckType =
1008                        HvLpEvent_AckType_ImmediateAck;
1009                    bevent−>event.xType = HvLpEvent_Type_VirtualIo;
1010                    bevent−>event.xSubtype = viocmd;
1011                    bevent−>event.xSourceLp = HvLpConfig_getLpIndex();
1012                    bevent−>event.xTargetLp = viopath_hostLp;
1013                    bevent−>event.xSizeMinus1 =
1014                        offsetof(struct vioblocklpevent,
1015                                 u.rwData.dmaInfo) +
1016                        (sizeof(bevent−>u.rwData.dmaInfo[0]) * (sgindex)) − 1;
1017                    bevent−>event.xSizeMinus1 =
1018                        sizeof(struct vioblocklpevent) − 1;
1019                    bevent−>event.xSourceInstanceId =
1020                        viopath_sourceinst(viopath_hostLp);
1021                    bevent−>event.xTargetInstanceId =
1022                        viopath_targetinst(viopath_hostLp);
1023                    bevent−>event.xCorrelationToken =
1024                        (u64) (unsigned long) req−>buffer;
1025                    bevent−>mVersion = VIOVERSION;
1026                    bevent−>mDisk = device_no;
1027                    bevent−>u.rwData.mOffset = start;
1028
1029                    /* Copy just the dma information from the sg list into the request */
1030                    for (sgindex = 0; sgindex < nsg; sgindex++) {
1031                            bevent−>u.rwData.dmaInfo[sgindex].mToken =
1032                                sg[sgindex].dma_address;
1033                            bevent−>u.rwData.dmaInfo[sgindex].mLen =
1034                                sg[sgindex].dma_length;
1035                    }
1036
1037                    /* Send the request                                                      */
1038                    hvrc = HvCallEvent_signalLpEvent(&bevent−>event);
1039                    vio_free_event_buffer(viomajorsubtype_blockio, bevent);
1040            }
1041
1042            if (hvrc != HvLpEvent_Rc_Good) {
1043                    printk(KERN_WARNING_VIO
1044                           "error sending disk event to OS/400 (rc %d)\n",
1045                           (int) hvrc);
1046                    return −1;
1047            } else {
1048                    /* If the request was successful, bump the number of outstanding */
1049                    num_req_outstanding++;
1050            }
1051            return 0;
1052    }
1053
1054    /* This is the external request processing routine
1055     */
1056    static void do_viodasd_request(request_queue_t * q)
1057    {
1058            int device_no;
1059            for (;;) {
1060                    struct request *req;
1061                    struct gendisk *gendisk;
1062
1063                    /* inlined INIT_REQUEST here because we don't define MAJOR_NR before blk.h */
1064                    if (list_empty(&q−>queue_head))
1065                            return;
1066                    req = blkdev_entry_next_request(&q−>queue_head);
1067                    if (major_to_index(MAJOR(req−>rq_dev)) < 0)
1068                            panic(VIOD_DEVICE_NAME ": request list destroyed");
1069                    if (req−>bh) {
1070                            if (!buffer_locked(req−>bh))
1071                                    panic(VIOD_DEVICE_NAME
1072                                          ": block not locked");
1073                    }
1074
1075                    gendisk = major_to_gendisk(MAJOR(req−>rq_dev));
1076
1077                    device_no = DEVICE_NR(req−>rq_dev);
1078                    if (device_no > MAX_DISKNO || device_no < 0) {
1079                            printk(KERN_WARNING_VIO "Invalid device # %d\n",
1080                                   device_no);
```

```
1081                            viodasd_end_request(req, 0);
1082                            continue;
1083                    }
1084
1085                    if (gendisk->sizes == NULL) {
1086                            printk(KERN_WARNING_VIO
1087                                    "Ouch! gendisk->sizes is NULL\n");
1088                            viodasd_end_request(req, 0);
1089                            continue;
1090                    }
1091
1092                    /* If the queue is plugged, don't dequeue anything right now */
1093                    if ((q) && (q->plugged)) {
1094                            return;
1095                    }
1096
1097                    /* If we already have the maximum number of requests outstanding to OS/400
1098                       just bail out. We'll come back later                                  */
1099                    if (num_req_outstanding >= VIOMAXREQ) {
1100                            return;
1101                    }
1102
1103                    /* get the current request, then dequeue it from the queue           */
1104                    blkdev_dequeue_request(req);
1105
1106                    /* Try sending the request                                           */
1107                    if (send_request(req) == 0) {
1108                            list_add_tail(&req->queue, &reqlist);
1109                    } else {
1110                            viodasd_end_request(req, 0);
1111                    }
1112            }
1113    }
1114
1115    /* Check for changed disks
1116     */
1117    static int viodasd_check_change(kdev_t dev)
1118    {
1119            struct viodasd_waitevent we;
1120            HvLpEvent_Rc hvrc;
1121            int device_no = DEVICE_NR(dev);
1122
1123            /* This semaphore is raised in the interrupt handler                    */
1124            DECLARE_MUTEX_LOCKED(Semaphore);
1125
1126            /* Check that we are dealing with a valid hosting partition             */
1127            if (viopath_hostLp == HvLpIndexInvalid) {
1128                    printk(KERN_WARNING_VIO "Invalid hosting partition\n");
1129                    return -EIO;
1130            }
1131
1132            we.sem = &Semaphore;
1133
1134            /* Send the open event to OS/400                                        */
1135            hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
1136                                                 HvLpEvent_Type_VirtualIo,
1137                                                 viomajorsubtype_blockio |
1138                                                 vioblockcheck,
1139                                                 HvLpEvent_AckInd_DoAck,
1140                                                 HvLpEvent_AckType_ImmediateAck,
1141                                                 viopath_sourceinst
1142                                                 (viopath_hostLp),
1143                                                 viopath_targetinst
1144                                                 (viopath_hostLp),
1145                                                 (u64) (unsigned long) &we,
1146                                                 VIOVERSION << 16,
1147                                                 ((u64) device_no << 48), 0, 0,
1148                                                 0);
1149
1150            if (hvrc != 0) {
1151                    printk(KERN_WARNING_VIO "bad rc on signalLpEvent %d\n",
1152                           (int) hvrc);
1153                    return -EIO;
1154            }
1155
1156            /* Wait for the interrupt handler to get the response                   */
1157            down(&Semaphore);
1158
1159            /* Check the return code.  If bad, assume no change                     */
1160            if (we.rc != 0) {
1161                    printk(KERN_WARNING_VIO
1162                           "bad rc %d on check_change. Assuming no change\n",
1163                           (int) we.rc);
1164                    return 0;
1165            }
1166
1167            return we.data.changed;
1168    }
1169
1170    /* Our file operations table
```

```
 */
static struct block_device_operations viodasd_fops = {
        open:viodasd_open,
        release:viodasd_release,
        ioctl:viodasd_ioctl,
        check_media_change:viodasd_check_change,
        revalidate:viodasd_revalidate
};

/* returns the total number of scatterlist elements converted */
static int block_event_to_scatterlist(const struct vioblocklpevent *bevent,
                                      struct scatterlist *sg,
                                      int *total_len)
{
        int i, numsg;
        const struct rwData *rwData = &bevent->u.rwData;
        static const int offset =
            offsetof(struct vioblocklpevent, u.rwData.dmaInfo);
        static const int element_size = sizeof(rwData->dmaInfo[0]);

        numsg = ((bevent->event.xSizeMinus1 + 1) - offset) / element_size;
        if (numsg > VIOMAXBLOCKDMA)
                numsg = VIOMAXBLOCKDMA;

        *total_len = 0;
        memset(sg, 0x00, sizeof(sg[0]) * VIOMAXBLOCKDMA);

        for (i = 0; (i < numsg) && (rwData->dmaInfo[i].mLen > 0); ++i) {
                sg[i].dma_address = rwData->dmaInfo[i].mToken;
                sg[i].dma_length = rwData->dmaInfo[i].mLen;
                *total_len += rwData->dmaInfo[i].mLen;
        }
        return i;
}

static struct request *find_request_with_token(u64 token)
{
        struct request *req = blkdev_entry_to_request(reqlist.next);
        while ((&req->queue != &reqlist) &&
               ((u64) (unsigned long) req->buffer != token))
                req = blkdev_entry_to_request(req->queue.next);
        if (&req->queue == &reqlist) {
                return NULL;
        }
        return req;
}

/* Restart all queues, starting with the one _after_ the major given, */
/* thus reducing the chance of starvation of disks with late majors. */
static void viodasd_restart_all_queues_starting_from(int first_major)
{
        int i, first_index = major_to_index(first_major);
        for(i = first_index + 1; i < NUM_MAJORS; ++i)
                do_viodasd_request(BLK_DEFAULT_QUEUE(major_table[i]));
        for(i = 0; i <= first_index; ++i)
                do_viodasd_request(BLK_DEFAULT_QUEUE(major_table[i]));
}

/* For read and write requests, decrement the number of outstanding requests,
 * Free the DMA buffers we allocated, and find the matching request by
 * using the buffer pointer we stored in the correlation token.
 */
static int viodasd_handleReadWrite(struct vioblocklpevent *bevent)
{
        int num_sg, num_sect, pci_direction, total_len, major;
        struct request *req;
        struct scatterlist sg[VIOMAXBLOCKDMA];
        struct HvLpEvent *event = &bevent->event;
        unsigned long irq_flags;

        num_sg = block_event_to_scatterlist(bevent, sg, &total_len);
        num_sect = total_len >> 9;
        if (event->xSubtype == (viomajorsubtype_blockio | vioblockread))
                pci_direction = PCI_DMA_FROMDEVICE;
        else
                pci_direction = PCI_DMA_TODEVICE;
        pci_unmap_sg(iSeries_vio_dev, sg, num_sg, pci_direction);


        /* Since this is running in interrupt mode, we need to make sure we're not
         * stepping on any global I/O operations
         */
        spin_lock_irqsave(&io_request_lock, irq_flags);

        num_req_outstanding--;

        /* Now find the matching request in OUR list (remember we moved the request
         * from the global list to our list when we got it)
         */
        req = find_request_with_token(bevent->event.xCorrelationToken);
```

```
1261          if (req == NULL) {
1262                  printk(KERN_WARNING_VIO
1263                         "Yikes! No request matching 0x%lx found\n",
1264                         bevent->event.xCorrelationToken);
1265                  spin_unlock_irqrestore(&io_request_lock, irq_flags);
1266                  return -1;
1267          }
1268
1269          /* Remove the request from our list */
1270          list_del(&req->queue);
1271          /* Record this event's major number so we can check that queue again */
1272          major = MAJOR(req->rq_dev);
1273
1274          if (!req->bh) {
1275                  if (event->xRc != HvLpEvent_Rc_Good) {
1276                          const struct vio_error_entry *err =
1277                              vio_lookup_rc(viodasd_err_table,
1278                                            bevent->mSubTypeRc);
1279                          printk(KERN_WARNING_VIO
1280                                 "read/write error %d:0x%04x (%s)\n",
1281                                 event->xRc, bevent->mSubTypeRc, err->msg);
1282                          viodasd_end_request(req, 0);
1283                  } else {
1284                          if (num_sect != req->current_nr_sectors) {
1285                                  printk(KERN_WARNING_VIO
1286                                         "Yikes...non bh i/o # sect doesn't match!!!\n");
1287                          }
1288                          viodasd_end_request(req, 1);
1289                  }
1290          } else {
1291                  /* record having received the answers we did */
1292                  while ((num_sect > 0) && (req->bh)) {
1293                          num_sect -= req->current_nr_sectors;
1294                          viodasd_end_request(req, 1);
1295                  }
1296                  /* if they somehow answered _more_ than we asked for...something weird happened */
1297                  if (num_sect)
1298                          printk(KERN_WARNING_VIO
1299                                 "Yikes...sectors left over on a request!!!\n");
1300
1301                  /* if they didn't answer the whole request this time, re-submit the request */
1302                  if (req->bh) {
1303                          if (send_request(req) == 0) {
1304                                  list_add_tail(&req->queue, &reqlist);
1305                          } else {
1306                                  viodasd_end_request(req, 0);
1307                          }
1308                  }
1309          }
1310
1311          /* Finally, try to get more requests off of this device's queue */
1312          viodasd_restart_all_queues_starting_from(major);
1313
1314          spin_unlock_irqrestore(&io_request_lock, irq_flags);
1315
1316          return 0;
1317  }
1318
1319  /* This routine handles incoming block LP events */
1320  static void vioHandleBlockEvent(struct HvLpEvent *event)
1321  {
1322          struct vioblocklpevent *bevent = (struct vioblocklpevent *) event;
1323          struct viodasd_waitevent *pwe;
1324
1325          if (event == NULL) {
1326                  /* Notification that a partition went away! */
1327                  return;
1328          }
1329          // First, we should NEVER get an int here...only acks
1330          if (event->xFlags.xFunction == HvLpEvent_Function_Int) {
1331                  printk(KERN_WARNING_VIO
1332                         "Yikes! got an int in viodasd event handler!\n");
1333                  if (event->xFlags.xAckInd == HvLpEvent_AckInd_DoAck) {
1334                          event->xRc = HvLpEvent_Rc_InvalidSubtype;
1335                          HvCallEvent_ackLpEvent(event);
1336                  }
1337          }
1338
1339          switch (event->xSubtype & VIOMINOR_SUBTYPE_MASK) {
1340
1341                  /* Handle a response to an open request.  We get all the disk information
1342                   * in the response, so update it.  The correlation token contains a pointer to
1343                   * a waitevent structure that has a semaphore in it.  update the return code
1344                   * in the waitevent structure and post the semaphore to wake up the guy who
1345                   * sent the request */
1346          case vioblockopen:
1347                  pwe =
1348                      (struct viodasd_waitevent *) (unsigned long) event->
1349                      xCorrelationToken;
1350                  pwe->rc = event->xRc;
```

```
1351                    pwe->data.subRC = bevent->mSubTypeRc;
1352                    if (event->xRc == HvLpEvent_Rc_Good) {
1353                            const struct openData *data = &bevent->u.openData;
1354                            struct viodasd_device *device =
1355                                &viodasd_devices[bevent->mDisk];
1356                            device->readOnly =
1357                                bevent->mFlags & vioblockflags_ro;
1358                            device->size = data->mDiskLen;
1359                            device->cylinders = data->mCylinders;
1360                            device->tracks = data->mTracks;
1361                            device->sectors = data->mSectors;
1362                            device->bytesPerSector = data->mBytesPerSector;
1363                            viodasd_max_disk = data->mMaxDisks;
1364                    }
1365                    up(pwe->sem);
1366                    break;
1367            case vioblockclose:
1368                    break;
1369            case vioblockcheck:
1370                    pwe =
1371                        (struct viodasd_waitevent *) (unsigned long) event->
1372                        xCorrelationToken;
1373                    pwe->rc = event->xRc;
1374                    pwe->data.changed = bevent->u.check.changed;
1375                    up(pwe->sem);
1376                    break;
1377            case vioblockflush:
1378                    up((void *) (unsigned long) event->xCorrelationToken);
1379                    break;
1380            case vioblockread:
1381            case vioblockwrite:
1382                    viodasd_handleReadWrite(bevent);
1383                    break;
1384
1385            default:
1386                    printk(KERN_WARNING_VIO "invalid subtype!");
1387                    if (event->xFlags.xAckInd == HvLpEvent_AckInd_DoAck) {
1388                            event->xRc = HvLpEvent_Rc_InvalidSubtype;
1389                            HvCallEvent_ackLpEvent(event);
1390                    }
1391            }
1392    }
1393
1394    static const char *major_name(int major)
1395    {
1396            static char major_names[NUM_MAJORS][MAX_MAJOR_NAME];
1397            int index = major_to_index(major);
1398
1399            if(index < 0)
1400                    return NULL;
1401            if(major_names[index][0] == '\0') {
1402                    if(index == 0)
1403                            strcpy(major_names[index], VIOD_GENHD_NAME);
1404                    else
1405                            sprintf(major_names[index], VIOD_GENHD_NAME"%d", index);
1406            }
1407            return major_names[index];
1408    }
1409
1410    static const char *device_name(int major)
1411    {
1412            static char device_names[NUM_MAJORS][MAX_MAJOR_NAME];
1413            int index = major_to_index(major);
1414
1415            if(index < 0)
1416                    return NULL;
1417            if(device_names[index][0] == '\0') {
1418    #ifdef CONFIG_VIODASD_IDE
1419                    sprintf(device_names[index], VIOD_DEVICE_NAME"%d", index);
1420    #else
1421                    strcpy(device_names[index], VIOD_DEVICE_NAME);
1422    #endif
1423            }
1424            return device_names[index];
1425    }
1426
1427    /* This routine tries to clean up anything we allocated/registered
1428     */
1429    static void viodasd_cleanup_major(int major)
1430    {
1431            const int num_partitions = DEV_PER_MAJOR << PARTITION_SHIFT;
1432            int minor;
1433
1434    #define CLEANIT(x) if (x) {kfree(x); x=NULL;}
1435
1436            for (minor = 0; minor < num_partitions; minor++)
1437                    fsync_dev(MKDEV(major, minor));
1438
1439            blk_cleanup_queue(BLK_DEFAULT_QUEUE(major));
1440
```

```
1441          read_ahead[major] = 0;
1442
1443          CLEANIT(blk_size[major]);
1444          CLEANIT(blksize_size[major]);
1445          CLEANIT(hardsect_size[major]);
1446          CLEANIT(max_sectors[major]);
1447          CLEANIT(major_to_gendisk(major)->part);
1448
1449          blk_cleanup_queue(BLK_DEFAULT_QUEUE(major));
1450
1451          devfs_unregister_blkdev(major, device_name(major));
1452  }
1453
1454  /* in case of bad return code, caller must cleanup2() for this major */
1455  static int viodasd_init_major(int major)
1456  {
1457          int i;
1458          const int numpart = DEV_PER_MAJOR << PARTITION_SHIFT;
1459          int *sizes, *sectsizes, *blksizes, *maxsectors;
1460          struct hd_struct *partitions;
1461          struct gendisk *gendisk = major_to_gendisk(major);
1462
1463          /*
1464           * Do the devfs_register.  This works even if devfs is not
1465           * configured
1466           */
1467          if (devfs_register_blkdev(major, device_name(major), &viodasd_fops)) {
1468                  printk(KERN_WARNING_VIO
1469                         "%s: can't register major number %d\n",
1470                         device_name(major), major);
1471                  return -1;
1472          }
1473
1474          blk_init_queue(BLK_DEFAULT_QUEUE(major), do_viodasd_request);
1475
1476          read_ahead[major] = 8;  /* 8 sector (4kB) read ahead */
1477
1478          /* initialize the struct */
1479          gendisk->major = major;
1480          gendisk->major_name = major_name(major);
1481          gendisk->minor_shift = PARTITION_SHIFT;
1482          gendisk->max_p = 1 << PARTITION_SHIFT;
1483          gendisk->nr_real = DEV_PER_MAJOR;
1484          gendisk->fops = &viodasd_fops;
1485
1486          /* to be assigned later */
1487          gendisk->next = NULL;
1488          gendisk->part = NULL;
1489          gendisk->sizes = NULL;
1490          gendisk->de_arr = NULL;
1491          gendisk->flags = NULL;
1492
1493          /* register us in the global list */
1494          add_gendisk(gendisk);
1495
1496          /*
1497           * Now fill in all the device driver info
1498           */
1499          sizes = kmalloc(numpart * sizeof(int), GFP_KERNEL);
1500          if (!sizes)
1501                  return -ENOMEM;
1502          memset(sizes, 0x00, numpart * sizeof(int));
1503          blk_size[major] = gendisk->sizes = sizes;
1504
1505          partitions =
1506              kmalloc(numpart * sizeof(struct hd_struct), GFP_KERNEL);
1507          if (!partitions)
1508                  return -ENOMEM;
1509          memset(partitions, 0x00, numpart * sizeof(struct hd_struct));
1510          gendisk->part = partitions;
1511
1512          blksizes = kmalloc(numpart * sizeof(int), GFP_KERNEL);
1513          if (!blksizes)
1514                  return -ENOMEM;
1515          for (i = 0; i < numpart; i++)
1516                  blksizes[i] = blksize;
1517          blksize_size[major] = blksizes;
1518
1519          sectsizes = kmalloc(numpart * sizeof(int), GFP_KERNEL);
1520          if (!sectsizes)
1521                  return -ENOMEM;
1522          for (i = 0; i < numpart; i++)
1523                  sectsizes[i] = 0;
1524          hardsect_size[major] = sectsizes;
1525
1526          maxsectors = kmalloc(numpart * sizeof(int), GFP_KERNEL);
1527          if (!maxsectors)
1528                  return -ENOMEM;
1529          for (i = 0; i < numpart; i++)
1530                  maxsectors[i] = VIODASD_MAXSECTORS;
```

```
1531          max_sectors[major] = maxsectors;
1532
1533          return 0;
1534  }
1535
1536  static void internal_register_disk(int diskno)
1537  {
1538          static int registered[MAX_DISKNO] = { 0, };
1539          int major = diskno_to_major(diskno);
1540          int dev_within_major = diskno % DEV_PER_MAJOR;
1541          struct gendisk *gendisk = major_to_gendisk(major);
1542          int i;
1543
1544          if(registered[diskno])
1545                  return;
1546          registered[diskno] = 1;
1547
1548          if (diskno == 0) {
1549                  printk(KERN_INFO_VIO
1550                         "%s: Currently %d disks connected\n",
1551                         VIOD_DEVICE_NAME, (int) viodasd_max_disk + 1);
1552                  if (viodasd_max_disk > MAX_DISKNO - 1)
1553                          printk(KERN_INFO_VIO
1554                                 "Only examining the first %d\n",
1555                                 MAX_DISKNO);
1556          }
1557
1558          register_disk(gendisk,
1559                        MKDEV(major,
1560                              dev_within_major <<
1561                              PARTITION_SHIFT),
1562                        1 << PARTITION_SHIFT, &viodasd_fops,
1563                        gendisk->
1564                        part[dev_within_major << PARTITION_SHIFT].nr_sects);
1565
1566          printk(KERN_INFO_VIO
1567                 "%s: Disk %2.2d size %dM, sectors %d, heads %d, cylinders %d, sectsize %d\n",
1568                 VIOD_DEVICE_NAME,
1569                 diskno,
1570                 (int) (viodasd_devices[diskno].size /
1571                        (1024 * 1024)),
1572                 (int) viodasd_devices[diskno].sectors,
1573                 (int) viodasd_devices[diskno].tracks,
1574                 (int) viodasd_devices[diskno].cylinders,
1575                 (int) hardsect_size[major][dev_within_major <<
1576                                            PARTITION_SHIFT]);
1577
1578          for (i = 1; i < (1 << PARTITION_SHIFT); ++i) {
1579                  int minor = (dev_within_major << PARTITION_SHIFT) + i;
1580                  struct hd_struct *partition = &gendisk->part[minor];
1581                  if (partition->nr_sects)
1582                          printk(KERN_INFO_VIO
1583                                 "%s: Disk %2.2d partition %2.2d start sector %ld, # sector %ld\n",
1584                                 VIOD_DEVICE_NAME, diskno, i,
1585                                 partition->start_sect, partition->nr_sects);
1586          }
1587  }
1588
1589  /* Initialize the whole device driver.  Handle module and non-module
1590   * versions
1591   */
1592  __init int viodasd_init(void)
1593  {
1594          int i, j;
1595          int rc;
1596
1597          /* Try to open to our host lp
1598           */
1599          if (viopath_hostLp == HvLpIndexInvalid) {
1600                  vio_set_hostlp();
1601          }
1602
1603          if (viopath_hostLp == HvLpIndexInvalid) {
1604                  printk(KERN_WARNING_VIO "%s: invalid hosting partition\n",
1605                         VIOD_DEVICE_NAME);
1606                  return -EIO;
1607          }
1608
1609          printk(KERN_INFO_VIO
1610                 "%s: Disk vers %s, major %d, max disks %d, hosting partition %d\n",
1611                 VIOD_DEVICE_NAME, VIODASD_VERS, major_table[0], MAX_DISKNO,
1612                 viopath_hostLp);
1613
1614          if (ROOT_DEV == NODEV) {
1615                  /* first disk, third partition */
1616                  ROOT_DEV = diskno_to_devt(0, 3);
1617
1618                  printk(KERN_INFO_VIO
1619                         "Claiming root file system as third partition of first virtual disk");
1620          }
```

```
1621
1622            /* Actually open the path to the hosting partition          */
1623            rc = viopath_open(viopath_hostLp, viomajorsubtype_blockio,
1624                              VIOMAXREQ + 2);
1625            if (rc) {
1626                    printk(KERN_WARNING_VIO
1627                           "error opening path to host partition %d\n",
1628                           viopath_hostLp);
1629                    return -EIO;
1630            } else {
1631                    printk("%s: opened path to hosting partition %d\n",
1632                           VIOD_DEVICE_NAME, viopath_hostLp);
1633            }
1634
1635            viodasd_devices =
1636                kmalloc(MAX_DISKNO * sizeof(struct viodasd_device),
1637                        GFP_KERNEL);
1638            if (!viodasd_devices)
1639                    return -ENOMEM;
1640            memset(viodasd_devices, 0x00,
1641                   MAX_DISKNO * sizeof(struct viodasd_device));
1642
1643            /*
1644             * Initialize our request handler
1645             */
1646            vio_setHandler(viomajorsubtype_blockio, vioHandleBlockEvent);
1647
1648            for (i = 0; i < NUM_MAJORS; ++i) {
1649                    int init_rc = viodasd_init_major(major_table[i]);
1650                    if (init_rc < 0) {
1651                            for (j = 0; j <= i; ++j)
1652                                    viodasd_cleanup_major(major_table[j]);
1653                            return init_rc;
1654                    }
1655            }
1656
1657            viodasd_max_disk = MAX_DISKNO - 1;
1658            for (i = 0; i <= viodasd_max_disk && i < MAX_DISKNO; i++) {
1659                    // Note that internal_open has side effects:
1660                    //  a) it updates the size of the disk
1661                    //  b) it updates viodasd_max_disk
1662                    //  c) it registers the disk if it has not done so already
1663                    if (internal_open(i, vioblockflags_ro) == 0)
1664                            internal_release(i, vioblockflags_ro);
1665            }
1666
1667            /*
1668             * Create the proc entry
1669             */
1670            iSeries_proc_callback(&viodasd_proc_init);
1671
1672            return 0;
1673    }
1674
1675    #ifdef MODULE
1676    void viodasd_exit(void)
1677    {
1678            int i;
1679            for(i = 0; i < NUM_MAJORS; ++i)
1680                    viodasd_cleanup_major(major_table[i]);
1681
1682            CLEANIT(viodasd_devices);
1683
1684            viopath_close(viopath_hostLp, viomajorsubtype_blockio, VIOMAXREQ + 2);
1685            iSeries_proc_callback(&viodasd_proc_delete);
1686
1687    }
1688    #endif
1689
1690    #ifdef MODULE
1691    module_init(viodasd_init);
1692    module_exit(viodasd_exit);
1693    #endif
```

```
/* −*− linux−c −*−
 *  drivers/char/vio.h
 *
 *  iSeries Virtual I/O Message Path header
 *
 *  Authors: Dave Boutcher <boutcher@us.ibm.com>
 *           Ryan Arnold <ryanarn@us.ibm.com>
 *           Colin Devilbiss <devilbis@us.ibm.com>
 *
 * (C) Copyright 2000 IBM Corporation
 *
 * This header file is used by the iSeries virtual I/O device
 * drivers.  It defines the interfaces to the common functions
 * (implemented in drivers/char/viopath.h) as well as defining
 * common functions and structures.  Currently (at the time I
 * wrote this comment) the iSeries virtual I/O device drivers
 * that use this are
 *   drivers/block/viodasd.c
 *   drivers/char/viocons.c
 *   drivers/char/viotape.c
 *   drivers/cdrom/viocd.c
 *
 * The iSeries virtual ethernet support (veth.c) uses a whole
 * different set of functions.
 *
 * 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) anyu later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 *
 */
#ifndef _VIO_H
#define _VIO_H

#include <asm/iSeries/HvTypes.h>
#include <asm/iSeries/HvLpEvent.h>

/* iSeries virtual I/O events use the subtype field in
 * HvLpEvent to figure out what kind of vio event is coming
 * in.  We use a table to route these, and this defines
 * the maximum number of distinct subtypes
 */
#define VIO_MAX_SUBTYPES 7

/* Each subtype can register a handler to process their events.
 * The handler must have this interface.
 */
typedef void (vio_event_handler_t) (struct HvLpEvent * event);

int viopath_open(HvLpIndex remoteLp, int subtype, int numReq);
int viopath_close(HvLpIndex remoteLp, int subtype, int numReq);
int vio_setHandler(int subtype, vio_event_handler_t * beh);
int vio_clearHandler(int subtype);
int viopath_isactive(HvLpIndex lp);
HvLpInstanceId viopath_sourceinst(HvLpIndex lp);
HvLpInstanceId viopath_targetinst(HvLpIndex lp);
void vio_set_hostlp(void);
void *vio_get_event_buffer(int subtype);
void vio_free_event_buffer(int subtype, void *buffer);

extern HvLpIndex viopath_hostLp;
extern HvLpIndex viopath_ourLp;

#define VIO_MESSAGE "iSeries virtual I/O: "
#define KERN_DEBUG_VIO KERN_DEBUG VIO_MESSAGE
#define KERN_INFO_VIO KERN_INFO VIO_MESSAGE
#define KERN_WARNING_VIO KERN_WARNING VIO_MESSAGE

#define VIOCHAR_MAX_DATA 200

#define VIOMAJOR_SUBTYPE_MASK 0xff00
#define VIOMINOR_SUBTYPE_MASK 0x00ff
#define VIOMAJOR_SUBTYPE_SHIFT 8

#define VIOVERSION            0x0101

/*
This is the general structure for VIO errors; each module should have a table
of them, and each table should be terminated by an entry of { 0, 0, NULL }.
Then, to find a specific error message, a module should pass its local table
and the return code.
```

```
91  */
92  struct vio_error_entry {
93          u16 rc;
94          int errno;
95          const char *msg;
96  };
97  const struct vio_error_entry *vio_lookup_rc(const struct vio_error_entry
98                                              *local_table, u16 rc);
99
100 enum viosubtypes {
101         viomajorsubtype_monitor = 0x0100,
102         viomajorsubtype_blockio = 0x0200,
103         viomajorsubtype_chario = 0x0300,
104         viomajorsubtype_config = 0x0400,
105         viomajorsubtype_cdio = 0x0500,
106         viomajorsubtype_tape = 0x0600
107 };
108
109
110 enum vioconfigsubtype {
111         vioconfigget = 0x0001,
112 };
113
114 enum viorc {
115         viorc_good = 0x0000,
116         viorc_noConnection = 0x0001,
117         viorc_noReceiver = 0x0002,
118         viorc_noBufferAvailable = 0x0003,
119         viorc_invalidMessageType = 0x0004,
120         viorc_invalidRange = 0x0201,
121         viorc_invalidToken = 0x0202,
122         viorc_DMAError = 0x0203,
123         viorc_useError = 0x0204,
124         viorc_releaseError = 0x0205,
125         viorc_invalidDisk = 0x0206,
126         viorc_openRejected = 0x0301
127 };
128
129
130 #endif                          /* _VIO_H */
```

```
/* −*− linux−c −*−
 *  arch/ppc64/viopath.c
 *
 *  iSeries Virtual I/O Message Path code
 *
 *  Authors: Dave Boutcher <boutcher@us.ibm.com>
 *           Ryan Arnold <ryanarn@us.ibm.com>
 *           Colin Devilbiss <devilbis@us.ibm.com>
 *
 * (C) Copyright 2000 IBM Corporation
 *
 * This code is used by the iSeries virtual disk, cd,
 * tape, and console to communicate with OS/400 in another
 * partition.
 *
 * 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) anyu later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 *
 */
#include <linux/config.h>
#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/vmalloc.h>
#include <linux/string.h>
#include <linux/proc_fs.h>
#include <linux/pci.h>
#include <linux/wait.h>

#include <asm/iSeries/LparData.h>
#include <asm/iSeries/HvLpEvent.h>
#include <asm/iSeries/HvLpConfig.h>
#include <asm/iSeries/HvCallCfg.h>
#include <asm/iSeries/mf.h>
#include <asm/iSeries/iSeries_proc.h>

#include "vio.h"

EXPORT_SYMBOL(viopath_hostLp);
EXPORT_SYMBOL(viopath_ourLp);
EXPORT_SYMBOL(vio_set_hostlp);
EXPORT_SYMBOL(vio_lookup_rc);
EXPORT_SYMBOL(viopath_open);
EXPORT_SYMBOL(viopath_close);
EXPORT_SYMBOL(viopath_isactive);
EXPORT_SYMBOL(viopath_sourceinst);
EXPORT_SYMBOL(viopath_targetinst);
EXPORT_SYMBOL(vio_setHandler);
EXPORT_SYMBOL(vio_clearHandler);
EXPORT_SYMBOL(vio_get_event_buffer);
EXPORT_SYMBOL(vio_free_event_buffer);

extern struct pci_dev * iSeries_vio_dev;

/* Status of the path to each other partition in the system.
 * This is overkill, since we will only ever establish connections
 * to our hosting partition and the primary partition on the system.
 * But this allows for other support in the future.
 */
static struct viopathStatus {
        int isOpen:1;           /* Did we open the path?            */
        int isActive:1;         /* Do we have a mon msg outstanding */
        int users[VIO_MAX_SUBTYPES];
        HvLpInstanceId mSourceInst;
        HvLpInstanceId mTargetInst;
        int numberAllocated;
} viopathStatus[HVMAXARCHITECTEDLPS];

static spinlock_t statuslock = SPIN_LOCK_UNLOCKED;

/*
 * For each kind of event we allocate a buffer that is
 * guaranteed not to cross a page boundary
 */
static void *event_buffer[VIO_MAX_SUBTYPES];
static atomic_t event_buffer_available[VIO_MAX_SUBTYPES];

static void handleMonitorEvent(struct HvLpEvent *event);
```

```
91
92  /* We use this structure to handle asynchronous responses.  The caller
93   * blocks on the semaphore and the handler posts the semaphore.
94   */
95  struct doneAllocParms_t {
96          struct semaphore *sem;
97          int number;
98  };
99
100 /* Put a sequence number in each mon msg.  The value is not
101  * important.  Start at something other than 0 just for
102  * readability.  wrapping this is ok.
103  */
104 static u8 viomonseq = 22;
105
106 /* Our hosting logical partition.  We get this at startup
107  * time, and different modules access this variable directly.
108  */
109 HvLpIndex viopath_hostLp = 0xff;        /* HvLpIndexInvalid */
110 HvLpIndex viopath_ourLp = 0xff;
111
112 /* For each kind of incoming event we set a pointer to a
113  * routine to call.
114  */
115 static vio_event_handler_t *vio_handler[VIO_MAX_SUBTYPES];
116
117 static char e2a(char x) {
118         switch (x) {
119         case 0xF0: return '0';
120         case 0xF1: return '1';
121         case 0xF2: return '2';
122         case 0xF3: return '3';
123         case 0xF4: return '4';
124         case 0xF5: return '5';
125         case 0xF6: return '6';
126         case 0xF7: return '7';
127         case 0xF8: return '8';
128         case 0xF9: return '9';
129         case 0xC1: return 'A';
130         case 0xC2: return 'B';
131         case 0xC3: return 'C';
132         case 0xC4: return 'D';
133         case 0xC5: return 'E';
134         case 0xC6: return 'F';
135         case 0xC7: return 'G';
136         case 0xC8: return 'H';
137         case 0xC9: return 'I';
138         case 0xD1: return 'J';
139         case 0xD2: return 'K';
140         case 0xD3: return 'L';
141         case 0xD4: return 'M';
142         case 0xD5: return 'N';
143         case 0xD6: return 'O';
144         case 0xD7: return 'P';
145         case 0xD8: return 'Q';
146         case 0xD9: return 'R';
147         case 0xE2: return 'S';
148         case 0xE3: return 'T';
149         case 0xE4: return 'U';
150         case 0xE5: return 'V';
151         case 0xE6: return 'W';
152         case 0xE7: return 'X';
153         case 0xE8: return 'Y';
154         case 0xE9: return 'Z';
155         }
156         return ' ';
157 }
158
159 /* Handle reads from the proc file system
160  */
161 static int proc_read(char *buf, char **start, off_t offset,
162                      int blen, int *eof, void *data)
163 {
164         HvLpEvent_Rc hvrc;
165         DECLARE_MUTEX_LOCKED(Semaphore);
166         dma_addr_t dmaa =
167             pci_map_single(iSeries_vio_dev, buf, PAGE_SIZE, PCI_DMA_FROMDEVICE);
168         int len = PAGE_SIZE;
169
170         if (len > blen)
171                 len = blen;
172
173         memset(buf, 0x00, len);
174         hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
175                                              HvLpEvent_Type_VirtualIo,
176                                              viomajorsubtype_config |
177                                              vioconfigget,
178                                              HvLpEvent_AckInd_DoAck,
179                                              HvLpEvent_AckType_ImmediateAck,
180                                              viopath_sourceinst
```

```
                                                (viopath_hostLp),
                                                viopath_targetinst
                                                (viopath_hostLp),
                                                (u64) (unsigned long)
                                                &Semaphore, VIOVERSION << 16,
                                                ((u64) dmaa) << 32, len, 0,
                                                0);
        if (hvrc != HvLpEvent_Rc_Good) {
                printk("viopath hv error on op %d\n", (int) hvrc);
        }

        down(&Semaphore);

        pci_unmap_single(iSeries_vio_dev, dmaa, PAGE_SIZE, PCI_DMA_FROMDEVICE);

        sprintf(buf+strlen(buf),"SRLNBR=");
        buf[strlen(buf)] = e2a(xItExtVpdPanel.mfgID[2]);
        buf[strlen(buf)] = e2a(xItExtVpdPanel.mfgID[3]);
        buf[strlen(buf)] = e2a(xItExtVpdPanel.systemSerial[1]);
        buf[strlen(buf)] = e2a(xItExtVpdPanel.systemSerial[2]);
        buf[strlen(buf)] = e2a(xItExtVpdPanel.systemSerial[3]);
        buf[strlen(buf)] = e2a(xItExtVpdPanel.systemSerial[4]);
        buf[strlen(buf)] = e2a(xItExtVpdPanel.systemSerial[5]);
        buf[strlen(buf)] = '\n';        *eof = 1;
        return strlen(buf);
}

/* Handle writes to our proc file system
 */
static int proc_write(struct file *file, const char *buffer,
                      unsigned long count, void *data)
{
        /* Doesn't do anything today!!!
         */
        return count;
}

/* setup our proc file system entries
 */
static void vio_proc_init(struct proc_dir_entry *iSeries_proc)
{
        struct proc_dir_entry *ent;
        ent = create_proc_entry("config", S_IFREG | S_IRUSR, iSeries_proc);
        if (!ent)
                return;
        ent->nlink = 1;
        ent->data = NULL;
        ent->read_proc = proc_read;
        ent->write_proc = proc_write;
}

/* See if a given LP is active.  Allow for invalid lps to be passed in
 * and just return invalid
 */
int viopath_isactive(HvLpIndex lp)
{
        if (lp == HvLpIndexInvalid)
                return 0;
        if (lp < HVMAXARCHITECTEDLPS)
                return viopathStatus[lp].isActive;
        else
                return 0;
}

/* We cache the source and target instance ids for each
 * partition.
 */
HvLpInstanceId viopath_sourceinst(HvLpIndex lp)
{
        return viopathStatus[lp].mSourceInst;
}

HvLpInstanceId viopath_targetinst(HvLpIndex lp)
{
        return viopathStatus[lp].mTargetInst;
}

/* Send a monitor message.  This is a message with the acknowledge
 * bit on that the other side will NOT explicitly acknowledge.  When
 * the other side goes down, the hypervisor will acknowledge any
 * outstanding messages....so we will know when the other side dies.
 */
static void sendMonMsg(HvLpIndex remoteLp)
{
        HvLpEvent_Rc hvrc;

        viopathStatus[remoteLp].mSourceInst =
            HvCallEvent_getSourceLpInstanceId(remoteLp,
                                              HvLpEvent_Type_VirtualIo);
        viopathStatus[remoteLp].mTargetInst =
```

```
271             HvCallEvent_getTargetLpInstanceId(remoteLp,
272                                               HvLpEvent_Type_VirtualIo);
273
274         /* Deliberately ignore the return code here.  if we call this
275          * more than once, we don't care.
276          */
277         vio_setHandler(viomajorsubtype_monitor, handleMonitorEvent);
278
279         hvrc = HvCallEvent_signalLpEventFast(remoteLp,
280                                              HvLpEvent_Type_VirtualIo,
281                                              viomajorsubtype_monitor,
282                                              HvLpEvent_AckInd_DoAck,
283                                              HvLpEvent_AckType_DeferredAck,
284                                              viopathStatus[remoteLp].
285                                              mSourceInst,
286                                              viopathStatus[remoteLp].
287                                              mTargetInst, viomonseq++,
288                                              0, 0, 0, 0, 0);
289
290         if (hvrc == HvLpEvent_Rc_Good) {
291                 viopathStatus[remoteLp].isActive = 1;
292         } else {
293                 printk(KERN_WARNING_VIO
294                        "could not connect to partition %d\n", remoteLp);
295                 viopathStatus[remoteLp].isActive = 0;
296         }
297 }
298
299 static void handleMonitorEvent(struct HvLpEvent *event)
300 {
301         HvLpIndex remoteLp;
302         int i;
303
304         /* This handler is _also_ called as part of the loop
305          * at the end of this routine, so it must be able to
306          * ignore NULL events...
307          */
308         if(!event)
309                 return;
310
311         /* First see if this is just a normal monitor message from the
312          * other partition
313          */
314         if (event->xFlags.xFunction == HvLpEvent_Function_Int) {
315                 remoteLp = event->xSourceLp;
316                 if (!viopathStatus[remoteLp].isActive)
317                         sendMonMsg(remoteLp);
318                 return;
319         }
320
321         /* This path is for an acknowledgement; the other partition
322          * died
323          */
324         remoteLp = event->xTargetLp;
325         if ((event->xSourceInstanceId !=
326              viopathStatus[remoteLp].mSourceInst)
327             || (event->xTargetInstanceId !=
328                 viopathStatus[remoteLp].mTargetInst)) {
329                 printk(KERN_WARNING_VIO
330                        "ignoring ack....mismatched instances\n");
331                 return;
332         }
333
334         printk(KERN_WARNING_VIO "partition %d ended\n", remoteLp);
335
336         viopathStatus[remoteLp].isActive = 0;
337
338         /* For each active handler, pass them a NULL
339          * message to indicate that the other partition
340          * died
341          */
342         for (i = 0; i < VIO_MAX_SUBTYPES; i++) {
343                 if (vio_handler[i] != NULL)
344                         (*vio_handler[i]) (NULL);
345         }
346 }
347
348 int vio_setHandler(int subtype, vio_event_handler_t * beh)
349 {
350         subtype = subtype >> VIOMAJOR_SUBTYPE_SHIFT;
351
352         if ((subtype < 0) || (subtype >= VIO_MAX_SUBTYPES))
353                 return -EINVAL;
354
355         if (vio_handler[subtype] != NULL)
356                 return -EBUSY;
357
358         vio_handler[subtype] = beh;
359         return 0;
360 }
```

```
361
362 int vio_clearHandler(int subtype)
363 {
364         subtype = subtype >> VIOMAJOR_SUBTYPE_SHIFT;
365
366         if ((subtype < 0) || (subtype >= VIO_MAX_SUBTYPES))
367                 return −EINVAL;
368
369         if (vio_handler[subtype] == NULL)
370                 return −EAGAIN;
371
372         vio_handler[subtype] = NULL;
373         return 0;
374 }
375
376 static void handleConfig(struct HvLpEvent *event)
377 {
378         if(!event)
379                 return;
380         if (event−>xFlags.xFunction == HvLpEvent_Function_Int) {
381                 printk(KERN_WARNING_VIO
382                        "unexpected config request from partition %d",
383                        event−>xSourceLp);
384
385                 if ((event−>xFlags.xFunction == HvLpEvent_Function_Int) &&
386                     (event−>xFlags.xAckInd == HvLpEvent_AckInd_DoAck)) {
387                         event−>xRc = HvLpEvent_Rc_InvalidSubtype;
388                         HvCallEvent_ackLpEvent(event);
389                 }
390                 return;
391         }
392
393         up((struct semaphore *) event−>xCorrelationToken);
394 }
395
396 /* Initialization of the hosting partition
397  */
398 void vio_set_hostlp(void)
399 {
400         /* If this has already been set then we DON'T want to either change
401          * it or re−register the proc file system
402          */
403         if (viopath_hostLp != HvLpIndexInvalid)
404                 return;
405
406         /* Figure out our hosting partition.  This isn't allowed to change
407          * while we're active
408          */
409         viopath_ourLp = HvLpConfig_getLpIndex();
410         viopath_hostLp = HvCallCfg_getHostingLpIndex(viopath_ourLp);
411
412         /* If we have a valid hosting LP, create a proc file system entry
413          * for config information
414          */
415         if (viopath_hostLp != HvLpIndexInvalid) {
416                 iSeries_proc_callback(&vio_proc_init);
417                 vio_setHandler(viomajorsubtype_config, handleConfig);
418         }
419 }
420
421 static void vio_handleEvent(struct HvLpEvent *event, struct pt_regs *regs)
422 {
423         HvLpIndex remoteLp;
424         int subtype =
425             (event−>
426              xSubtype & VIOMAJOR_SUBTYPE_MASK) >> VIOMAJOR_SUBTYPE_SHIFT;
427
428         if (event−>xFlags.xFunction == HvLpEvent_Function_Int) {
429                 remoteLp = event−>xSourceLp;
430                 /* The isActive is checked because if the hosting partition
431                  * went down and came back up it would not be active but it would have
432                  * different source and target instances, in which case we'd want to
433                  * reset them.  This case really protects against an unauthorized
434                  * active partition sending interrupts or acks to this linux partition.
435                  */
436                 if (viopathStatus[remoteLp].isActive && (event−>xSourceInstanceId !=
437                     viopathStatus[remoteLp].mTargetInst)) {
438                         printk(KERN_WARNING_VIO
439                                "message from invalid partition. "
440                                "int msg rcvd, source inst (%d) doesnt match (%d)\n",
441                                viopathStatus[remoteLp].mTargetInst,
442                                event−>xSourceInstanceId);
443                         return;
444                 }
445
446                 if (viopathStatus[remoteLp].isActive && (event−>xTargetInstanceId !=
447                     viopathStatus[remoteLp].mSourceInst)) {
448                         printk(KERN_WARNING_VIO
449                                "message from invalid partition. "
450                                "int msg rcvd, target inst (%d) doesnt match (%d)\n",
```

```
451                                 viopathStatus[remoteLp].mSourceInst,
452                                 event->xTargetInstanceId);
453                         return;
454                 }
455         } else {
456                 remoteLp = event->xTargetLp;
457                 if (event->xSourceInstanceId !=
458                     viopathStatus[remoteLp].mSourceInst) {
459                         printk(KERN_WARNING_VIO
460                                 "message from invalid partition. "
461                                 "ack msg rcvd, source inst (%d) doesnt match (%d)\n",
462                                 viopathStatus[remoteLp].mSourceInst,
463                                 event->xSourceInstanceId);
464                         return;
465                 }
466
467                 if (event->xTargetInstanceId !=
468                     viopathStatus[remoteLp].mTargetInst) {
469                         printk(KERN_WARNING_VIO
470                                 "message from invalid partition. "
471                                 "viopath: ack msg rcvd, target inst (%d) doesnt match (%d)\n",
472                                 viopathStatus[remoteLp].mTargetInst,
473                                 event->xTargetInstanceId);
474                         return;
475                 }
476         }
477
478         if (vio_handler[subtype] == NULL) {
479                 printk(KERN_WARNING_VIO
480                        "unexpected virtual io event subtype %d from partition %d\n",
481                        event->xSubtype, remoteLp);
482                 /* No handler.  Ack if necessary
483                  */
484                 if ((event->xFlags.xFunction == HvLpEvent_Function_Int) &&
485                     (event->xFlags.xAckInd == HvLpEvent_AckInd_DoAck)) {
486                         event->xRc = HvLpEvent_Rc_InvalidSubtype;
487                         HvCallEvent_ackLpEvent(event);
488                 }
489                 return;
490         }
491
492         /* This innocuous little line is where all the real work happens
493          */
494         (*vio_handler[subtype]) (event);
495 }
496
497 static void viopath_donealloc(void *parm, int number)
498 {
499         struct doneAllocParms_t *doneAllocParmsp =
500             (struct doneAllocParms_t *) parm;
501         doneAllocParmsp->number = number;
502         up(doneAllocParmsp->sem);
503 }
504
505 static int allocateEvents(HvLpIndex remoteLp, int numEvents)
506 {
507         struct doneAllocParms_t doneAllocParms;
508         DECLARE_MUTEX_LOCKED(Semaphore);
509         doneAllocParms.sem = &Semaphore;
510
511         mf_allocateLpEvents(remoteLp, HvLpEvent_Type_VirtualIo, 250,    /* It would be nice to put a real number here! */
512                             numEvents,
513                             &viopath_donealloc, &doneAllocParms);
514
515         down(&Semaphore);
516
517         return doneAllocParms.number;
518 }
519
520 int viopath_open(HvLpIndex remoteLp, int subtype, int numReq)
521 {
522         int i;
523         unsigned long flags;
524
525         if ((remoteLp >= HvMaxArchitectedLps)
526             || (remoteLp == HvLpIndexInvalid))
527                 return -EINVAL;
528
529         subtype = subtype >> VIOMAJOR_SUBTYPE_SHIFT;
530         if ((subtype < 0) || (subtype >= VIO_MAX_SUBTYPES))
531                 return -EINVAL;
532
533         spin_lock_irqsave(&statuslock, flags);
534
535         /* OK...we can fit 4 maximum-sized events (256 bytes) in
536          * each page (4096).  Get a new page every 4
537          */
538         if (event_buffer[0] == NULL) {
539                 for (i = 0; i < VIO_MAX_SUBTYPES; i++) {
```

```
                        if ((i % 4) == 0) {
                                event_buffer[i] =
                                    (void *) get_free_page(GFP_KERNEL);
                                if (event_buffer[i] == NULL) {
                                        spin_unlock_irqrestore(&statuslock, flags);
                                        return -ENOMEM;
                                }
                        } else {
                                event_buffer[i] =
                                    event_buffer[i - 1] + 256;
                        }
                        atomic_set(&event_buffer_available[i], 1);
                }
        }

        viopathStatus[remoteLp].users[subtype]++;

        if (!viopathStatus[remoteLp].isOpen) {
                HvCallEvent_openLpEventPath(remoteLp,
                                            HvLpEvent_Type_VirtualIo);

                viopathStatus[remoteLp].numberAllocated +=
                    allocateEvents(remoteLp, 1);

                if (viopathStatus[remoteLp].numberAllocated == 0) {
                        HvCallEvent_closeLpEventPath(remoteLp,
                                                     HvLpEvent_Type_VirtualIo);

                        spin_unlock_irqrestore(&statuslock, flags);
                        return -ENOMEM;
                }

                viopathStatus[remoteLp].mSourceInst =
                    HvCallEvent_getSourceLpInstanceId(remoteLp,
                                                      HvLpEvent_Type_VirtualIo);
                viopathStatus[remoteLp].mTargetInst =
                    HvCallEvent_getTargetLpInstanceId(remoteLp,
                                                      HvLpEvent_Type_VirtualIo);

                HvLpEvent_registerHandler(HvLpEvent_Type_VirtualIo,
                                          &vio_handleEvent);

                viopathStatus[remoteLp].isOpen = 1;

                sendMonMsg(remoteLp);

                printk(KERN_INFO_VIO
                        "Opening connection to partition %d, setting sinst %d, tinst %d\n",
                        remoteLp,
                        viopathStatus[remoteLp].mSourceInst,
                        viopathStatus[remoteLp].mTargetInst);
        }

        viopathStatus[remoteLp].numberAllocated +=
            allocateEvents(remoteLp, numReq);
        spin_unlock_irqrestore(&statuslock, flags);

        return 0;
}

int viopath_close(HvLpIndex remoteLp, int subtype, int numReq)
{
        unsigned long flags;
        int i;
        int numOpen;
        struct doneAllocParms_t doneAllocParms;
        DECLARE_MUTEX_LOCKED(Semaphore);
        doneAllocParms.sem = &Semaphore;

        if ((remoteLp >= HvMaxArchitectedLps)
            || (remoteLp == HvLpIndexInvalid))
                return -EINVAL;

        subtype = subtype >> VIOMAJOR_SUBTYPE_SHIFT;
        if ((subtype < 0) || (subtype >= VIO_MAX_SUBTYPES))
                return -EINVAL;

        spin_lock_irqsave(&statuslock, flags);

        viopathStatus[remoteLp].users[subtype]--;

        mf_deallocateLpEvents( remoteLp,HvLpEvent_Type_VirtualIo,
                               numReq,
                               &viopath_donealloc,
                               &doneAllocParms );
        down(&Semaphore);

        for (i = 0, numOpen = 0; i < VIO_MAX_SUBTYPES; i++) {
                numOpen += viopathStatus[remoteLp].users[i];
        }
```

```
630
631         if ((viopathStatus[remoteLp].isOpen) && (numOpen == 0)) {
632                 printk(KERN_INFO_VIO
633                        "Closing connection to partition %d", remoteLp);
634
635                 HvCallEvent_closeLpEventPath(remoteLp,
636                                              HvLpEvent_Type_VirtualIo);
637                 viopathStatus[remoteLp].isOpen = 0;
638                 viopathStatus[remoteLp].isActive = 0;
639
640                 for (i = 0; i < VIO_MAX_SUBTYPES; i++) {
641                         atomic_set(&event_buffer_available[i], 0);
642
643                         for (i = 0; i < VIO_MAX_SUBTYPES; i += 4) {
644                                 free_page((unsigned long) event_buffer[i]);
645                         }
646                 }
647
648         }
649         spin_unlock_irqrestore(&statuslock, flags);
650         return 0;
651 }
652
653 void *vio_get_event_buffer(int subtype)
654 {
655         subtype = subtype >> VIOMAJOR_SUBTYPE_SHIFT;
656         if ((subtype < 0) || (subtype >= VIO_MAX_SUBTYPES))
657                 return NULL;
658
659         if (atomic_dec_if_positive(&event_buffer_available[subtype]) == 0)
660                 return event_buffer[subtype];
661         else
662                 return NULL;
663 }
664
665 void vio_free_event_buffer(int subtype, void *buffer)
666 {
667         subtype = subtype >> VIOMAJOR_SUBTYPE_SHIFT;
668         if ((subtype < 0) || (subtype >= VIO_MAX_SUBTYPES)) {
669                 printk(KERN_WARNING_VIO
670                        "unexpected subtype %d freeing event buffer\n",
671                        subtype);
672                 return;
673         }
674
675         if (atomic_read(&event_buffer_available[subtype]) != 0) {
676                 printk(KERN_WARNING_VIO
677                        "freeing unallocated event buffer, subtype %d\n",
678                        subtype);
679                 return;
680         }
681
682         if (buffer != event_buffer[subtype]) {
683                 printk(KERN_WARNING_VIO
684                        "freeing invalid event buffer, subtype %d\n",
685                        subtype);
686         }
687
688         atomic_set(&event_buffer_available[subtype], 1);
689 }
690
691 static const struct vio_error_entry vio_no_error =
692     { 0, 0, "Non−VIO Error" };
693 static const struct vio_error_entry vio_unknown_error =
694     { 0, EIO, "Unknown Error" };
695
696 static const struct vio_error_entry vio_default_errors[] = {
697         {0x0001, EIO, "No Connection"},
698         {0x0002, EIO, "No Receiver"},
699         {0x0003, EIO, "No Buffer Available"},
700         {0x0004, EBADRQC, "Invalid Message Type"},
701         {0x0000, 0, NULL},
702 };
703
704 const struct vio_error_entry *vio_lookup_rc(const struct vio_error_entry
705                                             *local_table, u16 rc)
706 {
707         const struct vio_error_entry *cur;
708         if (!rc)
709                 return &vio_no_error;
710         if (local_table)
711                 for (cur = local_table; cur−>rc; ++cur)
712                         if (cur−>rc == rc)
713                                 return cur;
714         for (cur = vio_default_errors; cur−>rc; ++cur)
715                 if (cur−>rc == rc)
716                         return cur;
717         return &vio_unknown_error;
718 }
```

```
/* −*− linux−c −*−
 *  drivers/char/viotape.c
 *
 *  iSeries Virtual Tape
 ****************************************************************************
 *
 *  Authors: Dave Boutcher <boutcher@us.ibm.com>
 *           Ryan Arnold <ryanarn@us.ibm.com>
 *           Colin Devilbiss <devilbis@us.ibm.com>
 *
 * (C) Copyright 2000 IBM Corporation
 *
 * 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) anyu later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 *
 ****************************************************************************
 * This routine provides access to tape drives owned and managed by an OS/400
 * partition running on the same box as this Linux partition.
 *
 * All tape operations are performed by sending messages back and forth to
 * the OS/400 partition.  The format of the messages is defined in
 * iSeries/vio.h
 *
 */


#undef VIOT_DEBUG

#include <linux/config.h>
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/wait.h>
#include <linux/spinlock.h>
#include <asm/ioctls.h>
#include <linux/mtio.h>
#include <linux/pci.h>
#include <linux/devfs_fs.h>
#include <linux/devfs_fs_kernel.h>
#include <asm/uaccess.h>

#include "vio.h"
#include <asm/iSeries/HvLpEvent.h>
#include "asm/iSeries/HvCallEvent.h"
#include "asm/iSeries/HvLpConfig.h"
#include <asm/iSeries/iSeries_proc.h>

extern struct pci_dev * iSeries_vio_dev;

static int viotape_major = 230;
static int viotape_numdev = 0;

#define VIOTAPE_MAXREQ 1

/* version number for viotape driver */
static unsigned int version_major = 1;
static unsigned int version_minor = 0;

static u64 sndMsgSeq;
static u64 sndMsgAck;
static u64 rcvMsgSeq;
static u64 rcvMsgAck;

/*****************************************************************************
 * The minor number follows the conventions of the SCSI tape drives.  The
 * rewind and mode are encoded in the minor #.  We use this struct to break
 * them out
 *****************************************************************************/
struct viot_devinfo_struct {
        int major;
        int minor;
        int devno;
        int mode;
        int rewind;
};

```

```
91  #define VIOTAPOP_RESET          0
92  #define VIOTAPOP_FSF            1
93  #define VIOTAPOP_BSF            2
94  #define VIOTAPOP_FSR            3
95  #define VIOTAPOP_BSR            4
96  #define VIOTAPOP_WEOF           5
97  #define VIOTAPOP_REW            6
98  #define VIOTAPOP_NOP            7
99  #define VIOTAPOP_EOM            8
100 #define VIOTAPOP_ERASE          9
101 #define VIOTAPOP_SETBLK        10
102 #define VIOTAPOP_SETDENSITY    11
103 #define VIOTAPOP_SETPOS        12
104 #define VIOTAPOP_GETPOS        13
105 #define VIOTAPOP_SETPART       14
106
107 struct viotapelpevent {
108         struct HvLpEvent event;
109         u32 mReserved1;
110         u16 mVersion;
111         u16 mSubTypeRc;
112         u16 mTape;
113         u16 mFlags;
114         u32 mToken;
115         u64 mLen;
116         union {
117                 struct {
118                         u32 mTapeOp;
119                         u32 mCount;
120                 } tapeOp;
121                 struct {
122                         u32 mType;
123                         u32 mResid;
124                         u32 mDsreg;
125                         u32 mGstat;
126                         u32 mErreg;
127                         u32 mFileNo;
128                         u32 mBlkNo;
129                 } getStatus;
130                 struct {
131                         u32 mBlkNo;
132                 } getPos;
133         } u;
134 };
135 enum viotapesubtype {
136         viotapeopen = 0x0001,
137         viotapeclose = 0x0002,
138         viotaperead = 0x0003,
139         viotapewrite = 0x0004,
140         viotapegetinfo = 0x0005,
141         viotapeop = 0x0006,
142         viotapegetpos = 0x0007,
143         viotapesetpos = 0x0008,
144         viotapegetstatus = 0x0009
145 };
146
147 enum viotapeRc {
148         viotape_InvalidRange = 0x0601,
149         viotape_InvalidToken = 0x0602,
150         viotape_DMAError = 0x0603,
151         viotape_UseError = 0x0604,
152         viotape_ReleaseError = 0x0605,
153         viotape_InvalidTape = 0x0606,
154         viotape_InvalidOp = 0x0607,
155         viotape_TapeErr = 0x0608,
156
157         viotape_AllocTimedOut = 0x0640,
158         viotape_BOTEnc = 0x0641,
159         viotape_BlankTape = 0x0642,
160         viotape_BufferEmpty = 0x0643,
161         viotape_CleanCartFound = 0x0644,
162         viotape_CmdNotAllowed = 0x0645,
163         viotape_CmdNotSupported = 0x0646,
164         viotape_DataCheck = 0x0647,
165         viotape_DecompressErr = 0x0648,
166         viotape_DeviceTimeout = 0x0649,
167         viotape_DeviceUnavail = 0x064a,
168         viotape_DeviceBusy = 0x064b,
169         viotape_EndOfMedia = 0x064c,
170         viotape_EndOfTape = 0x064d,
171         viotape_EquipCheck = 0x064e,
172         viotape_InsufficientRs = 0x064f,
173         viotape_InvalidLogBlk = 0x0650,
174         viotape_LengthError = 0x0651,
175         viotape_LibDoorOpen = 0x0652,
176         viotape_LoadFailure = 0x0653,
177         viotape_NotCapable = 0x0654,
178         viotape_NotOperational = 0x0655,
179         viotape_NotReady = 0x0656,
180         viotape_OpCancelled = 0x0657,
```

```
181         viotape_PhyLinkErr = 0x0658,
182         viotape_RdyNotBOT = 0x0659,
183         viotape_TapeMark = 0x065a,
184         viotape_WriteProt = 0x065b
185 };
186
187 static const struct vio_error_entry viotape_err_table[] = {
188         {viotape_InvalidRange, EIO, "Internal error"},
189         {viotape_InvalidToken, EIO, "Internal error"},
190         {viotape_DMAError, EIO, "DMA error"},
191         {viotape_UseError, EIO, "Internal error"},
192         {viotape_ReleaseError, EIO, "Internal error"},
193         {viotape_InvalidTape, EIO, "Invalid tape device"},
194         {viotape_InvalidOp, EIO, "Invalid operation"},
195         {viotape_TapeErr, EIO, "Tape error"},
196         {viotape_AllocTimedOut, EBUSY, "Allocate timed out"},
197         {viotape_BOTEnc, EIO, "Beginning of tape encountered"},
198         {viotape_BlankTape, EIO, "Blank tape"},
199         {viotape_BufferEmpty, EIO, "Buffer empty"},
200         {viotape_CleanCartFound, ENOMEDIUM, "Cleaning cartridge found"},
201         {viotape_CmdNotAllowed, EIO, "Command not allowed"},
202         {viotape_CmdNotSupported, EIO, "Command not supported"},
203         {viotape_DataCheck, EIO, "Data check"},
204         {viotape_DecompressErr, EIO, "Decompression error"},
205         {viotape_DeviceTimeout, EBUSY, "Device timeout"},
206         {viotape_DeviceUnavail, EIO, "Device unavailable"},
207         {viotape_DeviceBusy, EBUSY, "Device busy"},
208         {viotape_EndOfMedia, ENOSPC, "End of media"},
209         {viotape_EndOfTape, ENOSPC, "End of tape"},
210         {viotape_EquipCheck, EIO, "Equipment check"},
211         {viotape_InsufficientRs, EOVERFLOW, "Insufficient tape resources"},
212         {viotape_InvalidLogBlk, EIO, "Invalid logical block location"},
213         {viotape_LengthError, EOVERFLOW, "Length error"},
214         {viotape_LibDoorOpen, EBUSY, "Door open"},
215         {viotape_LoadFailure, ENOMEDIUM, "Load failure"},
216         {viotape_NotCapable, EIO, "Not capable"},
217         {viotape_NotOperational, EIO, "Not operational"},
218         {viotape_NotReady, EIO, "Not ready"},
219         {viotape_OpCancelled, EIO, "Operation cancelled"},
220         {viotape_PhyLinkErr, EIO, "Physical link error"},
221         {viotape_RdyNotBOT, EIO, "Ready but not beginning of tape"},
222         {viotape_TapeMark, EIO, "Tape mark"},
223         {viotape_WriteProt, EROFS, "Write protection error"},
224         {0, 0, NULL},
225 };
226
227 /* Maximum # tapes we support
228  */
229 #define VIOTAPE_MAX_TAPE 8
230 #define MAX_PARTITIONS 4
231
232 /* defines for current tape state */
233 #define VIOT_IDLE 0
234 #define VIOT_READING 1
235 #define VIOT_WRITING 2
236
237 /* Our info on the tapes
238  */
239 struct tape_descr {
240         char rsrcname[10];
241         char type[4];
242         char model[3];
243 };
244
245 static struct tape_descr *viotape_unitinfo = NULL;
246
247 static const char *lasterr[VIOTAPE_MAX_TAPE];
248
249 static struct mtget viomtget[VIOTAPE_MAX_TAPE];
250
251 /* maintain the current state of each tape (and partition)
252    so that we know when to write EOF marks.
253 */
254 static struct {
255         unsigned char cur_part;
256         devfs_handle_t dev_handle;
257         struct {
258                 unsigned char rwi;
259         } part_stat[MAX_PARTITIONS];
260 } state[VIOTAPE_MAX_TAPE];
261
262 /* We single-thread
263  */
264 static struct semaphore reqSem;
265
266 /* When we send a request, we use this struct to get the response back
267  * from the interrupt handler
268  */
269 struct opStruct {
270         void *buffer;
```

```
271         dma_addr_t dmaaddr;
272         size_t count;
273         int rc;
274         struct semaphore *sem;
275         struct opStruct *free;
276 };
277
278 static spinlock_t opStructListLock;
279 static struct opStruct *opStructList;
280
281 /* forward declaration to resolve interdependence */
282 static int chg_state(int index, unsigned char new_state,
283                      struct file *file);
284
285 /* Decode the kdev_t into its parts
286  */
287 void getDevInfo(kdev_t dev, struct viot_devinfo_struct *devi)
288 {
289         devi->major = MAJOR(dev);
290         devi->minor = MINOR(dev);
291         devi->devno = devi->minor & 0x1F;
292         devi->mode = (devi->minor & 0x60) >> 5;
293         /* if bit is set in the minor, do _not_ rewind automatically */
294         devi->rewind = !(devi->minor & 0x80);
295 }
296
297
298 /* Allocate an op structure from our pool
299  */
300 static struct opStruct *getOpStruct(void)
301 {
302         struct opStruct *newOpStruct;
303         spin_lock(&opStructListLock);
304
305         if (opStructList == NULL) {
306                 newOpStruct = kmalloc(sizeof(struct opStruct), GFP_KERNEL);
307         } else {
308                 newOpStruct = opStructList;
309                 opStructList = opStructList->free;
310         }
311
312         if (newOpStruct)
313                 memset(newOpStruct, 0x00, sizeof(struct opStruct));
314
315         spin_unlock(&opStructListLock);
316
317         return newOpStruct;
318 }
319
320 /* Return an op structure to our pool
321  */
322 static void freeOpStruct(struct opStruct *opStruct)
323 {
324         spin_lock(&opStructListLock);
325         opStruct->free = opStructList;
326         opStructList = opStruct;
327         spin_unlock(&opStructListLock);
328 }
329
330 /* Map our tape return codes to errno values
331  */
332 int tapeRcToErrno(int tapeRc, char *operation, int tapeno)
333 {
334         const struct vio_error_entry *err;
335         if(tapeRc == 0)
336                 return 0;
337         err = vio_lookup_rc(viotape_err_table, tapeRc);
338
339         printk(KERN_WARNING_VIO "tape error 0x%04x on Device %d (%−10s): %s\n",
340                tapeRc, tapeno, viotape_unitinfo[tapeno].rsrcname, err->msg);
341
342         lasterr[tapeno] = err->msg;
343
344         return −err->errno;
345 }
346
347 /* Handle reads from the proc file system.
348  */
349 static int proc_read(char *buf, char **start, off_t offset,
350                      int blen, int *eof, void *data)
351 {
352         int len = 0;
353         int i;
354
355         len += sprintf(buf + len, "viotape driver version %d.%d\n",
356                        version_major, version_minor);
357
358         for (i = 0; i < viotape_numdev; i++) {
359
360                 len +=
```

```
361                     sprintf(buf + len,
362                             "viotape device %d is iSeries resource %10.10s type %4.4s, model %3.3s\n",
363                             i, viotape_unitinfo[i].rsrcname,
364                             viotape_unitinfo[i].type,
365                             viotape_unitinfo[i].model);
366             if (lasterr[i])
367                     len +=
368                         sprintf(buf + len, "  last error: %s\n",
369                                 lasterr[i]);
370         }
371
372         *eof = 1;
373         return len;
374 }
375
376 /* setup our proc file system entries
377  */
378 void viotape_proc_init(struct proc_dir_entry *iSeries_proc)
379 {
380         struct proc_dir_entry *ent;
381         ent =
382             create_proc_entry("viotape", S_IFREG | S_IRUSR, iSeries_proc);
383         if (!ent)
384                 return;
385         ent->nlink = 1;
386         ent->data = NULL;
387         ent->read_proc = proc_read;
388 }
389
390 /* clean up our proc file system entries
391  */
392 void viotape_proc_delete(struct proc_dir_entry *iSeries_proc)
393 {
394         remove_proc_entry("viotape", iSeries_proc);
395 }
396
397
398 /* Get info on all tapes from OS/400
399  */
400 static void get_viotape_info(void)
401 {
402         dma_addr_t dmaaddr;
403         HvLpEvent_Rc hvrc;
404         int i;
405         struct opStruct *op = getOpStruct();
406         DECLARE_MUTEX_LOCKED(Semaphore);
407         if (op == NULL)
408                 return;
409
410         if (viotape_unitinfo == NULL) {
411                 viotape_unitinfo =
412                     kmalloc(sizeof(struct tape_descr) * VIOTAPE_MAX_TAPE,
413                             GFP_KERNEL);
414         }
415         memset(viotape_unitinfo, 0x00,
416                sizeof(struct tape_descr) * VIOTAPE_MAX_TAPE);
417         memset(lasterr, 0x00, sizeof(lasterr));
418
419         op->sem = &Semaphore;
420
421         dmaaddr = pci_map_single(iSeries_vio_dev, viotape_unitinfo,
422                                  sizeof(struct tape_descr) *
423                                  VIOTAPE_MAX_TAPE, PCI_DMA_FROMDEVICE);
424         if (dmaaddr == 0xFFFFFFFF) {
425                 printk(KERN_WARNING_VIO "viotape error allocating tce\n");
426                 return;
427         }
428
429         hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
430                                              HvLpEvent_Type_VirtualIo,
431                                              viomajorsubtype_tape |
432                                              viotapegetinfo,
433                                              HvLpEvent_AckInd_DoAck,
434                                              HvLpEvent_AckType_ImmediateAck,
435                                              viopath_sourceinst
436                                              (viopath_hostLp),
437                                              viopath_targetinst
438                                              (viopath_hostLp),
439                                              (u64) (unsigned long) op,
440                                              VIOVERSION << 16, dmaaddr,
441                                              sizeof(struct tape_descr) *
442                                              VIOTAPE_MAX_TAPE, 0, 0);
443         if (hvrc != HvLpEvent_Rc_Good) {
444                 printk("viotape hv error on op %d\n", (int) hvrc);
445         }
446
447         down(&Semaphore);
448
449         freeOpStruct(op);
450
```

```
451
452         for (i = 0;
453              ((i < VIOTAPE_MAX_TAPE) && (viotape_unitinfo[i].rsrcname[0]));
454              i++) {
455                 printk("found a tape %10.10s\n",
456                        viotape_unitinfo[i].rsrcname);
457                 viotape_numdev++;
458         }
459 }
460
461
462 /* Write
463  */
464 static ssize_t viotap_write(struct file *file, const char *buf,
465                             size_t count, loff_t * ppos)
466 {
467         HvLpEvent_Rc hvrc;
468         kdev_t dev = file->f_dentry->d_inode->i_rdev;
469         unsigned short flags = file->f_flags;
470         struct opStruct *op = getOpStruct();
471         int noblock = ((flags & O_NONBLOCK) != 0);
472         int err;
473         struct viot_devinfo_struct devi;
474         DECLARE_MUTEX_LOCKED(Semaphore);
475
476         if (op == NULL)
477                 return -ENOMEM;
478
479         getDevInfo(dev, &devi);
480
481         /* We need to make sure we can send a request.  We use
482          * a semaphore to keep track of # requests in use.  If
483          * we are non-blocking, make sure we don't block on the
484          * semaphore
485          */
486         if (noblock) {
487                 if (down_trylock(&reqSem)) {
488                         freeOpStruct(op);
489                         return -EWOULDBLOCK;
490                 }
491         } else {
492                 down(&reqSem);
493         }
494
495         /* Allocate a DMA buffer */
496         op->buffer = pci_alloc_consistent(iSeries_vio_dev, count, &op->dmaaddr);
497
498         if ((op->dmaaddr == 0xFFFFFFFF) || (op->buffer == NULL)) {
499                 printk(KERN_WARNING_VIO
500                        "tape error allocating dma buffer for len %ld\n",
501                        count);
502                 freeOpStruct(op);
503                 up(&reqSem);
504                 return -EFAULT;
505         }
506
507         op->count = count;
508
509         /* Copy the data into the buffer */
510         err = copy_from_user(op->buffer, (const void *) buf, count);
511         if (err) {
512                 printk(KERN_WARNING_VIO
513                        "tape: error on copy from user\n");
514                 pci_free_consistent(iSeries_vio_dev, count, op->buffer, op->dmaaddr);
515                 freeOpStruct(op);
516                 up(&reqSem);
517                 return -EFAULT;
518         }
519
520         if (noblock) {
521                 op->sem = NULL;
522         } else {
523                 op->sem = &Semaphore;
524         }
525
526         hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
527                                              HvLpEvent_Type_VirtualIo,
528                                              viomajorsubtype_tape |
529                                              viotapewrite,
530                                              HvLpEvent_AckInd_DoAck,
531                                              HvLpEvent_AckType_ImmediateAck,
532                                              viopath_sourceinst
533                                              (viopath_hostLp),
534                                              viopath_targetinst
535                                              (viopath_hostLp),
536                                              (u64) (unsigned long) op,
537                                              VIOVERSION << 16,
538                                              ((u64) devi.
539                                               devno << 48) | op->dmaaddr,
540                                              count, 0, 0);
```

```
541         if (hvrc != HvLpEvent_Rc_Good) {
542                 printk("viotape hv error on op %d\n", (int) hvrc);
543                 pci_free_consistent(iSeries_vio_dev, count, op->buffer, op->dmaaddr);
544                 freeOpStruct(op);
545                 up(&reqSem);
546                 return -EIO;
547         }
548
549         if (noblock)
550                 return count;
551
552         down(&Semaphore);
553
554         err = op->rc;
555
556         /* Free the buffer */
557         pci_free_consistent(iSeries_vio_dev, count, op->buffer, op->dmaaddr);
558
559         count = op->count;
560
561         freeOpStruct(op);
562         up(&reqSem);
563         if (err)
564                 return tapeRcToErrno(err, "write", devi.devno);
565         else {
566                 chg_state(devi.devno, VIOT_WRITING, file);
567                 return count;
568         }
569 }
570
571 /* read
572  */
573 static ssize_t viotap_read(struct file *file, char *buf, size_t count,
574                            loff_t * ptr)
575 {
576         HvLpEvent_Rc hvrc;
577         kdev_t dev = file->f_dentry->d_inode->i_rdev;
578         unsigned short flags = file->f_flags;
579         struct opStruct *op = getOpStruct();
580         int noblock = ((flags & O_NONBLOCK) != 0);
581         int err;
582         struct viot_devinfo_struct devi;
583         DECLARE_MUTEX_LOCKED(Semaphore);
584
585         if (op == NULL)
586                 return -ENOMEM;
587
588         getDevInfo(dev, &devi);
589
590         /* We need to make sure we can send a request.  We use
591          * a semaphore to keep track of # requests in use.  If
592          * we are non-blocking, make sure we don't block on the
593          * semaphore
594          */
595         if (noblock) {
596                 if (down_trylock(&reqSem)) {
597                         freeOpStruct(op);
598                         return -EWOULDBLOCK;
599                 }
600         } else {
601                 down(&reqSem);
602         }
603
604         chg_state(devi.devno, VIOT_READING, file);
605
606         /* Allocate a DMA buffer */
607         op->buffer = pci_alloc_consistent(iSeries_vio_dev, count, &op->dmaaddr);
608
609         if ((op->dmaaddr == 0xFFFFFFFF) || (op->buffer == NULL)) {
610                 freeOpStruct(op);
611                 up(&reqSem);
612                 return -EFAULT;
613         }
614
615         op->count = count;
616
617         op->sem = &Semaphore;
618
619         hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
620                                              HvLpEvent_Type_VirtualIo,
621                                              viomajorsubtype_tape |
622                                              viotaperead,
623                                              HvLpEvent_AckInd_DoAck,
624                                              HvLpEvent_AckType_ImmediateAck,
625                                              viopath_sourceinst
626                                              (viopath_hostLp),
627                                              viopath_targetinst
628                                              (viopath_hostLp),
629                                              (u64) (unsigned long) op,
630                                              VIOVERSION << 16,
```

```
                                                ((u64) devi.
                                                 devno << 48) | op->dmaaddr,
                                                count, 0, 0);
        if (hvrc != HvLpEvent_Rc_Good) {
                printk(KERN_WARNING_VIO
                       "tape hv error on op %d\n", (int) hvrc);
                pci_free_consistent(iSeries_vio_dev, count, op->buffer, op->dmaaddr);
                freeOpStruct(op);
                up(&reqSem);
                return -EIO;
        }

        down(&Semaphore);

        if (op->rc == 0) {
                /* If we got data back        */
                if (op->count) {
                        /* Copy the data into the buffer */
                        err = copy_to_user(buf, op->buffer, count);
                        if (err) {
                                printk("error on copy_to_user\n");
                                pci_free_consistent(iSeries_vio_dev, count,
                                                    op->buffer,
                                                    op->dmaaddr);
                                freeOpStruct(op);
                                up(&reqSem);
                                return -EFAULT;
                        }
                }
        }

        err = op->rc;

        /* Free the buffer */
        pci_free_consistent(iSeries_vio_dev, count, op->buffer, op->dmaaddr);
        count = op->count;

        freeOpStruct(op);
        up(&reqSem);
        if (err)
                return tapeRcToErrno(err, "read", devi.devno);
        else
                return count;
}

/* read
 */
static int viotap_ioctl(struct inode *inode, struct file *file,
                        unsigned int cmd, unsigned long arg)
{
        HvLpEvent_Rc hvrc;
        int err;
        DECLARE_MUTEX_LOCKED(Semaphore);
        kdev_t dev = file->f_dentry->d_inode->i_rdev;
        struct opStruct *op = getOpStruct();
        struct viot_devinfo_struct devi;
        if (op == NULL)
                return -ENOMEM;

        getDevInfo(dev, &devi);

        down(&reqSem);

        switch (cmd) {
        case MTIOCTOP:{
                        struct mtop mtc;
                        u32 myOp;

                        /* inode is null if and only if we (the kernel) made the request */
                        if (inode == NULL)
                                memcpy(&mtc, (void *) arg,
                                       sizeof(struct mtop));
                        else if (copy_from_user
                                 ((char *) &mtc, (char *) arg,
                                  sizeof(struct mtop))) {
                                freeOpStruct(op);
                                up(&reqSem);
                                return -EFAULT;
                        }

                        switch (mtc.mt_op) {
                        case MTRESET:
                                myOp = VIOTAPOP_RESET;
                                break;
                        case MTFSF:
                                myOp = VIOTAPOP_FSF;
                                break;
                        case MTBSF:
                                myOp = VIOTAPOP_BSF;
                                break;
```

```
721                             case MTFSR:
722                                     myOp = VIOTAPOP_FSR;
723                                     break;
724                             case MTBSR:
725                                     myOp = VIOTAPOP_BSR;
726                                     break;
727                             case MTWEOF:
728                                     myOp = VIOTAPOP_WEOF;
729                                     break;
730                             case MTREW:
731                                     myOp = VIOTAPOP_REW;
732                                     break;
733                             case MTNOP:
734                                     myOp = VIOTAPOP_NOP;
735                                     break;
736                             case MTEOM:
737                                     myOp = VIOTAPOP_EOM;
738                                     break;
739                             case MTERASE:
740                                     myOp = VIOTAPOP_ERASE;
741                                     break;
742                             case MTSETBLK:
743                                     myOp = VIOTAPOP_SETBLK;
744                                     break;
745                             case MTSETDENSITY:
746                                     myOp = VIOTAPOP_SETDENSITY;
747                                     break;
748                             case MTTELL:
749                                     myOp = VIOTAPOP_GETPOS;
750                                     break;
751                             case MTSEEK:
752                                     myOp = VIOTAPOP_SETPOS;
753                                     break;
754                             case MTSETPART:
755                                     myOp = VIOTAPOP_SETPART;
756                                     break;
757                             default:
758                                     return -EIO;
759                             }
760
761     /* if we moved the head, we are no longer reading or writing */
762                             switch (mtc.mt_op) {
763                             case MTFSF:
764                             case MTBSF:
765                             case MTFSR:
766                             case MTBSR:
767                             case MTTELL:
768                             case MTSEEK:
769                             case MTREW:
770                                     chg_state(devi.devno, VIOT_IDLE, file);
771                             }
772
773                             op->sem = &Semaphore;
774                             hvrc =
775                                 HvCallEvent_signalLpEventFast(viopath_hostLp,
776                                                               HvLpEvent_Type_VirtualIo,
777                                                               viomajorsubtype_tape
778                                                               | viotapeop,
779                                                               HvLpEvent_AckInd_DoAck,
780                                                               HvLpEvent_AckType_ImmediateAck,
781                                                               viopath_sourceinst
782                                                               (viopath_hostLp),
783                                                               viopath_targetinst
784                                                               (viopath_hostLp),
785                                                               (u64) (unsigned
786                                                                      long) op,
787                                                               VIOVERSION << 16,
788                                                               ((u64) devi.
789                                                                devno << 48), 0,
790                                                               (((u64) myOp) <<
791                                                                32) | mtc.
792                                                               mt_count, 0);
793                             if (hvrc != HvLpEvent_Rc_Good) {
794                                     printk("viotape hv error on op %d\n",
795                                            (int) hvrc);
796                                     freeOpStruct(op);
797                                     up(&reqSem);
798                                     return -EIO;
799                             }
800                             down(&Semaphore);
801                             if (op->rc) {
802                                     freeOpStruct(op);
803                                     up(&reqSem);
804                                     return tapeRcToErrno(op->rc,
805                                                          "tape operation",
806                                                          devi.devno);
807                             } else {
808                                     freeOpStruct(op);
809                                     up(&reqSem);
810                                     return 0;
```

```
811                             }
812                             break;
813                     }
814
815             case MTIOCGET:
816                     op->sem = &Semaphore;
817                     hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
818                                                          HvLpEvent_Type_VirtualIo,
819                                                          viomajorsubtype_tape |
820                                                          viotapegetstatus,
821                                                          HvLpEvent_AckInd_DoAck,
822                                                          HvLpEvent_AckType_ImmediateAck,
823                                                          viopath_sourceinst
824                                                          (viopath_hostLp),
825                                                          viopath_targetinst
826                                                          (viopath_hostLp),
827                                                          (u64) (unsigned long)
828                                                          op, VIOVERSION << 16,
829                                                          ((u64) devi.
830                                                           devno << 48), 0, 0,
831                                                          0);
832                     if (hvrc != HvLpEvent_Rc_Good) {
833                             printk("viotape hv error on op %d\n", (int) hvrc);
834                             freeOpStruct(op);
835                             up(&reqSem);
836                             return -EIO;
837                     }
838                     down(&Semaphore);
839                     up(&reqSem);
840                     if (op->rc) {
841                             freeOpStruct(op);
842                             return tapeRcToErrno(op->rc, "get status",
843                                                  devi.devno);
844                     } else {
845                             freeOpStruct(op);
846                             err =
847                                 copy_to_user((void *) arg, &viomtget[dev],
848                                              sizeof(viomtget[0]));
849                             if (err) {
850                                     freeOpStruct(op);
851                                     return -EFAULT;
852                             }
853                             return 0;
854                     }
855                     break;
856             case MTIOCPOS:
857                     printk("Got an MTIOCPOS\n");
858             default:
859                     return -ENOSYS;
860             }
861             return 0;
862     }
863
864     /* Open
865      */
866     static int viotap_open(struct inode *inode, struct file *file)
867     {
868             DECLARE_MUTEX_LOCKED(Semaphore);
869             kdev_t dev = file->f_dentry->d_inode->i_rdev;
870             HvLpEvent_Rc hvrc;
871             struct opStruct *op = getOpStruct();
872             struct viot_devinfo_struct devi;
873             if (op == NULL)
874                     return -ENOMEM;
875
876             getDevInfo(dev, &devi);
877
878     // Note: We currently only support one mode!
879             if ((devi.devno >= viotape_numdev) || (devi.mode)) {
880                     freeOpStruct(op);
881                     return -ENODEV;
882             }
883
884             op->sem = &Semaphore;
885
886             hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
887                                                  HvLpEvent_Type_VirtualIo,
888                                                  viomajorsubtype_tape |
889                                                  viotapeopen,
890                                                  HvLpEvent_AckInd_DoAck,
891                                                  HvLpEvent_AckType_ImmediateAck,
892                                                  viopath_sourceinst
893                                                  (viopath_hostLp),
894                                                  viopath_targetinst
895                                                  (viopath_hostLp),
896                                                  (u64) (unsigned long) op,
897                                                  VIOVERSION << 16,
898                                                  ((u64) devi.devno << 48), 0,
899                                                  0, 0);
900
```

```
901
902         if (hvrc != 0) {
903                 printk("viotape bad rc on signalLpEvent %d\n", (int) hvrc);
904                 freeOpStruct(op);
905                 return -EIO;
906         }
907
908         down(&Semaphore);
909
910         if (op->rc) {
911                 freeOpStruct(op);
912                 return tapeRcToErrno(op->rc, "open", devi.devno);
913         } else {
914                 freeOpStruct(op);
915                 MOD_INC_USE_COUNT;
916                 return 0;
917         }
918 }
919
920
921 /* Release
922  */
923 static int viotap_release(struct inode *inode, struct file *file)
924 {
925         DECLARE_MUTEX_LOCKED(Semaphore);
926         kdev_t dev = file->f_dentry->d_inode->i_rdev;
927         HvLpEvent_Rc hvrc;
928         struct viot_devinfo_struct devi;
929         struct opStruct *op = getOpStruct();
930
931         if (op == NULL)
932                 return -ENOMEM;
933         op->sem = &Semaphore;
934
935         getDevInfo(dev, &devi);
936
937         if (devi.devno >= viotape_numdev) {
938                 freeOpStruct(op);
939                 return -ENODEV;
940         }
941
942         chg_state(devi.devno, VIOT_IDLE, file);
943
944         if (devi.rewind) {
945                 hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
946                                                      HvLpEvent_Type_VirtualIo,
947                                                      viomajorsubtype_tape |
948                                                      viotapeop,
949                                                      HvLpEvent_AckInd_DoAck,
950                                                      HvLpEvent_AckType_ImmediateAck,
951                                                      viopath_sourceinst
952                                                      (viopath_hostLp),
953                                                      viopath_targetinst
954                                                      (viopath_hostLp),
955                                                      (u64) (unsigned long)
956                                                      op, VIOVERSION << 16,
957                                                      ((u64) devi.
958                                                       devno << 48), 0,
959                                                      ((u64) VIOTAPOP_REW)
960                                                      << 32, 0);
961                 down(&Semaphore);
962
963                 if (op->rc) {
964                         tapeRcToErrno(op->rc, "rewind", devi.devno);
965                 }
966         }
967
968         hvrc = HvCallEvent_signalLpEventFast(viopath_hostLp,
969                                              HvLpEvent_Type_VirtualIo,
970                                              viomajorsubtype_tape |
971                                              viotapeclose,
972                                              HvLpEvent_AckInd_DoAck,
973                                              HvLpEvent_AckType_ImmediateAck,
974                                              viopath_sourceinst
975                                              (viopath_hostLp),
976                                              viopath_targetinst
977                                              (viopath_hostLp),
978                                              (u64) (unsigned long) op,
979                                              VIOVERSION << 16,
980                                              ((u64) devi.devno << 48), 0,
981                                              0, 0);
982
983
984         if (hvrc != 0) {
985                 printk("viotape: bad rc on signalLpEvent %d\n",
986                        (int) hvrc);
987                 return -EIO;
988         }
989
990         down(&Semaphore);
```

```
991
992             if (op->rc) {
993                     printk("viotape: close failed\n");
994             }
995             MOD_DEC_USE_COUNT;
996             return 0;
997     }
998
999     struct file_operations viotap_fops = {
1000            owner:THIS_MODULE,
1001            read:viotap_read,
1002            write:viotap_write,
1003            ioctl:viotap_ioctl,
1004            open:viotap_open,
1005            release:viotap_release,
1006    };
1007
1008    /* Handle interrupt events for tape
1009     */
1010    static void vioHandleTapeEvent(struct HvLpEvent *event)
1011    {
1012            int tapeminor;
1013            struct opStruct *op;
1014            struct viotapelpevent *tevent = (struct viotapelpevent *) event;
1015
1016            if (event == NULL) {
1017              /* Notification that a partition went away! */
1018              if (!viopath_isactive(viopath_hostLp)) {
1019                /* TODO! Clean up */
1020              }
1021              return;
1022            }
1023
1024            tapeminor = event->xSubtype & VIOMINOR_SUBTYPE_MASK;
1025            switch (tapeminor) {
1026            case viotapegetinfo:
1027            case viotapeopen:
1028            case viotapeclose:
1029                    op = (struct opStruct *) (unsigned long) event->
1030                        xCorrelationToken;
1031                    op->rc = tevent->mSubTypeRc;
1032                    up(op->sem);
1033                    break;
1034            case viotaperead:
1035            case viotapewrite:
1036                    op = (struct opStruct *) (unsigned long) event->
1037                        xCorrelationToken;
1038                    op->rc = tevent->mSubTypeRc;
1039                    op->count = tevent->mLen;
1040
1041                    if (op->sem) {
1042                            up(op->sem);
1043                    } else {
1044                            freeOpStruct(op);
1045                            up(&reqSem);
1046                    }
1047                    break;
1048            case viotapeop:
1049            case viotapegetpos:
1050            case viotapesetpos:
1051            case viotapegetstatus:
1052                    op = (struct opStruct *) (unsigned long) event->
1053                        xCorrelationToken;
1054                    if (op) {
1055                            op->count = tevent->u.tapeOp.mCount;
1056                            op->rc = tevent->mSubTypeRc;
1057
1058                            if (op->sem) {
1059                                    up(op->sem);
1060                            }
1061                    }
1062                    break;
1063            default:
1064                    printk("viotape: wierd ack\n");
1065            }
1066    }
1067
1068
1069    /* Do initialization
1070     */
1071    int __init viotap_init(void)
1072    {
1073            DECLARE_MUTEX_LOCKED(Semaphore);
1074            int rc;
1075            char tapename[32];
1076            int i;
1077
1078            printk("viotape driver version %d.%d\n", version_major,
1079                   version_minor);
1080
```

```
1081          sndMsgSeq = sndMsgAck = 0;
1082          rcvMsgSeq = rcvMsgAck = 0;
1083          opStructList = NULL;
1084          spin_lock_init(&opStructListLock);
1085
1086          sema_init(&reqSem, VIOTAPE_MAXREQ);
1087
1088          if (viopath_hostLp == HvLpIndexInvalid)
1089                  vio_set_hostlp();
1090
1091          /*
1092           * Open to our hosting lp
1093           */
1094          if (viopath_hostLp == HvLpIndexInvalid)
1095                  return -1;
1096
1097          printk("viotape: init - open path to hosting (%d)\n",
1098                 viopath_hostLp);
1099
1100          rc = viopath_open(viopath_hostLp, viomajorsubtype_tape, VIOTAPE_MAXREQ + 2);
1101          if (rc) {
1102                  printk("viotape: error on viopath_open to hostlp %d\n",
1103                         rc);
1104          }
1105
1106          vio_setHandler(viomajorsubtype_tape, vioHandleTapeEvent);
1107
1108          printk("viotape major is %d\n", viotape_major);
1109
1110          get_viotape_info();
1111
1112          if (devfs_register_chrdev(viotape_major, "viotape", &viotap_fops)) {
1113                  printk("Error registering viotape device\n");
1114                  return -1;
1115          }
1116
1117          for (i = 0; i < viotape_numdev; i++) {
1118                  int j;
1119                  state[i].cur_part = 0;
1120                  for (j = 0; j < MAX_PARTITIONS; ++j)
1121                          state[i].part_stat[j].rwi = VIOT_IDLE;
1122                  sprintf(tapename, "viotape%d", i);
1123                  state[i].dev_handle =
1124                      devfs_register(NULL, tapename, DEVFS_FL_DEFAULT,
1125                                     viotape_major, i,
1126                                     S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP |
1127                                     S_IWGRP, &viotap_fops, NULL);
1128                  printk
1129                      ("viotape device %s is iSeries resource %10.10s type %4.4s, model %3.3s\n",
1130                       tapename, viotape_unitinfo[i].rsrcname,
1131                       viotape_unitinfo[i].type, viotape_unitinfo[i].model);
1132          }
1133
1134          /*
1135           * Create the proc entry
1136           */
1137          iSeries_proc_callback(&viotape_proc_init);
1138
1139          return 0;
1140  }
1141
1142  /* Give a new state to the tape object
1143   */
1144  static int chg_state(int index, unsigned char new_state, struct file *file)
1145  {
1146          unsigned char *cur_state =
1147              &state[index].part_stat[state[index].cur_part].rwi;
1148          int rc = 0;
1149
1150          /* if the same state, don't bother */
1151          if (*cur_state == new_state)
1152                  return 0;
1153
1154          /* write an EOF if changing from writing to some other state */
1155          if (*cur_state == VIOT_WRITING) {
1156                  struct mtop write_eof = { MTWEOF, 1 };
1157                  rc = viotap_ioctl(NULL, file, MTIOCTOP,
1158                                    (unsigned long) &write_eof);
1159          }
1160          *cur_state = new_state;
1161          return rc;
1162  }
1163
1164  /* Cleanup
1165   */
1166  static void __exit viotap_exit(void)
1167  {
1168          int i, ret;
1169          for (i = 0; i < viotape_numdev; ++i)
1170                  devfs_unregister(state[i].dev_handle);
```

```
1171          ret = devfs_unregister_chrdev(viotape_major, "viotape");
1172          if (ret < 0)
1173                  printk("Error unregistering device: %d\n", ret);
1174          iSeries_proc_callback(&viotape_proc_delete);
1175          if (viotape_unitinfo != NULL) {
1176                  kfree(viotape_unitinfo);
1177                  viotape_unitinfo = NULL;
1178          }
1179          viopath_close(viopath_hostLp, viomajorsubtype_tape, VIOTAPE_MAXREQ + 2);
1180          vio_clearHandler(viomajorsubtype_tape);
1181  }
1182
1183  MODULE_LICENSE("GPL");
1184  module_init(viotap_init);
1185  module_exit(viotap_exit);
```

```
#ifndef _ABS_ADDR_H
#define _ABS_ADDR_H

#include <linux/config.h>

/*
 * c 2001 PPC 64 Team, IBM Corp
 *
 * 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.
 */

#include <asm/types.h>
#include <asm/page.h>
#include <asm/prom.h>
#include <asm/lmb.h>

typedef u32 msChunks_entry;
struct msChunks {
        unsigned long num_chunks;
        unsigned long chunk_size;
        unsigned long chunk_shift;
        unsigned long chunk_mask;
        msChunks_entry *abs;
};

extern struct msChunks msChunks;

extern unsigned long msChunks_alloc(unsigned long, unsigned long, unsigned long);
extern unsigned long reloc_offset(void);

#ifdef CONFIG_MSCHUNKS

static inline unsigned long
chunk_to_addr(unsigned long chunk)
{
        unsigned long offset = reloc_offset();
        struct msChunks *_msChunks = PTRRELOC(&msChunks);

        return chunk << _msChunks->chunk_shift;
}

static inline unsigned long
addr_to_chunk(unsigned long addr)
{
        unsigned long offset = reloc_offset();
        struct msChunks *_msChunks = PTRRELOC(&msChunks);

        return addr >> _msChunks->chunk_shift;
}

static inline unsigned long
chunk_offset(unsigned long addr)
{
        unsigned long offset = reloc_offset();
        struct msChunks *_msChunks = PTRRELOC(&msChunks);

        return addr & _msChunks->chunk_mask;
}

static inline unsigned long
abs_chunk(unsigned long pchunk)
{
        unsigned long offset = reloc_offset();
        struct msChunks *_msChunks = PTRRELOC(&msChunks);
        if ( pchunk >= _msChunks->num_chunks ) {
                return pchunk;
        }
        return PTRRELOC(_msChunks->abs)[pchunk];
}


static inline unsigned long
phys_to_absolute(unsigned long pa)
{
        return chunk_to_addr(abs_chunk(addr_to_chunk(pa))) + chunk_offset(pa);
}

static inline unsigned long
physRpn_to_absRpn(unsigned long rpn)
{
        unsigned long pa = rpn << PAGE_SHIFT;
        unsigned long aa = phys_to_absolute(pa);
        return (aa >> PAGE_SHIFT);
}

static inline unsigned long
absolute_to_phys(unsigned long aa)
```

```
91  {
92          return lmb_abs_to_phys(aa);
93  }
94
95  #else  /* !CONFIG_MSCHUNKS */
96
97  #define chunk_to_addr(chunk) ((unsigned long)(chunk))
98  #define addr_to_chunk(addr) (addr)
99  #define chunk_offset(addr) (0)
100 #define abs_chunk(pchunk) (pchunk)
101
102 #define phys_to_absolute(pa) (pa)
103 #define physRpn_to_absRpn(rpn) (rpn)
104 #define absolute_to_phys(aa) (aa)
105
106 #endif /* !CONFIG_MSCHUNKS */
107
108
109 static inline unsigned long
110 virt_to_absolute(unsigned long ea)
111 {
112         return phys_to_absolute(__pa(ea));
113 }
114
115 static inline unsigned long
116 absolute_to_virt(unsigned long aa)
117 {
118         return (unsigned long)__va(absolute_to_phys(aa));
119 }
120
121 #endif /* _ABS_ADDR_H */
```

```
/*
 * eeh.h
 * Copyright (C) 2001  Dave Engebretsen & Todd Inglett IBM Corporation.
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

/* Start Change Log
 * 2001/10/27 : engebret : Created.
 * End Change Log
 */

#ifndef _EEH_H
#define _EEH_H

struct pci_dev;

#define IO_UNMAPPED_REGION_ID 0xaUL

#define IO_TOKEN_TO_ADDR(token) ((((unsigned long)(token)) & 0xFFFFFFFF) | (0xEUL << 60))
/* Flag bits encoded in the 3 unused function bits of devfn */
#define EEH_TOKEN_DISABLED (1UL << 34UL)        /* eeh is disabled for this token */
#define IS_EEH_TOKEN_DISABLED(token) ((unsigned long)(token) & EEH_TOKEN_DISABLED)

#define EEH_STATE_OVERRIDE 1   /* IOA does not require eeh traps */
#define EEH_STATE_FAILURE  16  /* */

/* This is for profiling only */
extern unsigned long eeh_total_mmio_ffs;

extern int eeh_implemented;

void eeh_init(void);
static inline int is_eeh_implemented(void) { return eeh_implemented; }
int eeh_get_state(unsigned long ea);
unsigned long eeh_check_failure(void *token, unsigned long val);

#define EEH_DISABLE             0
#define EEH_ENABLE              1
#define EEH_RELEASE_LOADSTORE   2
#define EEH_RELEASE_DMA         3
int eeh_set_option(struct pci_dev *dev, int options);

/* Given a PCI device check if eeh should be configured or not.
 * This may look at firmware properties and/or kernel cmdline options.
 */
int is_eeh_configured(struct pci_dev *dev);

/* Generate an EEH token.
 * The high nibble of the offset is cleared, otherwise bounds checking is performed.
 * Use IO_TOKEN_TO_ADDR(token) to translate this token back to a mapped virtual addr.
 * Do NOT do this to perform IO -- use the read/write macros!
 */
unsigned long eeh_token(unsigned long phb,
                        unsigned long bus,
                        unsigned long devfn,
                        unsigned long offset);

extern void *memcpy(void *, const void *, unsigned long);
extern void *memset(void *,int, unsigned long);

/* EEH_POSSIBLE_ERROR() -- test for possible MMIO failure.
 *
 * Order this macro for performance.
 * If EEH is off for a device and it is a memory BAR, ioremap will
 * map it to the IOREGION.  In this case addr == vaddr and since these
 * should be in registers we compare them first.  Next we check for
 * all ones which is perhaps fastest as ~val.  Finally we weed out
 * EEH disabled IO BARs.
 *
 * If this macro yields TRUE, the caller relays to eeh_check_failure()
 * which does further tests out of line.
 */
/* #define EEH_POSSIBLE_ERROR(addr, vaddr, val) ((vaddr) != (addr) && ~(val) == 0 && !IS_EEH_TOKEN_DISABLED(addr)) */
/* This version is rearranged to collect some profiling data */
#define EEH_POSSIBLE_ERROR(addr, vaddr, val) (~(val) == 0 && (++eeh_total_mmio_ffs, (vaddr) != (addr) && !IS_EEH_TOKEN_DISABLED(addr)))
```

```
89
90  /*
91   * MMIO read/write operations with EEH support.
92   *
93   * addr: 64b token of the form 0xA0PPBBDDyyyyyyyy
94   *       0xA0     : Unmapped MMIO region
95   *       PP       : PHB index (starting at zero)
96   *       BB       : PCI Bus number under given PHB
97   *       DD       : PCI devfn under given bus
98   *       yyyyyyyy : Virtual address offset
99   *
100  * An actual virtual address is produced from this token
101  * by masking into the form:
102  *   0xE0000000yyyyyyyy
103  */
104 static inline u8 eeh_readb(void *addr) {
105         volatile u8 *vaddr = (volatile u8 *)IO_TOKEN_TO_ADDR(addr);
106         u8 val = in_8(vaddr);
107         if (EEH_POSSIBLE_ERROR(addr, vaddr, val))
108                 return eeh_check_failure(addr, val);
109         return val;
110 }
111 static inline void eeh_writeb(u8 val, void *addr) {
112         volatile u8 *vaddr = (volatile u8 *)IO_TOKEN_TO_ADDR(addr);
113         out_8(vaddr, val);
114 }
115 static inline u16 eeh_readw(void *addr) {
116         volatile u16 *vaddr = (volatile u16 *)IO_TOKEN_TO_ADDR(addr);
117         u16 val = in_le16(vaddr);
118         if (EEH_POSSIBLE_ERROR(addr, vaddr, val))
119                 return eeh_check_failure(addr, val);
120         return val;
121 }
122 static inline void eeh_writew(u16 val, void *addr) {
123         volatile u16 *vaddr = (volatile u16 *)IO_TOKEN_TO_ADDR(addr);
124         out_le16(vaddr, val);
125 }
126 static inline u32 eeh_readl(void *addr) {
127         volatile u32 *vaddr = (volatile u32 *)IO_TOKEN_TO_ADDR(addr);
128         u32 val = in_le32(vaddr);
129         if (EEH_POSSIBLE_ERROR(addr, vaddr, val))
130                 return eeh_check_failure(addr, val);
131         return val;
132 }
133 static inline void eeh_writel(u32 val, void *addr) {
134         volatile u32 *vaddr = (volatile u32 *)IO_TOKEN_TO_ADDR(addr);
135         out_le32(vaddr, val);
136 }
137
138 static inline void eeh_memset_io(void *addr, int c, unsigned long n) {
139         void *vaddr = (void *)IO_TOKEN_TO_ADDR(addr);
140         memset(vaddr, c, n);
141 }
142 static inline void eeh_memcpy_fromio(void *dest, void *src, unsigned long n) {
143         void *vsrc = (void *)IO_TOKEN_TO_ADDR(src);
144         memcpy(dest, vsrc, n);
145         /* look for ffff's here at dest[n] */
146 }
147 static inline void eeh_memcpy_toio(void *dest, void *src, unsigned long n) {
148         void *vdest = (void *)IO_TOKEN_TO_ADDR(dest);
149         memcpy(vdest, src, n);
150 }
151
152 static inline void eeh_insb(volatile u8 *addr, void *buf, int n) {
153         volatile u8 *vaddr = (volatile u8 *)IO_TOKEN_TO_ADDR(addr);
154         _insb(vaddr, buf, n);
155         /* ToDo: look for ff's in buf[n] */
156 }
157
158 static inline void eeh_outsb(volatile u8 *addr, const void *buf, int n) {
159         volatile u8 *vaddr = (volatile u8 *)IO_TOKEN_TO_ADDR(addr);
160         _outsb(vaddr, buf, n);
161 }
162
163 static inline void eeh_insw_ns(volatile u16 *addr, void *buf, int n) {
164         volatile u16 *vaddr = (volatile u16 *)IO_TOKEN_TO_ADDR(addr);
165         _insw_ns(vaddr, buf, n);
166         /* ToDo: look for ffff's in buf[n] */
167 }
168
169 static inline void eeh_outsw_ns(volatile u16 *addr, const void *buf, int n) {
170         volatile u16 *vaddr = (volatile u16 *)IO_TOKEN_TO_ADDR(addr);
171         _outsw_ns(vaddr, buf, n);
172 }
173
174 static inline void eeh_insl_ns(volatile u32 *addr, void *buf, int n) {
175         volatile u32 *vaddr = (volatile u32 *)IO_TOKEN_TO_ADDR(addr);
176         _insl_ns(vaddr, buf, n);
177         /* ToDo: look for ffffffff's in buf[n] */
178 }
```

```
179
180 static inline void eeh_outsl_ns(volatile u32 *addr, const void *buf, int n) {
181         volatile u32 *vaddr = (volatile u32 *)IO_TOKEN_TO_ADDR(addr);
182         _outsl_ns(vaddr, buf, n);
183 }
184
185
186 #endif /* _EEH_H */
```

```
/*************************************************************************
 * flight_recorder.h                                                     *
 *************************************************************************
 * This code supports the a generic flight recorder.                     *
 * Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                     *
 *                                                                       *
 * 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 program is distributed in the hope that it will be useful,       *
 * but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 * GNU General Public License for more details.                          *
 *                                                                       *
 * 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.,                                       *
 * 59 Temple Place, Suite 330,                                           *
 * Boston, MA  02111-1307  USA                                           *
 *************************************************************************
 * See the fight_recorder.c file for useage deails.                      *
 *************************************************************************/
#include <linux/kernel.h>

/*************************************************************************
 * Generic Flight Recorder Structure                                     *
 *************************************************************************/
struct flightRecorder {                 /* Structure Defination         */
        char  Signature[8];                /* Eye Catcher                  */
        int   Size;                                /* Size of Flight Recorder      */
        int   Flags;                       /* Format Flags.                */
        char* StartPointer;                /* Buffer Starting Address      */
        char* EndPointer;                  /* Buffer Ending Address        */
        char* NextPointer;                 /* Next Entry Address           */
        char* WrapPointer;                 /* Point at which buffer wraps  */
        char* Buffer;                      /* Where the data log is.       */
};
typedef struct flightRecorder FlightRecorder;

/*************************************************************************
 * Forware declares
 *************************************************************************/
FlightRecorder* alloc_Flight_Recorder(FlightRecorder* FrPtr, char* Signature, int SizeOfFr);
void            fr_Log_Entry(FlightRecorder* LogFr, const char *fmt, ...);
int             fr_Dump(FlightRecorder* Fr, char *Buffer, int BufferLen);

/*************************************************************************
 * Sample Macro to make life easier using the flight_recorder.
 * TestFr is a global value.
 * To use them: TESTFR("Test Loop value is &d",Loop");
 *************************************************************************/
#define LOGFR(...) (fr_Log_Entry(__VA_ARGS__))

```

```
#ifndef _PPC64_LMB_H
#define _PPC64_LMB_H

/*
 * Definitions for talking to the Open Firmware PROM on
 * Power Macintosh computers.
 *
 * Copyright (C) 2001 Peter Bergner, IBM Corp.
 *
 * 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.
 */

#include <asm/prom.h>

extern unsigned long reloc_offset(void);

#define MAX_LMB_REGIONS 64

union lmb_reg_property {
        struct reg_property32 addr32[MAX_LMB_REGIONS];
        struct reg_property64 addr64[MAX_LMB_REGIONS];
};

#define LMB_MEMORY_AREA 1
#define LMB_IO_AREA     2

#define LMB_ALLOC_ANYWHERE      0
#define LMB_ALLOC_FIRST4GBYTE   (1UL<<32)

struct lmb_property {
        unsigned long base;
        unsigned long physbase;
        unsigned long size;
        unsigned long type;
};

struct lmb_region {
        unsigned long cnt;
        unsigned long size;
        unsigned long iosize;
        unsigned long lcd_size;         /* Least Common Denominator */
        struct lmb_property region[MAX_LMB_REGIONS+1];
};

struct lmb {
        unsigned long debug;
        unsigned long rmo_size;
        struct lmb_region memory;
        struct lmb_region reserved;
};

extern struct lmb lmb;

extern void lmb_init(void);
extern void lmb_analyze(void);
extern long lmb_add(unsigned long, unsigned long);
#ifdef CONFIG_MSCHUNKS
extern long lmb_add_io(unsigned long base, unsigned long size);
#endif /* CONFIG_MSCHUNKS */
extern long lmb_reserve(unsigned long, unsigned long);
extern unsigned long lmb_alloc(unsigned long, unsigned long);
extern unsigned long lmb_alloc_base(unsigned long, unsigned long, unsigned long);
extern unsigned long lmb_phys_mem_size(void);
extern unsigned long lmb_end_of_DRAM(void);
extern unsigned long lmb_abs_to_phys(unsigned long);
extern void lmb_dump(char *);

static inline unsigned long
lmb_addrs_overlap(unsigned long base1, unsigned long size1,
                  unsigned long base2, unsigned long size2)
{
        return ((base1 < (base2+size2)) && (base2 < (base1+size1)));
}

static inline long
lmb_regions_overlap(struct lmb_region *rgn, unsigned long r1, unsigned long r2)
{
        unsigned long base1 = rgn->region[r1].base;
        unsigned long size1 = rgn->region[r1].size;
        unsigned long base2 = rgn->region[r2].base;
        unsigned long size2 = rgn->region[r2].size;

        return lmb_addrs_overlap(base1,size1,base2,size2);
}

static inline long
lmb_addrs_adjacent(unsigned long base1, unsigned long size1,
```

```
                    unsigned long base2, unsigned long size2)
{
        if ( base2 == base1 + size1 ) {
                return 1;
        } else if ( base1 == base2 + size2 ) {
                return -1;
        }
        return 0;
}

static inline long
lmb_regions_adjacent(struct lmb_region *rgn, unsigned long r1, unsigned long r2)
{
        unsigned long base1 = rgn->region[r1].base;
        unsigned long size1 = rgn->region[r1].size;
        unsigned long type1 = rgn->region[r1].type;
        unsigned long base2 = rgn->region[r2].base;
        unsigned long size2 = rgn->region[r2].size;
        unsigned long type2 = rgn->region[r2].type;

        return (type1 == type2) && lmb_addrs_adjacent(base1,size1,base2,size2);
}

#endif /* _PPC64_LMB_H */
```

```
#ifndef _NACA_H
#define _NACA_H

/*
 * c 2001 PPC 64 Team, IBM Corp
 *
 * 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.
 */

#include <asm/types.h>
#include <asm/systemcfg.h>

struct naca_struct {
        /*=================================================================
         * Cache line 1: 0x0000 − 0x007F
         * Kernel only data − undefined for user space
         *=================================================================
         */
        void *xItVpdAreas;              /* VPD Data                  0x00 */
        void *xRamDisk;                 /* iSeries ramdisk           0x08 */
        u64   xRamDiskSize;             /* In pages                  0x10 */
        struct paca_struct *paca;       /* Ptr to an array of pacas  0x18 */
        u64 debug_switch;               /* Debug print control       0x20 */
        u64 banner;                     /* Ptr to banner string      0x28 */
        u64 log;                        /* Ptr to log buffer         0x30 */
        u64 serialPortAddr;             /* Phy addr of serial port   0x38 */
        u64 interrupt_controller;       /* Type of int controller    0x40 */
        u64 slb_size;                   /* SLB size in entries       0x48 */
        u64 pftSize;                    /* Log 2 of page table size  0x50 */
        void *systemcfg;                /* Pointer to systemcfg data 0x58 */
        u32 dCacheL1LogLineSize;        /* L1 d−cache line size Log2 0x60 */
        u32 dCacheL1LinesPerPage;       /* L1 d−cache lines / page   0x64 */
        u32 iCacheL1LogLineSize;        /* L1 i−cache line size Log2 0x68 */
        u32 iCacheL1LinesPerPage;       /* L1 i−cache lines / page   0x6c */
        u64 resv0[2];                   /* Reserved           0x70 − 0x7F */
};

extern struct naca_struct *naca;

#endif /* _NACA_H */
```

```
#ifndef _PPC64_PACA_H
#define _PPC64_PACA_H

/*============================================================================
 *                                                          Header File Id
 * Name_____________:  paca.h
 *
 * Description_______:
 *
 * This control block defines the PACA which defines the processor
 * specific data for each logical processor on the system.
 * There are some pointers defined that are utilized by PLIC.
 *
 * C 2001 PPC 64 Team, IBM Corp
 *
 * 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.
 */
#include        <asm/types.h>

#define N_EXC_STACK    2

/*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * Other Includes
 *−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 */
#include        <asm/iSeries/ItLpPaca.h>
#include        <asm/iSeries/ItLpRegSave.h>
#include        <asm/iSeries/ItLpQueue.h>
#include        <asm/rtas.h>
#include        <asm/mmu.h>
#include        <asm/processor.h>

/* A paca entry is required for each logical processor.  On systems
 * that support hardware multi−threading, this is equal to twice the
 * number of physical processors.  On LPAR systems, we are required
 * to have space for the maximum number of logical processors we
 * could ever possibly have.  Currently, we are limited to allocating
 * 24 processors to a partition which gives 48 logical processors on
 * an HMT box.  Therefore, we reserve this many paca entries.
 */
#define MAX_PROCESSORS 24
#define MAX_PACAS MAX_PROCESSORS * 2

extern struct paca_struct paca[];
#define get_paca() ((struct paca_struct *)mfspr(SPRG3))

/*============================================================================
 * Name_______: paca
 *
 * Description:
 *
 *      Defines the layout of the paca.
 *
 *      This structure is not directly accessed by PLIC or the SP except
 *      for the first two pointers that point to the ItLpPaca area and the
 *      ItLpRegSave area for this processor.  Both the ItLpPaca and
 *      ItLpRegSave objects are currently contained within the
 *      PACA but they do not need to be.
 *
 *============================================================================
 */
struct paca_struct {
/*=============================================================================================
 * CACHE_LINE_1 0x0000 − 0x007F
 *=============================================================================================
 */
        struct ItLpPaca *xLpPacaPtr;    /* Pointer to LpPaca for PLIC           0x00 */
        struct ItLpRegSave *xLpRegSavePtr; /* Pointer to LpRegSave for PLIC     0x08 */
        u64 xCurrent;                   /* Pointer to current                   0x10 */
        u16 xPacaIndex;                 /* Logical processor number             0x18 */
        u16 xHwProcNum;                 /* Actual Hardware Processor Number     0x1a */
        u32 default_decr;               /* Default decrementer value            0x1c */
        u64 xHrdIntStack;               /* Stack for hardware interrupts        0x20 */
        u64 xKsave;                     /* Saved Kernel stack addr or zero      0x28 */
        u64 pvr;                        /* Processor version register           0x30 */
        u8 *exception_sp;               /*                                      0x38 */

        struct ItLpQueue *lpQueuePtr;   /* LpQueue handled by this processor    0x40 */
        u64  xTOC;                      /* Kernel TOC address                   0x48 */
        STAB xStab_data;                /* Segment table information            0x50,0x58,0x60 */
        u8 xSegments[STAB_CACHE_SIZE];  /* Cache of used stab entries           0x68,0x70 */
        u8 xProcEnabled;                /* 1=soft enabled                       0x78 */
        u8 xHrdIntCount;                /* Count of active hardware interrupts  0x79  */
        u8 resv1[6];                    /*                                      0x7B−0x7F */

/*=============================================================================================
 * CACHE_LINE_2 0x0080 − 0x00FF
```

```
 *=============================================================================
 */
        u64 *pgd_cache;                 /*                                      0x00 */
        u64 *pmd_cache;                 /*                                      0x08 */
        u64 *pte_cache;                 /*                                      0x10 */
        u64 pgtable_cache_sz;           /*                                      0x18 */
        u64 next_jiffy_update_tb;       /* TB value for next jiffy update       0x20 */
        u32 lpEvent_count;              /* lpEvents processed                   0x28 */
        u8  rsvd2[128-5*8-1*4];         /*                                      0x68 */

/*=============================================================================
 * CACHE_LINE_3 0x0100 - 0x017F
 *=============================================================================
 */
        u8              xProcStart;     /* At startup, processor spins until    0x100 */
                                        /* xProcStart becomes non-zero. */
        u8              rsvd3[127];

/*=============================================================================
 * CACHE_LINE_4-8  0x0180 - 0x03FF Contains ItLpPaca
 *=============================================================================
 */
        struct ItLpPaca xLpPaca;        /* Space for ItLpPaca */

/*=============================================================================
 * CACHE_LINE_9-16 0x0400 - 0x07FF Contains ItLpRegSave
 *=============================================================================
 */
        struct ItLpRegSave xRegSav;     /* Register save for proc */

/*=============================================================================
 * CACHE_LINE_17-18 0x0800 - 0x0EFF Reserved
 *=============================================================================
 */
        struct rtas_args xRtas;         /* Per processor RTAS struct */
        u64 xR1;                        /* r1 save for RTAS calls */
        u64 xSavedMsr;                  /* Old msr saved here by HvCall */
        u8 rsvd5[256-16-sizeof(struct rtas_args)];

/*=============================================================================
 * CACHE_LINE_19 - 20 Profile Data
 *=============================================================================
 */
        u32 pmc[12];                    /* Default pmc value            */
        u64 pmcc[8];                    /* Cumulative pmc counts        */
        u64 rsvd5a[2];

        u32 prof_multiplier;            /*                                      */
        u32 prof_shift;                 /* iSeries shift for profile bucket size */
        u32 *prof_buffer;               /* iSeries profiling buffer             */
        u32 *prof_stext;                /* iSeries start of kernel text         */
        u32 *prof_etext;                /* iSeries start of kernel text         */
        u32 prof_len;                   /* iSeries length of profile buffer -1  */
        u8  prof_mode;                  /* */
        u8  rsvv5b[3];
        u64 prof_counter;               /*                                      */
        u8  rsvd5c[128-8*6];

/*=============================================================================
 * CACHE_LINE_20-30
 *=============================================================================
 */
        u8 rsvd6[0x500];

/*=============================================================================
 * CACHE_LINE_31 0x0F00 - 0x0F7F Exception stack
 *=============================================================================
 */
        u8 exception_stack[N_EXC_STACK*EXC_FRAME_SIZE];

/*=============================================================================
 * CACHE_LINE_32 0x0F80 - 0x0FFF Reserved
 *=============================================================================
 */
        u8 rsvd7[0x80];                  /* Give the stack some rope ... */

/*=============================================================================
 * Page 2 Reserved for guard page.  Also used as a stack early in SMP boots before
 *        relocation is enabled.
 *=============================================================================
 */
        u8 guard[0x1000];               /* ... and then hang 'em         */
};

#endif /* _PPC64_PACA_H */
```

```
/*
 * This file contains the code to configure and utilize the ppc64 pmc hardware
 * Copyright (C) 2002 David Engebretsen <engebret@us.ibm.com>
 */

#ifndef __KERNEL__
#define INLINE_SYSCALL(arg1, arg2)       \
  ({                                     \
    register long r0 __asm__ ("r0");     \
    register long r3 __asm__ ("r3"); \
    register long r4 __asm__ ("r4"); \
    long ret, err;                       \
    r0 = 208; \
    r3 = (long) (arg1); \
    r4 = (long) (arg2); \
    __asm__ ("sc\n\t"                    \
             "mfcr    %1\n\t"            \
             : "=r" (r3), "=r" (err)      \
             : "r" (r0), "r" (r3), "r" (r4) \
             : "cc", "memory");          \
    ret = r3;                            \
  })
#endif

#ifndef __ASSEMBLY__
struct perfmon_base_struct {
        u64 profile_buffer;
        u64 profile_length;
        u64 trace_buffer;
        u64 trace_length;
        u64 trace_end;
        u64 state;
};

struct pmc_header {
        int type;
        int pid;
        int resv[30];
};

struct pmc_struct {
        int pmc[11];
};

struct pmc_info_struct {
        unsigned int mode, cpu;

        unsigned int  pmc_base[11];
        unsigned long pmc_cumulative[8];
};

struct perfmon_struct {
        struct pmc_header header;

        union {
                struct pmc_struct      pmc;
                struct pmc_info_struct pmc_info;
        } vdata;
};

enum {
        PMC_OP_ALLOC         = 1,
        PMC_OP_FREE          = 2,
        PMC_OP_CLEAR         = 4,
        PMC_OP_DUMP          = 5,
        PMC_OP_DUMP_HARDWARE = 6,
        PMC_OP_DECR_PROFILE  = 20,
        PMC_OP_PMC_PROFILE   = 21,
        PMC_OP_SET           = 30,
        PMC_OP_SET_USER      = 31,
        PMC_OP_END           = 30
};


#define PMC_TRACE_CMD 0xFF

enum {
        PMC_TYPE_DERC_PROFILE  = 1,
        PMC_TYPE_CYCLE         = 2,
        PMC_TYPE_PROFILE       = 3,
        PMC_TYPE_DCACHE        = 4,
        PMC_TYPE_L2_MISS       = 5,
        PMC_TYPE_LWARCX        = 6,
        PMC_TYPE_END           = 6
};
#endif

#define PMC_STATE_INITIAL         0x00
#define PMC_STATE_READY           0x01
#define PMC_STATE_DECR_PROFILE    0x10
```

```
91 #define PMC_STATE_PROFILE_KERN    0x11
92 #define PMC_STATE_TRACE_KERN      0x20
93 #define PMC_STATE_TRACE_USER      0x21
94
```

```
/*
 * pmc.h
 * Copyright (C) 2001  Dave Engebretsen & Mike Corrigan IBM Corporation.
 *
 * The PPC64 PMC subsystem encompases both the hardware PMC registers and
 * a set of software event counters.  An interface is provided via the
 * proc filesystem which can be used to access this subsystem.
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

/* Start Change Log
 * 2001/06/05 : engebret : Created.
 * End Change Log
 */

#ifndef _PPC64_TYPES_H
#include        <asm/types.h>
#endif

#ifndef _PMC_H
#define _PMC_H

#define STAB_ENTRY_MAX 64

struct _pmc_hw
{
        u64 mmcr0;
        u64 mmcr1;
        u64 mmcra;

        u64 pmc1;
        u64 pmc2;
        u64 pmc3;
        u64 pmc4;
        u64 pmc5;
        u64 pmc6;
        u64 pmc7;
        u64 pmc8;
};

struct _pmc_sw
{
        u64 stab_faults;           /* Count of faults on the stab      */
        u64 stab_capacity_castouts;/* Count of castouts from the stab  */
        u64 stab_invalidations;    /* Count of invalidations from the  */
                                   /*   stab, not including castouts   */
        u64 stab_entry_use[STAB_ENTRY_MAX];

        u64 htab_primary_overflows;
        u64 htab_capacity_castouts;
        u64 htab_read_to_write_fault;
};

#define PMC_HW_TEXT_ENTRY_COUNT (sizeof(struct _pmc_hw) / sizeof(u64))
#define PMC_SW_TEXT_ENTRY_COUNT (sizeof(struct _pmc_sw) / sizeof(u64))
#define PMC_TEXT_ENTRY_SIZE  64

struct _pmc_sw_text {
        char buffer[PMC_SW_TEXT_ENTRY_COUNT * PMC_TEXT_ENTRY_SIZE];
};

struct _pmc_hw_text {
        char buffer[PMC_HW_TEXT_ENTRY_COUNT * PMC_TEXT_ENTRY_SIZE];
};

extern struct _pmc_sw pmc_sw_system;
extern struct _pmc_sw pmc_sw_cpu[];

extern struct _pmc_sw_text pmc_sw_text;
extern struct _pmc_hw_text pmc_hw_text;
extern char *ppc64_pmc_stab(int file);
extern char *ppc64_pmc_htab(int file);
extern char *ppc64_pmc_hw(int file);

void *btmalloc(unsigned long size);
void btfree(void *addr);

```

```
91  #if 1
92  #define PMC_SW_PROCESSOR(F)      pmc_sw_cpu[smp_processor_id()].F++
93  #define PMC_SW_PROCESSOR_A(F, E) (pmc_sw_cpu[smp_processor_id()].F[(E)])++
94  #define PMC_SW_SYSTEM(F)         pmc_sw_system.F++
95  #else
96  #define PMC_SW_PROCESSOR(F)   do {;} while (0)
97  #define PMC_SW_PROCESSOR_A(F) do {;} while (0)
98  #define PMC_SW_SYSTEM(F)      do {;} while (0)
99  #endif
100
101 #define PMC_CONTROL_CPI 1
102 #define PMC_CONTROL_TLB 2
103
104 /* To find an entry in the bolted page-table-directory */
105 #define pgd_offset_b(address) (bolted_pgd + pgd_index(address))
106 #define BTMALLOC_START 0xB000000000000000
107 #define BTMALLOC_END   0xB0000000ffffffff /* 4 GB Max-more or less arbitrary */
108
109 #endif /* _PMC_H */
```

```
#ifndef __PPCDEBUG_H
#define __PPCDEBUG_H
/*********************************************************************
 * Author: Adam Litke, IBM Corp
 * (c) 2001
 *
 * This file contains definitions and macros for a runtime debugging
 * system for ppc64 (This should also work on 32 bit with a few
 * adjustments.
 *
 * 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.
 *
 *********************************************************************/

#include <linux/config.h>
#include <asm/udbg.h>
#include <stdarg.h>

#define PPCDBG_BITVAL(X)     ((1UL)<<((unsigned long)(X)))

/* Defined below are the bit positions of various debug flags in the
 * debug_switch variable (defined in naca.h).
 * -- When adding new values, please enter them into trace names below --
 *
 * Values 62 & 63 can be used to stress the hardware page table management
 * code.  They must be set statically, any attempt to change them dynamically
 * would be a very bad idea.
 */
#define PPCDBG_MMINIT        PPCDBG_BITVAL(0)
#define PPCDBG_MM            PPCDBG_BITVAL(1)
#define PPCDBG_SYS32         PPCDBG_BITVAL(2)
#define PPCDBG_SYS32NI       PPCDBG_BITVAL(3)
#define PPCDBG_SYS32X        PPCDBG_BITVAL(4)
#define PPCDBG_SYS32M        PPCDBG_BITVAL(5)
#define PPCDBG_SYS64         PPCDBG_BITVAL(6)
#define PPCDBG_SYS64NI       PPCDBG_BITVAL(7)
#define PPCDBG_SYS64X        PPCDBG_BITVAL(8)
#define PPCDBG_SIGNAL        PPCDBG_BITVAL(9)
#define PPCDBG_SIGNALXMON    PPCDBG_BITVAL(10)
#define PPCDBG_BINFMT32      PPCDBG_BITVAL(11)
#define PPCDBG_BINFMT64      PPCDBG_BITVAL(12)
#define PPCDBG_BINFMTXMON    PPCDBG_BITVAL(13)
#define PPCDBG_BINFMT_32ADDR PPCDBG_BITVAL(14)
#define PPCDBG_ALIGNFIXUP    PPCDBG_BITVAL(15)
#define PPCDBG_TCEINIT       PPCDBG_BITVAL(16)
#define PPCDBG_TCE           PPCDBG_BITVAL(17)
#define PPCDBG_PHBINIT       PPCDBG_BITVAL(18)
#define PPCDBG_SMP           PPCDBG_BITVAL(19)
#define PPCDBG_BOOT          PPCDBG_BITVAL(20)
#define PPCDBG_BUSWALK       PPCDBG_BITVAL(21)
#define PPCDBG_PROM          PPCDBG_BITVAL(22)
#define PPCDBG_RTAS          PPCDBG_BITVAL(23)
#define PPCDBG_HTABSTRESS    PPCDBG_BITVAL(62)
#define PPCDBG_HTABSIZE      PPCDBG_BITVAL(63)
#define PPCDBG_NONE          (0UL)
#define PPCDBG_ALL           (0xffffffffUL)

/* The default initial value for the debug switch */
#define PPC_DEBUG_DEFAULT    0
/* #define PPC_DEBUG_DEFAULT    PPCDBG_ALL        */

#define PPCDBG_NUM_FLAGS     64

#ifdef WANT_PPCDBG_TAB
/* A table of debug switch names to allow name lookup in xmon
 * (and whoever else wants it.
 */
char *trace_names[PPCDBG_NUM_FLAGS] = {
        /* Known debug names */
        "mminit",         "mm",
        "syscall32",      "syscall32_ni",  "syscall32x",          "syscall32m",
        "syscall64",      "syscall64_ni",  "syscall64x",
        "signal",         "signal_xmon",
        "binfmt32",       "binfmt64",          "binfmt_xmon",     "binfmt_32addr",
        "alignfixup",   "tceinit",       "tce",           "phb_init",
        "smp",            "boot",          "buswalk",         "prom",
        "rtas"
};
#else
extern char *trace_names[64];
#endif /* WANT_PPCDBG_TAB */

#ifdef CONFIG_PPCDBG
/* Macro to conditionally print debug based on debug_switch */
#define PPCDBG(...) udbg_ppcdbg(__VA_ARGS__)

/* Macro to conditionally call a debug routine based on debug_switch */
```

```
91  #define PPCDBGCALL(FLAGS,FUNCTION) ifppcdebug(FLAGS) FUNCTION
92
93  /* Macros to test for debug states */
94  #define ifppcdebug(FLAGS) if (udbg_ifdebug(FLAGS))
95  #define ppcdebugset(FLAGS) (udbg_ifdebug(FLAGS))
96  #define PPCDBG_BINFMT ((current->thread.flags & PPC_FLAG_32BIT) ? PPCDBG_BINFMT32 : PPCDBG_BINFMT64)
97
98  #ifdef CONFIG_XMON
99  #define PPCDBG_ENTER_DEBUGGER() xmon(0)
100 #define PPCDBG_ENTER_DEBUGGER_REGS(X) xmon(X)
101 #endif
102 #ifdef CONFIG_KDB
103 #include <linux/kdb.h>
104 #define PPCDBG_ENTER_DEBUGGER() kdb(KDB_REASON_CALL, 0, 0)
105 #endif
106
107 #else
108 #define PPCDBG(...) do {;} while (0)
109 #define PPCDBGCALL(FLAGS,FUNCTION) do {;} while (0)
110 #define ifppcdebug(...) if (0)
111 #define ppcdebugset(FLAGS) (0)
112 #endif /* CONFIG_PPCDBG */
113
114 #ifndef PPCDBG_ENTER_DEBUGGER
115 #define PPCDBG_ENTER_DEBUGGER() do {;} while(0)
116 #endif
117
118 #ifndef PPCDBG_ENTER_DEBUGGER_REGS
119 #define PPCDBG_ENTER_DEBUGGER_REGS(A) do {;} while(0)
120 #endif
121
122 #endif /*__PPCDEBUG_H */
```

```
#ifndef _PPC64_RTAS_H
#define _PPC64_RTAS_H

#include <linux/spinlock.h>
#include <asm/page.h>

/*
 * Definitions for talking to the RTAS on CHRP machines.
 *
 * Copyright (C) 2001 Peter Bergner
 * Copyright (C) 2001 PPC 64 Team, IBM Corp
 *
 * 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.
 */

#define RTAS_UNKNOWN_SERVICE (-1)
#define RTAS_INSTANTIATE_MAX (1UL<<30) /* Don't instantiate rtas at/above this value */

/*
 * In general to call RTAS use rtas_token("string") to lookup
 * an RTAS token for the given string (e.g. "event-scan").
 * To actually perform the call use
 *    ret = rtas_call(token, n_in, n_out, ...)
 * Where n_in is the number of input parameters and
 *       n_out is the number of output parameters
 *
 * If the "string" is invalid on this system, RTAS_UNKOWN_SERVICE
 * will be returned as a token.  rtas_call() does look for this
 * token and error out gracefully so rtas_call(rtas_token("str"), ...)
 * may be safely used for one-shot calls to RTAS.
 *
 */

typedef u32 rtas_arg_t;

struct rtas_args {
        u32 token;
        u32 nargs;
        u32 nret;
        rtas_arg_t args[16];
#if 0
        spinlock_t lock;
#endif
        rtas_arg_t *rets;     /* Pointer to return values in args[]. */
};

struct rtas_t {
        unsigned long entry;            /* physical address pointer */
        unsigned long base;             /* physical address pointer */
        unsigned long size;
        spinlock_t lock;

        struct device_node *dev;        /* virtual address pointer */
};

/* Event classes */
#define INTERNAL_ERROR          0x80000000 /* set bit 0 */
#define EPOW_WARNING            0x40000000 /* set bit 1 */
#define POWERMGM_EVENTS         0x20000000 /* set bit 2 */
#define HOTPLUG_EVENTS          0x10000000 /* set bit 3 */
#define EVENT_SCAN_ALL_EVENTS   0xf0000000

/* event-scan returns */
#define SEVERITY_FATAL          0x5
#define SEVERITY_ERROR          0x4
#define SEVERITY_ERROR_SYNC     0x3
#define SEVERITY_WARNING        0x2
#define SEVERITY_EVENT          0x1
#define SEVERITY_NO_ERROR       0x0
#define DISP_FULLY_RECOVERED    0x0
#define DISP_LIMITED_RECOVERY   0x1
#define DISP_NOT_RECOVERED      0x2
#define PART_PRESENT            0x0
#define PART_NOT_PRESENT        0x1
#define INITIATOR_UNKNOWN       0x0
#define INITIATOR_CPU           0x1
#define INITIATOR_PCI           0x2
#define INITIATOR_ISA           0x3
#define INITIATOR_MEMORY        0x4
#define INITIATOR_POWERMGM      0x5
#define TARGET_UNKNOWN          0x0
#define TARGET_CPU              0x1
#define TARGET_PCI              0x2
#define TARGET_ISA              0x3
#define TARGET_MEMORY           0x4
#define TARGET_POWERMGM         0x5
#define TYPE_RETRY              0x01
```

```
91  #define TYPE_TCE_ERR            0x02
92  #define TYPE_INTERN_DEV_FAIL    0x03
93  #define TYPE_TIMEOUT            0x04
94  #define TYPE_DATA_PARITY        0x05
95  #define TYPE_ADDR_PARITY        0x06
96  #define TYPE_CACHE_PARITY       0x07
97  #define TYPE_ADDR_INVALID       0x08
98  #define TYPE_ECC_UNCORR         0x09
99  #define TYPE_ECC_CORR           0x0a
100 #define TYPE_EPOW               0x40
101 /* I don't add PowerMGM events right now, this is a different topic */
102 #define TYPE_PMGM_POWER_SW_ON   0x60
103 #define TYPE_PMGM_POWER_SW_OFF  0x61
104 #define TYPE_PMGM_LID_OPEN      0x62
105 #define TYPE_PMGM_LID_CLOSE     0x63
106 #define TYPE_PMGM_SLEEP_BTN     0x64
107 #define TYPE_PMGM_WAKE_BTN      0x65
108 #define TYPE_PMGM_BATTERY_WARN  0x66
109 #define TYPE_PMGM_BATTERY_CRIT  0x67
110 #define TYPE_PMGM_SWITCH_TO_BAT 0x68
111 #define TYPE_PMGM_SWITCH_TO_AC  0x69
112 #define TYPE_PMGM_KBD_OR_MOUSE  0x6a
113 #define TYPE_PMGM_ENCLOS_OPEN   0x6b
114 #define TYPE_PMGM_ENCLOS_CLOSED 0x6c
115 #define TYPE_PMGM_RING_INDICATE 0x6d
116 #define TYPE_PMGM_LAN_ATTENTION 0x6e
117 #define TYPE_PMGM_TIME_ALARM    0x6f
118 #define TYPE_PMGM_CONFIG_CHANGE 0x70
119 #define TYPE_PMGM_SERVICE_PROC  0x71
120
121 struct rtas_error_log {
122         unsigned long version:8;                /* Architectural version */
123         unsigned long severity:3;               /* Severity level of error */
124         unsigned long disposition:2;            /* Degree of recovery */
125         unsigned long extended:1;               /* extended log present? */
126         unsigned long /* reserved */ :2;        /* Reserved for future use */
127         unsigned long initiator:4;              /* Initiator of event */
128         unsigned long target:4;                 /* Target of failed operation */
129         unsigned long type:8;                   /* General event or error*/
130         unsigned long extended_log_length:32;   /* length in bytes */
131         unsigned char buffer[1];                /* allocated by klimit bump */
132 };
133
134 struct flash_block {
135         char *data;
136         unsigned long length;
137 };
138
139 /* This struct is very similar but not identical to
140  * that needed by the rtas flash update.
141  * All we need to do for rtas is rewrite num_blocks
142  * into a version/length and translate the pointers
143  * to absolute.
144  */
145 #define FLASH_BLOCKS_PER_NODE ((PAGE_SIZE - 16) / sizeof(struct flash_block))
146 struct flash_block_list {
147         unsigned long num_blocks;
148         struct flash_block_list *next;
149         struct flash_block blocks[FLASH_BLOCKS_PER_NODE];
150 };
151 struct flash_block_list_header { /* just the header of flash_block_list */
152         unsigned long num_blocks;
153         struct flash_block_list *next;
154 };
155 extern struct flash_block_list_header rtas_firmware_flash_list;
156
157 extern struct rtas_t rtas;
158
159 extern void enter_rtas(struct rtas_args *);
160 extern int rtas_token(const char *service);
161 extern long rtas_call(int token, int, int, unsigned long *, ...);
162 extern void phys_call_rtas(int, int, int, ...);
163 extern void phys_call_rtas_display_status(char);
164 extern void call_rtas_display_status(char);
165 extern void rtas_restart(char *cmd);
166 extern void rtas_power_off(void);
167 extern void rtas_halt(void);
168
169 extern struct proc_dir_entry *rtas_proc_dir;
170
171 /* Some RTAS ops require a data buffer and that buffer must be < 4G.
172  * Rather than having a memory allocator, just use this buffer
173  * (get the lock first), make the RTAS call.  Copy the data instead
174  * of holding the buffer for long.
175  */
176 #define RTAS_DATA_BUF_SIZE 1024
177 extern spinlock_t rtas_data_buf_lock;
178 extern char rtas_data_buf[RTAS_DATA_BUF_SIZE];
179
180 #endif /* _PPC64_RTAS_H */
```

```
#ifndef __UDBG_HDR
#define __UDBG_HDR

/*
 * c 2001 PPC 64 Team, IBM Corp
 *
 * 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.
 */

void udbg_init_uart(void *comport);
void udbg_putc(unsigned char c);
unsigned char udbg_getc(void);
int udbg_getc_poll(void);
void udbg_puts(const char *s);
int udbg_write(const char *s, int n);
int udbg_read(char *buf, int buflen);
struct console;
void udbg_console_write(struct console *con, const char *s, unsigned int n);
void udbg_puthex(unsigned long val);
void udbg_printSP(const char *s);
void udbg_printf(const char *fmt, ...);
void udbg_ppcdbg(unsigned long flags, const char *fmt, ...);
unsigned long udbg_ifdebug(unsigned long flags);

void udbg_init_uart(void *comport);
#endif
```

```
/*
 * HvCallCfg.h
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

//============================================================================
//
//      This file contains the "hypervisor call" interface which is used to
//      drive the hypervisor from the OS.
//
//============================================================================

//----------------------------------------------------------------------
// Standard Includes
//----------------------------------------------------------------------
#ifndef  _HVCALLSC_H
#include "HvCallSc.h"
#endif

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif

//----------------------------------------------------------------------------------------
// Constants
//----------------------------------------------------------------------------------------
#ifndef _HVCALLCFG_H
#define _HVCALLCFG_H

enum HvCallCfg_ReqQual
{
        HvCallCfg_Cur   = 0,
        HvCallCfg_Init  = 1,
        HvCallCfg_Max   = 2,
        HvCallCfg_Min   = 3
};

#define HvCallCfgGetLps                                 HvCallCfg +  0
#define HvCallCfgGetActiveLpMap                         HvCallCfg +  1
#define HvCallCfgGetLpVrmIndex                          HvCallCfg +  2
#define HvCallCfgGetLpMinSupportedPlicVrmIndex          HvCallCfg +  3
#define HvCallCfgGetLpMinCompatablePlicVrmIndex         HvCallCfg +  4
#define HvCallCfgGetLpVrmName                           HvCallCfg +  5
#define HvCallCfgGetSystemPhysicalProcessors            HvCallCfg +  6
#define HvCallCfgGetPhysicalProcessors                  HvCallCfg +  7
#define HvCallCfgGetSystemMsChunks                      HvCallCfg +  8
#define HvCallCfgGetMsChunks                            HvCallCfg +  9
#define HvCallCfgGetInteractivePercentage               HvCallCfg + 10
#define HvCallCfgIsBusDedicated                         HvCallCfg + 11
#define HvCallCfgGetBusOwner                            HvCallCfg + 12
#define HvCallCfgGetBusAllocation                       HvCallCfg + 13
#define HvCallCfgGetBusUnitOwner                        HvCallCfg + 14
#define HvCallCfgGetBusUnitAllocation                   HvCallCfg + 15
#define HvCallCfgGetVirtualBusPool                      HvCallCfg + 16
#define HvCallCfgGetBusUnitInterruptProc                HvCallCfg + 17
#define HvCallCfgGetConfiguredBusUnitsForIntProc        HvCallCfg + 18
#define HvCallCfgGetRioSanBusPool                       HvCallCfg + 19
#define HvCallCfgGetSharedPoolIndex                     HvCallCfg + 20
#define HvCallCfgGetSharedProcUnits                     HvCallCfg + 21
#define HvCallCfgGetNumProcsInSharedPool                HvCallCfg + 22
#define HvCallCfgRouter23                               HvCallCfg + 23
#define HvCallCfgRouter24                               HvCallCfg + 24
#define HvCallCfgRouter25                               HvCallCfg + 25
#define HvCallCfgRouter26                               HvCallCfg + 26
#define HvCallCfgRouter27                               HvCallCfg + 27
#define HvCallCfgGetMinRuntimeMsChunks                  HvCallCfg + 28
#define HvCallCfgSetMinRuntimeMsChunks                  HvCallCfg + 29
#define HvCallCfgGetVirtualLanIndexMap                  HvCallCfg + 30
#define HvCallCfgGetLpExecutionMode                     HvCallCfg + 31
#define HvCallCfgGetHostingLpIndex                      HvCallCfg + 32

//=====================================================================
static inline HvLpIndex HvCallCfg_getLps(void)
{
        HvLpIndex retVal = HvCall0(HvCallCfgGetLps);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
```

```
91          return retVal;
92  }
93  //=====================================================================
94  static inline int               HvCallCfg_isBusDedicated(u64 busIndex)
95  {
96          int retVal = HvCall1(HvCallCfgIsBusDedicated,busIndex);
97          // getPaca()->adjustHmtForNoOfSpinLocksHeld();
98          return retVal;
99  }
100 //=====================================================================
101 static inline HvLpIndex HvCallCfg_getBusOwner(u64 busIndex)
102 {
103         HvLpIndex retVal = HvCall1(HvCallCfgGetBusOwner,busIndex);
104         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
105         return retVal;
106 }
107 //=====================================================================
108 static inline HvLpIndexMap      HvCallCfg_getBusAllocation(u64 busIndex)
109 {
110         HvLpIndexMap retVal = HvCall1(HvCallCfgGetBusAllocation,busIndex);
111         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
112         return retVal;
113 }
114 //=====================================================================
115 static inline HvLpIndexMap      HvCallCfg_getActiveLpMap(void)
116 {
117         HvLpIndexMap retVal = HvCall0(HvCallCfgGetActiveLpMap);
118         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
119         return retVal;
120 }
121 //=====================================================================
122 static inline HvLpVirtualLanIndexMap    HvCallCfg_getVirtualLanIndexMap(HvLpIndex lp)
123 {
124         // This is a new function in V5R1 so calls to this on older
125         // hypervisors will return -1
126         u64 retVal = HvCall1(HvCallCfgGetVirtualLanIndexMap, lp);
127         if(retVal == -1)
128                 retVal = 0;
129         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
130         return retVal;
131 }
132 //=====================================================================
133 static inline u64               HvCallCfg_getSystemMsChunks(void)
134 {
135         u64 retVal = HvCall0(HvCallCfgGetSystemMsChunks);
136         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
137         return retVal;
138 }
139 //=====================================================================
140 static inline u64               HvCallCfg_getMsChunks(HvLpIndex lp,enum HvCallCfg_ReqQual qual)
141 {
142         u64 retVal = HvCall2(HvCallCfgGetMsChunks,lp,qual);
143         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
144         return retVal;
145 }
146 //=====================================================================
147 static inline u64               HvCallCfg_getMinRuntimeMsChunks(HvLpIndex lp)
148 {
149         // NOTE: This function was added in v5r1 so older hypervisors will return a -1 value
150         u64 retVal = HvCall1(HvCallCfgGetMinRuntimeMsChunks,lp);
151         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
152         return retVal;
153 }
154 //=====================================================================
155 static inline u64               HvCallCfg_setMinRuntimeMsChunks(u64 chunks)
156 {
157         u64 retVal = HvCall1(HvCallCfgSetMinRuntimeMsChunks,chunks);
158         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
159         return retVal;
160 }
161 //=====================================================================
162 static inline u64               HvCallCfg_getSystemPhysicalProcessors(void)
163 {
164         u64 retVal = HvCall0(HvCallCfgGetSystemPhysicalProcessors);
165         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
166         return retVal;
167 }
168 //=====================================================================
169 static inline u64               HvCallCfg_getPhysicalProcessors(HvLpIndex lp,enum HvCallCfg_ReqQual qual)
170 {
171         u64 retVal = HvCall2(HvCallCfgGetPhysicalProcessors,lp,qual);
172         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
173         return retVal;
174 }
175 //=====================================================================
176 static inline u64               HvCallCfg_getConfiguredBusUnitsForInterruptProc(HvLpIndex lp,
177                                                                                 u16 hvLogicalProcIndex)
178 {
179         u64 retVal = HvCall2(HvCallCfgGetConfiguredBusUnitsForIntProc,lp,hvLogicalProcIndex);
180         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
```

```
181         return retVal;
182
183 }
184 //=====================================================================
185 static inline HvLpSharedPoolIndex       HvCallCfg_getSharedPoolIndex(HvLpIndex lp)
186 {
187         HvLpSharedPoolIndex retVal =
188                 HvCall1(HvCallCfgGetSharedPoolIndex,lp);
189         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
190         return retVal;
191
192 }
193 //=====================================================================
194 static inline u64       HvCallCfg_getSharedProcUnits(HvLpIndex lp,enum HvCallCfg_ReqQual qual)
195 {
196         u64 retVal = HvCall2(HvCallCfgGetSharedProcUnits,lp,qual);
197         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
198         return retVal;
199
200 }
201 //=====================================================================
202 static inline u64       HvCallCfg_getNumProcsInSharedPool(HvLpSharedPoolIndex sPI)
203 {
204         u16 retVal = HvCall1(HvCallCfgGetNumProcsInSharedPool,sPI);
205         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
206         return retVal;
207
208 }
209 //=====================================================================
210 static inline HvLpIndex HvCallCfg_getHostingLpIndex(HvLpIndex lp)
211 {
212         u64 retVal = HvCall1(HvCallCfgGetHostingLpIndex,lp);
213         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
214         return retVal;
215
216 }
217
218 #endif // _HVCALLCFG_H
219
```

```
/*
 * HvCallEvent.h
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

//============================================================
//
//      This file contains the "hypervisor call" interface which is used to
//      drive the hypervisor from the OS.
//
//============================================================

//----------------------------------------------------------------------
// Standard Includes
//----------------------------------------------------------------------
#ifndef  _HVCALLSC_H
#include <asm/iSeries/HvCallSc.h>
#endif

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif

#include <asm/abs_addr.h>

//----------------------------------------------------------------------
// Other Includes
//----------------------------------------------------------------------

//----------------------------------------------------------------------
// Constants
//----------------------------------------------------------------------
#ifndef _HVCALLEVENT_H
#define _HVCALLEVENT_H

struct HvLpEvent;

typedef u8 HvLpEvent_Type;
typedef u8 HvLpEvent_AckInd;
typedef u8 HvLpEvent_AckType;

struct  HvCallEvent_PackedParms
{
        u8              xAckType:1;
        u8              xAckInd:1;
        u8              xRsvd:1;
        u8              xTargetLp:5;
        u8              xType;
        u16             xSubtype;
        HvLpInstanceId  xSourceInstId;
        HvLpInstanceId  xTargetInstId;
};

typedef u8 HvLpDma_Direction;
typedef u8 HvLpDma_AddressType;

struct  HvCallEvent_PackedDmaParms
{
        u8              xDirection:1;
        u8              xLocalAddrType:1;
        u8              xRemoteAddrType:1;
        u8              xRsvd1:5;
        HvLpIndex       xRemoteLp;
        u8              xType;
        u8              xRsvd2;
        HvLpInstanceId  xLocalInstId;
        HvLpInstanceId  xRemoteInstId;
};

typedef u64 HvLpEvent_Rc;
typedef u64 HvLpDma_Rc;

#define HvCallEventAckLpEvent                           HvCallEvent +  0
#define HvCallEventCancelLpEvent                        HvCallEvent +  1
#define HvCallEventCloseLpEventPath                     HvCallEvent +  2
#define HvCallEventDmaBufList                           HvCallEvent +  3
```

```
91  #define HvCallEventDmaSingle                            HvCallEvent +  4
92  #define HvCallEventDmaToSp                              HvCallEvent +  5
93  #define HvCallEventGetOverflowLpEvents                  HvCallEvent +  6
94  #define HvCallEventGetSourceLpInstanceId                HvCallEvent +  7
95  #define HvCallEventGetTargetLpInstanceId                HvCallEvent +  8
96  #define HvCallEventOpenLpEventPath                      HvCallEvent +  9
97  #define HvCallEventSetLpEventStack                      HvCallEvent + 10
98  #define HvCallEventSignalLpEvent                        HvCallEvent + 11
99  #define HvCallEventSignalLpEventParms                   HvCallEvent + 12
100 #define HvCallEventSetInterLpQueueIndex                 HvCallEvent + 13
101 #define HvCallEventSetLpEventQueueInterruptProc         HvCallEvent + 14
102 #define HvCallEventRouter15                             HvCallEvent + 15
103
104 //=====================================================================
105 static inline void              HvCallEvent_getOverflowLpEvents(u8 queueIndex)
106 {
107         HvCall1(HvCallEventGetOverflowLpEvents,queueIndex);
108         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
109 }
110 //=====================================================================
111 static inline void              HvCallEvent_setInterLpQueueIndex(u8 queueIndex)
112 {
113         HvCall1(HvCallEventSetInterLpQueueIndex,queueIndex);
114         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
115 }
116 //=====================================================================
117 static inline void              HvCallEvent_setLpEventStack(u8 queueIndex,
118                                             char * eventStackAddr,
119                                             u32 eventStackSize)
120 {
121         u64 abs_addr;
122         abs_addr = virt_to_absolute( (unsigned long) eventStackAddr );
123
124         HvCall3(HvCallEventSetLpEventStack, queueIndex, abs_addr, eventStackSize);
125         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
126 }
127 //=====================================================================
128 static inline void              HvCallEvent_setLpEventQueueInterruptProc(u8 queueIndex,
129                                                          u16 lpLogicalProcIndex)
130 {
131         HvCall2(HvCallEventSetLpEventQueueInterruptProc,queueIndex,lpLogicalProcIndex);
132         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
133 }
134 //=====================================================================
135 static inline HvLpEvent_Rc HvCallEvent_signalLpEvent(struct HvLpEvent* event)
136 {
137         u64 abs_addr;
138         HvLpEvent_Rc retVal;
139 #ifdef DEBUG_SENDEVENT
140         printk("HvCallEvent_signalLpEvent: *event = %016lx\n ", (unsigned long)event);
141 #endif
142         abs_addr = virt_to_absolute( (unsigned long) event );
143         retVal = (HvLpEvent_Rc)HvCall1(HvCallEventSignalLpEvent, abs_addr);
144         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
145         return retVal;
146 }
147 //=====================================================================
148 static inline HvLpEvent_Rc  HvCallEvent_signalLpEventFast(HvLpIndex targetLp,
149                                           HvLpEvent_Type type,
150                                           u16 subtype,
151                                           HvLpEvent_AckInd ackInd,
152                                           HvLpEvent_AckType ackType,
153                                           HvLpInstanceId sourceInstanceId,
154                                           HvLpInstanceId targetInstanceId,
155                                           u64 correlationToken,
156                                           u64 eventData1,
157                                           u64 eventData2,
158                                           u64 eventData3,
159                                           u64 eventData4,
160                                           u64 eventData5)
161 {
162         HvLpEvent_Rc retVal;
163
164         // Pack the misc bits into a single Dword to pass to PLIC
165         union
166         {
167                 struct HvCallEvent_PackedParms  parms;
168                 u64             dword;
169         } packed;
170         packed.parms.xAckType   = ackType;
171         packed.parms.xAckInd    = ackInd;
172         packed.parms.xRsvd      = 0;
173         packed.parms.xTargetLp  = targetLp;
174         packed.parms.xType      = type;
175         packed.parms.xSubtype   = subtype;
176         packed.parms.xSourceInstId      = sourceInstanceId;
177         packed.parms.xTargetInstId      = targetInstanceId;
178
179         retVal = (HvLpEvent_Rc)HvCall7(HvCallEventSignalLpEventParms,
180                                        packed.dword,
```

```
                                        correlationToken,
                                        eventData1,eventData2,
                                        eventData3,eventData4,
                                        eventData5);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retVal;
}
//=====================================================================
static inline HvLpEvent_Rc      HvCallEvent_ackLpEvent(struct HvLpEvent* event)
{
        u64 abs_addr;
        HvLpEvent_Rc retVal;
        abs_addr = virt_to_absolute( (unsigned long) event );

        retVal = (HvLpEvent_Rc)HvCall1(HvCallEventAckLpEvent, abs_addr);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retVal;
}
//=====================================================================
static inline HvLpEvent_Rc   HvCallEvent_cancelLpEvent(struct HvLpEvent* event)
{
        u64 abs_addr;
        HvLpEvent_Rc retVal;
        abs_addr = virt_to_absolute( (unsigned long) event );

        retVal = (HvLpEvent_Rc)HvCall1(HvCallEventCancelLpEvent, abs_addr);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retVal;
}
//=====================================================================
static inline HvLpInstanceId    HvCallEvent_getSourceLpInstanceId(HvLpIndex targetLp, HvLpEvent_Type type)
{
        HvLpInstanceId retVal;
        retVal = HvCall2(HvCallEventGetSourceLpInstanceId,targetLp,type);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retVal;
}
//=====================================================================
static inline HvLpInstanceId    HvCallEvent_getTargetLpInstanceId(HvLpIndex targetLp, HvLpEvent_Type type)
{
        HvLpInstanceId retVal;
        retVal = HvCall2(HvCallEventGetTargetLpInstanceId,targetLp,type);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retVal;
}
//=====================================================================
static inline void              HvCallEvent_openLpEventPath(HvLpIndex targetLp,
                                                            HvLpEvent_Type type)
{
        HvCall2(HvCallEventOpenLpEventPath,targetLp,type);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
}
//=====================================================================
static inline void              HvCallEvent_closeLpEventPath(HvLpIndex targetLp,
                                                             HvLpEvent_Type type)
{
        HvCall2(HvCallEventCloseLpEventPath,targetLp,type);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
}
//=====================================================================
static inline HvLpDma_Rc        HvCallEvent_dmaBufList(HvLpEvent_Type type,
                                        HvLpIndex remoteLp,
                                        HvLpDma_Direction direction,
                                        HvLpInstanceId localInstanceId,
                                        HvLpInstanceId remoteInstanceId,
                                        HvLpDma_AddressType localAddressType,
                                        HvLpDma_AddressType remoteAddressType,
                                        // Do these need to be converted to
                                        // absolute addresses?
                                        u64 localBufList,
                                        u64 remoteBufList,

                                        u32 transferLength)
{
        HvLpDma_Rc retVal;
        // Pack the misc bits into a single Dword to pass to PLIC
        union
        {
                struct HvCallEvent_PackedDmaParms       parms;
                u64             dword;
        } packed;
        packed.parms.xDirection         = direction;
        packed.parms.xLocalAddrType     = localAddressType;
        packed.parms.xRemoteAddrType    = remoteAddressType;
        packed.parms.xRsvd1             = 0;
        packed.parms.xRemoteLp          = remoteLp;
        packed.parms.xType              = type;
        packed.parms.xRsvd2             = 0;
        packed.parms.xLocalInstId       = localInstanceId;
        packed.parms.xRemoteInstId      = remoteInstanceId;
```

```
271
272         retVal = (HvLpDma_Rc)HvCall4(HvCallEventDmaBufList,
273                                      packed.dword,
274                                      localBufList,
275                                      remoteBufList,
276                                      transferLength);
277         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
278         return retVal;
279 }
280 //=====================================================================
281 static inline HvLpDma_Rc        HvCallEvent_dmaSingle(HvLpEvent_Type type,
282                                       HvLpIndex remoteLp,
283                                       HvLpDma_Direction direction,
284                                       HvLpInstanceId localInstanceId,
285                                       HvLpInstanceId remoteInstanceId,
286                                       HvLpDma_AddressType localAddressType,
287                                       HvLpDma_AddressType remoteAddressType,
288                                       u64 localAddrOrTce,
289                                       u64 remoteAddrOrTce,
290                                       u32 transferLength)
291 {
292         HvLpDma_Rc retVal;
293         // Pack the misc bits into a single Dword to pass to PLIC
294         union
295         {
296                 struct HvCallEvent_PackedDmaParms       parms;
297                 u64             dword;
298         } packed;
299         packed.parms.xDirection         = direction;
300         packed.parms.xLocalAddrType     = localAddressType;
301         packed.parms.xRemoteAddrType    = remoteAddressType;
302         packed.parms.xRsvd1             = 0;
303         packed.parms.xRemoteLp          = remoteLp;
304         packed.parms.xType              = type;
305         packed.parms.xRsvd2             = 0;
306         packed.parms.xLocalInstId       = localInstanceId;
307         packed.parms.xRemoteInstId      = remoteInstanceId;
308
309         retVal = (HvLpDma_Rc)HvCall4(HvCallEventDmaSingle,
310                                      packed.dword,
311                                      localAddrOrTce,
312                                      remoteAddrOrTce,
313                                      transferLength);
314         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
315         return retVal;
316 }
317 //=====================================================================
318 static inline HvLpDma_Rc        HvCallEvent_dmaToSp(void* local, u32 remote, u32 length, HvLpDma_Direction dir)
319 {
320         u64 abs_addr;
321         HvLpDma_Rc retVal;
322         abs_addr = virt_to_absolute( (unsigned long) local );
323
324         retVal = (HvLpDma_Rc)HvCall4(HvCallEventDmaToSp,
325                                      abs_addr,
326                                      remote,
327                                      length,
328                                      dir);
329         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
330         return retVal;
331 }
332 //=====================================================================
333
334 #endif // _HVCALLEVENT_H
335
```

```
/*
 * HvCall.h
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

//============================================================================
//
//      This file contains the "hypervisor call" interface which is used to
//      drive the hypervisor from the OS.
//
//============================================================================

//-----------------------------------------------------------------------
// Standard Includes
//-----------------------------------------------------------------------
#ifndef  _HVCALLSC_H
#include "HvCallSc.h"
#endif

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif

#include <asm/paca.h>

//-----------------------------------------------------------------------
// Constants
//-----------------------------------------------------------------------
#ifndef _HVCALL_H
#define _HVCALL_H
/*
enum HvCall_ReturnCode
{
        HvCall_Good             = 0,
        HvCall_Partial          = 1,
        HvCall_NotOwned         = 2,
        HvCall_NotFreed         = 3,
        HvCall_UnspecifiedError = 4
};

enum HvCall_TypeOfSIT
{
        HvCall_ReduceOnly       = 0,
        HvCall_Unconditional    = 1
};

enum HvCall_TypeOfYield
{
        HvCall_YieldTimed       = 0,    // Yield until specified time
        HvCall_YieldToActive    = 1,    // Yield until all active procs have run
        HvCall_YieldToProc      = 2     // Yield until the specified processor has run
};

enum HvCall_InterruptMasks
{
        HvCall_MaskIPI          = 0x00000001,
        HvCall_MaskLpEvent      = 0x00000002,
        HvCall_MaskLpProd       = 0x00000004,
        HvCall_MaskTimeout      = 0x00000008
};

enum HvCall_VaryOffChunkRc
{
        HvCall_VaryOffSucceeded         = 0,
        HvCall_VaryOffWithdrawn         = 1,
        HvCall_ChunkInLoadArea          = 2,
        HvCall_ChunkInHPT               = 3,
        HvCall_ChunkNotAccessible       = 4,
        HvCall_ChunkInUse               = 5
};
*/

/* Type of yield for HvCallBaseYieldProcessor */
#define HvCall_YieldTimed       0       // Yield until specified time (tb)
#define HvCall_YieldToActive    1       // Yield until all active procs have run
#define HvCall_YieldToProc      2       // Yield until the specified processor has run
```

```
91
92  /* interrupt masks for setEnabledInterrupts */
93  #define HvCall_MaskIPI          0x00000001
94  #define HvCall_MaskLpEvent      0x00000002
95  #define HvCall_MaskLpProd       0x00000004
96  #define HvCall_MaskTimeout      0x00000008
97
98  /* Log buffer formats                       */
99  #define HvCall_LogBuffer_ASCII          0
100 #define HvCall_LogBuffer_EBCDIC         1
101
102 #define HvCallBaseAckDeferredInts                       HvCallBase +  0
103 #define HvCallBaseCpmPowerOff                           HvCallBase +  1
104 #define HvCallBaseGetHwPatch                            HvCallBase +  2
105 #define HvCallBaseReIplSpAttn                           HvCallBase +  3
106 #define HvCallBaseSetASR                                HvCallBase +  4
107 #define HvCallBaseSetASRAndRfi                          HvCallBase +  5
108 #define HvCallBaseSetIMR                                HvCallBase +  6
109 #define HvCallBaseSendIPI                               HvCallBase +  7
110 #define HvCallBaseTerminateMachine                      HvCallBase +  8
111 #define HvCallBaseTerminateMachineSrc                   HvCallBase +  9
112 #define HvCallBaseProcessPlicInterrupts                 HvCallBase + 10
113 #define HvCallBaseIsPrimaryCpmOrMsdIpl                  HvCallBase + 11
114 #define HvCallBaseSetVirtualSIT                         HvCallBase + 12
115 #define HvCallBaseVaryOffThisProcessor                  HvCallBase + 13
116 #define HvCallBaseVaryOffMemoryChunk                    HvCallBase + 14
117 #define HvCallBaseVaryOffInteractivePercentage          HvCallBase + 15
118 #define HvCallBaseSendLpProd                            HvCallBase + 16
119 #define HvCallBaseSetEnabledInterrupts                  HvCallBase + 17
120 #define HvCallBaseYieldProcessor                        HvCallBase + 18
121 #define HvCallBaseVaryOffSharedProcUnits                HvCallBase + 19
122 #define HvCallBaseSetVirtualDecr                        HvCallBase + 20
123 #define HvCallBaseClearLogBuffer                        HvCallBase + 21
124 #define HvCallBaseGetLogBufferCodePage                  HvCallBase + 22
125 #define HvCallBaseGetLogBufferFormat                    HvCallBase + 23
126 #define HvCallBaseGetLogBufferLength                    HvCallBase + 24
127 #define HvCallBaseReadLogBuffer                         HvCallBase + 25
128 #define HvCallBaseSetLogBufferFormatAndCodePage         HvCallBase + 26
129 #define HvCallBaseWriteLogBuffer                        HvCallBase + 27
130 #define HvCallBaseRouter28                              HvCallBase + 28
131 #define HvCallBaseRouter29                              HvCallBase + 29
132 #define HvCallBaseRouter30                              HvCallBase + 30
133
134 #define HvCallCcSetDABR                                 HvCallCc + 7
135
136 //=============================================================================
137 static inline void              HvCall_setVirtualDecr(void)
138 {
139         /* Ignore any error return codes − most likely means that the target value for the
140          * LP has been increased and this vary off would bring us below the new target. */
141         HvCall0(HvCallBaseSetVirtualDecr);
142 }
143 //=====================================================================
144 static inline void              HvCall_yieldProcessor(unsigned typeOfYield, u64 yieldParm)
145 {
146         HvCall2( HvCallBaseYieldProcessor, typeOfYield, yieldParm );
147 }
148 //=====================================================================
149 static inline void              HvCall_setEnabledInterrupts(u64 enabledInterrupts)
150 {
151         HvCall1(HvCallBaseSetEnabledInterrupts,enabledInterrupts);
152 }
153
154 //=====================================================================
155 static inline void              HvCall_clearLogBuffer(HvLpIndex lpindex)
156 {
157         HvCall1(HvCallBaseClearLogBuffer,lpindex);
158 }
159
160 //=====================================================================
161 static inline u32               HvCall_getLogBufferCodePage(HvLpIndex lpindex)
162 {
163         u32 retVal = HvCall1(HvCallBaseGetLogBufferCodePage,lpindex);
164         return retVal;
165 }
166
167 //=====================================================================
168 static inline int               HvCall_getLogBufferFormat(HvLpIndex lpindex)
169 {
170         int retVal = HvCall1(HvCallBaseGetLogBufferFormat,lpindex);
171         return retVal;
172 }
173
174 //=====================================================================
175 static inline u32               HvCall_getLogBufferLength(HvLpIndex lpindex)
176 {
177         u32 retVal = HvCall1(HvCallBaseGetLogBufferLength,lpindex);
178         return retVal;
179 }
180
```

```
181 //=============================================================================
182 static inline void             HvCall_setLogBufferFormatAndCodepage(int format, u32 codePage)
183 {
184         HvCall2(HvCallBaseSetLogBufferFormatAndCodePage,format, codePage);
185 }
186
187 //=============================================================================
188 int HvCall_readLogBuffer(HvLpIndex lpindex, void *buffer, u64 bufLen);
189 void HvCall_writeLogBuffer(const void *buffer, u64 bufLen);
190
191 //=============================================================================
192 static inline void             HvCall_sendIPI(struct paca_struct * targetPaca)
193 {
194         HvCall1( HvCallBaseSendIPI, targetPaca->xPacaIndex );
195 }
196
197 //=============================================================================
198 static inline void             HvCall_terminateMachineSrc(void)
199 {
200         HvCall0( HvCallBaseTerminateMachineSrc );
201 }
202
203 static inline void HvCall_setDABR(unsigned long val)
204 {
205         HvCall1(HvCallCcSetDABR, val);
206 }
207
208 #endif // _HVCALL_H
209
```

```
/*
 * HvCallHpt.h
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

//============================================================================
//
//      This file contains the "hypervisor call" interface which is used to
//      drive the hypervisor from the OS.
//
//============================================================================

//----------------------------------------------------------------------
// Standard Includes
//----------------------------------------------------------------------
#ifndef  _HVCALLSC_H
#include "HvCallSc.h"
#endif

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif

//----------------------------------------------------------------------
// Other Includes
//----------------------------------------------------------------------

#ifndef _PPC_MMU_H
#include <asm/mmu.h>
#endif

//-----------------------------------------------------------------------------
// Constants
//-----------------------------------------------------------------------------
#ifndef _HVCALLHPT_H
#define _HVCALLHPT_H

#define HvCallHptGetHptAddress          HvCallHpt +  0
#define HvCallHptGetHptPages            HvCallHpt +  1
#define HvCallHptSetPp                  HvCallHpt +  5
#define HvCallHptSetSwBits              HvCallHpt +  6
#define HvCallHptUpdate                 HvCallHpt +  7
#define HvCallHptInvalidateNoSyncICache HvCallHpt +  8
#define HvCallHptGet                    HvCallHpt + 11
#define HvCallHptFindNextValid          HvCallHpt + 12
#define HvCallHptFindValid              HvCallHpt + 13
#define HvCallHptAddValidate            HvCallHpt + 16
#define HvCallHptInvalidateSetSwBitsGet HvCallHpt + 18


//=============================================================================
static inline u64               HvCallHpt_getHptAddress(void)
{
        u64 retval = HvCall0(HvCallHptGetHptAddress);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retval;
}
//=============================================================================
static inline u64               HvCallHpt_getHptPages(void)
{
        u64 retval = HvCall0(HvCallHptGetHptPages);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        return retval;
}
//=============================================================================
static inline void              HvCallHpt_setPp(u32 hpteIndex, u8 value)
{
        HvCall2( HvCallHptSetPp, hpteIndex, value );
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
}
//=============================================================================
static inline void              HvCallHpt_setSwBits(u32 hpteIndex, u8 bitson, u8 bitsoff )
{
        HvCall3( HvCallHptSetSwBits, hpteIndex, bitson, bitsoff );
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
}
```

```
91  //================================================================================
92  static inline void              HvCallHpt_invalidateNoSyncICache(u32 hpteIndex)
93
94  {
95          HvCall1( HvCallHptInvalidateNoSyncICache, hpteIndex );
96          // getPaca()->adjustHmtForNoOfSpinLocksHeld();
97  }
98  //================================================================================
99  static inline u64               HvCallHpt_invalidateSetSwBitsGet(u32 hpteIndex, u8 bitson, u8 bitsoff )
100
101 {
102         u64 compressedStatus;
103         compressedStatus = HvCall4( HvCallHptInvalidateSetSwBitsGet, hpteIndex, bitson, bitsoff, 1 );
104         HvCall1( HvCallHptInvalidateNoSyncICache, hpteIndex );
105         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
106         return compressedStatus;
107 }
108 //================================================================================
109 static inline u64               HvCallHpt_findValid( struct _HPTE *hpte, u64 vpn )
110 {
111         u64 retIndex = HvCall3Ret16( HvCallHptFindValid, hpte, vpn, 0, 0 );
112         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
113         return retIndex;
114 }
115 //================================================================================
116 static inline u64               HvCallHpt_findNextValid( struct _HPTE *hpte, u32 hpteIndex, u8 bitson, u8 bitsoff
     )
117 {
118         u64 retIndex = HvCall3Ret16( HvCallHptFindNextValid, hpte, hpteIndex, bitson, bitsoff );
119         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
120         return retIndex;
121 }
122 //================================================================================
123 static inline void              HvCallHpt_get( struct _HPTE *hpte, u32 hpteIndex )
124 {
125         HvCall2Ret16( HvCallHptGet, hpte, hpteIndex, 0 );
126         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
127 }
128 //================================================================================
129 static inline void              HvCallHpt_addValidate( u32 hpteIndex,
130                                                        u32 hBit,
131                                                        struct _HPTE *hpte )
132
133 {
134         HvCall4( HvCallHptAddValidate, hpteIndex,
135                  hBit, (*((u64 *)hpte)), (*(((u64 *)hpte)+1)) );
136         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
137 }
138
139
140 //================================================================================
141
142 #endif // _HVCALLHPT_H
143
```

```
/************************************************************************/
/* Provides the Hypervisor PCI calls for iSeries Linux Parition.        */
/* Copyright (C) 20yy  <Wayne G Holm> <IBM Corporation>                 */
/*                                                                      */
/* 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 program is distributed in the hope that it will be useful,      */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of       */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        */
/* GNU General Public License for more details.                         */
/*                                                                      */
/* 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.,                                      */
/* 59 Temple Place, Suite 330,                                          */
/* Boston, MA  02111−1307  USA                                          */
/************************************************************************/
/* Change Activity:                                                     */
/*   Created, Jan 9, 2001                                              */
/************************************************************************/
//============================================================================
//                                                          Header File Id
// Name_____________:  HvCallPci.H
//
// Description_______:
//
//      This file contains the "hypervisor call" interface which is used to
//      drive the hypervisor from SLIC.
//
//============================================================================

//--------------------------------------------------------------------
// Forward declarations
//--------------------------------------------------------------------

//--------------------------------------------------------------------
// Standard Includes
//--------------------------------------------------------------------
#ifndef  _HVCALLSC_H
#include "HvCallSc.h"
#endif

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif

//--------------------------------------------------------------------
// Other Includes
//--------------------------------------------------------------------


//--------------------------------------------------------------------------------
// Constants
//--------------------------------------------------------------------------------
#ifndef _HVCALLPCI_H
#define _HVCALLPCI_H

struct HvCallPci_DsaAddr { // make sure this struct size is 64−bits total
        u16             busNumber;
        u8              subBusNumber;
        u8              deviceId;
        u8              barNumber;
        u8              reserved[3];
};
union HvDsaMap {
        u64     DsaAddr;
        struct HvCallPci_DsaAddr Dsa;
};

struct HvCallPci_LoadReturn {
        u64             rc;
        u64             value;
};

enum HvCallPci_DeviceType {HvCallPci_NodeDevice = 1,
                           HvCallPci_SpDevice      = 2,
                           HvCallPci_IopDevice     = 3,
                           HvCallPci_BridgeDevice  = 4,
                           HvCallPci_MultiFunctionDevice = 5,
                           HvCallPci_IoaDevice     = 6
};


struct HvCallPci_DeviceInfo {
        u32     deviceType;             // See DeviceType enum for values
};

```

```
91  struct HvCallPci_BusUnitInfo {
92          u32     sizeReturned;           // length of data returned
93          u32     deviceType;             // see DeviceType enum for values
94  };
95
96  struct HvCallPci_BridgeInfo {
97          struct HvCallPci_BusUnitInfo busUnitInfo;  // Generic bus unit info
98          u8              subBusNumber;           // Bus number of secondary bus
99          u8              maxAgents;              // Max idsels on secondary bus
100         u8              maxSubBusNumber;        // Max Sub Bus
101         u8              logicalSlotNumber;      // Logical Slot Number for IOA
102 };
103
104
105 //  Maximum BusUnitInfo buffer size.  Provided for clients so they can allocate
106 //  a buffer big enough for any type of bus unit.  Increase as needed.
107 enum {HvCallPci_MaxBusUnitInfoSize = 128};
108
109 struct HvCallPci_BarParms {
110         u64             vaddr;
111         u64             raddr;
112         u64             size;
113         u64             protectStart;
114         u64             protectEnd;
115         u64             relocationOffset;
116         u64             pciAddress;
117         u64             reserved[3];
118 };
119
120 enum HvCallPci_VpdType {
121         HvCallPci_BusVpd                = 1,
122         HvCallPci_BusAdapterVpd = 2
123 };
124
125 #define HvCallPciConfigLoad8            HvCallPci + 0
126 #define HvCallPciConfigLoad16           HvCallPci + 1
127 #define HvCallPciConfigLoad32           HvCallPci + 2
128 #define HvCallPciConfigStore8           HvCallPci + 3
129 #define HvCallPciConfigStore16          HvCallPci + 4
130 #define HvCallPciConfigStore32          HvCallPci + 5
131 #define HvCallPciEoi                    HvCallPci + 16
132 #define HvCallPciGetBarParms            HvCallPci + 18
133 #define HvCallPciMaskFisr               HvCallPci + 20
134 #define HvCallPciUnmaskFisr             HvCallPci + 21
135 #define HvCallPciSetSlotReset           HvCallPci + 25
136 #define HvCallPciGetDeviceInfo          HvCallPci + 27
137 #define HvCallPciGetCardVpd             HvCallPci + 28
138 #define HvCallPciBarLoad8               HvCallPci + 40
139 #define HvCallPciBarLoad16              HvCallPci + 41
140 #define HvCallPciBarLoad32              HvCallPci + 42
141 #define HvCallPciBarLoad64              HvCallPci + 43
142 #define HvCallPciBarStore8              HvCallPci + 44
143 #define HvCallPciBarStore16             HvCallPci + 45
144 #define HvCallPciBarStore32             HvCallPci + 46
145 #define HvCallPciBarStore64             HvCallPci + 47
146 #define HvCallPciMaskInterrupts         HvCallPci + 48
147 #define HvCallPciUnmaskInterrupts       HvCallPci + 49
148 #define HvCallPciGetBusUnitInfo         HvCallPci + 50
149
150 //=============================================================================
151 static inline u64       HvCallPci_configLoad8(u16 busNumber, u8 subBusNumber,
152                                               u8 deviceId, u32 offset,
153                                               u8 *value)
154 {
155         struct HvCallPci_DsaAddr dsa;
156         struct HvCallPci_LoadReturn retVal;
157
158         *((u64*)&dsa) = 0;
159
160         dsa.busNumber = busNumber;
161         dsa.subBusNumber = subBusNumber;
162         dsa.deviceId = deviceId;
163
164         HvCall3Ret16(HvCallPciConfigLoad8, &retVal, *(u64 *)&dsa, offset, 0);
165
166         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
167
168         *value = retVal.value;
169
170         return retVal.rc;
171 }
172 //=============================================================================
173 static inline u64       HvCallPci_configLoad16(u16 busNumber, u8 subBusNumber,
174                                               u8 deviceId, u32 offset,
175                                               u16 *value)
176 {
177         struct HvCallPci_DsaAddr dsa;
178         struct HvCallPci_LoadReturn retVal;
179
180         *((u64*)&dsa) = 0;
```

```
181
182         dsa.busNumber = busNumber;
183         dsa.subBusNumber = subBusNumber;
184         dsa.deviceId = deviceId;
185
186         HvCall3Ret16(HvCallPciConfigLoad16, &retVal, *(u64 *)&dsa, offset, 0);
187
188         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
189
190         *value = retVal.value;
191
192         return retVal.rc;
193 }
194 //=====================================================================
195 static inline u64       HvCallPci_configLoad32(u16 busNumber, u8 subBusNumber,
196                                                u8 deviceId, u32 offset,
197                                                u32 *value)
198 {
199         struct HvCallPci_DsaAddr dsa;
200         struct HvCallPci_LoadReturn retVal;
201
202         *((u64*)&dsa) = 0;
203
204         dsa.busNumber = busNumber;
205         dsa.subBusNumber = subBusNumber;
206         dsa.deviceId = deviceId;
207
208         HvCall3Ret16(HvCallPciConfigLoad32, &retVal, *(u64 *)&dsa, offset, 0);
209
210         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
211
212         *value = retVal.value;
213
214         return retVal.rc;
215 }
216 //=====================================================================
217 static inline u64       HvCallPci_configStore8(u16 busNumber, u8 subBusNumber,
218                                                u8 deviceId, u32 offset,
219                                                u8  value)
220 {
221         struct HvCallPci_DsaAddr dsa;
222         u64 retVal;
223
224         *((u64*)&dsa) = 0;
225
226         dsa.busNumber = busNumber;
227         dsa.subBusNumber = subBusNumber;
228         dsa.deviceId = deviceId;
229
230         retVal = HvCall4(HvCallPciConfigStore8, *(u64 *)&dsa, offset, value, 0);
231
232         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
233
234         return retVal;
235 }
236 //=====================================================================
237 static inline u64       HvCallPci_configStore16(u16 busNumber, u8 subBusNumber,
238                                                u8 deviceId, u32 offset,
239                                                u16  value)
240 {
241         struct HvCallPci_DsaAddr dsa;
242         u64 retVal;
243
244         *((u64*)&dsa) = 0;
245
246         dsa.busNumber = busNumber;
247         dsa.subBusNumber = subBusNumber;
248         dsa.deviceId = deviceId;
249
250         retVal = HvCall4(HvCallPciConfigStore16, *(u64 *)&dsa, offset, value, 0);
251
252         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
253
254         return retVal;
255 }
256 //=====================================================================
257 static inline u64       HvCallPci_configStore32(u16 busNumber, u8 subBusNumber,
258                                                u8 deviceId, u32 offset,
259                                                u32  value)
260 {
261         struct HvCallPci_DsaAddr dsa;
262         u64 retVal;
263
264         *((u64*)&dsa) = 0;
265
266         dsa.busNumber = busNumber;
267         dsa.subBusNumber = subBusNumber;
268         dsa.deviceId = deviceId;
269
270         retVal = HvCall4(HvCallPciConfigStore32, *(u64 *)&dsa, offset, value, 0);
```

```
271
272         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
273
274         return retVal;
275 }
276 //=====================================================================
277 static inline u64       HvCallPci_barLoad8(u16  busNumberParm,
278                                            u8           subBusParm,
279                                            u8           deviceIdParm,
280                                            u8           barNumberParm,
281                                            u64          offsetParm,
282                                            u8*          valueParm)
283 {
284         struct HvCallPci_DsaAddr dsa;
285         struct HvCallPci_LoadReturn retVal;
286
287         *((u64*)&dsa) = 0;
288
289         dsa.busNumber = busNumberParm;
290         dsa.subBusNumber = subBusParm;
291         dsa.deviceId = deviceIdParm;
292         dsa.barNumber = barNumberParm;
293
294         HvCall3Ret16(HvCallPciBarLoad8, &retVal, *(u64 *)&dsa, offsetParm, 0);
295
296         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
297
298         *valueParm = retVal.value;
299
300         return retVal.rc;
301 }
302 //=====================================================================
303 static inline u64       HvCallPci_barLoad16(u16 busNumberParm,
304                                             u8           subBusParm,
305                                             u8           deviceIdParm,
306                                             u8           barNumberParm,
307                                             u64          offsetParm,
308                                             u16*         valueParm)
309 {
310         struct HvCallPci_DsaAddr dsa;
311         struct HvCallPci_LoadReturn retVal;
312
313         *((u64*)&dsa) = 0;
314
315         dsa.busNumber = busNumberParm;
316         dsa.subBusNumber = subBusParm;
317         dsa.deviceId = deviceIdParm;
318         dsa.barNumber = barNumberParm;
319
320         HvCall3Ret16(HvCallPciBarLoad16, &retVal, *(u64 *)&dsa, offsetParm, 0);
321
322         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
323
324         *valueParm = retVal.value;
325
326         return retVal.rc;
327 }
328 //=====================================================================
329 static inline u64       HvCallPci_barLoad32(u16 busNumberParm,
330                                             u8           subBusParm,
331                                             u8           deviceIdParm,
332                                             u8           barNumberParm,
333                                             u64          offsetParm,
334                                             u32*         valueParm)
335 {
336         struct HvCallPci_DsaAddr dsa;
337         struct HvCallPci_LoadReturn retVal;
338
339         *((u64*)&dsa) = 0;
340
341         dsa.busNumber = busNumberParm;
342         dsa.subBusNumber = subBusParm;
343         dsa.deviceId = deviceIdParm;
344         dsa.barNumber = barNumberParm;
345
346         HvCall3Ret16(HvCallPciBarLoad32, &retVal, *(u64 *)&dsa, offsetParm, 0);
347
348         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
349
350         *valueParm = retVal.value;
351
352         return retVal.rc;
353 }
354 //=====================================================================
355 static inline u64       HvCallPci_barLoad64(u16 busNumberParm,
356                                             u8           subBusParm,
357                                             u8           deviceIdParm,
358                                             u8           barNumberParm,
359                                             u64          offsetParm,
360                                             u64*         valueParm)
```

```
361 {
362         struct HvCallPci_DsaAddr dsa;
363         struct HvCallPci_LoadReturn retVal;
364
365         *((u64*)&dsa) = 0;
366
367         dsa.busNumber = busNumberParm;
368         dsa.subBusNumber = subBusParm;
369         dsa.deviceId = deviceIdParm;
370         dsa.barNumber = barNumberParm;
371
372         HvCall3Ret16(HvCallPciBarLoad64, &retVal, *(u64 *)&dsa, offsetParm, 0);
373
374         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
375
376         *valueParm = retVal.value;
377
378         return retVal.rc;
379 }
380 //===================================================================================
381 static inline u64       HvCallPci_barStore8(u16 busNumberParm,
382                                             u8          subBusParm,
383                                             u8          deviceIdParm,
384                                             u8          barNumberParm,
385                                             u64         offsetParm,
386                                             u8          valueParm)
387 {
388         struct HvCallPci_DsaAddr dsa;
389         u64 retVal;
390
391         *((u64*)&dsa) = 0;
392
393         dsa.busNumber = busNumberParm;
394         dsa.subBusNumber = subBusParm;
395         dsa.deviceId = deviceIdParm;
396         dsa.barNumber = barNumberParm;
397
398         retVal = HvCall4(HvCallPciBarStore8, *(u64 *)&dsa, offsetParm, valueParm, 0);
399
400         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
401
402         return retVal;
403 }
404 //===================================================================================
405 static inline u64       HvCallPci_barStore16(u16        busNumberParm,
406                                              u8         subBusParm,
407                                              u8         deviceIdParm,
408                                              u8         barNumberParm,
409                                              u64        offsetParm,
410                                              u16        valueParm)
411 {
412         struct HvCallPci_DsaAddr dsa;
413         u64 retVal;
414
415         *((u64*)&dsa) = 0;
416
417         dsa.busNumber = busNumberParm;
418         dsa.subBusNumber = subBusParm;
419         dsa.deviceId = deviceIdParm;
420         dsa.barNumber = barNumberParm;
421
422         retVal = HvCall4(HvCallPciBarStore16, *(u64 *)&dsa, offsetParm, valueParm, 0);
423
424         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
425
426         return retVal;
427 }
428 //===================================================================================
429 static inline u64       HvCallPci_barStore32(u16        busNumberParm,
430                                              u8         subBusParm,
431                                              u8         deviceIdParm,
432                                              u8         barNumberParm,
433                                              u64        offsetParm,
434                                              u32        valueParm)
435 {
436         struct HvCallPci_DsaAddr dsa;
437         u64 retVal;
438
439         *((u64*)&dsa) = 0;
440
441         dsa.busNumber = busNumberParm;
442         dsa.subBusNumber = subBusParm;
443         dsa.deviceId = deviceIdParm;
444         dsa.barNumber = barNumberParm;
445
446         retVal = HvCall4(HvCallPciBarStore32, *(u64 *)&dsa, offsetParm, valueParm, 0);
447
448         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
449
450         return retVal;
```

```
}
//=============================================================================
static inline u64       HvCallPci_barStore64(u16        busNumberParm,
                                             u8         subBusParm,
                                             u8         deviceIdParm,
                                             u8         barNumberParm,
                                             u64        offsetParm,
                                             u64        valueParm)
{
        struct HvCallPci_DsaAddr dsa;
        u64 retVal;

        *((u64*)&dsa) = 0;

        dsa.busNumber = busNumberParm;
        dsa.subBusNumber = subBusParm;
        dsa.deviceId = deviceIdParm;
        dsa.barNumber = barNumberParm;

        retVal = HvCall4(HvCallPciBarStore64, *(u64 *)&dsa, offsetParm, valueParm, 0);

        // getPaca()->adjustHmtForNoOfSpinLocksHeld();

        return retVal;
}
//=============================================================================
static inline u64       HvCallPci_eoi(u16       busNumberParm,
                                      u8        subBusParm,
                                      u8        deviceIdParm)
{
        struct HvCallPci_DsaAddr dsa;
        struct HvCallPci_LoadReturn retVal;

        *((u64*)&dsa) = 0;

        dsa.busNumber = busNumberParm;
        dsa.subBusNumber = subBusParm;
        dsa.deviceId = deviceIdParm;

        HvCall1Ret16(HvCallPciEoi, &retVal, *(u64*)&dsa);

        // getPaca()->adjustHmtForNoOfSpinLocksHeld();

        return retVal.rc;
}
//=============================================================================
static inline u64       HvCallPci_getBarParms(u16       busNumberParm,
                                              u8        subBusParm,
                                              u8        deviceIdParm,
                                              u8        barNumberParm,
                                              u64       parms,
                                              u32       sizeofParms)
{
        struct HvCallPci_DsaAddr dsa;
        u64 retVal;

        *((u64*)&dsa) = 0;

        dsa.busNumber = busNumberParm;
        dsa.subBusNumber = subBusParm;
        dsa.deviceId = deviceIdParm;
        dsa.barNumber = barNumberParm;

        retVal = HvCall3(HvCallPciGetBarParms, *(u64*)&dsa, parms, sizeofParms);

        // getPaca()->adjustHmtForNoOfSpinLocksHeld();

        return retVal;
}
//=============================================================================
static inline u64       HvCallPci_maskFisr(u16  busNumberParm,
                                           u8   subBusParm,
                                           u8   deviceIdParm,
                                           u64  fisrMask)
{
        struct HvCallPci_DsaAddr dsa;
        u64 retVal;

        *((u64*)&dsa) = 0;

        dsa.busNumber = busNumberParm;
        dsa.subBusNumber = subBusParm;
        dsa.deviceId = deviceIdParm;

        retVal = HvCall2(HvCallPciMaskFisr, *(u64*)&dsa, fisrMask);

        // getPaca()->adjustHmtForNoOfSpinLocksHeld();

        return retVal;
}
```

```
541 //============================================================================
542 static inline u64       HvCallPci_unmaskFisr(u16        busNumberParm,
543                                              u8         subBusParm,
544                                              u8         deviceIdParm,
545                                              u64        fisrMask)
546 {
547         struct HvCallPci_DsaAddr dsa;
548         u64 retVal;
549
550         *((u64*)&dsa) = 0;
551
552         dsa.busNumber = busNumberParm;
553         dsa.subBusNumber = subBusParm;
554         dsa.deviceId = deviceIdParm;
555
556         retVal = HvCall2(HvCallPciUnmaskFisr, *(u64*)&dsa, fisrMask);
557
558         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
559
560         return retVal;
561 }
562 //============================================================================
563 static inline u64       HvCallPci_setSlotReset(u16              busNumberParm,
564                                                u8               subBusParm,
565                                                u8               deviceIdParm,
566                                                u64              onNotOff)
567 {
568         struct HvCallPci_DsaAddr dsa;
569         u64 retVal;
570
571         *((u64*)&dsa) = 0;
572
573         dsa.busNumber = busNumberParm;
574         dsa.subBusNumber = subBusParm;
575         dsa.deviceId = deviceIdParm;
576
577         retVal = HvCall2(HvCallPciSetSlotReset, *(u64*)&dsa, onNotOff);
578
579         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
580
581         return retVal;
582 }
583 //============================================================================
584 static inline u64       HvCallPci_getDeviceInfo(u16     busNumberParm,
585                                                 u8      subBusParm,
586                                                 u8      deviceNumberParm,
587                                                 u64     parms,
588                                                 u32     sizeofParms)
589 {
590         struct HvCallPci_DsaAddr dsa;
591         u64 retVal;
592
593         *((u64*)&dsa) = 0;
594
595         dsa.busNumber = busNumberParm;
596         dsa.subBusNumber = subBusParm;
597         dsa.deviceId = deviceNumberParm << 4;
598
599         retVal = HvCall3(HvCallPciGetDeviceInfo, *(u64*)&dsa, parms, sizeofParms);
600
601         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
602
603         return retVal;
604 }
605 //============================================================================
606 static inline u64       HvCallPci_maskInterrupts(u16    busNumberParm,
607                                                  u8     subBusParm,
608                                                  u8     deviceIdParm,
609                                                  u64    interruptMask)
610 {
611         struct HvCallPci_DsaAddr dsa;
612         u64 retVal;
613
614         *((u64*)&dsa) = 0;
615
616         dsa.busNumber = busNumberParm;
617         dsa.subBusNumber = subBusParm;
618         dsa.deviceId = deviceIdParm;
619
620         retVal = HvCall2(HvCallPciMaskInterrupts, *(u64*)&dsa, interruptMask);
621
622         // getPaca()->adjustHmtForNoOfSpinLocksHeld();
623
624         return retVal;
625 }
626 //============================================================================
627 static inline u64       HvCallPci_unmaskInterrupts(u16  busNumberParm,
628                                                    u8           subBusParm,
629                                                    u8           deviceIdParm,
630                                                    u64          interruptMask)
```

```
{
        struct HvCallPci_DsaAddr dsa;
        u64 retVal;

        *((u64*)&dsa) = 0;

        dsa.busNumber = busNumberParm;
        dsa.subBusNumber = subBusParm;
        dsa.deviceId = deviceIdParm;

        retVal = HvCall2(HvCallPciUnmaskInterrupts, *(u64*)&dsa, interruptMask);

        // getPaca()->adjustHmtForNoOfSpinLocksHeld();

        return retVal;
}
//=============================================================================

static inline u64       HvCallPci_getBusUnitInfo(u16            busNumberParm,
                                                 u8             subBusParm,
                                                 u8             deviceIdParm,
                                                 u64            parms,
                                                 u32            sizeofParms)
{
        struct HvCallPci_DsaAddr dsa;
        u64 retVal;

        *((u64*)&dsa) = 0;

        dsa.busNumber = busNumberParm;
        dsa.subBusNumber = subBusParm;
        dsa.deviceId = deviceIdParm;

        retVal = HvCall3(HvCallPciGetBusUnitInfo, *(u64*)&dsa, parms, sizeofParms);

        // getPaca()->adjustHmtForNoOfSpinLocksHeld();

        return retVal;
}
//=============================================================================

static inline int HvCallPci_getBusVpd(u16 busNumParm, u64 destParm, u16 sizeParm)
{
        int xRetSize;
        u64 xRc = HvCall4(HvCallPciGetCardVpd, busNumParm, destParm, sizeParm, HvCallPci_BusVpd);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        if (xRc == -1)
                xRetSize = -1;
        else
                xRetSize = xRc & 0xFFFF;
        return xRetSize;
}
//=============================================================================

static inline int HvCallPci_getBusAdapterVpd(u16 busNumParm, u64 destParm, u16 sizeParm)
{
        int xRetSize;
        u64 xRc = HvCall4(HvCallPciGetCardVpd, busNumParm, destParm, sizeParm, HvCallPci_BusAdapterVpd);
        // getPaca()->adjustHmtForNoOfSpinLocksHeld();
        if (xRc == -1)
                xRetSize = -1;
        else
                xRetSize = xRc & 0xFFFF;
        return xRetSize;
}
//=============================================================================
#endif // _HVCALLPCI_H
```

```
/*
 * HvLpConfig.h
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

//============================================================================
//
//      This file contains the interface to the LPAR configuration data
//  to determine which resources should be allocated to each partition.
//
//============================================================================

#ifndef  _HVCALLCFG_H
#include "HvCallCfg.h"
#endif

#ifndef  _HVTYPES_H
#include <asm/iSeries/HvTypes.h>
#endif

#ifndef  _ITLPNACA_H
#include <asm/iSeries/ItLpNaca.h>
#endif

#ifndef  _LPARDATA_H
#include <asm/iSeries/LparData.h>
#endif

#ifndef _HVLPCONFIG_H
#define _HVLPCONFIG_H

//---------------------------------------------------------------------
// Constants
//---------------------------------------------------------------------

extern HvLpIndex HvLpConfig_getLpIndex_outline(void);

//=====================================================================
static inline HvLpIndex HvLpConfig_getLpIndex(void)
{
        return itLpNaca.xLpIndex;
}
//=====================================================================
static inline HvLpIndex HvLpConfig_getPrimaryLpIndex(void)
{
        return itLpNaca.xPrimaryLpIndex;
}
//=====================================================================
static inline HvLpIndex HvLpConfig_getLps(void)
{
        return HvCallCfg_getLps();
}
//=====================================================================
static inline HvLpIndexMap      HvLpConfig_getActiveLpMap(void)
{
        return HvCallCfg_getActiveLpMap();
}
//=====================================================================
static inline u64               HvLpConfig_getSystemMsMegs(void)
{
        return HvCallCfg_getSystemMsChunks() / HVCHUNKSPERMEG;
}
//=====================================================================
static inline u64               HvLpConfig_getSystemMsChunks(void)
{
        return HvCallCfg_getSystemMsChunks();
}
//=====================================================================
static inline u64               HvLpConfig_getSystemMsPages(void)
{
        return HvCallCfg_getSystemMsChunks() * HVPAGESPERCHUNK;
}
//=====================================================================
static inline u64               HvLpConfig_getMsMegs(void)
{
        return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Cur) / HVCHUNKSPERMEG;
```

```
91  }
92  //=====================================================================
93  static inline u64               HvLpConfig_getMsChunks(void)
94  {
95          return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Cur);
96  }
97  //=====================================================================
98  static inline u64               HvLpConfig_getMsPages(void)
99  {
100         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Cur) * HVPAGESPERCHUNK;
101 }
102 //=====================================================================
103 static inline u64               HvLpConfig_getMinMsMegs(void)
104 {
105         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Min) / HVCHUNKSPERMEG;
106 }
107 //=====================================================================
108 static inline u64               HvLpConfig_getMinMsChunks(void)
109 {
110         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Min);
111 }
112 //=====================================================================
113 static inline u64               HvLpConfig_getMinMsPages(void)
114 {
115         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Min) * HVPAGESPERCHUNK;
116 }
117 //=====================================================================
118 static inline u64               HvLpConfig_getMinRuntimeMsMegs(void)
119 {
120         return HvCallCfg_getMinRuntimeMsChunks(HvLpConfig_getLpIndex()) / HVCHUNKSPERMEG;
121 }
122 //=====================================================================
123 static inline u64               HvLpConfig_getMinRuntimeMsChunks(void)
124 {
125         return HvCallCfg_getMinRuntimeMsChunks(HvLpConfig_getLpIndex());
126 }
127 //=====================================================================
128 static inline u64               HvLpConfig_getMinRuntimeMsPages(void)
129 {
130         return HvCallCfg_getMinRuntimeMsChunks(HvLpConfig_getLpIndex()) * HVPAGESPERCHUNK;
131 }
132 //=====================================================================
133 static inline u64               HvLpConfig_getMaxMsMegs(void)
134 {
135         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Max) / HVCHUNKSPERMEG;
136 }
137 //=====================================================================
138 static inline u64               HvLpConfig_getMaxMsChunks(void)
139 {
140         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Max);
141 }
142 //=====================================================================
143 static inline u64               HvLpConfig_getMaxMsPages(void)
144 {
145         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Max) * HVPAGESPERCHUNK;
146 }
147 //=====================================================================
148 static inline u64               HvLpConfig_getInitMsMegs(void)
149 {
150         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Init) / HVCHUNKSPERMEG;
151 }
152 //=====================================================================
153 static inline u64               HvLpConfig_getInitMsChunks(void)
154 {
155         return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Init);
156 }
157 //=====================================================================
158 static inline u64               HvLpConfig_getInitMsPages(void)
159 {    return HvCallCfg_getMsChunks(HvLpConfig_getLpIndex(),HvCallCfg_Init) * HVPAGESPERCHUNK;
160 }
161 //=====================================================================
162 static inline u64               HvLpConfig_getSystemPhysicalProcessors(void)
163 {
164         return HvCallCfg_getSystemPhysicalProcessors();
165 }
166 //=====================================================================
167 static inline u64               HvLpConfig_getSystemLogicalProcessors(void)
168 {
169         return HvCallCfg_getSystemPhysicalProcessors() * (/*getPaca()->getSecondaryThreadCount() +*/ 1);
170 }
171 //=====================================================================
172 static inline u64               HvLpConfig_getNumProcsInSharedPool(HvLpSharedPoolIndex sPI)
173 {
174         return HvCallCfg_getNumProcsInSharedPool(sPI);
175 }
176 //=====================================================================
177 static inline u64               HvLpConfig_getPhysicalProcessors(void)
178 {
179         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Cur);
180 }
```

```
181 //=====================================================================
182 static inline u64               HvLpConfig_getLogicalProcessors(void)
183 {
184         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Cur) * (/*getPaca()->getSecondar
    yThreadCount() +*/ 1);
185 }
186 //=====================================================================
187 static inline HvLpSharedPoolIndex       HvLpConfig_getSharedPoolIndex(void)
188 {
189         return HvCallCfg_getSharedPoolIndex(HvLpConfig_getLpIndex());
190 }
191 //=====================================================================
192 static inline u64               HvLpConfig_getSharedProcUnits(void)
193 {
194         return HvCallCfg_getSharedProcUnits(HvLpConfig_getLpIndex(),HvCallCfg_Cur);
195 }
196 //=====================================================================
197 static inline u64               HvLpConfig_getMinSharedProcUnits(void)
198 {
199         return HvCallCfg_getSharedProcUnits(HvLpConfig_getLpIndex(),HvCallCfg_Min);
200 }
201 //=====================================================================
202 static inline u64               HvLpConfig_getMaxSharedProcUnits(void)
203 {
204         return HvCallCfg_getSharedProcUnits(HvLpConfig_getLpIndex(),HvCallCfg_Max);
205 }
206 //=====================================================================
207 static inline u64               HvLpConfig_getMinPhysicalProcessors(void)
208 {
209         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Min);
210 }
211 //=====================================================================
212 static inline u64               HvLpConfig_getMinLogicalProcessors(void)
213 {
214         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Min) * (/*getPaca()->getSecondar
    yThreadCount() +*/ 1);
215 }
216 //=====================================================================
217 static inline u64               HvLpConfig_getMaxPhysicalProcessors(void)
218 {
219         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Max);
220 }
221 //=====================================================================
222 static inline u64               HvLpConfig_getMaxLogicalProcessors(void)
223 {
224         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Max) * (/*getPaca()->getSecondar
    yThreadCount() +*/ 1);
225 }
226 //=====================================================================
227 static inline u64               HvLpConfig_getInitPhysicalProcessors(void)
228 {
229         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Init);
230 }
231 //=====================================================================
232 static inline u64               HvLpConfig_getInitLogicalProcessors(void)
233 {
234         return HvCallCfg_getPhysicalProcessors(HvLpConfig_getLpIndex(),HvCallCfg_Init) * (/*getPaca()->getSeconda
    ryThreadCount() +*/ 1);
235 }
236 //=====================================================================
237 static inline HvLpVirtualLanIndexMap    HvLpConfig_getVirtualLanIndexMap(void)
238 {
239         return HvCallCfg_getVirtualLanIndexMap(HvLpConfig_getLpIndex_outline());
240 }
241 //=====================================================================
242 static inline HvLpVirtualLanIndexMap    HvLpConfig_getVirtualLanIndexMapForLp(HvLpIndex lp)
243 {
244         return HvCallCfg_getVirtualLanIndexMap(lp);
245 }
246 //=====================================================================
247 static inline HvLpIndex HvLpConfig_getBusOwner(HvBusNumber busNumber)
248 {
249         return HvCallCfg_getBusOwner(busNumber);
250 }
251 //=====================================================================
252 static inline int               HvLpConfig_isBusDedicated(HvBusNumber busNumber)
253 {
254         return HvCallCfg_isBusDedicated(busNumber);
255 }
256 //=====================================================================
257 static inline HvLpIndexMap      HvLpConfig_getBusAllocation(HvBusNumber busNumber)
258 {
259         return HvCallCfg_getBusAllocation(busNumber);
260 }
261 //=====================================================================
262 // returns the absolute real address of the load area
263 static inline u64               HvLpConfig_getLoadAddress(void)
264 {
265         return itLpNaca.xLoadAreaAddr & 0x7fffffffffffff;
266 }
```

```
267 //=====================================================================
268 static inline u64               HvLpConfig_getLoadPages(void)
269 {
270         return itLpNaca.xLoadAreaChunks * HVPAGESPERCHUNK;
271 }
272 //=====================================================================
273 static inline int               HvLpConfig_isBusOwnedByThisLp(HvBusNumber busNumber)
274 {
275         HvLpIndex busOwner = HvLpConfig_getBusOwner(busNumber);
276         return (busOwner == HvLpConfig_getLpIndex());
277 }
278 //=====================================================================
279 static inline int         HvLpConfig_doLpsCommunicateOnVirtualLan(HvLpIndex lp1, HvLpIndex lp2)
280 {
281         HvLpVirtualLanIndexMap virtualLanIndexMap1 = HvCallCfg_getVirtualLanIndexMap( lp1 );
282         HvLpVirtualLanIndexMap virtualLanIndexMap2 = HvCallCfg_getVirtualLanIndexMap( lp2 );
283         return ((virtualLanIndexMap1 & virtualLanIndexMap2) != 0);
284 }
285 //=====================================================================
286 static inline HvLpIndex         HvLpConfig_getHostingLpIndex(HvLpIndex lp)
287 {
288         return HvCallCfg_getHostingLpIndex(lp);
289 }
290 //=====================================================================
291
292 #endif // _HVLPCONFIG_H
```

```
/*
 * iSeries_dma.h
 * Copyright (C) 2001  Mike Corrigan IBM Corporation
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#ifndef _ISERIES_DMA_H
#define _ISERIES_DMA_H

#include <asm/types.h>
#ifndef __LINUX_SPINLOCK_H
#include <linux/spinlock.h>
#endif

// NUM_TCE_LEVELS defines the largest contiguous block
// of dma (tce) space we can get.  NUM_TCE_LEVELS = 10
// allows up to 2**9 pages (512 * 4096) = 2 MB
#define NUM_TCE_LEVELS 10

#define NO_TCE ((dma_addr_t)-1)

// Tces come in two formats, one for the virtual bus and a different
// format for PCI
#define TCE_VB  0
#define TCE_PCI 1


union Tce {
        u64     wholeTce;
        struct {
                u64     cacheBits       :6;     /* Cache hash bits - not used */
                u64     rsvd            :6;
                u64     rpn             :40;    /* Absolute page number */
                u64     valid           :1;     /* Tce is valid (vb only) */
                u64     allIo           :1;     /* Tce is valid for all lps (vb only) */
                u64     lpIndex         :8;     /* LpIndex for user of TCE (vb only) */
                u64     pciWrite        :1;     /* Write allowed (pci only) */
                u64     readWrite       :1;     /* Read allowed (pci), Write allowed
                                                   (vb) */
        } tceBits;
};

struct Bitmap {
        unsigned long   numBits;
        unsigned long   numBytes;
        unsigned char * map;
};

struct MultiLevelBitmap {
        unsigned long   maxLevel;
        struct Bitmap   level[NUM_TCE_LEVELS];
};

struct TceTable {
        u64     busNumber;
        u64     size;
        u64     startOffset;
        u64     index;
        spinlock_t      lock;
        struct MultiLevelBitmap mlbm;
};

struct HvTceTableManagerCB {
        u64     busNumber;              /* Bus number for this tce table */
        u64     start;                  /* Will be NULL for secondary */
        u64     totalSize;              /* Size (in pages) of whole table */
        u64     startOffset;            /* Index into real tce table of the
                                           start of our section */
        u64     size;                   /* Size (in pages) of our section */
        u64     index;                  /* Index of this tce table (token?) */
        u16     maxTceTableIndex;       /* Max number of tables for partition */
        u8      virtualBusFlag;         /* Flag to indicate virtual bus */
        u8      rsvd[5];
};

extern struct TceTable virtBusTceTable; /* Tce table for virtual bus */

```

```
91  extern struct TceTable * build_tce_table( struct HvTceTableManagerCB *,
92                                            struct TceTable *);
93  extern void              create_virtual_bus_tce_table( void );
94
95  extern void              create_pci_bus_tce_table( unsigned busNumber );
96
97  #endif // _ISERIES_DMA_H
```

```
#ifndef _ISERIES_FLIGHTRECORDER_H
#define _ISERIES_FLIGHTRECORDER_H
/*****************************************************************************/
/* File iSeries_FlightRecorder.h created by Allan Trautman Jan 22 2001. */
/*****************************************************************************/
/* This code supports the pci interface on the IBM iSeries systems.     */
/* Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                     */
/*                                                                       */
/* 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 program is distributed in the hope that it will be useful,       */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of        */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         */
/* GNU General Public License for more details.                          */
/*                                                                       */
/* 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.,                                       */
/* 59 Temple Place, Suite 330,                                           */
/* Boston, MA  02111-1307  USA                                           */
/*****************************************************************************/
/* Change Activity:                                                      */
/*   Created, Jan 22, 2001                                               */
/*   Added Time stamp methods. Apr 12, 2001                              */
/* End Change Activity                                                   */
/*****************************************************************************/
/* This is a generic Flight Recorder, simply stuffs line entries into a */
/* buffer for debug purposes.                                            */
/*                                                                       */
/* To use,                                                               */
/* 1. Create one, make it global so it isn't on the stack.               */
/*     FlightRecorder  PciFlightRecorder;                                */
/*                                                                       */
/* 2. Optionally create a pointer to it, just makes it easier to use.    */
/*     FlightRecorder* PciFr = &PciFlightRecorder;                       */
/*                                                                       */
/* 3. Initialize with you signature.                                     */
/*          iSeries_Fr_Initialize(PciFr, "Pci Flight Recorder");         */
/*                                                                       */
/* 4. Log entries.                                                       */
/*          PciFr->logEntry(PciFr,"In Main");                            */
/*                                                                       */
/* 5. Later, you can find the Flight Recorder by looking in the          */
/*    System.map                                                         */
/*****************************************************************************/
struct iSeries_FlightRecorder;          /* Forward declares              */
struct rtc_time;
void   logEntry(struct iSeries_FlightRecorder*, char* Text);
void   logTime( struct iSeries_FlightRecorder*, char* Text);
void   logDate( struct iSeries_FlightRecorder*, char* Text);
#define FlightRecorderSize 4096

/*****************************************************************************/
/* Generic Flight Recorder Structure                                         */
/*****************************************************************************/
struct iSeries_FlightRecorder {         /* Structure Defination          */
        char  Signature[16];               /* Eye Catcher                 */
        char* StartingPointer;             /* Buffer Starting Address     */
        char* CurrentPointer;              /* Next Entry Address          */
        int   WrapCount;                   /* Number of Buffer Wraps      */
        void  (*logEntry)(struct iSeries_FlightRecorder*,char*);
        void  (*logTime) (struct iSeries_FlightRecorder*,char*);
        void  (*logDate) (struct iSeries_FlightRecorder*,char*);
        char  Buffer[FlightRecorderSize];
};

typedef struct iSeries_FlightRecorder FlightRecorder;   /* Short Name   */
extern void iSeries_Fr_Initialize(FlightRecorder*, char* Signature);
/*****************************************************************************/
/* extern void iSeries_LogFr_Entry(  FlightRecorder*, char* Text);       */
/* extern void iSeries_LogFr_Date(   FlightRecorder*, char* Text);       */
/* extern void iSeries_LogFr_Time(   FlightRecorder*, char* Text);       */
/*****************************************************************************/
/* PCI Flight Recorder Helpers                                               */
/*****************************************************************************/
extern FlightRecorder* PciFr;            /* Ptr to Pci Fr                */
extern char*           PciFrBuffer;      /* Ptr to Fr Work Buffer        */
#define ISERIES_PCI_FR(buffer)      PciFr->logEntry(PciFr,buffer);
#define ISERIES_PCI_FR_TIME(buffer) PciFr->logTime(PciFr,buffer);
#define ISERIES_PCI_FR_DATE(buffer) PciFr->logDate(PciFr,buffer);

#endif /* _ISERIES_FLIGHTRECORDER_H */
```

```
#ifndef _ISERIES_64_PCI_H
#define _ISERIES_64_PCI_H
/*****************************************************************************/
/* File iSeries_pci.h created by Allan Trautman on Tue Feb 20, 2001.         */
/*****************************************************************************/
/* Define some useful macros for the iSeries pci routines.                   */
/* Copyright (C) 20yy  Allan H Trautman, IBM Corporation                     */
/*                                                                           */
/* 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 program is distributed in the hope that it will be useful,           */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of            */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the             */
/* GNU General Public License for more details.                              */
/*                                                                           */
/* 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.,                                           */
/* 59 Temple Place, Suite 330,                                               */
/* Boston, MA  02111-1307  USA                                               */
/*****************************************************************************/
/* Change Activity:                                                          */
/*   Created Feb 20, 2001                                                    */
/*   Added device reset, March 22, 2001                                      */
/*   Ported to ppc64, May 25, 2001                                           */
/* End Change Activity                                                       */
/*****************************************************************************/
#include <asm/iSeries/HvCallPci.h>

struct pci_dev;                         /* For Forward Reference             */
struct iSeries_Device_Node;
/*****************************************************************************/
/* Gets iSeries Bus, SubBus, of DevFn using pci_dev* structure               */
/*****************************************************************************/
#define ISERIES_BUS(DevPtr)    DevPtr->DsaAddr.busNumber
#define ISERIES_SUBBUS(DevPtr) DevPtr->DsaAddr.subBusNumber
#define ISERIES_DEVICE(DevPtr) DevPtr->DsaAddr.deviceId
#define ISERIES_DEVFUN(DevPtr) DevPtr->DevFn
#define ISERIES_DSA(DevPtr)   (*(u64*)&DevPtr->DsaAddr)
#define ISERIES_DEVNODE(PciDev) ((struct iSeries_Device_Node*)PciDev->sysdata)

#define EADsMaxAgents 7
/*****************************************************************************/
/* Decodes Linux DevFn to iSeries DevFn, bridge device, or function.         */
/* For Linux, see PCI_SLOT and PCI_FUNC in include/linux/pci.h               */
/*****************************************************************************/
#define ISERIES_DECODE_DEVFN(linuxdevfn)  (((linuxdevfn & 0x71) << 1) | (linuxdevfn & 0x07))
#define ISERIES_DECODE_DEVICE(linuxdevfn) (((linuxdevfn & 0x38) >> 3) |(((linuxdevfn & 0x40) >> 2) + 0x10))
#define ISERIES_DECODE_FUNCTION(linuxdevfn) (linuxdevfn & 0x07)
#define ISERIES_PCI_AGENTID(idsel,func) ((idsel & 0x0F) << 4) | (func  & 0x07)

#define ISERIES_GET_DEVICE_FROM_SUBBUS(subbus)   ((subbus >> 5) & 0x7)
#define ISERIES_GET_FUNCTION_FROM_SUBBUS(subbus) ((subbus >> 2) & 0x7)

#define ISERIES_ENCODE_DEVICE(agentid)  ((0x10) | ((agentid&0x20)>>2) | (agentid&07))
/*****************************************************************************/
/* Converts Virtual Address to Real Address for Hypervisor calls             */
/*****************************************************************************/
#define REALADDR(virtaddr)  (0x8000000000000000 | (virt_to_absolute((u64)virtaddr) ))

/*****************************************************************************/
/* Define TRUE and FALSE Values for Al                                       */
/*****************************************************************************/
#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

/*****************************************************************************/
/* iSeries Device Information                                                */
/*****************************************************************************/
struct iSeries_Device_Node {
        struct list_head Device_List;    /* Must be first for cast to wo*/
        struct pci_dev*  PciDev;         /* Pointer to pci_dev structure*/
        struct HvCallPci_DsaAddr DsaAddr;/* Direct Select Address       */
                                         /* busNumber,subBusNumber,     */
                                         /* deviceId, barNumber         */
        HvAgentId        AgentId;        /* Hypervisor DevFn            */
        int              DevFn;          /* Linux devfn                 */
        int              BarOffset;
        int              Irq;            /* Assigned IRQ                */
        int              ReturnCode;     /* Return Code Holder          */
        int              IoRetry;        /* Current Retry Count         */
        int              Flags;          /* Possible flags(disable/bist)*/
        u16              Vendor;         /* Vendor ID                   */
```

```
91          u8              LogicalSlot;    /* Hv Slot Index for Tces      */
92          struct TceTable* DevTceTable;   /* Device TCE Table            */
93          spinlock_t      IoLock;         /* Lock to single thread device*/
94          u8              PhbId;          /* Phb Card is on.             */
95          u16             Board;          /* Board Number                */
96          u8              FrameId;        /* iSeries spcn Frame Id       */
97          char            CardLocation[4];/* Char format of planar vpd   */
98          char            Location[20];   /* Frame  1, Card C10          */
99  };
100 /*********************************************************************/
101 /* Location Data extracted from the VPD list and device info.        */
102 /*********************************************************************/
103 struct LocationDataStruct {     /* Location data structure for device  */
104         u16  Bus;               /* iSeries Bus Number              0x00*/
105         u16  Board;             /* iSeries Board                   0x02*/
106         u8   FrameId;           /* iSeries spcn Frame Id           0x04*/
107         u8   PhbId;             /* iSeries Phb Location            0x05*/
108         u8   AgentId;           /* iSeries AgentId                 0x06*/
109         u8   Card;
110         char CardLocation[4];
111 };
112 typedef struct LocationDataStruct  LocationData;
113 #define LOCATION_DATA_SIZE      48
114 /*********************************************************************/
115 /* Flight Recorder tracing                                           */
116 /*********************************************************************/
117 extern int  iSeries_Set_PciTraceFlag(int TraceFlag);
118 extern int  iSeries_Get_PciTraceFlag(void);
119
120 /*********************************************************************/
121 /* Functions                                                         */
122 /*********************************************************************/
123 extern LocationData* iSeries_GetLocationData(struct pci_dev* PciDev);
124 extern int           iSeries_Device_Information(struct pci_dev*,char*, int);
125 extern void          iSeries_Get_Location_Code(struct iSeries_Device_Node*);
126 extern int           iSeries_Device_ToggleReset(struct pci_dev* PciDev, int AssertTime, int DelayTime);
127
128 #endif /* _ISERIES_64_PCI_H */
```

```
#ifndef _ISERIES_VPDINFO_H
#define _ISERIES_VPDINFO_H
/*********************************************************************/
/* File iSeries_VpdInfo.h created by Allan Trautman Feb 08 2001.     */
/*********************************************************************/
/* This code supports the location data fon on the IBM iSeries systems. */
/* Copyright (C) 20yy  <Allan H Trautman> <IBM Corp>                 */
/*                                                                   */
/* 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 program is distributed in the hope that it will be useful,   */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of    */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the     */
/* GNU General Public License for more details.                      */
/*                                                                   */
/* 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.,                                   */
/* 59 Temple Place, Suite 330,                                       */
/* Boston, MA  02111-1307  USA                                       */
/*********************************************************************/
/* Change Activity:                                                  */
/*   Created, Feg  8, 2001                                           */
/*   Reformated for Card, March 8, 2001                              */
/* End Change Activity                                               */
/*********************************************************************/

struct pci_dev;                 /* Forward Declare                   */
/*********************************************************************/
/* Location Data extracted from the VPD list and device info.        */
/*********************************************************************/
struct LocationDataStruct {     /* Location data structure for device   */
        u16  Bus;               /* iSeries Bus Number               0x00*/
        u16  Board;             /* iSeries Board                    0x02*/
        u8   FrameId;           /* iSeries spcn Frame Id            0x04*/
        u8   PhbId;             /* iSeries Phb Location             0x05*/
        u16  Card;              /* iSeries Card Slot                0x06*/
        char CardLocation[4];   /* Char format of planar vpd        0x08*/
        u8   AgentId;           /* iSeries AgentId                  0x0C*/
        u8   SecondaryAgentId;  /* iSeries Secondary Agent Id       0x0D*/
        u8   LinuxBus;          /* Linux Bus Number                 0x0E*/
        u8   LinuxDevFn;        /* Linux Device Function            0x0F*/
};
typedef struct LocationDataStruct  LocationData;
#define LOCATION_DATA_SIZE      16

/*********************************************************************/
/* Protypes                                                          */
/*********************************************************************/
extern LocationData* iSeries_GetLocationData(struct pci_dev* PciDev);
extern int           iSeries_Device_Information(struct pci_dev*,char*, int);

#endif /* _ISERIES_VPDINFO_H */
```

```
/*
 * mf.h
 * Copyright (C) 2001  Troy D. Armstrong IBM Corporation
 *
 * This modules exists as an interface between a Linux secondary partition
 * running on an iSeries and the primary partition's Virtual Service
 * Processor (VSP) object.  The VSP has final authority over powering on/off
 * all partitions in the iSeries.  It also provides miscellaneous low-level
 * machine facility type operations.
 *
 *
 * 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 program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#ifndef MF_H_INCLUDED
#define MF_H_INCLUDED

#include <asm/iSeries/HvTypes.h>
#include <asm/iSeries/HvLpEvent.h>

struct rtc_time;

typedef void (*MFCompleteHandler)( void * clientToken, int returnCode );

extern void mf_allocateLpEvents( HvLpIndex targetLp,
                                 HvLpEvent_Type type,
                                 unsigned size,
                                 unsigned amount,
                                 MFCompleteHandler hdlr,
                                 void * userToken );

extern void mf_deallocateLpEvents( HvLpIndex targetLp,
                                   HvLpEvent_Type type,
                                   unsigned count,
                                   MFCompleteHandler hdlr,
                                   void * userToken );

extern void mf_powerOff( void );

extern void mf_reboot( void );

extern void mf_displaySrc( u32 word );
extern void mf_displayProgress( u16 value );

extern void mf_clearSrc( void );

extern void mf_init( void );

extern void mf_setSide(char side);

extern char mf_getSide(void);

extern void mf_setCmdLine(const char *cmdline, int size, u64 side);

extern int  mf_getCmdLine(char *cmdline, int *size, u64 side);

extern void mf_getSrcHistory(char *buffer, int size);

extern int mf_setVmlinuxChunk(const char *buffer, int size, int offset, u64 side);

extern int mf_getVmlinuxChunk(char *buffer, int *size, int offset, u64 side);

extern int mf_setRtcTime(unsigned long time);

extern int mf_getRtcTime(unsigned long *time);

extern int mf_getRtc( struct rtc_time * tm );

extern int mf_setRtc( struct rtc_time * tm );

#endif /* MF_H_INCLUDED */
```

Table of Content

End of Table of Contents