101

```
/*
 * IBM Hot Plug Controller Driver
 *
 * Written By: Jyoti Shah, IBM Corporation
 *
 * Copyright (c) 2001−2002 IBM Corp.
 *
 * All rights reserved.
 *
 * 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, GOOD TITLE or
 * NON INFRINGEMENT.  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., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 * Send feedback to <gregkh@us.ibm.com>
 *                  <jshah@us.ibm.com>
 *
 */

#include <linux/wait.h>
#include <linux/time.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/smp_lock.h>
#include "ibmphp.h"

static int to_debug = FALSE;
#define debug_polling(fmt, arg...)      do { if (to_debug) debug (fmt, arg); } while (0)

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
// timeout values
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define CMD_COMPLETE_TOUT_SEC   60      // give HPC 60 sec to finish cmd
#define HPC_CTLR_WORKING_TOUT   60      // give HPC 60 sec to finish cmd
#define HPC_GETACCESS_TIMEOUT   60      // seconds
#define POLL_INTERVAL_SEC       2       // poll HPC every 2 seconds
#define POLL_LATCH_CNT          5       // poll latch 5 times, then poll slots

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
// Winnipeg Architected Register Offsets
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define WPG_I2CMBUFL_OFFSET     0x08    // I2C Message Buffer Low
#define WPG_I2CMOSUP_OFFSET     0x10    // I2C Master Operation Setup Reg
#define WPG_I2CMCNTL_OFFSET     0x20    // I2C Master Control Register
#define WPG_I2CPARM_OFFSET      0x40    // I2C Parameter Register
#define WPG_I2CSTAT_OFFSET      0x70    // I2C Status Register

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
// Winnipeg Store Type commands (Add this commands to the register offset)
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define WPG_I2C_AND             0x1000  // I2C AND operation
#define WPG_I2C_OR              0x2000  // I2C OR operation

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
// Command set for I2C Master Operation Setup Regisetr
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define WPG_READATADDR_MASK     0x00010000      // read,bytes,I2C shifted,index
#define WPG_WRITEATADDR_MASK    0x40010000      // write,bytes,I2C shifted,index
#define WPG_READDIRECT_MASK     0x10010000
#define WPG_WRITEDIRECT_MASK    0x60010000


//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
// bit masks for I2C Master Control Register
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define WPG_I2CMCNTL_STARTOP_MASK       0x00000002      // Start the Operation

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define WPG_I2C_IOREMAP_SIZE    0x2044  // size of linear address interval

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
// command index
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
#define WPG_1ST_SLOT_INDEX      0x01    // index − 1st slot for ctlr
#define WPG_CTLR_INDEX          0x0F    // index − ctlr
#define WPG_1ST_EXTSLOT_INDEX   0x10    // index − 1st ext slot for ctlr
#define WPG_1ST_BUS_INDEX       0x1F    // index − 1st bus for ctlr

```

```
91  //-----------------------------------------------------------------------------
92  // macro utilities
93  //-----------------------------------------------------------------------------
94  // if bits 20,22,25,26,27,29,30 are OFF return TRUE
95  #define HPC_I2CSTATUS_CHECK(s)  ((u8)((s & 0x00000A76) ? FALSE : TRUE))
96
97  //-----------------------------------------------------------------------------
98  // global variables
99  //-----------------------------------------------------------------------------
100 static int ibmphp_shutdown;
101 static int tid_poll;
102 static struct semaphore sem_hpcaccess;  // lock access to HPC
103 static struct semaphore semOperations;  // lock all operations and
104                                         // access to data structures
105 static struct semaphore sem_exit;       // make sure polling thread goes away
106 //-----------------------------------------------------------------------------
107 // local function prototypes
108 //-----------------------------------------------------------------------------
109 static u8 ctrl_read (struct controller *, void *, u8);
110 static u8 ctrl_write (struct controller *, void *, u8, u8);
111 static u8 hpc_writecmdtoindex (u8, u8);
112 static u8 hpc_readcmdtoindex (u8, u8);
113 static void get_hpc_access (void);
114 static void free_hpc_access (void);
115 static void poll_hpc (void);
116 static int update_slot (struct slot *, u8);
117 static int process_changeinstatus (struct slot *, struct slot *);
118 static int process_changeinlatch (u8, u8, struct controller *);
119 static int hpc_poll_thread (void *);
120 static int hpc_wait_ctlr_notworking (int, struct controller *, void *, u8 *);
121 //-----------------------------------------------------------------------------
122
123
124 /*-----------------------------------------------------------------------
125 * Name:    ibmphp_hpc_initvars
126 *
127 * Action:  initialize semaphores and variables
128 *-----------------------------------------------------------------------*/
129 void ibmphp_hpc_initvars (void)
130 {
131         debug ("%s – Entry\n", __FUNCTION__);
132
133         init_MUTEX (&sem_hpcaccess);
134         init_MUTEX (&semOperations);
135         init_MUTEX_LOCKED (&sem_exit);
136         to_debug = FALSE;
137         ibmphp_shutdown = FALSE;
138         tid_poll = 0;
139
140         debug ("%s – Exit\n", __FUNCTION__);
141 }
142
143 /*-----------------------------------------------------------------------
144 * Name:    ctrl_read
145 *
146 * Action:  read from HPC over I2C
147 *
148 *-----------------------------------------------------------------------*/
149 static u8 ctrl_read (struct controller *ctlr_ptr, void *WPGBbar, u8 index)
150 {
151         u8 status;
152         int i;
153         void *wpg_addr;         // base addr + offset
154         ulong wpg_data,         // data to/from WPG LOHI format
155         ultemp, data;           // actual data HILO format
156
157
158         debug_polling ("%s – Entry WPGBbar[%lx] index[%x] \n", __FUNCTION__, (ulong) WPGBbar, index);
159
160         //--------------------------------------------------------------------
161         // READ − step 1
162         // read at address, byte length, I2C address (shifted), index
163         // or read direct, byte length, index
164         if (ctlr_ptr->ctlr_type == 0x02) {
165                 data = WPG_READATADDR_MASK;
166                 // fill in I2C address
167                 ultemp = (ulong) ctlr_ptr->u.wpeg_ctlr.i2c_addr;
168                 ultemp = ultemp >> 1;
169                 data |= (ultemp << 8);
170
171                 // fill in index
172                 data |= (ulong) index;
173         } else if (ctlr_ptr->ctlr_type == 0x04) {
174                 data = WPG_READDIRECT_MASK;
175
176                 // fill in index
177                 ultemp = (ulong) index;
178                 ultemp = ultemp << 8;
179                 data |= ultemp;
180         } else {
```

```
181                     err ("this controller type is not supported \n");
182                     return HPC_ERROR;
183             }
184
185             wpg_data = swab32 (data);       // swap data before writing
186             (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMOSUP_OFFSET;
187             writel (wpg_data, wpg_addr);
188
189             //--------------------------------------------------------------------
190             // READ - step 2 : clear the message buffer
191             data = 0x00000000;
192             wpg_data = swab32 (data);
193             (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMBUFL_OFFSET;
194             writel (wpg_data, wpg_addr);
195
196             //--------------------------------------------------------------------
197             // READ - step 3 : issue start operation, I2C master control bit 30:ON
198             //                 2020 : [20] OR operation at [20] offset 0x20
199             data = WPG_I2CMCNTL_STARTOP_MASK;
200             wpg_data = swab32 (data);
201             (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMCNTL_OFFSET + (ulong) WPG_I2C_OR;
202             writel (wpg_data, wpg_addr);
203
204             //--------------------------------------------------------------------
205             // READ - step 4 : wait until start operation bit clears
206             i = CMD_COMPLETE_TOUT_SEC;
207             while (i) {
208                     long_delay (1 * HZ / 100);
209                     (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMCNTL_OFFSET;
210                     wpg_data = readl (wpg_addr);
211                     data = swab32 (wpg_data);
212                     if (!(data & WPG_I2CMCNTL_STARTOP_MASK))
213                             break;
214                     i--;
215             }
216             if (i == 0) {
217                     debug ("%s - Error : WPG timeout\n", __FUNCTION__);
218                     return HPC_ERROR;
219             }
220             //--------------------------------------------------------------------
221             // READ - step 5 : read I2C status register
222             i = CMD_COMPLETE_TOUT_SEC;
223             while (i) {
224                     long_delay (1 * HZ / 100);
225                     (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CSTAT_OFFSET;
226                     wpg_data = readl (wpg_addr);
227                     data = swab32 (wpg_data);
228                     if (HPC_I2CSTATUS_CHECK (data))
229                             break;
230                     i--;
231             }
232             if (i == 0) {
233                     debug ("ctrl_read - Exit Error:I2C timeout\n");
234                     return HPC_ERROR;
235             }
236
237             //--------------------------------------------------------------------
238             // READ - step 6 : get DATA
239             (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMBUFL_OFFSET;
240             wpg_data = readl (wpg_addr);
241             data = swab32 (wpg_data);
242
243             status = (u8) data;
244
245             debug_polling ("%s - Exit index[%x] status[%x]\n", __FUNCTION__, index, status);
246
247             return (status);
248     }
249
250     /*--------------------------------------------------------------------
251     * Name:    ctrl_write
252     *
253     * Action:  write to HPC over I2C
254     *
255     * Return   0 or error codes
256     *--------------------------------------------------------------------*/
257     static u8 ctrl_write (struct controller *ctlr_ptr, void *WPGBbar, u8 index, u8 cmd)
258     {
259             u8 rc;
260             void *wpg_addr;         // base addr + offset
261             ulong wpg_data,         // data to/from WPG LOHI format
262             ultemp, data;           // actual data HILO format
263             int i;
264
265
266             debug_polling ("%s - Entry WPGBbar[%lx] index[%x] cmd[%x]\n", __FUNCTION__, (ulong) WPGBbar, index, cmd);
267
268             rc = 0;
269             //--------------------------------------------------------------------
270             // WRITE - step 1
```

```
271         // write at address, byte length, I2C address (shifted), index
272         // or write direct, byte length, index
273         data = 0x00000000;
274
275         if (ctlr_ptr->ctlr_type == 0x02) {
276                 data = WPG_WRITEATADDR_MASK;
277                 // fill in I2C address
278                 ultemp = (ulong) ctlr_ptr->u.wpeg_ctlr.i2c_addr;
279                 ultemp = ultemp >> 1;
280                 data |= (ultemp << 8);
281
282                 // fill in index
283                 data |= (ulong) index;
284         } else if (ctlr_ptr->ctlr_type == 0x04) {
285                 data = WPG_WRITEDIRECT_MASK;
286
287                 // fill in index
288                 ultemp = (ulong) index;
289                 ultemp = ultemp << 8;
290                 data |= ultemp;
291         } else {
292                 err ("this controller type is not supported \n");
293                 return HPC_ERROR;
294         }
295
296         wpg_data = swab32 (data);       // swap data before writing
297         (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMOSUP_OFFSET;
298         writel (wpg_data, wpg_addr);
299
300         //--------------------------------------------------------------------
301         // WRITE - step 2 : clear the message buffer
302         data = 0x00000000 | (ulong) cmd;
303         wpg_data = swab32 (data);
304         (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMBUFL_OFFSET;
305         writel (wpg_data, wpg_addr);
306
307         //--------------------------------------------------------------------
308         // WRITE - step 3 : issue start operation,I2C master control bit 30:ON
309         //                  2020 : [20] OR operation at [20] offset 0x20
310         data = WPG_I2CMCNTL_STARTOP_MASK;
311         wpg_data = swab32 (data);
312         (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMCNTL_OFFSET + (ulong) WPG_I2C_OR;
313         writel (wpg_data, wpg_addr);
314
315         //--------------------------------------------------------------------
316         // WRITE - step 4 : wait until start operation bit clears
317         i = CMD_COMPLETE_TOUT_SEC;
318         while (i) {
319                 long_delay (1 * HZ / 100);
320                 (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CMCNTL_OFFSET;
321                 wpg_data = readl (wpg_addr);
322                 data = swab32 (wpg_data);
323                 if (!(data & WPG_I2CMCNTL_STARTOP_MASK))
324                         break;
325                 i--;
326         }
327         if (i == 0) {
328                 debug ("%s – Exit Error:WPG timeout\n", __FUNCTION__);
329                 rc = HPC_ERROR;
330         }
331
332         //--------------------------------------------------------------------
333         // WRITE - step 5 : read I2C status register
334         i = CMD_COMPLETE_TOUT_SEC;
335         while (i) {
336                 long_delay (1 * HZ / 100);
337                 (ulong) wpg_addr = (ulong) WPGBbar + (ulong) WPG_I2CSTAT_OFFSET;
338                 wpg_data = readl (wpg_addr);
339                 data = swab32 (wpg_data);
340                 if (HPC_I2CSTATUS_CHECK (data))
341                         break;
342                 i--;
343         }
344         if (i == 0) {
345                 debug ("ctrl_read – Error : I2C timeout\n");
346                 rc = HPC_ERROR;
347         }
348
349         debug_polling ("%s Exit rc[%x]\n", __FUNCTION__, rc);
350         return (rc);
351 }
352
353 /*--------------------------------------------------------------------
354 * Name:    hpc_writecmdtoindex()
355 *
356 * Action:  convert a write command to proper index within a controller
357 *
358 * Return   index, HPC_ERROR
359 *--------------------------------------------------------------------*/
360 static u8 hpc_writecmdtoindex (u8 cmd, u8 index)
```

```
{
        u8 rc;

        switch (cmd) {
        case HPC_CTLR_ENABLEIRQ:        // 0x00.N.15
        case HPC_CTLR_CLEARIRQ: // 0x06.N.15
        case HPC_CTLR_RESET:    // 0x07.N.15
        case HPC_CTLR_IRQSTEER: // 0x08.N.15
        case HPC_CTLR_DISABLEIRQ:       // 0x01.N.15
        case HPC_ALLSLOT_ON:    // 0x11.N.15
        case HPC_ALLSLOT_OFF:   // 0x12.N.15
                rc = 0x0F;
                break;

        case HPC_SLOT_OFF:      // 0x02.Y.0−14
        case HPC_SLOT_ON:       // 0x03.Y.0−14
        case HPC_SLOT_ATTNOFF:  // 0x04.N.0−14
        case HPC_SLOT_ATTNON:   // 0x05.N.0−14
        case HPC_SLOT_BLINKLED: // 0x13.N.0−14
                rc = index;
                break;

        case HPC_BUS_33CONVMODE:
        case HPC_BUS_66CONVMODE:
        case HPC_BUS_66PCIXMODE:
        case HPC_BUS_100PCIXMODE:
        case HPC_BUS_133PCIXMODE:
                rc = index + WPG_1ST_BUS_INDEX − 1;
                break;

        default:
                err ("hpc_writecmdtoindex − Error invalid cmd[%x]\n", cmd);
                rc = HPC_ERROR;
        }

        return rc;
}

/*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
* Name:    hpc_readcmdtoindex()
*
* Action:  convert a read command to proper index within a controller
*
* Return   index, HPC_ERROR
*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−*/
static u8 hpc_readcmdtoindex (u8 cmd, u8 index)
{
        u8 rc;

        switch (cmd) {
        case READ_CTLRSTATUS:
                rc = 0x0F;
                break;
        case READ_SLOTSTATUS:
        case READ_ALLSTAT:
                rc = index;
                break;
        case READ_EXTSLOTSTATUS:
                rc = index + WPG_1ST_EXTSLOT_INDEX;
                break;
        case READ_BUSSTATUS:
                rc = index + WPG_1ST_BUS_INDEX − 1;
                break;
        case READ_SLOTLATCHLOWREG:
                rc = 0x28;
                break;
        case READ_REVLEVEL:
                rc = 0x25;
                break;
        case READ_HPCOPTIONS:
                rc = 0x27;
                break;
        default:
                rc = HPC_ERROR;
        }
        return rc;
}

/*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
* Name:    HPCreadslot()
*
* Action:  issue a READ command to HPC
*
* Input:   pslot   − can not be NULL for READ_ALLSTAT
*          pstatus − can be NULL for READ_ALLSTAT
*
* Return   0 or error codes
*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−*/
int ibmphp_hpc_readslot (struct slot * pslot, u8 cmd, u8 * pstatus)
{
```

```
451         void *wpg_bbar;
452         struct controller *ctlr_ptr;
453         struct list_head *pslotlist;
454         u8 index, status;
455         int rc = 0;
456         int busindex;
457
458         debug_polling ("%s – Entry pslot[%lx] cmd[%x] pstatus[%lx]\n", __FUNCTION__, (ulong) pslot, cmd, (ulong) pstatus);
459
460         if ((pslot == NULL)
461             || ((pstatus == NULL) && (cmd != READ_ALLSTAT) && (cmd != READ_BUSSTATUS))) {
462                 rc = −EINVAL;
463                 err ("%s – Error invalid pointer, rc[%d]\n", __FUNCTION__, rc);
464                 return rc;
465         }
466
467         if (cmd == READ_BUSSTATUS) {
468                 busindex = ibmphp_get_bus_index (pslot->bus);
469                 if (busindex < 0) {
470                         rc = −EINVAL;
471                         err ("%s – Exit Error:invalid bus, rc[%d]\n", __FUNCTION__, rc);
472                         return rc;
473                 } else
474                         index = (u8) busindex;
475         } else
476                 index = pslot->ctlr_index;
477
478         index = hpc_readcmdtoindex (cmd, index);
479
480         if (index == HPC_ERROR) {
481                 rc = −EINVAL;
482                 err ("%s – Exit Error:invalid index, rc[%d]\n", __FUNCTION__, rc);
483                 return rc;
484         }
485
486         ctlr_ptr = pslot->ctrl;
487
488         get_hpc_access ();
489
490         //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
491         // map physical address to logical address
492         //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
493         wpg_bbar = ioremap (ctlr_ptr->u.wpeg_ctlr.wpegbbar, WPG_I2C_IOREMAP_SIZE);
494
495         //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
496         // check controller status before reading
497         //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
498         rc = hpc_wait_ctlr_notworking (HPC_CTLR_WORKING_TOUT, ctlr_ptr, wpg_bbar, &status);
499         if (!rc) {
500                 switch (cmd) {
501                 case READ_ALLSTAT:
502                         // update the slot structure
503                         pslot->ctrl->status = status;
504                         pslot->status = ctrl_read (ctlr_ptr, wpg_bbar, index);
505                         rc = hpc_wait_ctlr_notworking (HPC_CTLR_WORKING_TOUT, ctlr_ptr, wpg_bbar,
506                                                        &status);
507                         if (!rc)
508                                 pslot->ext_status = ctrl_read (ctlr_ptr, wpg_bbar, index + WPG_1ST_EXTSLOT_INDEX)
    ;
509
510                         break;
511
512                 case READ_SLOTSTATUS:
513                         // DO NOT update the slot structure
514                         *pstatus = ctrl_read (ctlr_ptr, wpg_bbar, index);
515                         break;
516
517                 case READ_EXTSLOTSTATUS:
518                         // DO NOT update the slot structure
519                         *pstatus = ctrl_read (ctlr_ptr, wpg_bbar, index);
520                         break;
521
522                 case READ_CTLRSTATUS:
523                         // DO NOT update the slot structure
524                         *pstatus = status;
525                         break;
526
527                 case READ_BUSSTATUS:
528                         pslot->busstatus = ctrl_read (ctlr_ptr, wpg_bbar, index);
529                         break;
530                 case READ_REVLEVEL:
531                         *pstatus = ctrl_read (ctlr_ptr, wpg_bbar, index);
532                         break;
533                 case READ_HPCOPTIONS:
534                         *pstatus = ctrl_read (ctlr_ptr, wpg_bbar, index);
535                         break;
536                 case READ_SLOTLATCHLOWREG:
537                         // DO NOT update the slot structure
538                         *pstatus = ctrl_read (ctlr_ptr, wpg_bbar, index);
539                         break;
```

```
540
541                     // Not used
542             case READ_ALLSLOT:
543                     list_for_each (pslotlist, &ibmphp_slot_head) {
544                             pslot = list_entry (pslotlist, struct slot, ibm_slot_list);
545                             index = pslot->ctlr_index;
546                             rc = hpc_wait_ctlr_notworking (HPC_CTLR_WORKING_TOUT, ctlr_ptr,
547                                                            wpg_bbar, &status);
548                             if (!rc) {
549                                     pslot->status = ctrl_read (ctlr_ptr, wpg_bbar, index);
550                                     rc = hpc_wait_ctlr_notworking (HPC_CTLR_WORKING_TOUT,
551                                                                    ctlr_ptr, wpg_bbar, &status);
552                                     if (!rc)
553                                             pslot->ext_status =
554                                                 ctrl_read (ctlr_ptr, wpg_bbar,
555                                                            index + WPG_1ST_EXTSLOT_INDEX);
556                             } else {
557                                     err ("%s – Error ctrl_read failed\n", __FUNCTION__);
558                                     rc = -EINVAL;
559                                     break;
560                             }
561                     }
562                     break;
563             default:
564                     rc = -EINVAL;
565                     break;
566             }
567     }
568     //--------------------------------------------------------------------
569     // cleanup
570     //--------------------------------------------------------------------
571     iounmap (wpg_bbar);     // remove physical to logical address mapping
572     free_hpc_access ();
573
574     debug_polling ("%s – Exit rc[%d]\n", __FUNCTION__, rc);
575     return rc;
576 }
577
578 /*--------------------------------------------------------------------
579 * Name:    ibmphp_hpc_writeslot()
580 *
581 * Action: issue a WRITE command to HPC
582 *--------------------------------------------------------------------*/
583 int ibmphp_hpc_writeslot (struct slot * pslot, u8 cmd)
584 {
585     void *wpg_bbar;
586     struct controller *ctlr_ptr;
587     u8 index, status;
588     int busindex;
589     u8 done;
590     int rc = 0;
591     int timeout;
592
593     debug_polling ("%s – Entry pslot[%lx] cmd[%x]\n", __FUNCTION__, (ulong) pslot, cmd);
594     if (pslot == NULL) {
595             rc = -EINVAL;
596             err ("%s – Error Exit rc[%d]\n", __FUNCTION__, rc);
597             return rc;
598     }
599
600     if ((cmd == HPC_BUS_33CONVMODE) || (cmd == HPC_BUS_66CONVMODE) ||
601             (cmd == HPC_BUS_66PCIXMODE) || (cmd == HPC_BUS_100PCIXMODE) ||
602             (cmd == HPC_BUS_133PCIXMODE)) {
603             busindex = ibmphp_get_bus_index (pslot->bus);
604             if (busindex < 0) {
605                     rc = -EINVAL;
606                     err ("%s – Exit Error:invalid bus, rc[%d]\n", __FUNCTION__, rc);
607                     return rc;
608             } else
609                     index = (u8) busindex;
610     } else
611             index = pslot->ctlr_index;
612
613     index = hpc_writecmdtoindex (cmd, index);
614
615     if (index == HPC_ERROR) {
616             rc = -EINVAL;
617             err ("%s – Error Exit rc[%d]\n", __FUNCTION__, rc);
618             return rc;
619     }
620
621     ctlr_ptr = pslot->ctrl;
622
623     get_hpc_access ();
624
625     //--------------------------------------------------------------------
626     // map physical address to logical address
627     //--------------------------------------------------------------------
628     wpg_bbar = ioremap (ctlr_ptr->u.wpeg_ctlr.wpegbbar, WPG_I2C_IOREMAP_SIZE);
629
```

```
630         debug ("%s – ctlr id[%x] physical[%lx] logical[%lx] i2c[%x]\n", __FUNCTION__,
631                 ctlr_ptr->ctlr_id, (ulong) (ctlr_ptr->u.wpeg_ctlr.wpegbbar), (ulong) wpg_bbar,
632                 ctlr_ptr->u.wpeg_ctlr.i2c_addr);
633
634         //--------------------------------------------------------------------
635         // check controller status before writing
636         //--------------------------------------------------------------------
637         rc = hpc_wait_ctlr_notworking (HPC_CTLR_WORKING_TOUT, ctlr_ptr, wpg_bbar, &status);
638         if (!rc) {
639
640                 ctrl_write (ctlr_ptr, wpg_bbar, index, cmd);
641
642                 //--------------------------------------------------------------------
643                 // check controller is still not working on the command
644                 //--------------------------------------------------------------------
645                 timeout = CMD_COMPLETE_TOUT_SEC;
646                 done = FALSE;
647                 while (!done) {
648                         rc = hpc_wait_ctlr_notworking (HPC_CTLR_WORKING_TOUT, ctlr_ptr, wpg_bbar,
649                                                        &status);
650                         if (!rc) {
651                                 if (NEEDTOCHECK_CMDSTATUS (cmd)) {
652                                         if (CTLR_FINISHED (status) == HPC_CTLR_FINISHED_YES)
653                                                 done = TRUE;
654                                 } else
655                                         done = TRUE;
656                         }
657                         if (!done) {
658                                 long_delay (1 * HZ);
659                                 if (timeout < 1) {
660                                         done = TRUE;
661                                         err ("%s – Error command complete timeout\n", __FUNCTION__);
662                                         rc = -EFAULT;
663                                 } else
664                                         timeout--;
665                         }
666                 }
667                 ctlr_ptr->status = status;
668         }
669         // cleanup
670         iounmap (wpg_bbar);     // remove physical to logical address mapping
671         free_hpc_access ();
672
673         debug_polling ("%s – Exit rc[%d]\n", __FUNCTION__, rc);
674         return rc;
675 }
676
677 /*--------------------------------------------------------------------
678  * Name:    get_hpc_access()
679  *
680  * Action: make sure only one process can access HPC at one time
681  *--------------------------------------------------------------------*/
682 static void get_hpc_access (void)
683 {
684         down (&sem_hpcaccess);
685 }
686
687 /*--------------------------------------------------------------------
688  * Name:    free_hpc_access()
689  *--------------------------------------------------------------------*/
690 void free_hpc_access (void)
691 {
692         up (&sem_hpcaccess);
693 }
694
695 /*--------------------------------------------------------------------
696  * Name:    ibmphp_lock_operations()
697  *
698  * Action: make sure only one process can change the data structure
699  *--------------------------------------------------------------------*/
700 void ibmphp_lock_operations (void)
701 {
702         down (&semOperations);
703 }
704
705 /*--------------------------------------------------------------------
706  * Name:    ibmphp_unlock_operations()
707  *--------------------------------------------------------------------*/
708 void ibmphp_unlock_operations (void)
709 {
710         debug ("%s – Entry\n", __FUNCTION__);
711         up (&semOperations);
712         debug ("%s – Exit\n", __FUNCTION__);
713 }
714
715 /*--------------------------------------------------------------------
716  * Name:    poll_hpc()
717  *--------------------------------------------------------------------*/
718 #define POLL_LATCH_REGISTER     0
719 #define POLL_SLOTS              1
```

```
720 #define POLL_SLEEP              2
721 static void poll_hpc (void)
722 {
723         struct slot myslot;
724         struct slot *pslot = NULL;
725         struct list_head *pslotlist;
726         int rc;
727         int poll_state = POLL_LATCH_REGISTER;
728         u8 oldlatchlow = 0x00;
729         u8 curlatchlow = 0x00;
730         int poll_count = 0;
731         u8 ctrl_count = 0x00;
732
733         debug ("%s – Entry\n", __FUNCTION__);
734
735         while (!ibmphp_shutdown) {
736                 /* try to get the lock to do some kind of harware access */
737                 down (&semOperations);
738
739                 switch (poll_state) {
740                         case POLL_LATCH_REGISTER:
741                                 oldlatchlow = curlatchlow;
742                                 ctrl_count = 0x00;
743                                 list_for_each (pslotlist, &ibmphp_slot_head) {
744                                         if (ctrl_count >= ibmphp_get_total_controllers())
745                                                 break;
746                                         pslot = list_entry (pslotlist, struct slot, ibm_slot_list);
747                                         if (pslot->ctrl->ctlr_relative_id == ctrl_count) {
748                                                 ctrl_count++;
749                                                 if (READ_SLOT_LATCH (pslot->ctrl)) {
750                                                         rc = ibmphp_hpc_readslot (pslot,
751                                                                                   READ_SLOTLATCHLOWREG,
752                                                                                   &curlatchlow);
753                                                         if (oldlatchlow != curlatchlow)
754                                                                 process_changeinlatch (oldlatchlow,
755                                                                                        curlatchlow,
756                                                                                        pslot->ctrl);
757                                                 }
758                                         }
759                                 }
760                                 poll_state = POLL_SLOTS;
761                                 break;
762
763                         case POLL_SLOTS:
764                                 list_for_each (pslotlist, &ibmphp_slot_head) {
765                                         pslot = list_entry (pslotlist, struct slot, ibm_slot_list);
766                                         // make a copy of the old status
767                                         memcpy ((void *) &myslot, (void *) pslot,
768                                                 sizeof (struct slot));
769                                         rc = ibmphp_hpc_readslot (pslot, READ_ALLSTAT, NULL);
770                                         if ((myslot.status != pslot->status)
771                                             || (myslot.ext_status != pslot->ext_status))
772                                                 process_changeinstatus (pslot, &myslot);
773                                 }
774
775                                 ctrl_count = 0x00;
776                                 list_for_each (pslotlist, &ibmphp_slot_head) {
777                                         if (ctrl_count >= ibmphp_get_total_controllers())
778                                                 break;
779                                         pslot = list_entry (pslotlist, struct slot, ibm_slot_list);
780                                         if (pslot->ctrl->ctlr_relative_id == ctrl_count) {
781                                                 ctrl_count++;
782                                                 if (READ_SLOT_LATCH (pslot->ctrl))
783                                                         rc = ibmphp_hpc_readslot (pslot,
784                                                                                   READ_SLOTLATCHLOWREG,
785                                                                                   &curlatchlow);
786                                         }
787                                 }
788                                 ++poll_count;
789                                 if (poll_count >= POLL_LATCH_CNT) {
790                                         poll_count = 0;
791                                         poll_state = POLL_SLEEP;
792                                 }
793                                 break;
794
795                         case POLL_SLEEP:
796                                 /* don't sleep with a lock on the hardware */
797                                 up (&semOperations);
798                                 long_delay (POLL_INTERVAL_SEC * HZ);
799                                 down (&semOperations);
800                                 poll_state = POLL_LATCH_REGISTER;
801                                 break;
802                 }
803
804                 /* give up the harware semaphore */
805                 up (&semOperations);
806
807                 /* sleep for a short time just for good measure */
808                 set_current_state (TASK_INTERRUPTIBLE);
809                 schedule_timeout (HZ/10);
```

```
810         }
811
812         up (&sem_exit);
813         debug ("%s – Exit\n", __FUNCTION__);
814 }
815
816
817 /* ------------------------------------------------------------------------
818  *  Name:    ibmphp_hpc_fillhpslotinfo(hotplug_slot * phpslot)
819  *
820  *  Action:  fill out the hotplug_slot info
821  *
822  *  Input:   pointer to hotplug_slot
823  *
824  *  Return
825  *  Value:   0 or error codes
826  *------------------------------------------------------------------------*/
827 int ibmphp_hpc_fillhpslotinfo (struct hotplug_slot *phpslot)
828 {
829         int rc = 0;
830         struct slot *pslot;
831
832         if (phpslot && phpslot->private) {
833                 pslot = (struct slot *) phpslot->private;
834                 rc = update_slot (pslot, (u8) TRUE);
835                 if (!rc) {
836
837                         // power − enabled:1  not:0
838                         phpslot->info->power_status = SLOT_POWER (pslot->status);
839
840                         // attention − off:0, on:1, blinking:2
841                         phpslot->info->attention_status = SLOT_ATTN (pslot->status, pslot->ext_status);
842
843                         // latch − open:1 closed:0
844                         phpslot->info->latch_status = SLOT_LATCH (pslot->status);
845
846                         // pci board − present:1 not:0
847                         if (SLOT_PRESENT (pslot->status))
848                                 phpslot->info->adapter_status = 1;
849                         else
850                                 phpslot->info->adapter_status = 0;
851 /*
852                         if (pslot->bus_on->supported_bus_mode
853                                 && (pslot->bus_on->supported_speed == BUS_SPEED_66))
854                                 phpslot->info->max_bus_speed_status = BUS_SPEED_66PCIX;
855                         else
856                                 phpslot->info->max_bus_speed_status = pslot->bus_on->supported_speed;
857 */              } else
858                         rc = −EINVAL;
859         } else
860                 rc = −EINVAL;
861
862         return rc;
863 }
864
865 /*------------------------------------------------------------------------
866  * Name:    update_slot
867  *
868  * Action:  fill out slot status and extended status, controller status
869  *
870  * Input:   pointer to slot struct
871  *------------------------------------------------------------------------*/
872 static int update_slot (struct slot *pslot, u8 update)
873 {
874         int rc = 0;
875
876         debug ("%s – Entry pslot[%lx]\n", __FUNCTION__, (ulong) pslot);
877         rc = ibmphp_hpc_readslot (pslot, READ_ALLSTAT, NULL);
878         debug ("%s – Exit rc[%d]\n", __FUNCTION__, rc);
879         return rc;
880 }
881
882 /*------------------------------------------------------------------------
883  * Name:    process_changeinstatus
884  *
885  * Action:  compare old and new slot status, process the change in status
886  *
887  * Input:   pointer to slot struct, old slot struct
888  *
889  * Return   0 or error codes
890  * Value:
891  *
892  * Side
893  * Effects: None.
894  *
895  * Notes:
896  *------------------------------------------------------------------------*/
897 static int process_changeinstatus (struct slot *pslot, struct slot *poldslot)
898 {
899         u8 status;
```

```
900         int rc = 0;
901         u8 disable = FALSE;
902         u8 update = FALSE;
903
904         debug ("process_changeinstatus – Entry pslot[%lx], poldslot[%lx]\n", (ulong) pslot,
905                (ulong) poldslot);
906
907         // bit 0 − HPC_SLOT_POWER
908         if ((pslot->status & 0x01) != (poldslot->status & 0x01))
909                 update = TRUE;
910
911         // bit 1 − HPC_SLOT_CONNECT
912         // ignore
913
914         // bit 2 − HPC_SLOT_ATTN
915         if ((pslot->status & 0x04) != (poldslot->status & 0x04))
916                 update = TRUE;
917
918         // bit 3 − HPC_SLOT_PRSNT2
919         // bit 4 − HPC_SLOT_PRSNT1
920         if (((pslot->status & 0x08) != (poldslot->status & 0x08))
921                 || ((pslot->status & 0x10) != (poldslot->status & 0x10)))
922                 update = TRUE;
923
924         // bit 5 − HPC_SLOT_PWRGD
925         if ((pslot->status & 0x20) != (poldslot->status & 0x20))
926                 // OFF −> ON: ignore, ON −> OFF: disable slot
927                 if ((poldslot->status & 0x20) && (SLOT_CONNECT (poldslot->status) == HPC_SLOT_CONNECTED) && (SLOT
    _PRESENT (poldslot->status)))
928                         disable = TRUE;
929
930         // bit 6 − HPC_SLOT_BUS_SPEED
931         // ignore
932
933         // bit 7 − HPC_SLOT_LATCH
934         if ((pslot->status & 0x80) != (poldslot->status & 0x80)) {
935                 update = TRUE;
936                 // OPEN −> CLOSE
937                 if (pslot->status & 0x80) {
938                         if (SLOT_PWRGD (pslot->status)) {
939                                 // power goes on and off after closing latch
940                                 // check again to make sure power is still ON
941                                 long_delay (1 * HZ);
942                                 rc = ibmphp_hpc_readslot (pslot, READ_SLOTSTATUS, &status);
943                                 if (SLOT_PWRGD (status))
944                                         update = TRUE;
945                                 else    // overwrite power in pslot to OFF
946                                         pslot->status &= ~HPC_SLOT_POWER;
947                         }
948                 }
949                 // CLOSE −> OPEN
950                 else if ((SLOT_PWRGD (poldslot->status) == HPC_SLOT_PWRGD_GOOD)
951                         && (SLOT_CONNECT (poldslot->status) == HPC_SLOT_CONNECTED) && (SLOT_PRESENT (poldslot->st
    atus))) {
952                         disable = TRUE;
953                 }
954                 // else − ignore
955         }
956         // bit 4 − HPC_SLOT_BLINK_ATTN
957         if ((pslot->ext_status & 0x08) != (poldslot->ext_status & 0x08))
958                 update = TRUE;
959
960         if (disable) {
961                 debug ("process_changeinstatus – disable slot\n");
962                 pslot->flag = FALSE;
963                 rc = ibmphp_disable_slot (pslot->hotplug_slot);
964         }
965
966         if (update || disable) {
967                 ibmphp_update_slot_info (pslot);
968         }
969
970         debug ("%s – Exit rc[%d] disable[%x] update[%x]\n", __FUNCTION__, rc, disable, update);
971
972         return rc;
973 }
974
975 /*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
976  * Name:    process_changeinlatch
977  *
978  * Action:  compare old and new latch reg status, process the change
979  *
980  * Input:   old and current latch register status
981  *
982  * Return   0 or error codes
983  * Value:
984  *−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−*/
985 static int process_changeinlatch (u8 old, u8 new, struct controller *ctrl)
986 {
987         struct slot myslot, *pslot;
```

```
988             u8 i;
989             u8 mask;
990             int rc = 0;
991
992             debug ("%s – Entry old[%x], new[%x]\n", __FUNCTION__, old, new);
993             // bit 0 reserved, 0 is LSB, check bit 1-6 for 6 slots
994
995             for (i = ctrl->starting_slot_num; i <= ctrl->ending_slot_num; i++) {
996                     mask = 0x01 << i;
997                     if ((mask & old) != (mask & new)) {
998                             pslot = ibmphp_get_slot_from_physical_num (i);
999                             if (pslot) {
1000                                    memcpy ((void *) &myslot, (void *) pslot, sizeof (struct slot));
1001                                    rc = ibmphp_hpc_readslot (pslot, READ_ALLSTAT, NULL);
1002                                    debug ("%s – call process_changeinstatus for slot[%d]\n", __FUNCTION__, i);
1003                                    process_changeinstatus (pslot, &myslot);
1004                            } else {
1005                                    rc = -EINVAL;
1006                                    err ("%s – Error bad pointer for slot[%d]\n", __FUNCTION__, i);
1007                            }
1008                    }
1009            }
1010            debug ("%s – Exit rc[%d]\n", __FUNCTION__, rc);
1011            return rc;
1012    }
1013
1014    /*----------------------------------------------------------------------
1015    * Name:    hpc_poll_thread
1016    *
1017    * Action:  polling
1018    *
1019    * Return   0
1020    * Value:
1021    *----------------------------------------------------------------------*/
1022    static int hpc_poll_thread (void *data)
1023    {
1024            debug ("%s – Entry\n", __FUNCTION__);
1025            lock_kernel ();
1026            daemonize ();
1027            reparent_to_init ();
1028
1029            //  New name
1030            strcpy (current->comm, "hpc_poll");
1031
1032            unlock_kernel ();
1033
1034            poll_hpc ();
1035
1036            tid_poll = 0;
1037            debug ("%s – Exit\n", __FUNCTION__);
1038            return 0;
1039    }
1040
1041
1042    /*----------------------------------------------------------------------
1043    * Name:    ibmphp_hpc_start_poll_thread
1044    *
1045    * Action:  start polling thread
1046    *----------------------------------------------------------------------*/
1047    int ibmphp_hpc_start_poll_thread (void)
1048    {
1049            int rc = 0;
1050
1051            debug ("ibmphp_hpc_start_poll_thread – Entry\n");
1052
1053            tid_poll = kernel_thread (hpc_poll_thread, 0, 0);
1054            if (tid_poll < 0) {
1055                    err ("ibmphp_hpc_start_poll_thread – Error, thread not started\n");
1056                    rc = -1;
1057            }
1058
1059            debug ("ibmphp_hpc_start_poll_thread – Exit tid_poll[%d] rc[%d]\n", tid_poll, rc);
1060            return rc;
1061    }
1062
1063    /*----------------------------------------------------------------------
1064    * Name:    ibmphp_hpc_stop_poll_thread
1065    *
1066    * Action:  stop polling thread and cleanup
1067    *----------------------------------------------------------------------*/
1068    void ibmphp_hpc_stop_poll_thread (void)
1069    {
1070            debug ("ibmphp_hpc_stop_poll_thread – Entry\n");
1071
1072            ibmphp_shutdown = TRUE;
1073            ibmphp_lock_operations ();
1074
1075            // wait for poll thread to exit
1076            down (&sem_exit);
1077
```

```
1078          // cleanup
1079          free_hpc_access ();
1080          ibmphp_unlock_operations ();
1081          up (&sem_exit);
1082
1083          debug ("ibmphp_hpc_stop_poll_thread – Exit\n");
1084  }
1085
1086  /*----------------------------------------------------------------------
1087  * Name:    hpc_wait_ctlr_notworking
1088  *
1089  * Action:  wait until the controller is in a not working state
1090  *
1091  * Return   0, HPC_ERROR
1092  * Value:
1093  *--------------------------------------------------------------------*/
1094  static int hpc_wait_ctlr_notworking (int timeout, struct controller *ctlr_ptr, void *wpg_bbar,
1095                                       u8 * pstatus)
1096  {
1097          int rc = 0;
1098          u8 done = FALSE;
1099
1100          debug_polling ("hpc_wait_ctlr_notworking – Entry timeout[%d]\n", timeout);
1101
1102          while (!done) {
1103                  *pstatus = ctrl_read (ctlr_ptr, wpg_bbar, WPG_CTLR_INDEX);
1104                  if (*pstatus == HPC_ERROR) {
1105                          rc = HPC_ERROR;
1106                          done = TRUE;
1107                  }
1108                  if (CTLR_WORKING (*pstatus) == HPC_CTLR_WORKING_NO)
1109                          done = TRUE;
1110                  if (!done) {
1111                          long_delay (1 * HZ);
1112                          if (timeout < 1) {
1113                                  done = TRUE;
1114                                  err ("HPCreadslot – Error ctlr timeout\n");
1115                                  rc = HPC_ERROR;
1116                          } else
1117                                  timeout--;
1118                  }
1119          }
1120          debug_polling ("hpc_wait_ctlr_notworking – Exit rc[%x] status[%x]\n", rc, *pstatus);
1121          return rc;
1122  }
```

Table of Content