```
/*
 * BK Id: SCCS/s.xics.c 1.8 12/19/01 09:48:40 trini
 */
/*
 * arch/ppc/kernel/xics.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.
 */
#include <linux/config.h>
#include <linux/types.h>
#include <linux/threads.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <asm/prom.h>
#include <asm/io.h>
#include "i8259.h"
#include "xics.h"

void xics_enable_irq(u_int irq);
void xics_disable_irq(u_int irq);
void xics_mask_and_ack_irq(u_int irq);
void xics_end_irq(u_int irq);

struct hw_interrupt_type xics_pic = {
        " XICS   ",
        NULL,
        NULL,
        xics_enable_irq,
        xics_disable_irq,
        xics_mask_and_ack_irq,
        xics_end_irq
};

struct hw_interrupt_type xics_8259_pic = {
        " XICS/8259",
        NULL,
        NULL,
        NULL,
        NULL,
        xics_mask_and_ack_irq,
        NULL
};

#define XICS_IPI                2
#define XICS_IRQ_8259_CASCADE   0x2c
#define XICS_IRQ_OFFSET         16
#define XICS_IRQ_SPURIOUS       0

#define DEFAULT_SERVER          0
#define DEFAULT_PRIORITY        0

struct xics_ipl {
        union {
                u32     word;
                u8      bytes[4];
        } xirr_poll;
        union {
                u32 word;
                u8      bytes[4];
        } xirr;
        u32     dummy;
        union {
                u32     word;
                u8      bytes[4];
        } qirr;
};

struct xics_info {
        volatile struct xics_ipl *      per_cpu[NR_CPUS];
};

struct xics_info        xics_info;

#define xirr_info(n_cpu)        (xics_info.per_cpu[n_cpu]->xirr.word)
#define cppr_info(n_cpu)        (xics_info.per_cpu[n_cpu]->xirr.bytes[0])
#define poll_info(n_cpu)        (xics_info.per_cpu[n_cpu]->xirr_poll.word)
#define qirr_info(n_cpu)        (xics_info.per_cpu[n_cpu]->qirr.bytes[0])

void
xics_enable_irq(
        u_int   irq
        )
{
        int     status;
        int     call_status;
```

```
91
92          irq −= XICS_IRQ_OFFSET;
93          if (irq == XICS_IPI)
94                  return;
95          call_status = call_rtas("ibm,set−xive", 3, 1, (ulong*)&status,
96                                  irq, DEFAULT_SERVER, DEFAULT_PRIORITY);
97          if( call_status != 0 ) {
98                  printk("xics_enable_irq: irq=%x: call_rtas failed; retn=%x, status=%x\n",
99                         irq, call_status, status);
100                 return;
101         }
102 }
103
104 void
105 xics_disable_irq(
106         u_int   irq
107         )
108 {
109         int     status;
110         int     call_status;
111
112         irq −= XICS_IRQ_OFFSET;
113         call_status = call_rtas("ibm,int−off", 1, 1, (ulong*)&status, irq);
114         if( call_status != 0 ) {
115                 printk("xics_disable_irq: irq=%x: call_rtas failed, retn=%x\n",
116                        irq, call_status);
117                 return;
118         }
119 }
120
121 void
122 xics_end_irq(
123         u_int   irq
124         )
125 {
126         int cpu = smp_processor_id();
127
128         cppr_info(cpu) = 0; /* actually the value overwritten by ack */
129         xirr_info(cpu) = (0xff<<24) | (irq−XICS_IRQ_OFFSET);
130 }
131
132 void
133 xics_mask_and_ack_irq(
134         u_int   irq
135         )
136 {
137         int cpu = smp_processor_id();
138
139         if( irq < XICS_IRQ_OFFSET ) {
140                 i8259_pic.ack(irq);
141                 xirr_info(cpu) = (0xff<<24) | XICS_IRQ_8259_CASCADE;
142         }
143         else {
144                 cppr_info(cpu) = 0xff;
145         }
146 }
147
148 int
149 xics_get_irq(struct pt_regs *regs)
150 {
151         u_int   cpu = smp_processor_id();
152         u_int   vec;
153         int irq;
154
155         vec = xirr_info(cpu);
156         /*  (vec >> 24) == old priority */
157         vec &= 0x00ffffff;
158         /* for sanity, this had better be < NR_IRQS − 16 */
159         if( vec == XICS_IRQ_8259_CASCADE )
160                 irq = i8259_poll();
161         else if( vec == XICS_IRQ_SPURIOUS )
162                 irq = −1;
163         else
164                 irq = vec + XICS_IRQ_OFFSET;
165         return irq;
166 }
167
168 #ifdef CONFIG_SMP
169 void xics_ipi_action(int irq, void *dev_id, struct pt_regs *regs)
170 {
171         qirr_info(smp_processor_id()) = 0xff;
172         smp_message_recv(MSG_RESCHEDULE, regs);
173 }
174
175 void xics_cause_IPI(int cpu)
176 {
177         qirr_info(cpu) = 0;
178 }
179
180 void xics_setup_cpu(void)
```

```
{
        int cpu = smp_processor_id();

        cppr_info(cpu) = 0xff;
}
#endif /* CONFIG_SMP */

void
xics_init_IRQ( void )
{
        int i;
        extern unsigned long smp_chrp_cpu_nr;

#ifdef CONFIG_SMP
        for (i = 0; i < smp_chrp_cpu_nr; ++i)
                xics_info.per_cpu[i] =
                        ioremap(0xfe000000 + smp_hw_index[i] * 0x1000, 0x20);
#else
        xics_info.per_cpu[0] = ioremap(0xfe000000, 0x20);
#endif /* CONFIG_SMP */
        xics_8259_pic.enable = i8259_pic.enable;
        xics_8259_pic.disable = i8259_pic.disable;
        for (i = 0; i < 16; ++i)
                irq_desc[i].handler = &xics_8259_pic;
        for (; i < NR_IRQS; ++i)
                irq_desc[i].handler = &xics_pic;

        cppr_info(0) = 0xff;
        if (request_irq(XICS_IRQ_8259_CASCADE + XICS_IRQ_OFFSET, no_action,
                        0, "8259 cascade", 0))
                printk(KERN_ERR "xics_init_IRQ: couldn't get 8259 cascade\n");
        i8259_init();

#ifdef CONFIG_SMP
        request_irq(XICS_IPI + XICS_IRQ_OFFSET, xics_ipi_action, 0, "IPI", 0);
#endif
}
```

```
/*
 * iSeries_pci.c
 *
 * Copyright (C) 2001 Allan Trautman, IBM Corporation
 *
 * iSeries specific routines for PCI.
 *
 * Based on code from pci.c and iSeries_pci.c 32bit
 *
 * 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/list.h>
#include <linux/string.h>
#include <linux/init.h>
#include <linux/ide.h>
#include <linux/pci.h>
#include <linux/rtc.h>
#include <linux/time.h>

#include <asm/io.h>
#include <asm/irq.h>
#include <asm/prom.h>
#include <asm/machdep.h>
#include <asm/pci-bridge.h>
#include <asm/ppcdebug.h>
#include <asm/naca.h>
#include <asm/flight_recorder.h>
#include <asm/hardirq.h>
#include <asm/time.h>
#include <asm/pci_dma.h>

#include <asm/iSeries/HvCallPci.h>
#include <asm/iSeries/HvCallSm.h>
#include <asm/iSeries/HvCallXm.h>
#include <asm/iSeries/LparData.h>
#include <asm/iSeries/iSeries_irq.h>
#include <asm/iSeries/iSeries_pci.h>
#include <asm/iSeries/mf.h>

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

extern struct pci_controller* hose_head;
extern struct pci_controller** hose_tail;
extern int    global_phb_number;
extern int    panic_timeout;

extern struct device_node *allnodes;
extern unsigned long phb_tce_table_init(struct pci_controller *phb);
extern unsigned long iSeries_Base_Io_Memory;

extern struct pci_ops iSeries_pci_ops;
extern struct flightRecorder* PciFr;
extern struct TceTable* tceTables[256];

/*****************************************************************
 * Counters and control flags.
 *****************************************************************/
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;

extern int    Pci_Retry_Max;
extern int    Pci_Error_Flag;
extern int    Pci_Trace_Flag;

/*****************************************************************
 * Forward declares of prototypes.
 *****************************************************************/
struct iSeries_Device_Node* find_Device_Node(struct pci_dev* PciDev);
struct iSeries_Device_Node* get_Device_Node(struct pci_dev* PciDev);

unsigned long find_and_init_phbs(void);
void          fixup_resources(struct pci_dev *dev);
```

```
91  void          iSeries_pcibios_fixup(void);
92  struct        pci_controller* alloc_phb(struct device_node *dev, char *model, unsigned int addr_size_words) ;
93
94  void  iSeries_Scan_PHBs_Slots(struct pci_controller* Phb);
95  void  iSeries_Scan_EADs_Bridge(HvBusNumber Bus, HvSubBusNumber SubBus, int IdSel);
96  int   iSeries_Scan_Bridge_Slot(HvBusNumber Bus, struct HvCallPci_BridgeInfo* Info);
97  void  list_device_nodes(void);
98
99  struct pci_dev;
100
101 extern struct list_head iSeries_Global_Device_List;
102
103 int DeviceCount = 0;
104
105 /*****************************************************************************
106  * Log Error infor in Flight Recorder to system Console.
107  * Filter out the device not there errors.
108  * PCI: EADs Connect Failed 0x18.58.10 Rc: 0x00xx
109  * PCI: Read Vendor Failed 0x18.58.10 Rc: 0x00xx
110  * PCI: Connect Bus Unit Failed 0x18.58.10 Rc: 0x00xx
111  *****************************************************************************/
112 void  pci_Log_Error(char* Error_Text, int Bus, int SubBus, int AgentId, int HvRc)
113 {
114         if( HvRc != 0x0302) {
115                 char ErrorString[128];
116                 sprintf(ErrorString,"%s Failed: 0x%02X.%02X.%02X Rc: 0x%04X",Error_Text,Bus,SubBus,AgentId,HvRc);
117                 PCIFR(ErrorString);
118                 printk("PCI: %s\n",ErrorString);
119         }
120 }
121
122 /*****************************************************************************
123  * Dump the iSeries Temp Device Node
124  *<4>buswalk [swapper : − DeviceNode: 0xC000000000634300
125  *<4>00. Device Node   = 0xC000000000634300
126  *<4>    − PciDev      = 0x0000000000000000
127  *<4>    − tDevice     = 0x  17:01.00  0x1022 00
128  *<4>  4. Device Node = 0xC000000000634480
129  *<4>    − PciDev    = 0x0000000000000000
130  *<4>    − Device    = 0x  18:38.16 Irq:0xA7 Vendor:0x1014  Flags:0x00
131  *<4>    − Devfn     = 0xB0: 22.18
132  *****************************************************************************/
133 void dumpDevice_Node(struct iSeries_Device_Node* DevNode)
134 {
135         udbg_printf("Device Node    = 0x%p\n",DevNode);
136         udbg_printf("  − PciDev   = 0x%p\n",DevNode−>PciDev);
137         udbg_printf("  − Device   = 0x%4X:%02X.%02X (0x%02X)\n",
138                     ISERIES_BUS(DevNode),
139                     ISERIES_SUBBUS(DevNode),
140                     DevNode−>AgentId,
141                     DevNode−>DevFn);
142         udbg_printf("  − LSlot    = 0x%02X\n",DevNode−>LogicalSlot);
143         udbg_printf("  − TceTable = 0x%p\n ",DevNode−>DevTceTable);
144
145         udbg_printf("  − DSA    = 0x%04X\n",ISERIES_DSA(DevNode)>>32 );
146
147         udbg_printf("       = Irq:0x%02X Vendor:0x%04X  Flags:0x%02X\n",
148                     DevNode−>Irq,
149                     DevNode−>Vendor,
150                     DevNode−>Flags );
151         udbg_printf("  − Location = %s\n",DevNode−>CardLocation);
152
153
154 }
155 /*****************************************************************************
156  * Walk down the device node chain
157  *****************************************************************************/
158 void  list_device_nodes(void)
159 {
160         struct list_head* Device_Node_Ptr = iSeries_Global_Device_List.next;
161         while(Device_Node_Ptr != &iSeries_Global_Device_List) {
162                 dumpDevice_Node( (struct iSeries_Device_Node*)Device_Node_Ptr );
163                 Device_Node_Ptr = Device_Node_Ptr−>next;
164         }
165 }
166
167
168 /*********************************************************************
169  * build_device_node(u16 Bus, int SubBus, u8 DevFn)
170  *
171  *********************************************************************/
172 struct iSeries_Device_Node* build_device_node(HvBusNumber Bus, HvSubBusNumber  SubBus, int AgentId, int Function)
173 {
174         struct iSeries_Device_Node*  DeviceNode;
175
176         PPCDBG(PPCDBG_BUSWALK,"−build_device_node 0x%02X.%02X.%02X Function: %02X\n",Bus,SubBus,AgentId, Function);
177
178         DeviceNode = kmalloc(sizeof(struct iSeries_Device_Node), GFP_KERNEL);
179         if(DeviceNode == NULL) return NULL;
180
```

```
181         memset(DeviceNode,0,sizeof(struct iSeries_Device_Node) );
182         list_add_tail(&DeviceNode->Device_List,&iSeries_Global_Device_List);
183         /*DeviceNode->DsaAddr      = ((u64)Bus<<48)+((u64)SubBus<<40)+((u64)0x10<<32); */
184         ISERIES_BUS(DeviceNode)       = Bus;
185         ISERIES_SUBBUS(DeviceNode)    = SubBus;
186         DeviceNode->DsaAddr.deviceId  = 0x10;
187         DeviceNode->DsaAddr.barNumber = 0;
188         DeviceNode->AgentId           = AgentId;
189         DeviceNode->DevFn             = PCI_DEVFN(ISERIES_ENCODE_DEVICE(AgentId),Function );
190         DeviceNode->IoRetry           = 0;
191         iSeries_Get_Location_Code(DeviceNode);
192         PCIFR("Device 0x%02X.%2X, Node:0x%p ",ISERIES_BUS(DeviceNode),ISERIES_DEVFUN(DeviceNode),DeviceNode);
193         return DeviceNode;
194 }
195 /*****************************************************************************
196 *
197 * Allocate pci_controller(phb) initialized common variables.
198 *
199 *****************************************************************************/
200 struct pci_controller* pci_alloc_pci_controllerX(char *model, enum phb_types controller_type)
201 {
202         struct pci_controller *hose;
203         hose = (struct pci_controller*)kmalloc(sizeof(struct pci_controller), GFP_KERNEL);
204         if(hose == NULL) return NULL;
205
206         memset(hose, 0, sizeof(struct pci_controller));
207         if(strlen(model) < 8) strcpy(hose->what,model);
208         else                  memcpy(hose->what,model,7);
209         hose->type = controller_type;
210         hose->global_number = global_phb_number;
211         global_phb_number++;
212
213         *hose_tail = hose;
214         hose_tail = &hose->next;
215         return hose;
216 }
217
218 /*****************************************************************************
219  *
220  * unsigned int __init find_and_init_phbs(void)
221  *
222  * Description:
223  *   This function checks for all possible system PCI host bridges that connect
224  *   PCI buses.  The system hypervisor is queried as to the guest partition
225  *   ownership status.  A pci_controller is build for any bus which is partially
226  *   owned or fully owned by this guest partition.
227  *****************************************************************************/
228 unsigned long __init find_and_init_phbs(void)
229 {
230         struct      pci_controller* phb;
231         HvBusNumber BusNumber;
232
233         PPCDBG(PPCDBG_BUSWALK,"find_and_init_phbs Entry\n");
234
235         /* Check all possible buses. */
236         for (BusNumber = 0; BusNumber < 256; BusNumber++) {
237                 int RtnCode = HvCallXm_testBus(BusNumber);
238                 if (RtnCode == 0) {
239                         phb = pci_alloc_pci_controllerX("PHB HV", phb_type_hypervisor);
240                         if(phb == NULL) {
241                                 printk("PCI: Allocate pci_controller failed.\n");
242                                 PCIFR(      "Allocate pci_controller failed.");
243                                 return -1;
244                         }
245                         phb->pci_mem_offset = phb->local_number = BusNumber;
246                         phb->first_busno  = BusNumber;
247                         phb->last_busno   = BusNumber;
248                         phb->ops          = &iSeries_pci_ops;
249
250                         PPCDBG(PPCDBG_BUSWALK, "PCI:Create iSeries pci_controller(%p), Bus: %04X\n",phb,BusNumber);
251                         PCIFR("Create iSeries PHB controller: %04X",BusNumber);
252
253                         /************************************************/
254                         /* Find and connect the devices.                */
255                         /************************************************/
256                         iSeries_Scan_PHBs_Slots(phb);
257                 }
258                 /* Check for Unexpected Return code, a clue that something */
259                 /* has gone wrong.                                         */
260                 else if(RtnCode != 0x0301) {
261                         PCIFR("Unexpected Return on Probe(0x%04X): 0x%04X",BusNumber,RtnCode);
262                 }
263
264         }
265         return 0;
266 }
267 /*****************************************************************
268  * ppc64_pcibios_init
269  *
270  * Chance to initialize and structures or variable before PCI Bus walk.
```

```
271   *
272   *<4>buswalk [swapper : iSeries_pcibios_init Entry.
273   *<4>buswalk [swapper : IoMmTable Initialized 0xC00000000034BD30
274   *<4>buswalk [swapper : find_and_init_phbs Entry
275   *<4>buswalk [swapper : Create iSeries pci_controller:(0xC00000001F5C7000), Bus 0x0017
276   *<4>buswalk [swapper : Connect EADs: 0x17.00.12 = 0x00
277   *<4>buswalk [swapper : iSeries_assign_IRQ   0x0017.00.12 = 0x0091
278   *<4>buswalk [swapper : − allocate and assign IRQ 0x17.00.12 = 0x91
279   *<4>buswalk [swapper : − FoundDevice: 0x17.28.10 = 0x12AE
280   *<4>buswalk [swapper : − build_device_node 0x17.28.12
281   *<4>buswalk [swapper : iSeries_pcibios_init Exit.
282   *****************************************************************/
283  void iSeries_pcibios_init(void)
284  {
285          PPCDBG(PPCDBG_BUSWALK,"iSeries_pcibios_init Entry.\n");
286
287          iSeries_IoMmTable_Initialize();
288
289          find_and_init_phbs();
290
291          pci_assign_all_busses = 0;
292          PPCDBG(PPCDBG_BUSWALK,"iSeries_pcibios_init Exit.\n");
293  }
294  /*****************************************************************
295   * iSeries_pcibios_fixup(void)
296   *****************************************************************/
297  void __init iSeries_pcibios_fixup(void)
298  {
299          struct pci_dev* PciDev;
300          struct iSeries_Device_Node* DeviceNode;
301          char   Buffer[256];
302          int    DeviceCount = 0;
303
304          PPCDBG(PPCDBG_BUSWALK,"iSeries_pcibios_fixup Entry.\n");
305          /*************************************************/
306          /* Fix up at the device node and pci_dev relationship */
307          /*************************************************/
308          mf_displaySrc(0xC9000100);
309
310          pci_for_each_dev(PciDev) {
311                  DeviceNode = find_Device_Node(PciDev);
312                  if(DeviceNode != NULL) {
313                          ++DeviceCount;
314                          PciDev->sysdata    = (void*)DeviceNode;
315                          DeviceNode->PciDev = PciDev;
316
317                          PPCDBG(PPCDBG_BUSWALK,"PciDev 0x%p <==> DevNode 0x%p\n",PciDev,DeviceNode );
318
319                          iSeries_allocateDeviceBars(PciDev);
320
321                          PPCDBGCALL(PPCDBG_BUSWALK,dumpPci_Dev(PciDev) );
322
323                          iSeries_Device_Information(PciDev,Buffer, sizeof(Buffer) );
324                          printk("%d. %s\n",DeviceCount,Buffer);
325
326                          create_pci_bus_tce_table((unsigned long)DeviceNode);
327                  } else {
328                          printk("PCI: Device Tree not found for 0x%016lX\n",(unsigned long)PciDev);
329                  }
330          }
331          iSeries_IoMmTable_Status();
332
333          iSeries_activate_IRQs();
334
335          mf_displaySrc(0xC9000200);
336  }
337
338  /*****************************************************************
339   * iSeries_pcibios_fixup_bus(int Bus)
340   *
341   *****************************************************************/
342  void iSeries_pcibios_fixup_bus(struct pci_bus* PciBus)
343  {
344          PPCDBG(PPCDBG_BUSWALK,"iSeries_pcibios_fixup_bus(0x%04X) Entry.\n",PciBus->number);
345
346  }
347
348
349  /*****************************************************************
350   * fixup_resources(struct pci_dev *dev)
351   *
352   *****************************************************************/
353  void fixup_resources(struct pci_dev *PciDev)
354  {
355          PPCDBG(PPCDBG_BUSWALK,"fixup_resources PciDev %p\n",PciDev);
356  }
357
358
359  /*****************************************************************************
360  * Loop through each node function to find usable EADs bridges.
```

```
361 ******************************************************************************/
362 void  iSeries_Scan_PHBs_Slots(struct pci_controller* Phb)
363 {
364         struct HvCallPci_DeviceInfo* DevInfo;
365         HvBusNumber    Bus       = Phb->local_number;       /* System Bus        */
366         HvSubBusNumber SubBus    = 0;                       /* EADs is always 0. */
367         int            HvRc      = 0;
368         int            IdSel     = 1;
369         int            MaxAgents = 8;
370
371         DevInfo    = (struct HvCallPci_DeviceInfo*)kmalloc(sizeof(struct HvCallPci_DeviceInfo), GFP_KERNEL);
372         if(DevInfo == NULL) return;
373
374         /*************************************************************************
375          * Probe for EADs Bridges
376          *************************************************************************/
377         for (IdSel=1; IdSel < MaxAgents; ++IdSel) {
378                 HvRc = HvCallPci_getDeviceInfo(Bus, SubBus, IdSel,REALADDR(DevInfo), sizeof(struct HvCallPci_Devi
    ceInfo));
379                 if (HvRc == 0) {
380                         if(DevInfo->deviceType == HvCallPci_NodeDevice) {
381                                 iSeries_Scan_EADs_Bridge(Bus, SubBus, IdSel);
382                         }
383                         else {
384                                 printk("PCI: Invalid System Configuration(0x%02X).\n",DevInfo->deviceType);
385                                 PCIFR(      "Invalid System Configuration(0x%02X).",  DevInfo->deviceType);
386                         }
387                 }
388                 else pci_Log_Error("getDeviceInfo",Bus, SubBus, IdSel,HvRc);
389         }
390         kfree(DevInfo);
391 }
392
393 /*****************************************************************************
394 *
395 ******************************************************************************/
396 void  iSeries_Scan_EADs_Bridge(HvBusNumber Bus, HvSubBusNumber SubBus, int IdSel)
397 {
398         struct HvCallPci_BridgeInfo* BridgeInfo;
399         HvAgentId      AgentId;
400         int            Function;
401         int            HvRc;
402
403         BridgeInfo = (struct HvCallPci_BridgeInfo*)kmalloc(sizeof(struct HvCallPci_BridgeInfo), GFP_KERNEL);
404         if(BridgeInfo == NULL) return;
405
406         /*********************************************************************
407          * Note: hvSubBus and irq is always be 0 at this level!
408          *********************************************************************/
409         for (Function=0; Function < 8; ++Function) {
410                 AgentId = ISERIES_PCI_AGENTID(IdSel, Function);
411                 HvRc = HvCallXm_connectBusUnit(Bus, SubBus, AgentId, 0);
412                 if (HvRc == 0) {
413                         /*  Connect EADs: 0x18.00.12 = 0x00 */
414                         PPCDBG(PPCDBG_BUSWALK,"PCI:Connect EADs: 0x%02X.%02X.%02X\n",Bus, SubBus, AgentId);
415                         PCIFR(                    "Connect EADs: 0x%02X.%02X.%02X",  Bus, SubBus, AgentId);
416                         HvRc = HvCallPci_getBusUnitInfo(Bus, SubBus, AgentId,
417                                                         REALADDR(BridgeInfo), sizeof(struct HvCallPci_BridgeInfo)
    );
418                         if (HvRc == 0) {
419                                 PPCDBG(PPCDBG_BUSWALK,"PCI: BridgeInfo, Type:0x%02X, SubBus:0x%02X, MaxAgents:0x%02X, MaxSubBus
    : 0x%02X, LSlot: 0x%02X\n",
420                                        BridgeInfo->busUnitInfo.deviceType,
421                                        BridgeInfo->subBusNumber,
422                                        BridgeInfo->maxAgents,
423                                        BridgeInfo->maxSubBusNumber,
424                                        BridgeInfo->logicalSlotNumber);
425                                 PCIFR(                     "BridgeInfo, Type:0x%02X, SubBus:0x%02X, MaxAgents:0x%02X, MaxSubB
    us: 0x%02X, LSlot: 0x%02X",
426                                        BridgeInfo->busUnitInfo.deviceType,
427                                        BridgeInfo->subBusNumber,
428                                        BridgeInfo->maxAgents,
429                                        BridgeInfo->maxSubBusNumber,
430                                        BridgeInfo->logicalSlotNumber);
431
432                                 if (BridgeInfo->busUnitInfo.deviceType == HvCallPci_BridgeDevice)  {
433                                         /* Scan_Bridge_Slot...: 0x18.00.12 */
434                                         iSeries_Scan_Bridge_Slot(Bus,BridgeInfo);
435                                 }
436                                 else printk("PCI: Invalid Bridge Configuration(0x%02X)",BridgeInfo->busUnitInfo.deviceType);
437                         }
438                 }
439                 else if(HvRc != 0x000B) pci_Log_Error("EADs Connect",Bus,SubBus,AgentId,HvRc);
440         }
441         kfree(BridgeInfo);
442 }
443
444 /*****************************************************************************
445 *
446 * This assumes that the node slot is always on the primary bus!
```

```
447 *
448 *****************************************************************************/
449 int iSeries_Scan_Bridge_Slot(HvBusNumber Bus, struct HvCallPci_BridgeInfo* BridgeInfo)
450 {
451         struct iSeries_Device_Node* DeviceNode;
452         HvSubBusNumber SubBus = BridgeInfo->subBusNumber;
453         u16       VendorId    = 0;
454         int       HvRc        = 0;
455         u8        Irq         = 0;
456         int       IdSel       = ISERIES_GET_DEVICE_FROM_SUBBUS(SubBus);
457         int       Function    = ISERIES_GET_FUNCTION_FROM_SUBBUS(SubBus);
458         HvAgentId AgentId     = ISERIES_PCI_AGENTID(IdSel, Function);
459         HvAgentId EADsIdSel   = ISERIES_PCI_AGENTID(IdSel, Function);
460         int       FirstSlotId = 0;
461
462         /*******************************************************/
463         /* iSeries_allocate_IRQ.: 0x18.00.12(0xA3)             */
464         /*******************************************************/
465         Irq   = iSeries_allocate_IRQ(Bus, 0, AgentId);
466         iSeries_assign_IRQ(Irq, Bus, 0, AgentId);
467         PPCDBG(PPCDBG_BUSWALK,"PCI:- allocate and assign IRQ 0x%02X.%02X.%02X = 0x%02X\n",Bus, 0, AgentId, Irq );
468
469         /*********************************************************************
470          * Connect all functions of any device found.
471          *********************************************************************/
472         for (IdSel = 1; IdSel <= BridgeInfo->maxAgents; ++IdSel) {
473                 for (Function = 0; Function < 8; ++Function) {
474                         AgentId = ISERIES_PCI_AGENTID(IdSel, Function);
475                         HvRc = HvCallXm_connectBusUnit(Bus, SubBus, AgentId, Irq);
476                         if( HvRc == 0) {
477                                 HvRc = HvCallPci_configLoad16(Bus, SubBus, AgentId, PCI_VENDOR_ID, &VendorId);
478                                 if( HvRc == 0) {
479                                         /*******************************************************/
480                                         /* FoundDevice: 0x18.28.10 = 0x12AE                    */
481                                         /*******************************************************/
482                                         PPCDBG(PPCDBG_BUSWALK,"PCI:- FoundDevice: 0x%02X.%02X.%02X = 0x%04X\n",
483                                                                         Bus, SubBus, AgentId, VendorId);
484
485                                         HvRc = HvCallPci_configStore8(Bus, SubBus, AgentId, PCI_INTERRUPT_LINE, I
    rq);
486                                         if( HvRc != 0) {
487                                                 pci_Log_Error("PciCfgStore Irq Failed!",Bus,SubBus,AgentId,HvRc);
488                                         }
489
490                                         ++DeviceCount;
491                                         DeviceNode = build_device_node(Bus, SubBus, EADsIdSel, Function);
492                                         DeviceNode->Vendor      = VendorId;
493                                         DeviceNode->Irq         = Irq;
494                                         DeviceNode->LogicalSlot = BridgeInfo->logicalSlotNumber;
495                                         PCIFR("Device(%4d): 0x%02X.%02X.%02X 0x%02X 0x%04X",
496                                               DeviceCount,Bus, SubBus, AgentId,
497                                               DeviceNode->LogicalSlot,DeviceNode->Vendor);
498
499                                         /*******************************************************
500                                          * On the first device/function, assign irq to slot
501                                          *******************************************************/
502                                         if(Function == 0) {
503                                                 FirstSlotId = AgentId;
504                                                 // AHT iSeries_assign_IRQ(Irq, Bus, SubBus, AgentId);
505                                         }
506                                 }
507                                 else pci_Log_Error("Read Vendor",Bus,SubBus,AgentId,HvRc);
508                         }
509                         else pci_Log_Error("Connect Bus Unit",Bus,SubBus, AgentId,HvRc);
510                 } /* for (Function = 0; Function < 8; ++Function) */
511         } /* for (IdSel = 1; IdSel <= MaxAgents; ++IdSel) */
512         return HvRc;
513 }
514 /*****************************************************************************/
515 /* I/0 Memory copy MUST use mmio commands on iSeries                         */
516 /* To do; For performance, include the hv call directly                      */
517 /*****************************************************************************/
518 void* iSeries_memset_io(void* dest, char c, size_t Count)
519 {
520         u8    ByteValue     = c;
521         long  NumberOfBytes = Count;
522         char* IoBuffer      = dest;
523         while(NumberOfBytes > 0) {
524                 iSeries_Write_Byte( ByteValue, (void*)IoBuffer );
525                 ++IoBuffer;
526                 -- NumberOfBytes;
527         }
528         return dest;
529 }
530 void* iSeries_memcpy_toio(void *dest, void *source, size_t count)
531 {
532         char *dst           = dest;
533         char *src           = source;
534         long  NumberOfBytes = count;
535         while(NumberOfBytes > 0) {
```

```
536                 iSeries_Write_Byte(*src++, (void*)dst++);
537                 -- NumberOfBytes;
538         }
539         return dest;
540 }
541 void* iSeries_memcpy_fromio(void *dest, void *source, size_t count)
542 {
543         char *dst = dest;
544         char *src = source;
545         long  NumberOfBytes = count;
546         while(NumberOfBytes > 0) {
547                 *dst++ = iSeries_Read_Byte( (void*)src++);
548                 -- NumberOfBytes;
549         }
550         return dest;
551 }
552 /*****************************************************************************
553  * Look down the chain to find the matching Device Device
554  *****************************************************************************/
555 struct iSeries_Device_Node* find_Device_Node(struct pci_dev* PciDev)
556 {
557         struct list_head* Device_Node_Ptr = iSeries_Global_Device_List.next;
558         int Bus   = PciDev->bus->number;
559         int DevFn = PciDev->devfn;
560
561         while(Device_Node_Ptr != &iSeries_Global_Device_List) {
562                 struct iSeries_Device_Node* DevNode = (struct iSeries_Device_Node*)Device_Node_Ptr;
563                 if(Bus == ISERIES_BUS(DevNode) && DevFn == DevNode->DevFn) {
564                         return DevNode;
565                 }
566                 Device_Node_Ptr = Device_Node_Ptr->next;
567         }
568         return NULL;
569 }
570 /*********************************************************************/
571 /* Returns the device node for the passed pci_dev                    */
572 /* Sanity Check Node PciDev to passed pci_dev                        */
573 /* If none is found, returns a NULL which the client must handle. */
574 /*********************************************************************/
575 struct iSeries_Device_Node* get_Device_Node(struct pci_dev* PciDev)
576 {
577         struct iSeries_Device_Node* Node;
578         Node = (struct iSeries_Device_Node*)PciDev->sysdata;
579         if(Node == NULL ) {
580                 Node = find_Device_Node(PciDev);
581         }
582         else if(Node->PciDev != PciDev) {
583                 Node = find_Device_Node(PciDev);
584         }
585         return Node;
586 }
587 /*********************************************************************/
588 /* Set and reset Device Node Lock                                    */
589 /*********************************************************************/
590 #define setIoLock() \
591     unsigned long IrqFlags; \
592     spin_lock_irqsave(&DevNode->IoLock, IrqFlags );
593
594 #define resetIoLock() \
595     int RtnCode = DevNode->ReturnCode; \
596     spin_unlock_irqrestore( &DevNode->IoLock, IrqFlags ); \
597     return RtnCode;
598
599 /*****************************************************************************
600  *
601  * Read PCI Config Space Code
602  *
603  *****************************************************************************/
604 /** BYTE  *********************************************************************/
605 int iSeries_Node_read_config_byte(struct iSeries_Device_Node* DevNode, int Offset, u8* ReadValue)
606 {
607         u8  ReadData;
608         setIoLock();
609         ++Pci_Cfg_Read_Count;
610         DevNode->ReturnCode = HvCallPci_configLoad8(ISERIES_BUS(DevNode),ISERIES_SUBBUS(DevNode),0x10,
611                                                     Offset,&ReadData);
612         if(Pci_Trace_Flag == 1) {
613                 PCIFR("RCB: 0x%04X.%02X 0x%04X = 0x%02X",ISERIES_BUS(DevNode),DevNode->DevFn,Offset,ReadData);
614         }
615         if(DevNode->ReturnCode != 0 ) {
616                 printk("PCI: RCB: 0x%04X.%02X  Error: 0x%04X\n",ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCode)
    ;
617                 PCIFR(      "RCB: 0x%04X.%02X  Error: 0x%04X",  ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCod
    e);
618         }
619         *ReadValue = ReadData;
620         resetIoLock();
621 }
622 /** WORD  *********************************************************************/
623 int iSeries_Node_read_config_word(struct iSeries_Device_Node* DevNode, int Offset, u16* ReadValue)
```

```
624 {
625         u16  ReadData;
626         setIoLock();
627         ++Pci_Cfg_Read_Count;
628         DevNode->ReturnCode = HvCallPci_configLoad16(ISERIES_BUS(DevNode),ISERIES_SUBBUS(DevNode),0x10,
629                                                      Offset,&ReadData);
630         if(Pci_Trace_Flag == 1) {
631                 PCIFR("RCW: 0x%04X.%02X 0x%04X = 0x%04X",ISERIES_BUS(DevNode),DevNode->DevFn,Offset,ReadData);
632         }
633         if(DevNode->ReturnCode != 0 ) {
634                 printk("PCI: RCW: 0x%04X.%02X  Error: 0x%04X\n",ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCode
    );
635                 PCIFR(      "RCW: 0x%04X.%02X  Error: 0x%04X",  ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCo
    de);
636
637         }
638         *ReadValue = ReadData;
639         resetIoLock();
640 }
641 /** DWORD ********************************************************************/
642 int iSeries_Node_read_config_dword(struct iSeries_Device_Node* DevNode, int Offset, u32* ReadValue)
643 {
644         u32  ReadData;
645         setIoLock();
646         ++Pci_Cfg_Read_Count;
647         DevNode->ReturnCode = HvCallPci_configLoad32(ISERIES_BUS(DevNode),ISERIES_SUBBUS(DevNode),0x10,
648                                                      Offset,&ReadData);
649         if(Pci_Trace_Flag == 1) {
650                 PCIFR("RCL: 0x%04X.%02X 0x%04X = 0x%08X",ISERIES_BUS(DevNode),DevNode->DevFn,Offset,ReadData);
651         }
652         if(DevNode->ReturnCode != 0 ) {
653                 printk("PCI: RCL: 0x%04X.%02X  Error: 0x%04X\n",ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCode)
    ;
654                 PCIFR(      "RCL: 0x%04X.%02X  Error: 0x%04X",  ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCod
    e);
655         }
656         *ReadValue = ReadData;
657         resetIoLock();
658 }
659 int iSeries_pci_read_config_byte(struct pci_dev* PciDev, int Offset, u8* ReadValue) {
660         struct iSeries_Device_Node* DevNode = get_Device_Node(PciDev);
661         if(DevNode == NULL) return 0x0301;
662         return iSeries_Node_read_config_byte( DevNode ,Offset,ReadValue);
663 }
664 int iSeries_pci_read_config_word(struct pci_dev* PciDev, int Offset, u16* ReadValue) {
665         struct iSeries_Device_Node* DevNode = get_Device_Node(PciDev);
666         if(DevNode == NULL) return 0x0301;
667         return iSeries_Node_read_config_word( DevNode ,Offset,ReadValue );
668 }
669 int iSeries_pci_read_config_dword(struct pci_dev* PciDev, int Offset, u32* ReadValue) {
670         struct iSeries_Device_Node* DevNode = get_Device_Node(PciDev);
671         if(DevNode == NULL) return 0x0301;
672         return iSeries_Node_read_config_dword(DevNode ,Offset,ReadValue  );
673 }
674 /*****************************************************************************/
675 /*                                                                           */
676 /* Write PCI Config Space                                                    */
677 /*                                                                           */
678 /** BYTE  ********************************************************************/
679 int iSeries_Node_write_config_byte(struct iSeries_Device_Node* DevNode, int Offset, u8 WriteData)
680 {
681         setIoLock();
682         ++Pci_Cfg_Write_Count;
683         DevNode->ReturnCode = HvCallPci_configStore8(ISERIES_BUS(DevNode),ISERIES_SUBBUS(DevNode),0x10,
684                                                      Offset,WriteData);
685         if(Pci_Trace_Flag == 1) {
686                 PCIFR("WCB: 0x%04X.%02X 0x%04X = 0x%02X",ISERIES_BUS(DevNode),DevNode->DevFn,Offset,WriteData);
687         }
688         if(DevNode->ReturnCode != 0 ) {
689                 printk("PCI: WCB: 0x%04X.%02X  Error: 0x%04X\n",ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCode
    );
690                 PCIFR(      "WCB: 0x%04X.%02X  Error: 0x%04X",  ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCo
    de);
691         }
692         resetIoLock();
693 }
694 /** WORD  ********************************************************************/
695 int iSeries_Node_write_config_word(struct iSeries_Device_Node* DevNode, int Offset, u16 WriteData)
696 {
697         setIoLock();
698         ++Pci_Cfg_Write_Count;
699         DevNode->ReturnCode = HvCallPci_configStore16(ISERIES_BUS(DevNode),ISERIES_SUBBUS(DevNode),0x10,
700                                                      Offset,WriteData);
701         if(Pci_Trace_Flag == 1) {
702                 PCIFR("WCW: 0x%04X.%02X 0x%04X = 0x%04X",ISERIES_BUS(DevNode),DevNode->DevFn,Offset,WriteData);
703         }
704         if(DevNode->ReturnCode != 0 ) {
705                 printk("PCI: WCW: 0x%04X.%02X  Error: 0x%04X\n",ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCode
    );
706                 PCIFR(      "WCW: 0x%04X.%02X  Error: 0x%04X",  ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCo
```

```
de);
707             }
708             resetIoLock();
709     }
710     /** DWORD ********************************************************************/
711     int iSeries_Node_write_config_dword(struct iSeries_Device_Node* DevNode, int Offset, u32 WriteData)
712     {
713             setIoLock();
714             ++Pci_Cfg_Write_Count;
715             DevNode->ReturnCode = HvCallPci_configStore32(ISERIES_BUS(DevNode),ISERIES_SUBBUS(DevNode),0x10,
716                                                           Offset,WriteData);
717             if(Pci_Trace_Flag == 1) {
718                     PCIFR("WCL: 0x%04X.%02X 0x%04X = 0x%08X",ISERIES_BUS(DevNode),DevNode->DevFn,Offset,WriteData);
719             }
720             if(DevNode->ReturnCode != 0 ) {
721                     printk("PCI: WCL: 0x%04X.%02X  Error: 0x%04X\n",ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCode
    );
722                     PCIFR(      "WCL: 0x%04X.%02X  Error: 0x%04X",  ISERIES_BUS(DevNode),DevNode->DevFn,DevNode->ReturnCo
    de);
723             }
724             resetIoLock();
725     }
726     int iSeries_pci_write_config_byte( struct pci_dev* PciDev,int Offset, u8 WriteValue)
727     {
728             struct iSeries_Device_Node* DevNode = get_Device_Node(PciDev);
729             if(DevNode == NULL) return 0x0301;
730             return iSeries_Node_write_config_byte( DevNode,Offset,WriteValue);
731     }
732     int iSeries_pci_write_config_word( struct pci_dev* PciDev,int Offset,u16 WriteValue)
733     {
734             struct iSeries_Device_Node* DevNode = get_Device_Node(PciDev);
735             if(DevNode == NULL) return 0x0301;
736             return iSeries_Node_write_config_word( DevNode,Offset,WriteValue);
737     }
738     int iSeries_pci_write_config_dword(struct pci_dev* PciDev,int Offset,u32 WriteValue)
739     {
740             struct iSeries_Device_Node* DevNode = get_Device_Node(PciDev);
741             if(DevNode == NULL) return 0x0301;
742             return iSeries_Node_write_config_dword(DevNode,Offset,WriteValue);
743     }
744
745     /*****************************************************************************/
746     /* Branch Table                                                              */
747     /*****************************************************************************/
748     struct pci_ops iSeries_pci_ops = {
749             iSeries_pci_read_config_byte,
750             iSeries_pci_read_config_word,
751             iSeries_pci_read_config_dword,
752             iSeries_pci_write_config_byte,
753             iSeries_pci_write_config_word,
754             iSeries_pci_write_config_dword
755     };
756
757     /*****************************************************************************
758      * Log Pci Error and check Retry Count
759      * -> On Failure, print and log information.
760      *    Increment Retry Count, if exceeds max, panic partition.
761      * -> If in retry, print and log success
762      *****************************************************************************/
763     void logPciError(char* ErrorTxt, void* IoAddress, struct iSeries_Device_Node* DevNode, u64 RtnCode)
764     {
765             ++DevNode->IoRetry;
766             ++Pci_Error_Count;
767
768             PCIFR("%s: I/O Error(%1d/%1d):0x%04X  IoAddress:0x%p  Device:0x%04X:%02X",
769                   ErrorTxt, DevNode->IoRetry, in_interrupt(), RtnCode, IoAddress, ISERIES_BUS(DevNode),DevNode->Agent
    Id);
770             /*****************************************************/
771             /* Filter out EADs freeze and alignment errors       */
772             /*****************************************************/
773             if(RtnCode == 0x0102) {
774                     PCIFR("EADS Freeze error.....Panic.....");
775                     mf_displaySrc(0xB6000103);
776                     panic_timeout = 0;
777                     panic("PCI: EADs Freeze error SRC B6000103\n");
778             }
779             else if(RtnCode == 0x0241) {
780                     PCIFR("MMIO Alignment error: 0x%p",IoAddress);
781                     mf_displaySrc(0xB6000103);
782                     panic_timeout = 0;
783                     panic("PCI: MMIO Alignment error. SRC B6000103\n");
784             }
785             /*****************************************************/
786             /* Bump the retry and check for retry count exceeded.  */
787             /* If, Exceeded, panic the system.                     */
788             /*****************************************************/
789             if(DevNode->IoRetry > Pci_Retry_Max && Pci_Error_Flag != 0 ) {
790                     mf_displaySrc(0xB6000103);
791                     panic_timeout = 0;
792                     panic("PCI: Hardware I/O Error, SRC B6000103, Automatic Reboot Disabled.\n");
```

```
793         }
794         /*************************************************************/
795         /* Wait x ms before retrying I/O to give I/O time to recover. */
796         /* Retry wait delay logic.                                    */
797         /* − On first retry, no delay, maybe just glitch.             */
798         /* − On successify retries, vary the delay to avoid being in a*/
799         /*   repetitive timing window.                                */
800         /*************************************************************/
801         if(DevNode−>IoRetry > 0) {
802                 udelay(DevNode->IoRetry * 50);
803         }
804 }
805
806 /*********************************************************************
807  * Retry was successful
808  *********************************************************************/
809 void  pciRetrySuccessful(struct iSeries_Device_Node* DevNode)
810 {
811         struct  timeval  TimeClock;
812         struct  rtc_time CurTime;
813         do_gettimeofday(&TimeClock);
814         to_tm(TimeClock.tv_sec, &CurTime);
815
816         PCIFR("Retry Successful(%2d) on Device 0x%04X:%02X at %02d.%02d.%02d",
817               DevNode−>IoRetry,ISERIES_BUS(DevNode),DevNode−>AgentId,
818               CurTime.tm_hour,CurTime.tm_min,CurTime.tm_sec);
819
820         DevNode−>IoRetry = 0;
821 }
822
823 /*********************************************************************/
824 /* Translate the I/O Address into a device node, bar, and bar offset.   */
825 /* Note: Make sure the passed variable end up on the stack to avoid     */
826 /* the exposure of being device global.                                 */
827 /* The Device Node is Lock to block other I/O to device.                */
828 /*********************************************************************/
829 #define setUpMmIo(IoAddress,Type) \
830 unsigned long   IrqFlags; \
831 struct HvCallPci_LoadReturn Return; \
832 union HvDsaMap  DsaData;  \
833 u64             BarOffset;\
834 unsigned long BaseIoAddr = (unsigned long)IoAddress−iSeries_Base_Io_Memory; \
835 long          TableIndex = (BaseIoAddr/iSeries_IoMmTable_Entry_Size);       \
836 struct iSeries_Device_Node* DevNode = *(iSeries_IoMmTable+TableIndex);      \
837 if(DevNode != NULL) { \
838     DsaData.DsaAddr       = ISERIES_DSA(DevNode); \
839     DsaData.Dsa.barNumber = *(iSeries_IoBarTable+TableIndex); \
840     BarOffset             = BaseIoAddr % iSeries_IoMmTable_Entry_Size; \
841     ++Pci_Io_##Type##_Count; \
842     spin_lock_irqsave(&DevNode->IoLock, IrqFlags ); \
843 } \
844 else panic("PCI: Invalid PCI IoAddress detected 0x%p!\n",IoAddress);
845 /*********************************************************************/
846 /* Read MM I/O Instructions for the iSeries                          */
847 /* On MM I/O error, all ones are returned and iSeries_pci_IoError is cal*/
848 /* else, data is returned in big Endian format.                      */
849 /*********************************************************************/
850 /* iSeries_Read_Byte = Read Byte  ( 8 bit)                           */
851 /* iSeries_Read_Word = Read Word  (16 bit)                           */
852 /* iSeries_Read_Long = Read Long  (32 bit)                           */
853 /*********************************************************************/
854 u8  iSeries_Read_Byte(void* IoAddress)
855 {
856         setUpMmIo(IoAddress,Read);
857         do {
858                 HvCall3Ret16(HvCallPciBarLoad8, &Return, DsaData.DsaAddr,BarOffset, 0);
859                 if(Return.rc != 0 ) {
860                         logPciError("RDB",IoAddress, DevNode, Return.rc);
861                 }
862                 else if ( DevNode−>IoRetry > 0) {
863                         pciRetrySuccessful(DevNode);
864                 }
865         } while (Return.rc != 0);
866         spin_unlock_irqrestore(&DevNode−>IoLock, IrqFlags );
867         if(Pci_Trace_Flag == 1 ) PCIFR("RDB: IoAddress 0x%p = 0x%02X",IoAddress, (u8)Return.value);
868         return (u8)Return.value;
869 }
870 int Retry_Test = 0;
871 u16  iSeries_Read_Word(void* IoAddress)
872 {
873         setUpMmIo(IoAddress,Read);
874         do {
875                 HvCall3Ret16(HvCallPciBarLoad16,&Return, DsaData.DsaAddr,BarOffset, 0);
876
877                 if(Return.rc != 0 ) {
878                         logPciError("RDW",IoAddress, DevNode, Return.rc);
879                 }
880                 else if ( DevNode−>IoRetry > 0) {
881                         pciRetrySuccessful(DevNode);
882                 }
```

```
883         } while (Return.rc != 0);
884         spin_unlock_irqrestore(&DevNode->IoLock, IrqFlags );
885         if(Pci_Trace_Flag == 1 ) PCIFR("RDW: IoAddress 0x%p = 0x%04X",IoAddress, (u16)Return.value);
886         return swab16((u16)Return.value);
887 }
888 u32  iSeries_Read_Long(void* IoAddress)
889 {
890         setUpMmIo(IoAddress,Read);
891         do {
892                 HvCall3Ret16(HvCallPciBarLoad32,&Return, DsaData.DsaAddr,BarOffset, 0);
893
894                 if(Return.rc != 0 ) {
895                         logPciError("RDL",IoAddress, DevNode, Return.rc);
896                 }
897                 else if ( DevNode->IoRetry > 0) {
898                         pciRetrySuccessful(DevNode);
899                 }
900         } while (Return.rc != 0);
901         spin_unlock_irqrestore(&DevNode->IoLock, IrqFlags );
902         if(Pci_Trace_Flag == 1 ) PCIFR("RDL: IoAddress 0x%p = 0x%08X",IoAddress, swab32((u32)Return.value));
903         return swab32((u32)Return.value);
904 }
905 /*********************************************************************/
906 /* Write MM I/O Instructions for the iSeries                         */
907 /*********************************************************************/
908 /* iSeries_Write_Byte = Write Byte (8 bit)                           */
909 /* iSeries_Write_Word = Write Word(16 bit)                           */
910 /* iSeries_Write_Long = Write Long(32 bit)                           */
911 /*********************************************************************/
912 void iSeries_Write_Byte(u8 Data, void* IoAddress)
913 {
914         setUpMmIo(IoAddress,Write);
915         do {
916                 Return.rc = HvCall4(HvCallPciBarStore8, DsaData.DsaAddr,BarOffset, Data, 0);
917                 if(Return.rc != 0 ) {
918                         logPciError("WWB",IoAddress, DevNode, Return.rc);
919                 }
920                 else if ( DevNode->IoRetry > 0) {
921                         pciRetrySuccessful(DevNode);
922                 }
923         } while (Return.rc != 0);
924         spin_unlock_irqrestore(&DevNode->IoLock, IrqFlags );
925         if(Pci_Trace_Flag == 1) PCIFR("WWB: IoAddress 0x%p = 0x%02X",IoAddress,Data);
926 }
927 void iSeries_Write_Word(u16 Data, void* IoAddress)
928 {
929         setUpMmIo(IoAddress,Write);
930         do {
931                 Return.rc = HvCall4(HvCallPciBarStore16,DsaData.DsaAddr,BarOffset, swab16(Data), 0);
932                 if(Return.rc != 0 ) {
933                         logPciError("WWW",IoAddress, DevNode, Return.rc);
934                 }
935                 else if ( DevNode->IoRetry > 0) {
936                         pciRetrySuccessful(DevNode);
937                 }
938         } while (Return.rc != 0);
939         spin_unlock_irqrestore(&DevNode->IoLock, IrqFlags );
940         if(Pci_Trace_Flag == 1) PCIFR("WWW: IoAddress 0x%p = 0x%04X",IoAddress,Data);
941 }
942 void iSeries_Write_Long(u32 Data, void* IoAddress)
943 {
944         setUpMmIo(IoAddress,Write);
945         do {
946                 Return.rc = HvCall4(HvCallPciBarStore32,DsaData.DsaAddr,BarOffset, swab32(Data), 0);
947                 if(Return.rc != 0 ) {
948                         logPciError("WWL",IoAddress, DevNode, Return.rc);
949                 }
950                 else if ( DevNode->IoRetry > 0) {
951                         pciRetrySuccessful(DevNode);
952                 }
953         } while (Return.rc != 0);
954         spin_unlock_irqrestore(&DevNode->IoLock, IrqFlags );
955         if(Pci_Trace_Flag == 1) PCIFR("WWL: IoAddress 0x%p = 0x%08X",IoAddress, Data);
956 }
957 /*
958  * This is called very early before the page table is setup.
959  * There are warnings here because of type mismatches.. Okay for now. AHT
960  */
961 void
962 iSeries_pcibios_init_early(void)
963 {
964         //ppc_md.pcibios_read_config_byte   = iSeries_Node_read_config_byte;
965         //ppc_md.pcibios_read_config_word   = iSeries_Node_read_config_word;
966         //ppc_md.pcibios_read_config_dword  = iSeries_Node_read_config_dword;
967         //ppc_md.pcibios_write_config_byte  = iSeries_Node_write_config_byte;
968         //ppc_md.pcibios_write_config_word  = iSeries_Node_write_config_word;
969         //ppc_md.pcibios_write_config_dword = iSeries_Node_write_config_dword;
970 }
971
972 /*********************************************************************/
```

```
973 /* Set the slot reset line to the state passed in.                      */
974 /* This is the platform specific for code for the pci_reset_device      */
975 /* function.                                                            */
976 /**********************************************************************/
977 int pci_set_reset(struct pci_dev* PciDev, int State) {
978         struct iSeries_Device_Node* DeviceNode = (struct iSeries_Device_Node*)PciDev->sysdata;
979         if (DeviceNode == NULL) {
980                 printk("PCI: Pci Reset Failed, Device Node not found for pci_dev %p\n",PciDev);
981                 return -1;
982         }
983         DeviceNode->ReturnCode = HvCallPci_setSlotReset(ISERIES_BUS(DeviceNode),0x00,DeviceNode->AgentId,State);
984         return DeviceNode->ReturnCode;
985 }
```

```
/*
  * iSeries_proc.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
  */


/* Change Activity: */
/* End Change Activity */

#include <linux/proc_fs.h>
#include <linux/spinlock.h>
#ifndef _ISERIES_PROC_H
#include <asm/iSeries/iSeries_proc.h>
#endif


static struct proc_dir_entry * iSeries_proc_root = NULL;
static int iSeries_proc_initializationDone = 0;
static spinlock_t iSeries_proc_lock;

struct iSeries_proc_registration
{
        struct iSeries_proc_registration *next;
        iSeriesProcFunction functionMember;
};


struct iSeries_proc_registration preallocated[16];
#define MYQUEUETYPE(T) struct MYQueue##T
#define MYQUEUE(T) \
MYQUEUETYPE(T) \
{ \
        struct T *head; \
        struct T *tail; \
}
#define MYQUEUECTOR(q) do { (q)->head = NULL; (q)->tail = NULL; } while(0)
#define MYQUEUEENQ(q, p) \
do { \
        (p)->next = NULL; \
        if ((q)->head != NULL) { \
                (q)->head->next = (p); \
                (q)->head = (p); \
        } else { \
                (q)->tail = (q)->head = (p); \
        } \
} while(0)

#define MYQUEUEDEQ(q,p) \
do { \
        (p) = (q)->tail; \
        if ((p) != NULL) { \
                (q)->tail = (p)->next; \
                (p)->next = NULL; \
        } \
        if ((q)->tail == NULL) \
                (q)->head = NULL; \
} while(0)
MYQUEUE(iSeries_proc_registration);
typedef MYQUEUETYPE(iSeries_proc_registration) aQueue;


aQueue iSeries_free;
aQueue iSeries_queued;

void iSeries_proc_early_init(void)
{
        int i = 0;
        unsigned long flags;
        iSeries_proc_initializationDone = 0;
        spin_lock_init(&iSeries_proc_lock);
        MYQUEUECTOR(&iSeries_free);
        MYQUEUECTOR(&iSeries_queued);

        spin_lock_irqsave(&iSeries_proc_lock, flags);
        for (i = 0; i < 16; ++i) {
                MYQUEUEENQ(&iSeries_free, preallocated+i);
```

```
91          }
92          spin_unlock_irqrestore(&iSeries_proc_lock, flags);
93  }
94
95  void iSeries_proc_create(void)
96  {
97          unsigned long flags;
98          struct iSeries_proc_registration *reg = NULL;
99          spin_lock_irqsave(&iSeries_proc_lock, flags);
100         printk("iSeries_proc: Creating /proc/iSeries\n");
101
102         iSeries_proc_root = proc_mkdir("iSeries", 0);
103         if (!iSeries_proc_root) return;
104
105         MYQUEUEDEQ(&iSeries_queued, reg);
106
107         while (reg != NULL) {
108                 (*(reg->functionMember))(iSeries_proc_root);
109
110                 MYQUEUEDEQ(&iSeries_queued, reg);
111         }
112
113         iSeries_proc_initializationDone = 1;
114         spin_unlock_irqrestore(&iSeries_proc_lock, flags);
115 }
116
117 void iSeries_proc_callback(iSeriesProcFunction initFunction)
118 {
119         unsigned long flags;
120         spin_lock_irqsave(&iSeries_proc_lock, flags);
121
122         if (iSeries_proc_initializationDone) {
123                 (*initFunction)(iSeries_proc_root);
124         } else {
125                 struct iSeries_proc_registration *reg = NULL;
126
127                 MYQUEUEDEQ(&iSeries_free, reg);
128
129                 if (reg != NULL) {
130                         /* printk("Registering %p in reg %p\n", initFunction, reg); */
131                         reg->functionMember = initFunction;
132
133                         MYQUEUEENQ(&iSeries_queued, reg);
134                 } else {
135                         printk("Couldn't get a queue entry\n");
136                 }
137         }
138
139         spin_unlock_irqrestore(&iSeries_proc_lock, flags);
140 }
141
142
```

```
/*
 * mf_proc.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
 */


/* Change Activity: */
/* End Change Activity */

#ifndef _MF_PROC_H
#include <asm/iSeries/mf_proc.h>
#endif
#ifndef MF_H_INCLUDED
#include <asm/iSeries/mf.h>
#endif
#include <asm/uaccess.h>

static struct proc_dir_entry *mf_proc_root = NULL;

int proc_mf_dump_cmdline
(char *page, char **start, off_t off, int count, int *eof, void *data);

int proc_mf_dump_vmlinux
(char *page, char **start, off_t off, int count, int *eof, void *data);

int proc_mf_dump_side
(char *page, char **start, off_t off, int count, int *eof, void *data);

int proc_mf_change_side
(struct file *file, const char *buffer, unsigned long count, void *data);

int proc_mf_dump_src
(char *page, char **start, off_t off, int count, int *eof, void *data);
int proc_mf_change_src (struct file *file, const char *buffer, unsigned long count, void *data);
int proc_mf_change_cmdline(struct file *file, const char *buffer, unsigned long count, void *data);
int proc_mf_change_vmlinux(struct file *file, const char *buffer, unsigned long count, void *data);


void mf_proc_init(struct proc_dir_entry *iSeries_proc)
{
        struct proc_dir_entry *ent = NULL;
        struct proc_dir_entry *mf_a = NULL;
        struct proc_dir_entry *mf_b = NULL;
        struct proc_dir_entry *mf_c = NULL;
        struct proc_dir_entry *mf_d = NULL;

        mf_proc_root = proc_mkdir("mf", iSeries_proc);
        if (!mf_proc_root) return;

        mf_a = proc_mkdir("A", mf_proc_root);
        if (!mf_a) return;

        ent = create_proc_entry("cmdline", S_IFREG|S_IRUSR|S_IWUSR, mf_a);
        if (!ent) return;
        ent->nlink = 1;
        ent->data = (void *)0;
        ent->read_proc = proc_mf_dump_cmdline;
        ent->write_proc = proc_mf_change_cmdline;

        ent = create_proc_entry("vmlinux", S_IFREG|S_IWUSR, mf_a);
        if (!ent) return;
        ent->nlink = 1;
        ent->data = (void *)0;
        ent->write_proc = proc_mf_change_vmlinux;
        ent->read_proc = NULL;

        mf_b = proc_mkdir("B", mf_proc_root);
        if (!mf_b) return;

        ent = create_proc_entry("cmdline", S_IFREG|S_IRUSR|S_IWUSR, mf_b);
        if (!ent) return;
        ent->nlink = 1;
        ent->data = (void *)1;
        ent->read_proc = proc_mf_dump_cmdline;
        ent->write_proc = proc_mf_change_cmdline;

```

```
91          ent = create_proc_entry("vmlinux", S_IFREG|S_IWUSR, mf_b);
92          if (!ent) return;
93          ent->nlink = 1;
94          ent->data = (void *)1;
95          ent->write_proc = proc_mf_change_vmlinux;
96          ent->read_proc = NULL;
97
98          mf_c = proc_mkdir("C", mf_proc_root);
99          if (!mf_c) return;
100
101         ent = create_proc_entry("cmdline", S_IFREG|S_IRUSR|S_IWUSR, mf_c);
102         if (!ent) return;
103         ent->nlink = 1;
104         ent->data = (void *)2;
105         ent->read_proc = proc_mf_dump_cmdline;
106         ent->write_proc = proc_mf_change_cmdline;
107
108         ent = create_proc_entry("vmlinux", S_IFREG|S_IWUSR, mf_c);
109         if (!ent) return;
110         ent->nlink = 1;
111         ent->data = (void *)2;
112         ent->write_proc = proc_mf_change_vmlinux;
113         ent->read_proc = NULL;
114
115         mf_d = proc_mkdir("D", mf_proc_root);
116         if (!mf_d) return;
117
118
119         ent = create_proc_entry("cmdline", S_IFREG|S_IRUSR|S_IWUSR, mf_d);
120         if (!ent) return;
121         ent->nlink = 1;
122         ent->data = (void *)3;
123         ent->read_proc = proc_mf_dump_cmdline;
124         ent->write_proc = proc_mf_change_cmdline;
125 #if 0
126         ent = create_proc_entry("vmlinux", S_IFREG|S_IRUSR, mf_d);
127         if (!ent) return;
128         ent->nlink = 1;
129         ent->data = (void *)3;
130         ent->read_proc = proc_mf_dump_vmlinux;
131         ent->write_proc = NULL;
132 #endif
133         ent = create_proc_entry("side", S_IFREG|S_IRUSR|S_IWUSR, mf_proc_root);
134         if (!ent) return;
135         ent->nlink = 1;
136         ent->data = (void *)0;
137         ent->read_proc = proc_mf_dump_side;
138         ent->write_proc = proc_mf_change_side;
139
140         ent = create_proc_entry("src", S_IFREG|S_IRUSR|S_IWUSR, mf_proc_root);
141         if (!ent) return;
142         ent->nlink = 1;
143         ent->data = (void *)0;
144         ent->read_proc = proc_mf_dump_src;
145         ent->write_proc = proc_mf_change_src;
146 }
147
148 int proc_mf_dump_cmdline
149 (char *page, char **start, off_t off, int count, int *eof, void *data)
150 {
151         int             len = count;
152         char *p;
153
154         len = mf_getCmdLine(page, &len, (u64)data);
155
156         p = page + len - 1;
157         while ( p > page ) {
158                 if ( (*p == 0) || (*p == ' ') )
159                         --p;
160                 else
161                         break;
162         }
163         if ( *p != '\n' ) {
164                 ++p;
165                 *p = '\n';
166         }
167         ++p;
168         *p = 0;
169         len = p - page;
170
171         len -= off;
172         if (len < count) {
173                 *eof = 1;
174                 if (len <= 0)
175                         return 0;
176         } else
177                 len = count;
178         *start = page + off;
179         return len;
180 }
```

```
181
182 int proc_mf_dump_vmlinux
183 (char *page, char **start, off_t off, int count, int *eof, void *data)
184 {
185         int sizeToGet = count;
186         if (!capable(CAP_SYS_ADMIN))
187                 return −EACCES;
188
189         if (mf_getVmlinuxChunk(page, &sizeToGet, off, (u64)data) == 0)
190         {
191                 if (sizeToGet != 0)
192                 {
193                         *start = page + off;
194                         return sizeToGet;
195                 } else {
196                         *eof = 1;
197                         return 0;
198                 }
199         } else {
200                 *eof = 1;
201                 return 0;
202         }
203 }
204
205 int proc_mf_dump_side
206 (char *page, char **start, off_t off, int count, int *eof, void *data)
207 {
208         int             len = 0;
209
210         char mf_current_side = mf_getSide();
211         len = sprintf(page, "%c\n", mf_current_side);
212
213         if (len <= off+count) *eof = 1;
214         *start = page + off;
215         len −= off;
216         if (len>count) len = count;
217         if (len<0) len = 0;
218         return len;
219 }
220
221 int proc_mf_change_side(struct file *file, const char *buffer, unsigned long count, void *data)
222 {
223         if (!capable(CAP_SYS_ADMIN))
224                 return −EACCES;
225
226         if ((*buffer != 'A') &&
227             (*buffer != 'B') &&
228             (*buffer != 'C') &&
229             (*buffer != 'D'))
230         {
231                 printk(KERN_ERR "mf_proc.c: proc_mf_change_side: invalid side\n");
232                 return −EINVAL;
233         }
234
235         mf_setSide(*buffer);
236
237         return count;
238 }
239
240 int proc_mf_dump_src
241 (char *page, char **start, off_t off, int count, int *eof, void *data)
242 {
243         int             len = 0;
244         mf_getSrcHistory(page, count);
245         len = count;
246         len −= off;
247         if (len < count) {
248                 *eof = 1;
249                 if (len <= 0)
250                         return 0;
251         } else
252                 len = count;
253         *start = page + off;
254         return len;
255 }
256
257 int proc_mf_change_src(struct file *file, const char *buffer, unsigned long count, void *data)
258 {
259         if (!capable(CAP_SYS_ADMIN))
260                 return −EACCES;
261
262         if ((count < 4) && (count != 1))
263         {
264                 printk(KERN_ERR "mf_proc: invalid src\n");
265                 return −EINVAL;
266         }
267
268         if ((count == 1) && ((*buffer) == '\0'))
269         {
270                 mf_clearSrc();
```

```
271         } else {
272                 mf_displaySrc(*(u32 *)buffer);
273         }
274
275         return count;
276 }
277
278 int proc_mf_change_cmdline(struct file *file, const char *buffer, unsigned long count, void *data)
279 {
280         if (!capable(CAP_SYS_ADMIN))
281                 return -EACCES;
282
283         mf_setCmdLine(buffer, count, (u64)data);
284
285         return count;
286 }
287
288 int proc_mf_change_vmlinux(struct file *file, const char *buffer, unsigned long count, void *data)
289 {
290         if (!capable(CAP_SYS_ADMIN))
291                 return -EACCES;
292
293         mf_setVmlinuxChunk(buffer, count, file->f_pos, (u64)data);
294         file->f_pos += count;
295
296         return count;
297 }
```

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

#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 <asm/pmc.h>
#include <asm/uaccess.h>
#include <asm/naca.h>
#include <asm/perfmon.h>

extern char _stext[], _etext[], _end[];
struct perfmon_base_struct perfmon_base = {0, 0, 0, 0, 0, PMC_STATE_INITIAL};

int alloc_perf_buffer(int size);
int free_perf_buffer(void);
int clear_buffers(void);
void pmc_stop(void *data);
void pmc_start(void *data);
void pmc_touch_bolted(void *data);
void dump_pmc_struct(struct perfmon_struct *perfdata);
void dump_hardware_pmc_struct(void *perfdata);
int  decr_profile(struct perfmon_struct *perfdata);
int  pmc_profile(struct perfmon_struct *perfdata);
int  pmc_set_general(struct perfmon_struct *perfdata);
int  pmc_set_user_general(struct perfmon_struct *perfdata);
void pmc_configure(void *data);

asmlinkage int
sys_perfmonctl (int cmd, void *data)
{
        struct perfmon_struct *pdata;
        int err;

        printk("sys_perfmonctl: cmd = 0x%x\n", cmd);
        pdata = kmalloc(sizeof(struct perfmon_struct), GFP_USER);
        err = __copy_from_user(pdata, data, sizeof(struct perfmon_struct));
        switch(cmd) {
        case PMC_OP_ALLOC:
                alloc_perf_buffer(0);
                break;
        case PMC_OP_FREE:
                free_perf_buffer();
                break;
        case PMC_OP_CLEAR:
                clear_buffers();
                break;
        case PMC_OP_DUMP:
                dump_pmc_struct(pdata);
                copy_to_user(data, pdata, sizeof(struct perfmon_struct));
                break;
        case PMC_OP_DUMP_HARDWARE:
                dump_hardware_pmc_struct(pdata);
                smp_call_function(dump_hardware_pmc_struct, (void *)pdata, 0, 1);
                break;
        case PMC_OP_DECR_PROFILE: /* NIA time sampling */
                decr_profile(pdata);
                break;
        case PMC_OP_PMC_PROFILE:
                pmc_profile(pdata);
                break;
        case PMC_OP_SET:
                pmc_set_general(pdata);
                break;
        case PMC_OP_SET_USER:
                pmc_set_user_general(pdata);
                break;
        default:
                printk("Perfmon: Unknown operation\n");
                break;
        }

        kfree(pdata);
        return 0;
}

int alloc_perf_buffer(int size)
{
        int i;

        printk("Perfmon: allocate buffer\n");
        if(perfmon_base.state == PMC_STATE_INITIAL) {
                perfmon_base.profile_length = (((unsigned long) &_etext −
                                        (unsigned long) &_stext) >> 2) * sizeof(int);
```

```
91                  perfmon_base.profile_buffer = (unsigned long)btmalloc(perfmon_base.profile_length);
92                  perfmon_base.trace_length = 1024*1024*16;
93                  perfmon_base.trace_buffer = (unsigned long)btmalloc(perfmon_base.trace_length);
94
95                  if(perfmon_base.profile_buffer && perfmon_base.trace_buffer) {
96                          memset((char *)perfmon_base.profile_buffer, 0, perfmon_base.profile_length);
97                          printk("Profile buffer created at address 0x%lx of length 0x%lx\n",
98                                  perfmon_base.profile_buffer, perfmon_base.profile_length);
99                  } else {
100                         printk("Profile buffer creation failed\n");
101                         return 0;
102                 }
103
104                 /* Fault in the first bolted segment − it then remains in the stab for all time */
105                 pmc_touch_bolted(NULL);
106                 smp_call_function(pmc_touch_bolted, (void *)NULL, 0, 1);
107
108                 for (i=0; i<MAX_PACAS; ++i) {
109                         paca[i].prof_shift = 2;
110                         paca[i].prof_len = perfmon_base.profile_length;
111                         paca[i].prof_buffer = (unsigned *)(perfmon_base.profile_buffer);
112                         paca[i].prof_stext = (unsigned *)&_stext;
113
114                         paca[i].prof_etext = (unsigned *)&_etext;
115                         mb();
116                 }
117
118                 perfmon_base.state = PMC_STATE_READY;
119         }
120
121         return 0;
122 }
123
124 int free_perf_buffer()
125 {
126         printk("Perfmon: free buffer\n");
127
128         if(perfmon_base.state == PMC_STATE_INITIAL) {
129                 printk("Perfmon: free buffer failed – no buffer was allocated.\n");
130                 return −1;
131         }
132
133         btfree((void *)perfmon_base.profile_buffer);
134         btfree((void *)perfmon_base.trace_buffer);
135
136         perfmon_base.profile_length = 0;
137         perfmon_base.profile_buffer = 0;
138         perfmon_base.trace_buffer   = 0;
139         perfmon_base.trace_length   = 0;
140         perfmon_base.trace_end      = 0;
141         perfmon_base.state = PMC_STATE_INITIAL;
142
143         return(0);
144 }
145
146 int clear_buffers()
147 {
148         if(perfmon_base.state == PMC_STATE_INITIAL) {
149                 printk("Perfmon: clear buffer failed – no buffer was allocated.\n");
150                 return −1;
151         }
152
153         printk("Perfmon: clear buffer\n");
154
155         /* Stop counters on all processors −− blocking */
156         pmc_stop(NULL);
157         smp_call_function(pmc_stop, (void *)NULL, 0, 1);
158
159         /* Clear the buffers */
160         memset((char *)perfmon_base.profile_buffer, 0, perfmon_base.profile_length);
161         memset((char *)perfmon_base.trace_buffer, 0, perfmon_base.trace_length);
162
163         /* Reset the trace buffer point */
164         perfmon_base.trace_end = 0;
165
166         /* Restart counters on all processors −− blocking */
167         pmc_start(NULL);
168         smp_call_function(pmc_start, (void *)NULL, 0, 1);
169
170         return(0);
171 }
172
173 void pmc_stop(void *data)
174 {
175         /* Freeze all counters, leave everything else alone */
176         mtspr( MMCR0, mfspr( MMCR0 ) | 0x80000000 );
177 }
178
179 void pmc_start(void *data)
180 {
```

```
181         /* Free all counters, leave everything else alone */
182         mtspr( MMCR0, mfspr( MMCR0 ) & 0x7fffffff );
183 }
184
185 void pmc_touch_bolted(void *data)
186 {
187         volatile int touch;
188
189         /* Hack to fault the buffer into the segment table */
190         touch = *((int *)(perfmon_base.profile_buffer));
191 }
192
193 void dump_pmc_struct(struct perfmon_struct *perfdata)
194 {
195         unsigned int cpu = perfdata->vdata.pmc_info.cpu, i;
196
197         if(cpu > MAX_PACAS) return;
198
199         printk("PMC Control Mode: 0x%lx\n", perfmon_base.state);
200         printk("PMC[1 – 2] = 0x%16.16lx 0x%16.16lx\n",
201                paca[cpu].pmcc[0], paca[cpu].pmcc[1]);
202         printk("PMC[3 – 4] = 0x%16.16lx 0x%16.16lx\n",
203                paca[cpu].pmcc[2], paca[cpu].pmcc[3]);
204         printk("PMC[5 – 6] = 0x%16.16lx 0x%16.16lx\n",
205                paca[cpu].pmcc[4], paca[cpu].pmcc[5]);
206         printk("PMC[7 – 8] = 0x%16.16lx 0x%16.16lx\n",
207                paca[cpu].pmcc[6], paca[cpu].pmcc[7]);
208
209         perfdata->vdata.pmc_info.mode = perfmon_base.state;
210         for(i = 0; i < 11; i++)
211                 perfdata->vdata.pmc_info.pmc_base[i]  = paca[cpu].pmc[i];
212
213         for(i = 0; i < 8; i++)
214                 perfdata->vdata.pmc_info.pmc_cumulative[i]  = paca[cpu].pmcc[i];
215 }
216
217 void dump_hardware_pmc_struct(void *perfdata)
218 {
219         unsigned int cpu = smp_processor_id();
220
221         printk("PMC[%2.2d][1 – 4]  = 0x%8.8x 0x%8.8x 0x%8.8x 0x%8.8x\n",
222                cpu, (u32) mfspr(PMC1),(u32) mfspr(PMC2),(u32) mfspr(PMC3),(u32) mfspr(PMC4));
223         printk("PMC[%2.2d][5 – 8]  = 0x%8.8x 0x%8.8x 0x%8.8x 0x%8.8x\n",
224                cpu, (u32) mfspr(PMC5),(u32) mfspr(PMC6),(u32) mfspr(PMC7),(u32) mfspr(PMC8));
225         printk("MMCR[%2.2d][0,1,A] = 0x%8.8x 0x%8.8x 0x%8.8x\n",
226                cpu, (u32) mfspr(MMCR0),(u32) mfspr(MMCR1),(u32) mfspr(MMCRA));
227 }
228
229 int decr_profile(struct perfmon_struct *perfdata)
230 {
231         int i;
232
233         printk("Perfmon: NIA decrementer profile\n");
234
235         if(perfmon_base.state == PMC_STATE_INITIAL) {
236                 printk("Perfmon: failed – no buffer was allocated.\n");
237                 return −1;
238         }
239
240         /* Stop counters on all processors −− blocking */
241         pmc_stop(NULL);
242         smp_call_function(pmc_stop, (void *)NULL, 0, 1);
243
244         for (i=0; i<MAX_PACAS; ++i) {
245                 paca[i].prof_mode = PMC_STATE_DECR_PROFILE;
246         }
247
248         perfmon_base.state = PMC_STATE_DECR_PROFILE;
249         mb();
250
251         return 0;
252 }
253
254 int pmc_profile(struct perfmon_struct *perfdata)
255 {
256         struct pmc_struct *pdata = &(perfdata->vdata.pmc);
257         int i;
258
259         printk("Perfmon: NIA PMC profile and CPI\n");
260
261         if(perfmon_base.state == PMC_STATE_INITIAL) {
262                 printk("Perfmon: failed – no buffer was allocated.\n");
263                 return −1;
264         }
265
266         /* Stop counters on all processors −− blocking */
267         pmc_stop(NULL);
268         smp_call_function(pmc_stop, (void *)NULL, 0, 1);
269
270         for (i=0; i<MAX_PACAS; ++i) {
```

```
271                     paca[i].prof_mode = PMC_STATE_PROFILE_KERN;
272             }
273             perfmon_base.state = PMC_STATE_PROFILE_KERN;
274
275             pdata->pmc[0] = 0x7f000000;
276             for(i = 1; i < 8; i++)
277                     pdata->pmc[i] = 0x0;
278             pdata->pmc[8] = 0x26000000 | (0x01 << (31 - 25) | (0x1));
279             pdata->pmc[9] = (0x3 << (31-4)); /* Instr completed */
280             pdata->pmc[10] = 0x00000000 | (0x1 << (31 - 30));
281
282             mb();
283
284             pmc_configure((void *)perfdata);
285             smp_call_function(pmc_configure, (void *)perfdata, 0, 0);
286
287             return 0;
288     }
289
290     int pmc_set_general(struct perfmon_struct *perfdata)
291     {
292             int i;
293
294             printk("Perfmon: PMC sampling - General\n");
295
296             if(perfmon_base.state == PMC_STATE_INITIAL) {
297                     printk("Perfmon: failed - no buffer was allocated.\n");
298                     return -1;
299             }
300
301             /* Stop counters on all processors -- blocking */
302             pmc_stop(NULL);
303             smp_call_function(pmc_stop, (void *)NULL, 0, 1);
304
305             for (i=0; i<MAX_PACAS; ++i) {
306                     paca[i].prof_mode = PMC_STATE_TRACE_KERN;
307             }
308             perfmon_base.state = PMC_STATE_TRACE_KERN;
309             mb();
310
311             pmc_configure((void *)perfdata);
312             smp_call_function(pmc_configure, (void *)perfdata, 0, 0);
313
314             return 0;
315     }
316
317     int pmc_set_user_general(struct perfmon_struct *perfdata)
318     {
319             struct pmc_struct *pdata = &(perfdata->vdata.pmc);
320             int pid = perfdata->header.pid;
321             struct task_struct *task;
322             int i;
323
324             printk("Perfmon: PMC sampling - general user\n");
325
326             if(perfmon_base.state == PMC_STATE_INITIAL) {
327                     printk("Perfmon: failed - no buffer was allocated.\n");
328                     return -1;
329             }
330
331             if(pid) {
332                     printk("Perfmon: pid = 0x%x\n", pid);
333                     read_lock(&tasklist_lock);
334                     task = find_task_by_pid(pid);
335                     if (task) {
336                             printk("Perfmon: task = 0x%lx\n", (u64) task);
337                             task->thread.regs->msr |= 0x4;
338     #if 0
339                             for(i = 0; i < 11; i++)
340                                     task->thread.pmc[i] = pdata->pmc[i];
341     #endif
342                     } else {
343                             printk("Perfmon: task not found\n");
344                             read_unlock(&tasklist_lock);
345                             return -1;
346                     }
347             }
348             read_unlock(&tasklist_lock);
349
350             /* Stop counters on all processors -- blocking */
351             pmc_stop(NULL);
352             smp_call_function(pmc_stop, (void *)NULL, 0, 1);
353
354             for (i=0; i<MAX_PACAS; ++i) {
355                     paca[i].prof_mode = PMC_STATE_TRACE_USER;
356             }
357             perfmon_base.state = PMC_STATE_TRACE_USER;
358             mb();
359
360             pmc_configure((void *)perfdata);
```

```
361         smp_call_function(pmc_configure, (void *)perfdata, 0, 0);
362
363         return 0;
364 }
365
366 void pmc_configure(void *data)
367 {
368         struct paca_struct *lpaca = get_paca();
369         struct perfmon_struct *perfdata = (struct perfmon_struct *)data;
370         struct pmc_struct *pdata = &(perfdata->vdata.pmc);
371         unsigned long cmd_rec, i;
372
373         /* Indicate to hypervisor that we are using the PMCs */
374         if(systemcfg->platform == PLATFORM_ISERIES_LPAR)
375                 lpaca->xLpPacaPtr->xPMCRegsInUse = 1;
376
377         /* Freeze all counters */
378         mtspr( MMCR0, 0x80000000 ); mtspr( MMCR1, 0x00000000 );
379
380         cmd_rec = 0xFFUL << 56;
381         cmd_rec |= perfdata->header.type;
382         *((unsigned long *)(perfmon_base.trace_buffer + perfmon_base.trace_end)) = cmd_rec;
383         perfmon_base.trace_end += 8;
384
385         /* Clear all the PMCs */
386         mtspr( PMC1, 0 ); mtspr( PMC2, 0 ); mtspr( PMC3, 0 );
387         mtspr( PMC4, 0 ); mtspr( PMC5, 0 ); mtspr( PMC6, 0 );
388         mtspr( PMC7, 0 ); mtspr( PMC8, 0 );
389
390         for(i = 0; i < 11; i++)
391                 lpaca->pmc[i]  = pdata->pmc[i];
392
393         mtspr(PMC1, lpaca->pmc[0]); mtspr(PMC2, lpaca->pmc[1]);
394         mtspr(PMC3, lpaca->pmc[2]); mtspr(PMC4, lpaca->pmc[3]);
395         mtspr(PMC5, lpaca->pmc[4]); mtspr(PMC6, lpaca->pmc[5]);
396         mtspr(PMC7, lpaca->pmc[6]); mtspr(PMC8, lpaca->pmc[7]);
397         mtspr(MMCR1, lpaca->pmc[9]); mtspr(MMCRA, lpaca->pmc[10]);
398
399         mb();
400
401         /* Start all counters */
402         mtspr( MMCR0, lpaca->pmc[8]);
403 }
404
```

```
/*
 * proc_pmc.c
 * Copyright (C) 2001 Mike Corrigan & Dave Engebretsen 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       : mikec    : Created
 * 2001/06/05 : engebret : Software event count support.
 * 2001/08/03 : trautman : Added PCI Flight Recorder
 * End Change Activity
 */

#include <asm/proc_fs.h>
#include <asm/paca.h>
#include <asm/iSeries/ItLpPaca.h>
#include <asm/iSeries/ItLpQueue.h>
#include <asm/iSeries/HvCallXm.h>
#include <asm/iSeries/IoHriMainStore.h>
#include <asm/processor.h>
#include <asm/time.h>
#include <asm/iSeries/LparData.h>

#include <linux/config.h>
#include <linux/proc_fs.h>
#include <linux/spinlock.h>
#include <asm/pmc.h>
#include <asm/uaccess.h>
#include <asm/naca.h>
#include <asm/rtas.h>
#include <asm/perfmon.h>

/* pci Flight Recorder AHT */
extern void proc_pciFr_init(struct proc_dir_entry *proc_ppc64_root);

static int proc_pmc_control_mode = 0;

static struct proc_dir_entry *proc_ppc64_root = NULL;
static struct proc_dir_entry *proc_ppc64_pmc_root = NULL;
static struct proc_dir_entry *proc_ppc64_pmc_system_root = NULL;
static struct proc_dir_entry *proc_ppc64_pmc_cpu_root[NR_CPUS] = {NULL, };

static spinlock_t proc_ppc64_lock;
static int proc_ppc64_page_read(char *page, char **start, off_t off,
                                int count, int *eof, void *data);
static void proc_ppc64_create_paca(int num, struct proc_dir_entry *paca_dir);
int proc_ppc64_pmc_find_file(void *data);
int proc_ppc64_pmc_read(char *page, char **start, off_t off,
                        int count, int *eof, char *buffer);
int proc_ppc64_pmc_stab_read(char *page, char **start, off_t off,
                             int count, int *eof, void *data);
int proc_ppc64_pmc_htab_read(char *page, char **start, off_t off,
                             int count, int *eof, void *data);
int proc_ppc64_pmc_profile_read(char *page, char **start, off_t off,
                                int count, int *eof, void *data);
int proc_ppc64_pmc_profile_read(char *page, char **start, off_t off,
                                int count, int *eof, void *data);
int proc_ppc64_pmc_hw_read(char *page, char **start, off_t off,
                           int count, int *eof, void *data);

static struct proc_dir_entry *pmc_proc_root = NULL;

int proc_get_lpevents( char *page, char **start, off_t off, int count, int *eof, void *data);
int proc_reset_lpevents( struct file *file, const char *buffer, unsigned long count, void *data);

int proc_get_titanTod( char *page, char **start, off_t off, int count, int *eof, void *data);

int proc_pmc_get_control( char *page, char **start, off_t off, int count, int *eof, void *data);

int proc_pmc_set_control( struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_mmcr0( struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_mmcr1( struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_mmcra( struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc1(  struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc2(  struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc3(  struct file *file, const char *buffer, unsigned long count, void *data);
```

```
int proc_pmc_set_pmc4(  struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc5(  struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc6(  struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc7(  struct file *file, const char *buffer, unsigned long count, void *data);
int proc_pmc_set_pmc8(  struct file *file, const char *buffer, unsigned long count, void *data);

static loff_t  nacamap_seek( struct file *file, loff_t off, int whence);
static ssize_t nacamap_read( struct file *file, char *buf, size_t nbytes, loff_t *ppos);
static int     nacamap_mmap( struct file *file, struct vm_area_struct *vma );

static struct file_operations nacamap_fops = {
        llseek: nacamap_seek,
        read:   nacamap_read,
        mmap:   nacamap_mmap
};

static ssize_t read_profile(struct file *file, char *buf, size_t count, loff_t *ppos);
static ssize_t write_profile(struct file * file, const char * buf,
                             size_t count, loff_t *ppos);
static ssize_t read_trace(struct file *file, char *buf, size_t count, loff_t *ppos);
static ssize_t write_trace(struct file * file, const char * buf,
                             size_t count, loff_t *ppos);

static struct file_operations proc_profile_operations = {
        read:           read_profile,
        write:          write_profile,
};

static struct file_operations proc_trace_operations = {
        read:           read_trace,
        write:          write_trace,
};

extern struct perfmon_base_struct perfmon_base;

void proc_ppc64_init(void)
{
        unsigned long i;
        struct proc_dir_entry *ent = NULL;
        char buf[256];

        printk("proc_ppc64: Creating /proc/ppc64/pmc\n");

        /*
         * Create the root, system, and cpu directories as follows:
         *   /proc/ppc64/pmc/system
         *   /proc/ppc64/pmc/cpu0
         */
        spin_lock(&proc_ppc64_lock);
        proc_ppc64_root = proc_mkdir("ppc64", 0);
        if (!proc_ppc64_root) return;
        spin_unlock(&proc_ppc64_lock);

        ent = create_proc_entry("naca", S_IFREG|S_IRUGO, proc_ppc64_root);
        if ( ent ) {
                ent->nlink = 1;
                ent->data = naca;
                ent->size = 4096;
                ent->proc_fops = &nacamap_fops;
        }

        ent = create_proc_entry("systemcfg", S_IFREG|S_IRUGO, proc_ppc64_root);
        if ( ent ) {
                ent->nlink = 1;
                ent->data = systemcfg;
                ent->size = 4096;
                ent->proc_fops = &nacamap_fops;
        }

        /* /proc/ppc64/paca/XX -- raw paca contents.  Only readable to root */
        ent = proc_mkdir("paca", proc_ppc64_root);
        if (ent) {
                for (i = 0; i < systemcfg->processorCount; i++)
                        proc_ppc64_create_paca(i, ent);
        }

        /* Placeholder for rtas interfaces. */
        rtas_proc_dir = proc_mkdir("rtas", proc_ppc64_root);

        /* Create the /proc/ppc64/pcifr for the Pci Flight Recorder.     */
        proc_pciFr_init(proc_ppc64_root);

        proc_ppc64_pmc_root = proc_mkdir("pmc", proc_ppc64_root);

        proc_ppc64_pmc_system_root = proc_mkdir("system", proc_ppc64_pmc_root);
        for (i = 0; i < systemcfg->processorCount; i++) {
                sprintf(buf, "cpu%ld", i);
                proc_ppc64_pmc_cpu_root[i] = proc_mkdir(buf, proc_ppc64_pmc_root);
        }

```

```
181
182         /* Create directories for the software counters. */
183         for (i = 0; i < systemcfg->processorCount; i++) {
184                 ent = create_proc_entry("stab", S_IRUGO | S_IWUSR,
185                                         proc_ppc64_pmc_cpu_root[i]);
186                 if (ent) {
187                         ent->nlink = 1;
188                         ent->data = (void *)proc_ppc64_pmc_cpu_root[i];
189                         ent->read_proc = (void *)proc_ppc64_pmc_stab_read;
190                         ent->write_proc = (void *)proc_ppc64_pmc_stab_read;
191                 }
192
193                 ent = create_proc_entry("htab", S_IRUGO | S_IWUSR,
194                                         proc_ppc64_pmc_cpu_root[i]);
195                 if (ent) {
196                         ent->nlink = 1;
197                         ent->data = (void *)proc_ppc64_pmc_cpu_root[i];
198                         ent->read_proc = (void *)proc_ppc64_pmc_htab_read;
199                         ent->write_proc = (void *)proc_ppc64_pmc_htab_read;
200                 }
201         }
202
203         ent = create_proc_entry("stab", S_IRUGO | S_IWUSR,
204                                 proc_ppc64_pmc_system_root);
205         if (ent) {
206                 ent->nlink = 1;
207                 ent->data = (void *)proc_ppc64_pmc_system_root;
208                 ent->read_proc = (void *)proc_ppc64_pmc_stab_read;
209                 ent->write_proc = (void *)proc_ppc64_pmc_stab_read;
210         }
211
212         ent = create_proc_entry("htab", S_IRUGO | S_IWUSR,
213                                 proc_ppc64_pmc_system_root);
214         if (ent) {
215                 ent->nlink = 1;
216                 ent->data = (void *)proc_ppc64_pmc_system_root;
217                 ent->read_proc = (void *)proc_ppc64_pmc_htab_read;
218                 ent->write_proc = (void *)proc_ppc64_pmc_htab_read;
219         }
220
221         ent = create_proc_entry("profile", S_IWUSR | S_IRUGO, proc_ppc64_pmc_system_root);
222         if (ent) {
223                 ent->nlink = 1;
224                 ent->proc_fops = &proc_profile_operations;
225                 /* ent->size = (1+prof_len) * sizeof(unsigned int); */
226         }
227
228         ent = create_proc_entry("trace", S_IWUSR | S_IRUGO, proc_ppc64_pmc_system_root);
229         if (ent) {
230                 ent->nlink = 1;
231                 ent->proc_fops = &proc_trace_operations;
232                 /* ent->size = (1+prof_len) * sizeof(unsigned int); */
233         }
234
235         /* Create directories for the hardware counters. */
236         for (i = 0; i < systemcfg->processorCount; i++) {
237                 ent = create_proc_entry("hardware", S_IRUGO | S_IWUSR,
238                                         proc_ppc64_pmc_cpu_root[i]);
239                 if (ent) {
240                         ent->nlink = 1;
241                         ent->data = (void *)proc_ppc64_pmc_cpu_root[i];
242                         ent->read_proc = (void *)proc_ppc64_pmc_hw_read;
243                         ent->write_proc = (void *)proc_ppc64_pmc_hw_read;
244                 }
245         }
246
247         ent = create_proc_entry("hardware", S_IRUGO | S_IWUSR,
248                                 proc_ppc64_pmc_system_root);
249         if (ent) {
250                 ent->nlink = 1;
251                 ent->data = (void *)proc_ppc64_pmc_system_root;
252                 ent->read_proc = (void *)proc_ppc64_pmc_hw_read;
253                 ent->write_proc = (void *)proc_ppc64_pmc_hw_read;
254         }
255 }
256
257 /* Read a page of raw data.  "data" points to the start addr.
258  * Intended as a proc read function.
259  */
260 static int proc_ppc64_page_read(char *page, char **start, off_t off,
261                                 int count, int *eof, void *data)
262 {
263         int len = PAGE_SIZE - off;
264         char *p = (char *)data;
265
266         if (len > count)
267                 len = count;
268         if (len <= 0)
269                 return 0;
270         /* Rely on a "hack" in fs/proc/generic.c.
```

```
271             * If we could return a ptr to our own data this would be
272             * trivial (currently *start must be either an offset, or
273             * point into the given page).
274             */
275            memcpy(page, p+off, len);
276            *start = (char *)(unsigned long)len;
277            return len;
278    }
279
280    /* NOTE: since paca data is always in flux the values will never be a consistant set.
281     * In theory it could be made consistent if we made the corresponding cpu
282     * copy the page for us (via an IPI).  Probably not worth it.
283     *
284     */
285    static void proc_ppc64_create_paca(int num, struct proc_dir_entry *paca_dir)
286    {
287            struct proc_dir_entry *ent;
288            struct paca_struct *lpaca = paca + num;
289            char buf[16];
290
291            sprintf(buf, "%02x", num);
292            ent = create_proc_read_entry(buf, S_IRUSR, paca_dir, proc_ppc64_page_read, lpaca);
293    }
294
295    /*
296     * Find the requested 'file' given a proc token.
297     *
298     * Inputs: void * data: proc token
299     * Output: int        : (0, ..., +N) = CPU number.
300     *                      −1           = System.
301     */
302    int proc_ppc64_pmc_find_file(void *data)
303    {
304            int i;
305
306            if ((unsigned long)data ==
307               (unsigned long) proc_ppc64_pmc_system_root) {
308                    return(−1);
309            } else {
310                    for (i = 0; i < systemcfg->processorCount; i++) {
311                            if ((unsigned long)data ==
312                               (unsigned long)proc_ppc64_pmc_cpu_root[i]) {
313                                    return(i);
314                            }
315                    }
316            }
317
318            /* On error, just default to a type of system. */
319            printk("proc_ppc64_pmc_find_file: failed to find file token.\n");
320            return(−1);
321    }
322
323    int
324    proc_ppc64_pmc_read(char *page, char **start, off_t off,
325                        int count, int *eof, char *buffer)
326    {
327            int buffer_size, n;
328
329            if (count < 0) return 0;
330
331            if (buffer == NULL) {
332                    *eof = 1;
333                    return 0;
334            }
335
336            /* Check for read beyond EOF */
337            buffer_size = n = strlen(buffer);
338            if (off >= buffer_size) {
339                    *eof = 1;
340                    return 0;
341            }
342            if (n > (buffer_size − off)) n = buffer_size − off;
343
344            /* Never return more than was requested */
345            if (n > count) {
346                    n = count;
347            } else {
348                    *eof = 1;
349            }
350
351            memcpy(page, buffer + off, n);
352
353            *start = page;
354
355            return n;
356    }
357
358    int
359    proc_ppc64_pmc_stab_read(char *page, char **start, off_t off,
360                             int count, int *eof, void *data)
```

```
{
        int n, file;
        char *buffer = NULL;

        if (count < 0) return 0;
        spin_lock(&proc_ppc64_lock);

        /* Figure out which file is being request. */
        file = proc_ppc64_pmc_find_file(data);

        /* Update the counters and the text buffer representation. */
        buffer = ppc64_pmc_stab(file);

        /* Put the data into the requestor's buffer. */
        n = proc_ppc64_pmc_read(page, start, off, count, eof, buffer);

        spin_unlock(&proc_ppc64_lock);
        return n;
}

int
proc_ppc64_pmc_htab_read(char *page, char **start, off_t off,
                         int count, int *eof, void *data)
{
        int n, file;
        char *buffer = NULL;

        if (count < 0) return 0;
        spin_lock(&proc_ppc64_lock);

        /* Figure out which file is being request. */
        file = proc_ppc64_pmc_find_file(data);

        /* Update the counters and the text buffer representation. */
        buffer = ppc64_pmc_htab(file);

        /* Put the data into the requestor's buffer. */
        n = proc_ppc64_pmc_read(page, start, off, count, eof, buffer);

        spin_unlock(&proc_ppc64_lock);
        return n;
}

static ssize_t read_profile(struct file *file, char *buf,
                            size_t count, loff_t *ppos)
{
        unsigned long p = *ppos;
        ssize_t read;
        char * pnt;
        unsigned int sample_step = 4;

        if (p >= (perfmon_base.profile_length+1)) return 0;
        if (count > (perfmon_base.profile_length+1) − p)
                count = (perfmon_base.profile_length+1) − p;
        read = 0;

        while (p < sizeof(unsigned int) && count > 0) {
                put_user(*((char *)(&sample_step)+p),buf);
                buf++; p++; count−−; read++;
        }
        pnt = (char *)(perfmon_base.profile_buffer) + p − sizeof(unsigned int);
        copy_to_user(buf,(void *)pnt,count);
        read += count;
        *ppos += read;
        return read;
}

static ssize_t read_trace(struct file *file, char *buf,
                            size_t count, loff_t *ppos)
{
        unsigned long p = *ppos;
        ssize_t read;
        char * pnt;

        if (p >= (perfmon_base.trace_length)) return 0;
        if (count > (perfmon_base.trace_length) − p)
                count = (perfmon_base.trace_length) − p;
        read = 0;

        pnt = (char *)(perfmon_base.trace_buffer) + p; //  − sizeof(unsigned int);
        copy_to_user(buf,(void *)pnt,count);
        read += count;
        *ppos += read;
        return read;
}

static ssize_t write_trace(struct file * file, const char * buf,
                            size_t count, loff_t *ppos)
{
}
```

```
451
452 static ssize_t write_profile(struct file * file, const char * buf,
453                               size_t count, loff_t *ppos)
454 {
455 }
456
457 int
458 proc_ppc64_pmc_hw_read(char *page, char **start, off_t off,
459                               int count, int *eof, void *data)
460 {
461         int n, file;
462         char *buffer = NULL;
463
464         if (count < 0) return 0;
465         spin_lock(&proc_ppc64_lock);
466
467         /* Figure out which file is being request. */
468         file = proc_ppc64_pmc_find_file(data);
469
470         /* Update the counters and the text buffer representation. */
471         buffer = ppc64_pmc_hw(file);
472
473         /* Put the data into the requestor's buffer. */
474         n = proc_ppc64_pmc_read(page, start, off, count, eof, buffer);
475
476         spin_unlock(&proc_ppc64_lock);
477         return n;
478 }
479
480 /*
481  * DRENG the remainder of these functions still need work ...
482  */
483 void pmc_proc_init(struct proc_dir_entry *iSeries_proc)
484 {
485     struct proc_dir_entry *ent = NULL;
486
487     ent = create_proc_entry("lpevents", S_IFREG|S_IRUGO, iSeries_proc);
488     if (!ent) return;
489     ent->nlink = 1;
490     ent->data = (void *)0;
491     ent->read_proc = proc_get_lpevents;
492     ent->write_proc = proc_reset_lpevents;
493
494     ent = create_proc_entry("titanTod", S_IFREG|S_IRUGO, iSeries_proc);
495     if (!ent) return;
496     ent->nlink = 1;
497     ent->data = (void *)0;
498     ent->size = 0;
499     ent->read_proc = proc_get_titanTod;
500     ent->write_proc = NULL;
501
502     pmc_proc_root = proc_mkdir("pmc", iSeries_proc);
503     if (!pmc_proc_root) return;
504
505     ent = create_proc_entry("control", S_IFREG|S_IRUSR|S_IWUSR, pmc_proc_root);
506     if (!ent) return;
507     ent->nlink = 1;
508     ent->data = (void *)0;
509     ent->read_proc = proc_pmc_get_control;
510     ent->write_proc = proc_pmc_set_control;
511
512 }
513
514 static int pmc_calc_metrics( char *page, char **start, off_t off, int count, int *eof, int len)
515 {
516         if ( len <= off+count)
517                 *eof = 1;
518         *start = page+off;
519         len −= off;
520         if ( len > count )
521                 len = count;
522         if ( len < 0 )
523                 len = 0;
524         return len;
525 }
526
527 static char * lpEventTypes[9] = {
528         "Hypervisor\t\t",
529         "Machine Facilities\t",
530         "Session Manager\t",
531         "SPD I/O\t\t",
532         "Virtual Bus\t\t",
533         "PCI I/O\t\t",
534         "RIO I/O\t\t",
535         "Virtual Lan\t\t",
536         "Virtual I/O\t\t"
537         };
538
539
540 int proc_get_lpevents
```

```
(char *page, char **start, off_t off, int count, int *eof, void *data)
{
        unsigned i;
        int len = 0;

        len += sprintf( page+len, "LpEventQueue 0\n" );
        len += sprintf( page+len, " events processed:\t%lu\n",
                        (unsigned long)xItLpQueue.xLpIntCount );
        for (i=0; i<9; ++i) {
                len += sprintf( page+len, "  %s %10lu\n",
                        lpEventTypes[i],
                        (unsigned long)xItLpQueue.xLpIntCountByType[i] );
        }
        len += sprintf( page+len, "\n  events processed by processor:\n" );
        for (i=0; i<systemcfg->processorCount; ++i) {
                len += sprintf( page+len, "  CPU%02d  %10u\n",
                        i, paca[i].lpEvent_count );
        }

        return pmc_calc_metrics( page, start, off, count, eof, len );

}

int proc_reset_lpevents( struct file *file, const char *buffer, unsigned long count, void *data )
{
        return count;
}

static unsigned long startTitan = 0;
static unsigned long startTb = 0;


int proc_get_titanTod
(char *page, char **start, off_t off, int count, int *eof, void *data)
{
        int len = 0;
        unsigned long tb0, titan_tod;

        tb0 = get_tb();
        titan_tod = HvCallXm_loadTod();

        len += sprintf( page+len, "Titan\n" );
        len += sprintf( page+len, " time base =      %016lx\n", tb0 );
        len += sprintf( page+len, " titan tod =      %016lx\n", titan_tod );
        len += sprintf( page+len, " xProcFreq =      %016x\n", xIoHriProcessorVpd[0].xProcFreq );
        len += sprintf( page+len, " xTimeBaseFreq =    %016x\n", xIoHriProcessorVpd[0].xTimeBaseFreq );
        len += sprintf( page+len, " tb_ticks_per_jiffy = %lu\n", tb_ticks_per_jiffy );
        len += sprintf( page+len, " tb_ticks_per_usec  = %lu\n", tb_ticks_per_usec );

        if ( !startTitan ) {
                startTitan = titan_tod;
                startTb = tb0;
        }
        else {
                unsigned long titan_usec = (titan_tod − startTitan) >> 12;
                unsigned long tb_ticks = (tb0 − startTb);
                unsigned long titan_jiffies = titan_usec / (1000000/HZ);
                unsigned long titan_jiff_usec = titan_jiffies * (1000000/HZ);
                unsigned long titan_jiff_rem_usec = titan_usec − titan_jiff_usec;
                unsigned long tb_jiffies = tb_ticks / tb_ticks_per_jiffy;
                unsigned long tb_jiff_ticks = tb_jiffies * tb_ticks_per_jiffy;
                unsigned long tb_jiff_rem_ticks = tb_ticks − tb_jiff_ticks;
                unsigned long tb_jiff_rem_usec = tb_jiff_rem_ticks / tb_ticks_per_usec;
                unsigned long new_tb_ticks_per_jiffy = (tb_ticks * (1000000/HZ))/titan_usec;

                len += sprintf( page+len, " titan elapsed = %lu uSec\n", titan_usec);
                len += sprintf( page+len, " tb elapsed    = %lu ticks\n", tb_ticks);
                len += sprintf( page+len, " titan jiffies = %lu.%04lu \n", titan_jiffies, titan_jiff_rem_usec );

                len += sprintf( page+len, " tb jiffies    = %lu.%04lu\n", tb_jiffies, tb_jiff_rem_usec );
                len += sprintf( page+len, " new tb_ticks_per_jiffy = %lu\n", new_tb_ticks_per_jiffy );

        }

        return pmc_calc_metrics( page, start, off, count, eof, len );
}

int proc_pmc_get_control
(char *page, char **start, off_t off, int count, int *eof, void *data)
{
        int len = 0;

        if ( proc_pmc_control_mode == PMC_CONTROL_CPI ) {
                unsigned long mach_cycles   = mfspr( PMC5 );
                unsigned long inst_complete = mfspr( PMC4 );
                unsigned long inst_dispatch = mfspr( PMC3 );
                unsigned long thread_active_run = mfspr( PMC1 );
                unsigned long thread_active  = mfspr( PMC2 );
                unsigned long cpi = 0;
                unsigned long cpithou = 0;
```

```
630                     unsigned long remain;
631
632                     if ( inst_complete ) {
633                             cpi = thread_active_run / inst_complete;
634                             remain = thread_active_run % inst_complete;
635                             if ( inst_complete > 1000000 )
636                                     cpithou = remain / ( inst_complete / 1000 );
637                             else
638                                     cpithou = ( remain * 1000 ) / inst_complete;
639                     }
640                     len += sprintf( page+len, "PMC CPI Mode\nRaw Counts\n" );
641                     len += sprintf( page+len, "machine cycles       : %12lu\n", mach_cycles );
642                     len += sprintf( page+len, "thread active cycles   : %12lu\n\n", thread_active );
643
644                     len += sprintf( page+len, "instructions completed  : %12lu\n", inst_complete );
645                     len += sprintf( page+len, "instructions dispatched : %12lu\n", inst_dispatch );
646                     len += sprintf( page+len, "thread active run cycles : %12lu\n", thread_active_run );
647
648                     len += sprintf( page+len, "thread active run cycles/instructions completed\n" );
649                     len += sprintf( page+len, "CPI = %lu.%03lu\n", cpi, cpithou );
650
651             }
652             else if ( proc_pmc_control_mode == PMC_CONTROL_TLB ) {
653                     len += sprintf( page+len, "PMC TLB Mode\n" );
654                     len += sprintf( page+len, "I−miss count         : %12lu\n", mfspr( PMC1 ) );
655                     len += sprintf( page+len, "I−miss latency       : %12lu\n", mfspr( PMC2 ) );
656                     len += sprintf( page+len, "D−miss count         : %12lu\n", mfspr( PMC3 ) );
657                     len += sprintf( page+len, "D−miss latency       : %12lu\n", mfspr( PMC4 ) );
658                     len += sprintf( page+len, "IERAT miss count     : %12lu\n", mfspr( PMC5 ) );
659                     len += sprintf( page+len, "D−reference count    : %12lu\n", mfspr( PMC6 ) );
660                     len += sprintf( page+len, "miss PTEs searched   : %12lu\n", mfspr( PMC7 ) );
661                     len += sprintf( page+len, "miss >8 PTEs searched  : %12lu\n", mfspr( PMC8 ) );
662             }
663             /* IMPLEMENT ME */
664             return pmc_calc_metrics( page, start, off, count, eof, len );
665     }
666
667     unsigned long proc_pmc_conv_int( const char *buf, unsigned count )
668     {
669             const char * p;
670             char b0, b1;
671             unsigned v, multiplier, mult, i;
672             unsigned long val;
673             multiplier = 10;
674             p = buf;
675             if ( count >= 3 ) {
676                     b0 = buf[0];
677                     b1 = buf[1];
678                     if ( ( b0 == '0' ) &&
679                          ( ( b1 == 'x' ) || ( b1 == 'X' ) ) ) {
680                             p = buf + 2;
681                             count −= 2;
682                             multiplier = 16;
683                     }
684
685             }
686             val = 0;
687             for ( i=0; i<count; ++i ) {
688                     b0 = *p++;
689                     v = 0;
690                     mult = multiplier;
691                     if ( ( b0 >= '0' ) && ( b0 <= '9' ) )
692                             v = b0 − '0';
693                     else if ( multiplier == 16 ) {
694                             if ( ( b0 >= 'a' ) && ( b0 <= 'f' ) )
695                                     v = b0 − 'a' + 10;
696                             else if ( ( b0 >= 'A' ) && ( b0 <= 'F' ) )
697                                     v = b0 − 'A' + 10;
698                             else
699                                     mult = 1;
700                     }
701                     else
702                             mult = 1;
703                     val *= mult;
704                     val += v;
705             }
706
707             return val;
708
709     }
710
711     static inline void proc_pmc_stop(void)
712     {
713             /* Freeze all counters, leave everything else alone */
714             mtspr( MMCR0, mfspr( MMCR0 ) | 0x80000000 );
715     }
716
717     static inline void proc_pmc_start(void)
718     {
719             /* Unfreeze all counters, leave everything else alone */
```

```
720         mtspr( MMCR0, mfspr( MMCR0 ) & ~0x80000000 );
721
722 }
723
724 static inline void proc_pmc_reset(void)
725 {
726         /* Clear all the PMCs to zeros
727          * Assume a "stop" has already frozen the counters
728          * Clear all the PMCs
729          */
730         mtspr( PMC1, 0 );
731         mtspr( PMC2, 0 );
732         mtspr( PMC3, 0 );
733         mtspr( PMC4, 0 );
734         mtspr( PMC5, 0 );
735         mtspr( PMC6, 0 );
736         mtspr( PMC7, 0 );
737         mtspr( PMC8, 0 );
738
739 }
740
741 static inline void proc_pmc_cpi(void)
742 {
743         /* Configure the PMC registers to count cycles and instructions */
744         /* so we can compute cpi */
745         /*
746          * MMCRA[30]    = 1     Don't count in wait state (CTRL[31]=0)
747          * MMCR0[6]     = 1     Freeze counters when any overflow
748          * MMCR0[19:25] = 0x01  PMC1 counts Thread Active Run Cycles
749          * MMCR0[26:31] = 0x05  PMC2 counts Thread Active Cycles
750          * MMCR1[0:4]   = 0x07  PMC3 counts Instructions Dispatched
751          * MMCR1[5:9]   = 0x03  PMC4 counts Instructions Completed
752          * MMCR1[10:14] = 0x06  PMC5 counts Machine Cycles
753          *
754          */
755
756         proc_pmc_control_mode = PMC_CONTROL_CPI;
757
758         /* Indicate to hypervisor that we are using the PMCs */
759         get_paca()->xLpPacaPtr->xPMCRegsInUse = 1;
760
761         /* Freeze all counters */
762         mtspr( MMCR0, 0x80000000 );
763         mtspr( MMCR1, 0x00000000 );
764
765         /* Clear all the PMCs */
766         mtspr( PMC1, 0 );
767         mtspr( PMC2, 0 );
768         mtspr( PMC3, 0 );
769         mtspr( PMC4, 0 );
770         mtspr( PMC5, 0 );
771         mtspr( PMC6, 0 );
772         mtspr( PMC7, 0 );
773         mtspr( PMC8, 0 );
774
775         /* Freeze counters in Wait State (CTRL[31]=0) */
776         mtspr( MMCRA, 0x00000002 );
777
778         /* PMC3<−0x07, PMC4<−0x03, PMC5<−0x06 */
779         mtspr( MMCR1, 0x38cc0000 );
780
781         mb();
782
783         /* PMC1<−0x01, PMC2<−0x05
784          * Start all counters
785          */
786         mtspr( MMCR0, 0x02000045 );
787
788 }
789
790 static inline void proc_pmc_tlb(void)
791 {
792         /* Configure the PMC registers to count tlb misses  */
793         /*
794          * MMCR0[6]     = 1     Freeze counters when any overflow
795          * MMCR0[19:25] = 0x55  Group count
796          *   PMC1 counts  I misses
797          *   PMC2 counts  I miss duration (latency)
798          *   PMC3 counts  D misses
799          *   PMC4 counts  D miss duration (latency)
800          *   PMC5 counts  IERAT misses
801          *   PMC6 counts  D references (including PMC7)
802          *   PMC7 counts  miss PTEs searched
803          *   PMC8 counts  miss >8 PTEs searched
804          *
805          */
806
807         proc_pmc_control_mode = PMC_CONTROL_TLB;
808
809         /* Indicate to hypervisor that we are using the PMCs */
```

```
810         get_paca()->xLpPacaPtr->xPMCRegsInUse = 1;
811
812         /* Freeze all counters */
813         mtspr( MMCR0, 0x80000000 );
814         mtspr( MMCR1, 0x00000000 );
815
816         /* Clear all the PMCs */
817         mtspr( PMC1, 0 );
818         mtspr( PMC2, 0 );
819         mtspr( PMC3, 0 );
820         mtspr( PMC4, 0 );
821         mtspr( PMC5, 0 );
822         mtspr( PMC6, 0 );
823         mtspr( PMC7, 0 );
824         mtspr( PMC8, 0 );
825
826         mtspr( MMCRA, 0x00000000 );
827
828         mb();
829
830         /* PMC1<-0x55
831          * Start all counters
832          */
833         mtspr( MMCR0, 0x02001540 );
834
835 }
836
837 int proc_pmc_set_control( struct file *file, const char *buffer, unsigned long count, void *data )
838 {
839         if      ( ! strncmp( buffer, "stop", 4 ) )
840                 proc_pmc_stop();
841         else if ( ! strncmp( buffer, "start", 5 ) )
842                 proc_pmc_start();
843         else if ( ! strncmp( buffer, "reset", 5 ) )
844                 proc_pmc_reset();
845         else if ( ! strncmp( buffer, "cpi", 3 ) )
846                 proc_pmc_cpi();
847         else if ( ! strncmp( buffer, "tlb", 3 ) )
848                 proc_pmc_tlb();
849
850         /* IMPLEMENT ME */
851         return count;
852 }
853
854 int proc_pmc_set_mmcr0( struct file *file, const char *buffer, unsigned long count, void *data )
855 {
856         unsigned long v;
857         v = proc_pmc_conv_int( buffer, count );
858         v = v & ~0x04000000;    /* Don't allow interrupts for now */
859         if ( v & ~0x80000000 )  /* Inform hypervisor we are using PMCs */
860                 get_paca()->xLpPacaPtr->xPMCRegsInUse = 1;
861         else
862                 get_paca()->xLpPacaPtr->xPMCRegsInUse = 0;
863         mtspr( MMCR0, v );
864
865         return count;
866 }
867
868 int proc_pmc_set_mmcr1( struct file *file, const char *buffer, unsigned long count, void *data )
869 {
870         unsigned long v;
871         v = proc_pmc_conv_int( buffer, count );
872         mtspr( MMCR1, v );
873
874         return count;
875 }
876
877 int proc_pmc_set_mmcra( struct file *file, const char *buffer, unsigned long count, void *data )
878 {
879         unsigned long v;
880         v = proc_pmc_conv_int( buffer, count );
881         v = v & ~0x00008000;    /* Don't allow interrupts for now */
882         mtspr( MMCRA, v );
883
884         return count;
885 }
886
887
888 int proc_pmc_set_pmc1( struct file *file, const char *buffer, unsigned long count, void *data )
889 {
890         unsigned long v;
891         v = proc_pmc_conv_int( buffer, count );
892         mtspr( PMC1, v );
893
894         return count;
895 }
896
897 int proc_pmc_set_pmc2( struct file *file, const char *buffer, unsigned long count, void *data )
898 {
899         unsigned long v;
```

```
900         v = proc_pmc_conv_int( buffer, count );
901         mtspr( PMC2, v );
902
903         return count;
904 }
905
906 int proc_pmc_set_pmc3( struct file *file, const char *buffer, unsigned long count, void *data )
907 {
908         unsigned long v;
909         v = proc_pmc_conv_int( buffer, count );
910         mtspr( PMC3, v );
911
912         return count;
913 }
914
915 int proc_pmc_set_pmc4( struct file *file, const char *buffer, unsigned long count, void *data )
916 {
917         unsigned long v;
918         v = proc_pmc_conv_int( buffer, count );
919         mtspr( PMC4, v );
920
921         return count;
922 }
923
924 int proc_pmc_set_pmc5( struct file *file, const char *buffer, unsigned long count, void *data )
925 {
926         unsigned long v;
927         v = proc_pmc_conv_int( buffer, count );
928         mtspr( PMC5, v );
929
930         return count;
931 }
932
933 int proc_pmc_set_pmc6( struct file *file, const char *buffer, unsigned long count, void *data )
934 {
935         unsigned long v;
936         v = proc_pmc_conv_int( buffer, count );
937         mtspr( PMC6, v );
938
939         return count;
940 }
941
942 int proc_pmc_set_pmc7( struct file *file, const char *buffer, unsigned long count, void *data )
943 {
944         unsigned long v;
945         v = proc_pmc_conv_int( buffer, count );
946         mtspr( PMC7, v );
947
948         return count;
949 }
950
951 int proc_pmc_set_pmc8( struct file *file, const char *buffer, unsigned long count, void *data )
952 {
953         unsigned long v;
954         v = proc_pmc_conv_int( buffer, count );
955         mtspr( PMC8, v );
956
957         return count;
958 }
959
960 static loff_t nacamap_seek( struct file *file, loff_t off, int whence)
961 {
962         loff_t new;
963         struct proc_dir_entry *dp;
964
965         dp = file->f_dentry->d_inode->u.generic_ip;
966
967         switch(whence) {
968         case 0:
969                 new = off;
970                 break;
971         case 1:
972                 new = file->f_pos + off;
973                 break;
974         case 2:
975                 new = dp->size + off;
976                 break;
977         default:
978                 return -EINVAL;
979         }
980         if ( new < 0 || new > dp->size )
981                 return -EINVAL;
982         return (file->f_pos = new);
983 }
984
985 static ssize_t nacamap_read( struct file *file, char *buf, size_t nbytes, loff_t *ppos)
986 {
987         unsigned pos = *ppos;
988         struct proc_dir_entry *dp;
989
```

```
990         dp = file->f_dentry->d_inode->u.generic_ip;
991
992         if ( pos >= dp->size )
993                 return 0;
994         if ( nbytes >= dp->size )
995                 nbytes = dp->size;
996         if ( pos + nbytes > dp->size )
997                 nbytes = dp->size - pos;
998
999         copy_to_user( buf, (char *)dp->data + pos, nbytes );
1000        *ppos = pos + nbytes;
1001        return nbytes;
1002 }
1003
1004 static int nacamap_mmap( struct file *file, struct vm_area_struct *vma )
1005 {
1006        struct proc_dir_entry *dp;
1007
1008        dp = file->f_dentry->d_inode->u.generic_ip;
1009
1010        vma->vm_flags |= VM_SHM | VM_LOCKED;
1011
1012        if ((vma->vm_end - vma->vm_start) > dp->size)
1013                return -EINVAL;
1014
1015        remap_page_range( vma->vm_start, __pa(dp->data), dp->size, vma->vm_page_prot );
1016        return 0;
1017 }
1018
```

```
/*
 * Copyright (C) 2001 Anton Blanchard <anton@au.ibm.com>, IBM
 *
 * 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.
 *
 * Communication to userspace based on kernel/printk.c
 */

#include <linux/types.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/poll.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <linux/vmalloc.h>

#include <asm/uaccess.h>
#include <asm/io.h>
#include <asm/rtas.h>
#include <asm/prom.h>

#if 0
#define DEBUG(A...)     printk(KERN_ERR A)
#else
#define DEBUG(A...)
#endif

static spinlock_t rtas_log_lock = SPIN_LOCK_UNLOCKED;

DECLARE_WAIT_QUEUE_HEAD(rtas_log_wait);

#define LOG_NUMBER              64              /* must be a power of two */
#define LOG_NUMBER_MASK         (LOG_NUMBER-1)

static char *rtas_log_buf;
static unsigned long rtas_log_start;
static unsigned long rtas_log_size;

static int surveillance_requested;
static unsigned int rtas_event_scan_rate;
static unsigned int rtas_error_log_max;

#define EVENT_SCAN_ALL_EVENTS   0xf0000000
#define SURVEILLANCE_TOKEN      9000
#define SURVEILLANCE_TIMEOUT    1
#define SURVEILLANCE_SCANRATE   1

/*
 * Since we use 32 bit RTAS, the physical address of this must be below
 * 4G or else bad things happen. Allocate this in the kernel data and
 * make it big enough.
 */
#define RTAS_ERROR_LOG_MAX 1024
static unsigned char logdata[RTAS_ERROR_LOG_MAX];

static int rtas_log_open(struct inode * inode, struct file * file)
{
        return 0;
}

static int rtas_log_release(struct inode * inode, struct file * file)
{
        return 0;
}

static ssize_t rtas_log_read(struct file * file, char * buf,
                         size_t count, loff_t *ppos)
{
        int error;
        char *tmp;
        unsigned long offset;

        if (!buf || count < rtas_error_log_max)
                return -EINVAL;

        count = rtas_error_log_max;

        error = verify_area(VERIFY_WRITE, buf, count);
        if (error)
                return -EINVAL;

        tmp = kmalloc(rtas_error_log_max, GFP_KERNEL);
        if (!tmp)
                return -ENOMEM;

        error = wait_event_interruptible(rtas_log_wait, rtas_log_size);
```

```
91          if (error)
92                  goto out;
93
94          spin_lock(&rtas_log_lock);
95          offset = rtas_error_log_max * (rtas_log_start & LOG_NUMBER_MASK);
96          memcpy(tmp, &rtas_log_buf[offset], count);
97          rtas_log_start += 1;
98          rtas_log_size −= 1;
99          spin_unlock(&rtas_log_lock);
100
101         error = copy_to_user(buf, tmp, count) ? −EFAULT : count;
102 out:
103         kfree(tmp);
104         return error;
105 }
106
107 static unsigned int rtas_log_poll(struct file *file, poll_table * wait)
108 {
109         poll_wait(file, &rtas_log_wait, wait);
110         if (rtas_log_size)
111                 return POLLIN | POLLRDNORM;
112         return 0;
113 }
114
115 struct file_operations proc_rtas_log_operations = {
116         read:           rtas_log_read,
117         poll:           rtas_log_poll,
118         open:           rtas_log_open,
119         release:        rtas_log_release,
120 };
121
122
123 #define RTAS_ERR KERN_ERR "RTAS: "
124
125 /* Extended error log header (12 bytes) */
126 struct exthdr {
127         unsigned int valid:1;
128         unsigned int unrecoverable:1;
129         unsigned int recoverable:1;
130         unsigned int unrecoverable_bypassed:1;  /* i.e. degraded performance */
131         unsigned int predictive:1;
132         unsigned int newlog:1;
133         unsigned int bigendian:1;               /* always 1 */
134         unsigned int /* reserved */:1;
135
136         unsigned int platform_specific:1;       /* only in version 3+ */
137         unsigned int /* reserved */:3;
138         unsigned int platform_value:4;          /* valid iff platform_specific */
139
140         unsigned int power_pc:1;                /* always 1 */
141         unsigned int /* reserved */:2;
142         unsigned int addr_invalid:1;            /* failing_address is invalid */
143         unsigned int format_type:4;
144 #define EXTLOG_FMT_CPU 1
145 #define EXTLOG_FMT_MEMORY 2
146 #define EXTLOG_FMT_IO 3
147 #define EXTLOG_FMT_POST 4
148 #define EXTLOG_FMT_ENV 5
149 #define EXTLOG_FMT_POW 6
150 #define EXTLOG_FMT_IBMDIAG 12
151 #define EXTLOG_FMT_IBMSP 13
152
153         /* This group is in version 3+ only */
154         unsigned int non_hardware:1;            /* Firmware or software is suspect */
155         unsigned int hot_plug:1;                /* Failing component may be hot plugged */
156         unsigned int group_failure:1;           /* Group of components should be replaced */
157         unsigned int /* reserved */:1;
158
159         unsigned int residual:1;                /* Residual error from previous boot (maybe a crash) */
160         unsigned int boot:1;                    /* Error during boot */
161         unsigned int config_change:1;           /* Configuration changed since last boot */
162         unsigned int post:1;                    /* Error during POST */
163
164         unsigned int bcdtime:32;                /* Time of error in BCD HHMMSS00 */
165         unsigned int bcddate:32;                /* Time of error in BCD YYYYMMDD */
166 };
167
168 struct cpuhdr {
169         unsigned int internal:1;
170         unsigned int intcache:1;
171         unsigned int extcache_parity:1; /* or multi−bit ECC */
172         unsigned int extcache_ecc:1;
173         unsigned int sysbus_timeout:1;
174         unsigned int io_timeout:1;
175         unsigned int sysbus_parity:1;
176         unsigned int sysbus_protocol:1;
177         unsigned int cpuid:8;
178         unsigned int element:16;
179         unsigned int failing_address_hi:32;
180         unsigned int failing_address_lo:32;
```

```
181
182         /* These are version 4+ */
183         unsigned int try_reboot:1;      /* 1 => fault may be fixed by reboot */
184         unsigned int /* reserved */:7;
185         /* 15 bytes reserved here */
186 };
187
188 struct memhdr {
189         unsigned int uncorrectable:1;
190         unsigned int ECC:1;
191         unsigned int threshold_exceeded:1;
192         unsigned int control_internal:1;
193         unsigned int bad_address:1;
194         unsigned int bad_data:1;
195         unsigned int bus:1;
196         unsigned int timeout:1;
197         unsigned int sysbus_parity:1;
198         unsigned int sysbus_timeout:1;
199         unsigned int sysbus_protocol:1;
200         unsigned int hostbridge_timeout:1;
201         unsigned int hostbridge_parity:1;
202         unsigned int reserved1:1;
203         unsigned int support:1;
204         unsigned int sysbus_internal:1;
205         unsigned int mem_controller_detected:8; /* who detected fault? */
206         unsigned int mem_controller_faulted:8;  /* who caused fault? */
207         unsigned int failing_address_hi:32;
208         unsigned int failing_address_lo:32;
209         unsigned int ecc_syndrome:16;
210         unsigned int memory_card:8;
211         unsigned int reserved2:8;
212         unsigned int sub_elements:32;           /* one bit per element */
213         unsigned int element:16;
214 };
215
216 struct iohdr {
217         unsigned int bus_addr_parity:1;
218         unsigned int bus_data_parity:1;
219         unsigned int bus_timeout:1;
220         unsigned int bridge_internal:1;
221         unsigned int non_pci:1;         /* i.e. secondary bus such as ISA */
222         unsigned int mezzanine_addr_parity:1;
223         unsigned int mezzanine_data_parity:1;
224         unsigned int mezzanine_timeout:1;
225
226         unsigned int bridge_via_sysbus:1;
227         unsigned int bridge_via_mezzanine:1;
228         unsigned int bridge_via_expbus:1;
229         unsigned int detected_by_expbus:1;
230         unsigned int expbus_data_parity:1;
231         unsigned int expbus_timeout:1;
232         unsigned int expbus_connection_failure:1;
233         unsigned int expbus_not_operating:1;
234
235         /* IOA signalling the error */
236         unsigned int pci_sig_busno:8;
237         unsigned int pci_sig_devfn:8;
238         unsigned int pci_sig_deviceid:16;
239         unsigned int pci_sig_vendorid:16;
240         unsigned int pci_sig_revisionid:8;
241         unsigned int pci_sig_slot:8;    /* 00 => system board, ff => multiple */
242
243         /* IOA sending at time of error */
244         unsigned int pci_send_busno:8;
245         unsigned int pci_send_devfn:8;
246         unsigned int pci_send_deviceid:16;
247         unsigned int pci_send_vendorid:16;
248         unsigned int pci_send_revisionid:8;
249         unsigned int pci_send_slot:8;   /* 00 => system board, ff => multiple */
250 };
251
252 struct posthdr {
253         unsigned int firmware:1;
254         unsigned int config:1;
255         unsigned int cpu:1;
256         unsigned int memory:1;
257         unsigned int io:1;
258         unsigned int keyboard:1;
259         unsigned int mouse:1;
260         unsigned int display:1;
261
262         unsigned int ipl_floppy:1;
263         unsigned int ipl_controller:1;
264         unsigned int ipl_cdrom:1;
265         unsigned int ipl_disk:1;
266         unsigned int ipl_net:1;
267         unsigned int ipl_other:1;
268         unsigned int /* reserved */:1;
269         unsigned int firmware_selftest:1;
270
```

```
271         char         devname[12];
272         unsigned int post_code:4;
273         unsigned int firmware_rev:2;
274         unsigned int loc_code:8;        /* currently unused */
275 };
276
277 struct epowhdr {
278         unsigned int epow_sensor_value:32;
279         unsigned int sensor:1;
280         unsigned int power_fault:1;
281         unsigned int fan:1;
282         unsigned int temp:1;
283         unsigned int redundancy:1;
284         unsigned int CUoD:1;
285         unsigned int /* reserved */:2;
286
287         unsigned int general:1;
288         unsigned int power_loss:1;
289         unsigned int power_supply:1;
290         unsigned int power_switch:1;
291         unsigned int /* reserved */:4;
292
293         unsigned int /* reserved */:16;
294         unsigned int sensor_token:32;
295         unsigned int sensor_index:32;
296         unsigned int sensor_value:32;
297         unsigned int sensor_status:32;
298 };
299
300 struct pm_eventhdr {
301         unsigned int event_id:32;
302 };
303
304 struct sphdr {
305         unsigned int ibm:32;    /* "IBM\0" */
306
307         unsigned int timeout:1;
308         unsigned int i2c_bus:1;
309         unsigned int i2c_secondary_bus:1;
310         unsigned int sp_memory:1;
311         unsigned int sp_registers:1;
312         unsigned int sp_communication:1;
313         unsigned int sp_firmware:1;
314         unsigned int sp_hardware:1;
315
316         unsigned int vpd_eeprom:1;
317         unsigned int op_panel:1;
318         unsigned int power_controller:1;
319         unsigned int fan_sensor:1;
320         unsigned int thermal_sensor:1;
321         unsigned int voltage_sensor:1;
322         unsigned int reserved1:2;
323
324         unsigned int serial_port:1;
325         unsigned int nvram:1;
326         unsigned int rtc:1;
327         unsigned int jtag:1;
328         unsigned int tod_battery:1;
329         unsigned int reserved2:1;
330         unsigned int heartbeat:1;
331         unsigned int surveillance:1;
332
333         unsigned int pcn_connection:1;  /* power control network */
334         unsigned int pcn_node:1;
335         unsigned int reserved3:2;
336         unsigned int pcn_access:1;
337         unsigned int reserved:3;
338
339         unsigned int sensor_token:32;   /* zero if undef */
340         unsigned int sensor_index:32;   /* zero if undef */
341 };
342
343
344 static char *severity_names[] = {
345         "NO ERROR", "EVENT", "WARNING", "ERROR_SYNC", "ERROR", "FATAL", "(6)", "(7)"
346 };
347 static char *rtas_disposition_names[] = {
348         "FULLY RECOVERED", "LIMITED RECOVERY", "NOT RECOVERED", "(4)"
349 };
350 static char *entity_names[] = { /* for initiator & targets */
351         "UNKNOWN", "CPU", "PCI", "ISA", "MEMORY", "POWER MANAGEMENT", "HOT PLUG", "(7)", "(8)",
352         "(9)", "(10)", "(11)", "(12)", "(13)", "(14)", "(15)"
353 };
354 static char *error_type[] = {   /* Not all types covered here so need to bounds check */
355         "UNKNOWN", "RETRY", "TCE_ERR", "INTERN_DEV_FAIL",
356         "TIMEOUT", "DATA_PARITY", "ADDR_PARITY", "CACHE_PARITY",
357         "ADDR_INVALID", "ECC_UNCORR", "ECC_CORR",
358 };
359
360 static char *rtas_error_type(int type)
```

```
{
        if (type < 11)
                return error_type[type];
        if (type == 64)
                return "SENSOR";
        if (type >=96 && type <= 159)
                return "POWER";
        return error_type[0];
}

static void printk_cpu_failure(int version, struct exthdr *exthdr, char *data)
{
        struct cpuhdr cpuhdr;

        memcpy(&cpuhdr, data, sizeof(cpuhdr));

        if (cpuhdr.internal) printk(RTAS_ERR "Internal error (not cache)\n");
        if (cpuhdr.intcache) printk(RTAS_ERR "Internal cache\n");
        if (cpuhdr.extcache_parity) printk(RTAS_ERR "External cache parity (or multi−bit)\n");
        if (cpuhdr.extcache_ecc) printk(RTAS_ERR "External cache ECC\n");
        if (cpuhdr.sysbus_timeout) printk(RTAS_ERR "System bus timeout\n");
        if (cpuhdr.io_timeout) printk(RTAS_ERR "I/O timeout\n");
        if (cpuhdr.sysbus_parity) printk(RTAS_ERR "System bus parity\n");
        if (cpuhdr.sysbus_protocol) printk(RTAS_ERR "System bus protocol/transfer\n");
        printk(RTAS_ERR "CPU id: %d\n", cpuhdr.cpuid);
        printk(RTAS_ERR "Failing element: 0x%04x\n", cpuhdr.element);
        if (!exthdr−>addr_invalid)
                printk(RTAS_ERR "Failing address: %08x%08x\n", cpuhdr.failing_address_hi, cpuhdr.failing_address_lo);
        if (version >= 4 && cpuhdr.try_reboot)
                printk(RTAS_ERR "A reboot of the system may correct the problem\n");
}

static void printk_mem_failure(int version, struct exthdr *exthdr, char *data)
{
        struct memhdr memhdr;

        memcpy(&memhdr, data, sizeof(memhdr));
        if (memhdr.uncorrectable) printk(RTAS_ERR "Uncorrectable Memory error\n");
        if (memhdr.ECC) printk(RTAS_ERR "ECC Correctable error\n");
        if (memhdr.threshold_exceeded) printk(RTAS_ERR "Correctable threshold exceeded\n");
        if (memhdr.control_internal) printk(RTAS_ERR "Memory Controller internal error\n");
        if (memhdr.bad_address) printk(RTAS_ERR "Memory Address error\n");
        if (memhdr.bad_data) printk(RTAS_ERR "Memory Data error\n");
        if (memhdr.bus) printk(RTAS_ERR "Memory bus/switch internal error\n");
        if (memhdr.timeout) printk(RTAS_ERR "Memory timeout\n");
        if (memhdr.sysbus_parity) printk(RTAS_ERR "System bus parity\n");
        if (memhdr.sysbus_timeout) printk(RTAS_ERR "System bus timeout\n");
        if (memhdr.sysbus_protocol) printk(RTAS_ERR "System bus protocol/transfer\n");
        if (memhdr.hostbridge_timeout) printk(RTAS_ERR "I/O Host Bridge timeout\n");
        if (memhdr.hostbridge_parity) printk(RTAS_ERR "I/O Host Bridge parity\n");
        if (memhdr.support) printk(RTAS_ERR "System support function error\n");
        if (memhdr.sysbus_internal) printk(RTAS_ERR "System bus internal hardware/switch error\n");
        printk(RTAS_ERR "Memory Controller that detected failure: %d\n", memhdr.mem_controller_detected);
        printk(RTAS_ERR "Memory Controller that faulted: %d\n", memhdr.mem_controller_faulted);
        if (!exthdr−>addr_invalid)
                printk(RTAS_ERR "Failing address: 0x%016x%016x\n", memhdr.failing_address_hi, memhdr.failing_address_lo)
;
        printk(RTAS_ERR "ECC syndrome bits: 0x%04x\n", memhdr.ecc_syndrome);
        printk(RTAS_ERR "Memory Card: %d\n", memhdr.memory_card);
        printk(RTAS_ERR "Failing element: 0x%04x\n", memhdr.element);
        printk(RTAS_ERR "Sub element bits: 0x%08x\n", memhdr.sub_elements);
}

static void printk_io_failure(int version, struct exthdr *exthdr, char *data)
{
        struct iohdr iohdr;

        memcpy(&iohdr, data, sizeof(iohdr));
        if (iohdr.bus_addr_parity) printk(RTAS_ERR "I/O bus address parity\n");
        if (iohdr.bus_data_parity) printk(RTAS_ERR "I/O bus data parity\n");
        if (iohdr.bus_timeout) printk(RTAS_ERR "I/O bus timeout, access or other\n");
        if (iohdr.bridge_internal) printk(RTAS_ERR "I/O bus bridge/device internal\n");
        if (iohdr.non_pci) printk(RTAS_ERR "Signaling IOA is a PCI to non−PCI bridge (e.g. ISA)\n");
        if (iohdr.mezzanine_addr_parity) printk(RTAS_ERR "Mezzanine/System bus address parity\n");
        if (iohdr.mezzanine_data_parity) printk(RTAS_ERR "Mezzanine/System bus data parity\n");
        if (iohdr.mezzanine_timeout) printk(RTAS_ERR "Mezzanine/System bus timeout, transfer or protocol\n");
        if (iohdr.bridge_via_sysbus) printk(RTAS_ERR "Bridge is connected to system bus\n");
        if (iohdr.bridge_via_mezzanine) printk(RTAS_ERR "Bridge is connected to memory controller via mezzanine bus\n");
        if (iohdr.bridge_via_expbus) printk(RTAS_ERR "Bridge is connected to I/O expansion bus\n");
        if (iohdr.detected_by_expbus) printk(RTAS_ERR "Error on system bus detected by I/O expansion bus controller\n");
        if (iohdr.expbus_data_parity) printk(RTAS_ERR "I/O expansion bus data error\n");
        if (iohdr.expbus_timeout) printk(RTAS_ERR "I/O expansion bus timeout, access or other\n");
        if (iohdr.expbus_connection_failure) printk(RTAS_ERR "I/O expansion bus connection failure\n");
        if (iohdr.expbus_not_operating) printk(RTAS_ERR "I/O expansion unit not in an operating state (powered off, off−line)\n");

        printk(RTAS_ERR "IOA Signaling the error: %d:%d.%d vendor:%04x device:%04x rev:%02x slot:%d\n",
               iohdr.pci_sig_busno, iohdr.pci_sig_devfn >> 3, iohdr.pci_sig_devfn & 0x7,
               iohdr.pci_sig_vendorid, iohdr.pci_sig_deviceid, iohdr.pci_sig_revisionid, iohdr.pci_sig_slot);
        printk(RTAS_ERR "IOA Sending during the error: %d:%d.%d vendor:%04x device:%04x rev:%02x slot:%d\n",
               iohdr.pci_send_busno, iohdr.pci_send_devfn >> 3, iohdr.pci_send_devfn & 0x7,
```

```
450                 iohdr.pci_send_vendorid, iohdr.pci_send_deviceid, iohdr.pci_send_revisionid, iohdr.pci_send_slot);
451
452 }
453
454 static void printk_post_failure(int version, struct exthdr *exthdr, char *data)
455 {
456         struct posthdr posthdr;
457
458         memcpy(&posthdr, data, sizeof(posthdr));
459
460         if (posthdr.devname[0]) printk(RTAS_ERR "Failing Device: %s\n", posthdr.devname);
461         if (posthdr.firmware) printk(RTAS_ERR "Firmware Error\n");
462         if (posthdr.config) printk(RTAS_ERR "Configuration Error\n");
463         if (posthdr.cpu) printk(RTAS_ERR "CPU POST Error\n");
464         if (posthdr.memory) printk(RTAS_ERR "Memory POST Error\n");
465         if (posthdr.io) printk(RTAS_ERR "I/O Subsystem POST Error\n");
466         if (posthdr.keyboard) printk(RTAS_ERR "Keyboard POST Error\n");
467         if (posthdr.mouse) printk(RTAS_ERR "Mouse POST Error\n");
468         if (posthdr.display) printk(RTAS_ERR "Display POST Error\n");
469
470         if (posthdr.ipl_floppy) printk(RTAS_ERR "Floppy IPL Error\n");
471         if (posthdr.ipl_controller) printk(RTAS_ERR "Drive Controller Error during IPL\n");
472         if (posthdr.ipl_cdrom) printk(RTAS_ERR "CDROM IPL Error\n");
473         if (posthdr.ipl_disk) printk(RTAS_ERR "Disk IPL Error\n");
474         if (posthdr.ipl_net) printk(RTAS_ERR "Network IPL Error\n");
475         if (posthdr.ipl_other) printk(RTAS_ERR "Other (tape,flash) IPL Error\n");
476         if (posthdr.firmware_selftest) printk(RTAS_ERR "Self−test error in firmware extended diagnostics\n");
477         printk(RTAS_ERR "POST Code: %d\n", posthdr.post_code);
478         printk(RTAS_ERR "Firmware Revision Code: %d\n", posthdr.firmware_rev);
479 }
480
481 static void printk_epow_warning(int version, struct exthdr *exthdr, char *data)
482 {
483         struct epowhdr epowhdr;
484
485         memcpy(&epowhdr, data, sizeof(epowhdr));
486         printk(RTAS_ERR "EPOW Sensor Value: 0x%08x\n", epowhdr.epow_sensor_value);
487         if (epowhdr.sensor) {
488                 printk(RTAS_ERR "EPOW detected by a sensor\n");
489                 printk(RTAS_ERR "Sensor Token: 0x%08x\n", epowhdr.sensor_token);
490                 printk(RTAS_ERR "Sensor Index: 0x%08x\n", epowhdr.sensor_index);
491                 printk(RTAS_ERR "Sensor Value: 0x%08x\n", epowhdr.sensor_value);
492                 printk(RTAS_ERR "Sensor Status: 0x%08x\n", epowhdr.sensor_status);
493         }
494         if (epowhdr.power_fault) printk(RTAS_ERR "EPOW caused by a power fault\n");
495         if (epowhdr.fan) printk(RTAS_ERR "EPOW caused by fan failure\n");
496         if (epowhdr.temp) printk(RTAS_ERR "EPOW caused by over−temperature condition\n");
497         if (epowhdr.redundancy) printk(RTAS_ERR "EPOW warning due to loss of redundancy\n");
498         if (epowhdr.CUoD) printk(RTAS_ERR "EPOW warning due to CUoD Entitlement Exceeded\n");
499
500         if (epowhdr.general) printk(RTAS_ERR "EPOW general power fault\n");
501         if (epowhdr.power_loss) printk(RTAS_ERR "EPOW power fault due to loss of power source\n");
502         if (epowhdr.power_supply) printk(RTAS_ERR "EPOW power fault due to internal power supply failure\n");
503         if (epowhdr.power_switch) printk(RTAS_ERR "EPOW power fault due to activation of power switch\n");
504 }
505
506 static void printk_pm_event(int version, struct exthdr *exthdr, char *data)
507 {
508         struct pm_eventhdr pm_eventhdr;
509
510         memcpy(&pm_eventhdr, data, sizeof(pm_eventhdr));
511         printk(RTAS_ERR "Event id: 0x%08x\n", pm_eventhdr.event_id);
512 }
513
514 static void printk_sp_log_msg(int version, struct exthdr *exthdr, char *data)
515 {
516         struct sphdr sphdr;
517         u32 eyecatcher;
518
519         memcpy(&sphdr, data, sizeof(sphdr));
520
521         eyecatcher = sphdr.ibm;
522         if (strcmp((char *)&eyecatcher, "IBM") != 0)
523                 printk(RTAS_ERR "This log entry may be corrupt (IBM signature malformed)\n");
524         if (sphdr.timeout) printk(RTAS_ERR "Timeout on communication response from service processor\n");
525         if (sphdr.i2c_bus) printk(RTAS_ERR "I2C general bus error\n");
526         if (sphdr.i2c_secondary_bus) printk(RTAS_ERR "I2C secondary bus error\n");
527         if (sphdr.sp_memory) printk(RTAS_ERR "Internal service processor memory error\n");
528         if (sphdr.sp_registers) printk(RTAS_ERR "Service processor error accessing special registers\n");
529         if (sphdr.sp_communication) printk(RTAS_ERR "Service processor reports unknown communcation error\n");
530         if (sphdr.sp_firmware) printk(RTAS_ERR "Internal service processor firmware error\n");
531         if (sphdr.sp_hardware) printk(RTAS_ERR "Other internal service processor hardware error\n");
532         if (sphdr.vpd_eeprom) printk(RTAS_ERR "Service processor error accessing VPD EEPROM\n");
533         if (sphdr.op_panel) printk(RTAS_ERR "Service processor error accessing Operator Panel\n");
534         if (sphdr.power_controller) printk(RTAS_ERR "Service processor error accessing Power Controller\n");
535         if (sphdr.fan_sensor) printk(RTAS_ERR "Service processor error accessing Fan Sensor\n");
536         if (sphdr.thermal_sensor) printk(RTAS_ERR "Service processor error accessing Thermal Sensor\n");
537         if (sphdr.voltage_sensor) printk(RTAS_ERR "Service processor error accessing Voltage Sensor\n");
538         if (sphdr.serial_port) printk(RTAS_ERR "Service processor error accessing serial port\n");
539         if (sphdr.nvram) printk(RTAS_ERR "Service processor detected NVRAM error\n");
```

```
540         if (sphdr.rtc) printk(RTAS_ERR "Service processor error accessing real time clock\n");
541         if (sphdr.jtag) printk(RTAS_ERR "Service processor error accessing JTAG/COP\n");
542         if (sphdr.tod_battery) printk(RTAS_ERR "Service processor or RTAS detects loss of voltage from TOD battery\n");
543         if (sphdr.heartbeat) printk(RTAS_ERR "Loss of heartbeat from Service processor\n");
544         if (sphdr.surveillance) printk(RTAS_ERR "Service processor detected a surveillance timeout\n");
545         if (sphdr.pcn_connection) printk(RTAS_ERR "Power Control Network general connection failure\n");
546         if (sphdr.pcn_node) printk(RTAS_ERR "Power Control Network node failure\n");
547         if (sphdr.pcn_access) printk(RTAS_ERR "Service processor error accessing Power Control Network\n");
548
549         if (sphdr.sensor_token) printk(RTAS_ERR "Sensor Token 0x%08x (%d)\n", sphdr.sensor_token, sphdr.sensor_token);
550         if (sphdr.sensor_index) printk(RTAS_ERR "Sensor Index 0x%08x (%d)\n", sphdr.sensor_index, sphdr.sensor_index);
551 }
552
553
554 static void printk_ext_raw_data(char *data)
555 {
556         int i;
557         printk(RTAS_ERR "raw ext data: ");
558         for (i = 0; i < 40; i++) {
559                 printk("%02x", data[i]);
560         }
561         printk("\n");
562 }
563
564 static void printk_ext_log_data(int version, char *buf)
565 {
566         char *data = buf+12;
567         struct exthdr exthdr;
568         memcpy(&exthdr, buf, sizeof(exthdr));   /* copy for alignment */
569         if (!exthdr.valid) {
570                 if (exthdr.bigendian && exthdr.power_pc)
571                         printk(RTAS_ERR "extended log data is not valid\n");
572                 else
573                         printk(RTAS_ERR "extended log data can not be decoded\n");
574                 return;
575         }
576
577         /* Dump useful stuff in the exthdr */
578         printk(RTAS_ERR "Status:%s%s%s%s%s\n",
579                exthdr.unrecoverable ? " unrecoverable" : "",
580                exthdr.recoverable ? " recoverable" : "",
581                exthdr.unrecoverable_bypassed ? " bypassed" : "",
582                exthdr.predictive ? " predictive" : "",
583                exthdr.newlog ? " new" : "");
584         printk(RTAS_ERR "Date/Time: %08x %08x\n", exthdr.bcddate, exthdr.bcdtime);
585         switch (exthdr.format_type) {
586             case EXTLOG_FMT_CPU:
587                 printk(RTAS_ERR "CPU Failure\n");
588                 printk_cpu_failure(version, &exthdr, data);
589                 break;
590             case EXTLOG_FMT_MEMORY:
591                 printk(RTAS_ERR "Memory Failure\n");
592                 printk_mem_failure(version, &exthdr, data);
593                 break;
594             case EXTLOG_FMT_IO:
595                 printk(RTAS_ERR "I/O Failure\n");
596                 printk_io_failure(version, &exthdr, data);
597                 break;
598             case EXTLOG_FMT_POST:
599                 printk(RTAS_ERR "POST Failure\n");
600                 printk_post_failure(version, &exthdr, data);
601                 break;
602             case EXTLOG_FMT_ENV:
603                 printk(RTAS_ERR "Environment and Power Warning\n");
604                 printk_epow_warning(version, &exthdr, data);
605                 break;
606             case EXTLOG_FMT_POW:
607                 printk(RTAS_ERR "Power Management Event\n");
608                 printk_pm_event(version, &exthdr, data);
609                 break;
610             case EXTLOG_FMT_IBMDIAG:
611                 printk(RTAS_ERR "IBM Diagnostic Log\n");
612                 printk_ext_raw_data(data);
613                 break;
614             case EXTLOG_FMT_IBMSP:
615                 printk(RTAS_ERR "IBM Service Processor Log\n");
616                 printk_sp_log_msg(version, &exthdr, data);
617                 break;
618             default:
619                 printk(RTAS_ERR "Unknown ext format type %d\n", exthdr.format_type);
620                 printk_ext_raw_data(data);
621                 break;
622         }
623 }
624
625 #define MAX_LOG_DEBUG 10
626 #define MAX_LOG_DEBUG_LEN 900
627 /* Print log debug data.  This appears after the location code.
628  * We limit the number of debug logs in case the data is somehow corrupt.
629  */
```

```
630 static void printk_log_debug(char *buf)
631 {
632         unsigned char *p = (unsigned char *)_ALIGN((unsigned long)buf, 4);
633         int len, n, logged;
634
635         logged = 0;
636         while ((logged < MAX_LOG_DEBUG) && (len = ((p[0] << 8) | p[1])) >= 4) {
637                 if (len > MAX_LOG_DEBUG_LEN)
638                         len = MAX_LOG_DEBUG_LEN;        /* bound it */
639                 printk("RTAS: Log Debug: %c%c ", p[2], p[3]);
640                 for (n=4; n < len; n++)
641                         printk("%02x", p[n]);
642                 printk("\n");
643                 p = (unsigned char *)_ALIGN((unsigned long)p+len, 4);
644                 logged++;
645                 if (len == MAX_LOG_DEBUG_LEN)
646                         return; /* no point continuing */
647         }
648         if (logged == 0)
649                 printk("RTAS: no log debug data present\n");
650 }
651
652
653 /* Yeah, the output here is ugly, but we want a CE to be
654  * able to grep RTAS /var/log/messages and see all the info
655  * collected together with obvious begin/end.
656  */
657 static void printk_log_rtas(char *buf)
658 {
659         struct rtas_error_log *err = (struct rtas_error_log *)buf;
660
661         printk(RTAS_ERR "−−−−−−−− event−scan begin −−−−−−−−\n");
662         if (strcmp(buf+8+40, "IBM") == 0) {
663                 /* Location code follows */
664                 char *loc = buf+8+40+4;
665                 int len = strlen(loc);
666                 if (len < 64) { /* Sanity check */
667                         printk(RTAS_ERR "Location Code: %s\n", loc);
668                         printk_log_debug(loc+len+1);
669                 }
670         }
671
672         printk(RTAS_ERR "%s: (%s) type: %s\n",
673                severity_names[err->severity],
674                rtas_disposition_names[err->disposition],
675                rtas_error_type(err->type));
676         printk(RTAS_ERR "initiator: %s  target: %s\n",
677                entity_names[err->initiator], entity_names[err->target]);
678         if (err->extended_log_length)
679                 printk_ext_log_data(err->version, buf+8);
680         printk(RTAS_ERR "−−−−−−−− event−scan end −−−−−−−−−−\n");
681 }
682
683
684 static void log_rtas(char *buf)
685 {
686         unsigned long offset;
687
688         DEBUG("logging rtas event\n");
689
690         /* Temporary -- perhaps we can do this when nobody has the log open? */
691         printk_log_rtas(buf);
692
693         spin_lock(&rtas_log_lock);
694
695         offset = rtas_error_log_max *
696                         ((rtas_log_start+rtas_log_size) & LOG_NUMBER_MASK);
697
698         memcpy(&rtas_log_buf[offset], buf, rtas_error_log_max);
699
700         if (rtas_log_size < LOG_NUMBER)
701                 rtas_log_size += 1;
702         else
703                 rtas_log_start += 1;
704
705         spin_unlock(&rtas_log_lock);
706         wake_up_interruptible(&rtas_log_wait);
707 }
708
709 static int enable_surveillance(void)
710 {
711         int error;
712
713         error = rtas_call(rtas_token("set−indicator"), 3, 1, NULL, SURVEILLANCE_TOKEN,
714                         0, SURVEILLANCE_TIMEOUT);
715
716         if (error) {
717                 printk(KERN_ERR "rtasd: could not enable surveillance\n");
718                 return −1;
719         }
```

```
720
721         rtas_event_scan_rate = SURVEILLANCE_SCANRATE;
722
723         return 0;
724 }
725
726 static int get_eventscan_parms(void)
727 {
728         struct device_node *node;
729         int *ip;
730
731         node = find_path_device("/rtas");
732
733         ip = (int *)get_property(node, "rtas−event−scan−rate", NULL);
734         if (ip == NULL) {
735                 printk(KERN_ERR "rtasd: no rtas−event−scan−rate\n");
736                 return −1;
737         }
738         rtas_event_scan_rate = *ip;
739         DEBUG("rtas−event−scan−rate %d\n", rtas_event_scan_rate);
740
741         ip = (int *)get_property(node, "rtas−error−log−max", NULL);
742         if (ip == NULL) {
743                 printk(KERN_ERR "rtasd: no rtas−error−log−max\n");
744                 return −1;
745         }
746         rtas_error_log_max = *ip;
747         DEBUG("rtas−error−log−max %d\n", rtas_error_log_max);
748
749         if (rtas_error_log_max > RTAS_ERROR_LOG_MAX) {
750                 printk(KERN_ERR "rtasd: truncated error log from %d to %d bytes\n", rtas_error_log_max, RTAS_ERROR_LOG_MAX);
751                 rtas_error_log_max = RTAS_ERROR_LOG_MAX;
752         }
753
754         return 0;
755 }
756
757 extern long sys_sched_get_priority_max(int policy);
758
759 static int rtasd(void *unused)
760 {
761         int cpu = 0;
762         int error;
763         int first_pass = 1;
764         int event_scan = rtas_token("event−scan");
765
766         if (event_scan == RTAS_UNKNOWN_SERVICE || get_eventscan_parms() == −1)
767                 goto error;
768
769         rtas_log_buf = vmalloc(rtas_error_log_max*LOG_NUMBER);
770         if (!rtas_log_buf) {
771                 printk(KERN_ERR "rtasd: no memory\n");
772                 goto error;
773         }
774
775         DEBUG("will sleep for %d jiffies\n", (HZ*60/rtas_event_scan_rate) / 2);
776
777         daemonize();
778         sigfillset(&current−>blocked);
779         sprintf(current−>comm, "rtasd");
780
781         /* Rusty unreal time task */
782         current−>policy = SCHED_FIFO;
783         /* XXX FIXME */
784         /* current−>nice = sys_sched_get_priority_max(SCHED_FIFO) + 1; */
785
786         cpu = 0;
787         current−>cpus_allowed = 1UL << cpu_logical_map(cpu);
788         schedule();
789
790         while(1) {
791                 do {
792                         memset(logdata, 0, rtas_error_log_max);
793                         error = rtas_call(event_scan, 4, 1, NULL,
794                                           EVENT_SCAN_ALL_EVENTS, 0,
795                                           __pa(logdata), rtas_error_log_max);
796                         if (error == −1) {
797                                 printk(KERN_ERR "event−scan failed\n");
798                                 break;
799                         }
800
801                         if (error == 0)
802                                 log_rtas(logdata);
803
804                 } while(error == 0);
805
806                 DEBUG("watchdog scheduled on cpu %d\n", smp_processor_id());
807
808                 cpu++;
809                 if (cpu >= smp_num_cpus) {
```

```
810
811                         if (first_pass && surveillance_requested) {
812                                 DEBUG("enabling surveillance\n");
813                                 if (enable_surveillance())
814                                         goto error_vfree;
815                                 DEBUG("surveillance enabled\n");
816                         }
817
818                         first_pass = 0;
819                         cpu = 0;
820                 }
821
822                 current->cpus_allowed = 1UL << cpu_logical_map(cpu);
823
824                 /* Check all cpus for pending events before sleeping*/
825                 if (first_pass) {
826                         schedule();
827                 } else {
828                         set_current_state(TASK_INTERRUPTIBLE);
829                         schedule_timeout((HZ*60/rtas_event_scan_rate) / 2);
830                 }
831         }
832
833 error_vfree:
834         vfree(rtas_log_buf);
835 error:
836         /* Should delete proc entries */
837         return -EINVAL;
838 }
839
840 static void __init rtas_init(void)
841 {
842         struct proc_dir_entry *rtas_dir, *entry;
843
844         rtas_dir = proc_mkdir("rtas", 0);
845         if (!rtas_dir) {
846                 printk(KERN_ERR "Failed to create rtas proc directory\n");
847         } else {
848                 entry = create_proc_entry("error_log", S_IRUSR, rtas_dir);
849                 if (entry)
850                         entry->proc_fops = &proc_rtas_log_operations;
851                 else
852                         printk(KERN_ERR "Failed to create rtas/error_log proc entry\n");
853         }
854
855         if (kernel_thread(rtasd, 0, CLONE_FS) < 0)
856                 printk(KERN_ERR "Failed to start RTAS daemon\n");
857
858         printk(KERN_ERR "RTAS daemon started\n");
859 }
860
861 static int __init surveillance_setup(char *str)
862 {
863         int i;
864
865         if (get_option(&str,&i)) {
866                 if (i == 1)
867                         surveillance_requested = 1;
868         }
869
870         return 1;
871 }
872
873 __initcall(rtas_init);
874 __setup("surveillance=", surveillance_setup);
```

```
/*
 * arch/ppc/kernel/xics.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.
 */
#include <linux/config.h>
#include <linux/types.h>
#include <linux/threads.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <asm/prom.h>
#include <asm/io.h>
#include <asm/pgtable.h>
#include <asm/smp.h>
#include <asm/naca.h>
#include <asm/rtas.h>
#include "i8259.h"
#include "xics.h"
#include <asm/ppcdebug.h>

void xics_enable_irq(u_int irq);
void xics_disable_irq(u_int irq);
void xics_mask_and_ack_irq(u_int irq);
void xics_end_irq(u_int irq);
void xics_set_affinity(unsigned int irq_nr, unsigned long cpumask);

struct hw_interrupt_type xics_pic = {
        " XICS   ",
        NULL,
        NULL,
        xics_enable_irq,
        xics_disable_irq,
        xics_mask_and_ack_irq,
        xics_end_irq,
        xics_set_affinity
};

struct hw_interrupt_type xics_8259_pic = {
        " XICS/8259",
        NULL,
        NULL,
        NULL,
        NULL,
        xics_mask_and_ack_irq,
        NULL
};

#define XICS_IPI                2
#define XICS_IRQ_OFFSET         0x10
#define XICS_IRQ_SPURIOUS       0

/* Want a priority other than 0.  Various HW issues require this. */
#define DEFAULT_PRIORITY        5

struct xics_ipl {
        union {
                u32     word;
                u8      bytes[4];
        } xirr_poll;
        union {
                u32 word;
                u8      bytes[4];
        } xirr;
        u32     dummy;
        union {
                u32     word;
                u8      bytes[4];
        } qirr;
};

struct xics_info {
        volatile struct xics_ipl *      per_cpu[NR_CPUS];
};

struct xics_info        xics_info;

unsigned long long intr_base = 0;
int xics_irq_8259_cascade = 0;
int xics_irq_8259_cascade_real = 0;
unsigned int default_server = 0xFF;
unsigned int default_distrib_server = 0;

/* RTAS service tokens */
int ibm_get_xive;
int ibm_set_xive;
```

```
91  int ibm_int_off;
92
93  struct xics_interrupt_node {
94          unsigned long long addr;
95          unsigned long long size;
96  } inodes[NR_CPUS*2];
97
98  typedef struct {
99          int (*xirr_info_get)(int cpu);
100         void (*xirr_info_set)(int cpu, int val);
101         void (*cppr_info)(int cpu, u8 val);
102         void (*qirr_info)(int cpu, u8 val);
103 } xics_ops;
104
105
106 static int pSeries_xirr_info_get(int n_cpu)
107 {
108         return (xics_info.per_cpu[n_cpu]->xirr.word);
109 }
110
111 static void pSeries_xirr_info_set(int n_cpu, int value)
112 {
113         xics_info.per_cpu[n_cpu]->xirr.word = value;
114 }
115
116 static void pSeries_cppr_info(int n_cpu, u8 value)
117 {
118         xics_info.per_cpu[n_cpu]->xirr.bytes[0] = value;
119 }
120
121 static void pSeries_qirr_info(int n_cpu , u8 value)
122 {
123         xics_info.per_cpu[n_cpu]->qirr.bytes[0] = value;
124 }
125
126 static xics_ops pSeries_ops = {
127         pSeries_xirr_info_get,
128         pSeries_xirr_info_set,
129         pSeries_cppr_info,
130         pSeries_qirr_info
131 };
132
133 static xics_ops *ops = &pSeries_ops;
134 extern xics_ops pSeriesLP_ops;
135
136
137 void
138 xics_enable_irq(
139         u_int   virq
140         )
141 {
142         u_int           irq;
143         unsigned long   status;
144         long            call_status;
145
146         virq -= XICS_IRQ_OFFSET;
147         irq = virt_irq_to_real(virq);
148         if (irq == XICS_IPI)
149                 return;
150 #ifdef CONFIG_IRQ_ALL_CPUS
151         call_status = rtas_call(ibm_set_xive, 3, 1, (unsigned long*)&status,
152                                 irq, smp_threads_ready ? default_distrib_server : default_server, DEFAULT_PRIORIT
    Y);
153 #else
154         call_status = rtas_call(ibm_set_xive, 3, 1, (unsigned long*)&status,
155                                 irq, default_server, DEFAULT_PRIORITY);
156 #endif
157         if( call_status != 0 ) {
158                 printk("xics_enable_irq: irq=%x: rtas_call failed; retn=%lx, status=%lx\n",
159                        irq, call_status, status);
160                 return;
161         }
162 }
163
164 void
165 xics_disable_irq(
166         u_int   virq
167         )
168 {
169         u_int           irq;
170         unsigned long   status;
171         long            call_status;
172
173         virq -= XICS_IRQ_OFFSET;
174         irq = virt_irq_to_real(virq);
175         call_status = rtas_call(ibm_int_off, 1, 1, (unsigned long*)&status,
176                                 irq);
177         if( call_status != 0 ) {
178                 printk("xics_disable_irq: irq=%x: rtas_call failed, retn=%lx\n",
179                        irq, call_status);
```

```
180                     return;
181             }
182     }
183
184     void
185     xics_end_irq(
186             u_int   irq
187             )
188     {
189             int cpu = smp_processor_id();
190
191             ops->cppr_info(cpu, 0); /* actually the value overwritten by ack */
192             iosync();
193             ops->xirr_info_set(cpu, ((0xff<<24) | (virt_irq_to_real(irq-XICS_IRQ_OFFSET))));
194             iosync();
195     }
196
197     void
198     xics_mask_and_ack_irq(u_int     irq)
199     {
200             int cpu = smp_processor_id();
201
202             if( irq < XICS_IRQ_OFFSET ) {
203                     i8259_pic.ack(irq);
204                     iosync();
205                     ops->xirr_info_set(cpu, ((0xff<<24) | xics_irq_8259_cascade_real));
206                     iosync();
207             }
208             else {
209                     ops->cppr_info(cpu, 0xff);
210                     iosync();
211             }
212     }
213
214     int
215     xics_get_irq(struct pt_regs *regs)
216     {
217             u_int   cpu = smp_processor_id();
218             u_int   vec;
219             int irq;
220
221             vec = ops->xirr_info_get(cpu);
222             /*  (vec >> 24) == old priority */
223             vec &= 0x00ffffff;
224             /* for sanity, this had better be < NR_IRQS - 16 */
225             if( vec == xics_irq_8259_cascade_real ) {
226                     irq = i8259_irq(cpu);
227                     if(irq == -1) {
228                             /* Spurious cascaded interrupt.  Still must ack xics */
229                             xics_end_irq(XICS_IRQ_OFFSET + xics_irq_8259_cascade);
230                             irq = -1;
231                     }
232             } else if( vec == XICS_IRQ_SPURIOUS ) {
233                     irq = -1;
234             } else {
235                     irq = real_irq_to_virt(vec) + XICS_IRQ_OFFSET;
236             }
237             return irq;
238     }
239
240
241     #ifdef CONFIG_SMP
242     void xics_ipi_action(int irq, void *dev_id, struct pt_regs *regs)
243     {
244             extern volatile unsigned long xics_ipi_message[];
245             int cpu = smp_processor_id();
246
247             ops->qirr_info(cpu, 0xff);
248             while (xics_ipi_message[cpu]) {
249                     if (test_and_clear_bit(PPC_MSG_CALL_FUNCTION, &xics_ipi_message[cpu])) {
250                             mb();
251                             smp_message_recv(PPC_MSG_CALL_FUNCTION, regs);
252                     }
253                     if (test_and_clear_bit(PPC_MSG_RESCHEDULE, &xics_ipi_message[cpu])) {
254                             mb();
255                             smp_message_recv(PPC_MSG_RESCHEDULE, regs);
256                     }
257     #ifdef CONFIG_XMON
258                     if (test_and_clear_bit(PPC_MSG_XMON_BREAK, &xics_ipi_message[cpu])) {
259                             mb();
260                             smp_message_recv(PPC_MSG_XMON_BREAK, regs);
261                     }
262     #endif
263             }
264     }
265
266     void xics_cause_IPI(int cpu)
267     {
268             ops->qirr_info(cpu,0) ;
269     }
```

```
270
271 void xics_setup_cpu(void)
272 {
273         int cpu = smp_processor_id();
274
275         ops->cppr_info(cpu, 0xff);
276         iosync();
277 }
278 #endif /* CONFIG_SMP */
279
280 void
281 xics_init_IRQ( void )
282 {
283         int i;
284         unsigned long intr_size = 0;
285         struct device_node *np;
286         uint *ireg, ilen, indx=0;
287
288         ppc64_boot_msg(0x20, "XICS Init");
289
290         ibm_get_xive = rtas_token("ibm,get-xive");
291         ibm_set_xive = rtas_token("ibm,set-xive");
292         ibm_int_off = rtas_token("ibm,int-off");
293
294         np = find_type_devices("PowerPC-External-Interrupt-Presentation");
295         if (!np) {
296                 printk(KERN_WARNING "Can't find Interrupt Presentation\n");
297                 udbg_printf("Can't find Interrupt Presentation\n");
298                 while (1);
299         }
300 nextnode:
301         ireg = (uint *)get_property(np, "ibm,interrupt-server-ranges", 0);
302         if (ireg) {
303                 /*
304                  * set node starting index for this node
305                  */
306                 indx = *ireg;
307         }
308
309         ireg = (uint *)get_property(np, "reg", &ilen);
310         if (!ireg) {
311                 printk(KERN_WARNING "Can't find Interrupt Reg Property\n");
312                 udbg_printf("Can't find Interrupt Reg Property\n");
313                 while (1);
314         }
315
316         while (ilen) {
317                 inodes[indx].addr = (unsigned long long)*ireg++ << 32;
318                 ilen -= sizeof(uint);
319                 inodes[indx].addr |= *ireg++;
320                 ilen -= sizeof(uint);
321                 inodes[indx].size = (unsigned long long)*ireg++ << 32;
322                 ilen -= sizeof(uint);
323                 inodes[indx].size |= *ireg++;
324                 ilen -= sizeof(uint);
325                 indx++;
326                 if (indx >= NR_CPUS) break;
327         }
328
329         np = np->next;
330         if ((indx < NR_CPUS) && np) goto nextnode;
331
332         /* Find the server numbers for the boot cpu. */
333         for (np = find_type_devices("cpu"); np; np = np->next) {
334                 ireg = (uint *)get_property(np, "reg", &ilen);
335                 if (ireg && ireg[0] == hard_smp_processor_id()) {
336                         ireg = (uint *)get_property(np, "ibm,ppc-interrupt-gserver#s", &ilen);
337                         i = ilen / sizeof(int);
338                         if (ireg && i > 0) {
339                                 default_server = ireg[0];
340                                 default_distrib_server = ireg[i-1]; /* take last element */
341                         }
342                         break;
343                 }
344         }
345
346         intr_base = inodes[0].addr;
347         intr_size = (ulong)inodes[0].size;
348
349         np = find_type_devices("interrupt-controller");
350         if (!np) {
351                 printk(KERN_WARNING "xics:  no ISA Interrupt Controller\n");
352                 xics_irq_8259_cascade_real = -1;
353                 xics_irq_8259_cascade = -1;
354         } else {
355                 ireg = (uint *) get_property(np, "interrupts", 0);
356                 if (!ireg) {
357                         printk(KERN_WARNING "Can't find ISA Interrupts Property\n");
358                         udbg_printf("Can't find ISA Interrupts Property\n");
359                         while (1);
```

```
360                 }
361                 xics_irq_8259_cascade_real = *ireg;
362                 xics_irq_8259_cascade = virt_irq_create_mapping(xics_irq_8259_cascade_real);
363         }
364
365         if (systemcfg->platform == PLATFORM_PSERIES) {
366 #ifdef CONFIG_SMP
367                 for (i = 0; i < systemcfg->processorCount; ++i) {
368                         xics_info.per_cpu[i] =
369                           __ioremap((ulong)inodes[get_hard_smp_processor_id(i)].addr,
370                                     (ulong)inodes[get_hard_smp_processor_id(i)].size, _PAGE_NO_CACHE);
371                 }
372 #else
373                 xics_info.per_cpu[0] = __ioremap((ulong)intr_base, intr_size, _PAGE_NO_CACHE);
374 #endif /* CONFIG_SMP */
375 #ifdef CONFIG_PPC_PSERIES
376         /* actually iSeries does not use any of xics...but it has link dependencies
377          * for now, except this new one...
378          */
379         } else if (systemcfg->platform == PLATFORM_PSERIES_LPAR) {
380                 ops = &pSeriesLP_ops;
381 #endif
382         }
383
384         xics_8259_pic.enable = i8259_pic.enable;
385         xics_8259_pic.disable = i8259_pic.disable;
386         for (i = 0; i < 16; ++i)
387                 irq_desc[i].handler = &xics_8259_pic;
388         for (; i < NR_IRQS; ++i)
389                 irq_desc[i].handler = &xics_pic;
390
391         ops->cppr_info(0, 0xff);
392         iosync();
393         if (xics_irq_8259_cascade != -1) {
394                 if (request_irq(xics_irq_8259_cascade + XICS_IRQ_OFFSET, no_action,
395                                 0, "8259 cascade", 0))
396                         printk(KERN_ERR "xics_init_IRQ: couldn't get 8259 cascade\n");
397                 i8259_init();
398         }
399
400 #ifdef CONFIG_SMP
401         real_irq_to_virt_map[XICS_IPI] = virt_irq_to_real_map[XICS_IPI] = XICS_IPI;
402         request_irq(XICS_IPI + XICS_IRQ_OFFSET, xics_ipi_action, 0, "IPI", 0);
403         irq_desc[XICS_IPI+XICS_IRQ_OFFSET].status |= IRQ_PER_CPU;
404 #endif
405         ppc64_boot_msg(0x21, "XICS Done");
406 }
407
408 void xics_isa_init(void)
409 {
410         return;
411         if (request_irq(xics_irq_8259_cascade + XICS_IRQ_OFFSET, no_action,
412                         0, "8259 cascade", 0))
413                 printk(KERN_ERR "xics_init_IRQ: couldn't get 8259 cascade\n");
414         i8259_init();
415 }
416
417 /*
418  * Find first logical cpu and return its physical cpu number
419  */
420 static inline u32 physmask(u32 cpumask)
421 {
422         int i;
423
424         for (i = 0; i < smp_num_cpus; ++i, cpumask >>= 1) {
425                 if (cpumask & 1)
426                         return get_hard_smp_processor_id(i);
427         }
428
429         printk(KERN_ERR "xics_set_affinity: invalid irq mask\n");
430
431         return default_distrib_server;
432 }
433
434 void xics_set_affinity(unsigned int virq, unsigned long cpumask)
435 {
436         irq_desc_t *desc = irq_desc + virq;
437         unsigned int irq;
438         unsigned long flags;
439         long status;
440         unsigned long xics_status[2];
441         u32 newmask;
442
443         virq -= XICS_IRQ_OFFSET;
444         irq = virt_irq_to_real(virq);
445         if (irq == XICS_IPI)
446                 return;
447
448         spin_lock_irqsave(&desc->lock, flags);
449
```

```
450         status = rtas_call(ibm_get_xive, 1, 3, (void *)&xics_status, irq);
451
452         if (status) {
453                 printk("xics_set_affinity: irq=%d ibm,get−xive returns %ld\n",
454                                 irq, status);
455                 goto out;
456         }
457
458         /* For the moment only implement delivery to all cpus or one cpu */
459         if (cpumask == 0xffffffff)
460                 newmask = default_distrib_server;
461         else
462                 newmask = physmask(cpumask);
463
464         status = rtas_call(ibm_set_xive, 3, 1, NULL,
465                                 irq, newmask, xics_status[1]);
466
467         if (status) {
468                 printk("xics_set_affinity irq=%d ibm,set−xive returns %ld\n",
469                                 irq, status);
470                 goto out;
471         }
472
473 out:
474         spin_unlock_irqrestore(&desc−>lock, flags);
475 }
```

Table of Content