```
**********************************************************
   47022 Mon Oct  1 13:29:57 2012
new/usr/src/uts/i86pc/io/pci/pci_common.c
3235 pci: pci_common_intr_ops() leaks ddi_acc_handle_t
Reviewed by: Dan McDonald <danmcd@nexenta.com>
Reviewed by: Boris Protopopov <boris.protopopov@nexenta.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 * Copyright (c) 2012, Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 *      File that has code which is common between pci(7d) and npe(7d)
 *      It shares the following:
 *      - interrupt code
 *      - pci_tools ioctl code
 *      - name_child code
 *      - set_parent_private_data code
 */

#include <sys/conf.h>
#include <sys/pci.h>
#include <sys/sunndi.h>
#include <sys/mach_intr.h>
#include <sys/pci_intr_lib.h>
#include <sys/psm.h>
#include <sys/policy.h>
#include <sys/sysmacros.h>
#include <sys/clock.h>
#include <sys/apic.h>
#include <sys/pci_tools.h>
#include <io/pci/pci_var.h>
#include <io/pci/pci_tools_ext.h>
#include <io/pci/pci_common.h>
#include <sys/pci_cfgspace.h>
#include <sys/pci_impl.h>
#include <sys/pci_cap.h>

/*
 * Function prototypes
 */
static int      pci_get_priority(dev_info_t *, ddi_intr_handle_impl_t *, int *);
```



```
static int      pci_enable_intr(dev_info_t *, dev_info_t *,
                    ddi_intr_handle_impl_t *, uint32_t);
static void     pci_disable_intr(dev_info_t *, dev_info_t *,
                    ddi_intr_handle_impl_t *, uint32_t);
static int      pci_alloc_intr_fixed(dev_info_t *, dev_info_t *,
                    ddi_intr_handle_impl_t *, void *);
static int      pci_free_intr_fixed(dev_info_t *, dev_info_t *,
                    ddi_intr_handle_impl_t *);

/* Extern declarations for PSM module */
extern int      (*psm_intr_ops)(dev_info_t *, ddi_intr_handle_impl_t *,
                    psm_intr_op_t, int *);
extern ddi_irm_pool_t *apix_irm_pool_p;

/*
 * pci_name_child:
 *
 *      Assign the address portion of the node name
 */
int
pci_common_name_child(dev_info_t *child, char *name, int namelen)
{
        int             dev, func, length;
        char            **unit_addr;
        uint_t          n;
        pci_regspec_t   *pci_rp;

        if (ndi_dev_is_persistent_node(child) == 0) {
                /*
                 * For .conf node, use "unit-address" property
                 */
                if (ddi_prop_lookup_string_array(DDI_DEV_T_ANY, child,
                    DDI_PROP_DONTPASS, "unit-address", &unit_addr, &n) !=
                    DDI_PROP_SUCCESS) {
                        cmn_err(CE_WARN, "cannot find unit-address in %s.conf",
                            ddi_get_name(child));
                        return (DDI_FAILURE);
                }
                if (n != 1 || *unit_addr == NULL || **unit_addr == 0) {
                        cmn_err(CE_WARN, "unit-address property in %s.conf"
                            " not well-formed", ddi_get_name(child));
                        ddi_prop_free(unit_addr);
                        return (DDI_FAILURE);
                }
                (void) snprintf(name, namelen, "%s", *unit_addr);
                ddi_prop_free(unit_addr);
                return (DDI_SUCCESS);
        }

        if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, child, DDI_PROP_DONTPASS,
            "reg", (int **)&pci_rp, (uint_t *)&length) != DDI_PROP_SUCCESS) {
                cmn_err(CE_WARN, "cannot find reg property in %s",
                    ddi_get_name(child));
                return (DDI_FAILURE);
        }

        /* copy the device identifications */
        dev = PCI_REG_DEV_G(pci_rp->pci_phys_hi);
        func = PCI_REG_FUNC_G(pci_rp->pci_phys_hi);

        /*
         * free the memory allocated by ddi_prop_lookup_int_array
         */
        ddi_prop_free(pci_rp);

        if (func != 0) {
```

```
126                 (void) snprintf(name, namelen, "%x,%x", dev, func);
127         } else {
128                 (void) snprintf(name, namelen, "%x", dev);
129         }

131         return (DDI_SUCCESS);
132 }

134 /*
135  * Interrupt related code:
136  *
137  * The following busop is common to npe and pci drivers
138  *      bus_introp
139  */

141 /*
142  * Create the ddi_parent_private_data for a pseudo child.
143  */
144 void
145 pci_common_set_parent_private_data(dev_info_t *dip)
146 {
147         struct ddi_parent_private_data *pdptr;

149         pdptr = (struct ddi_parent_private_data *)kmem_zalloc(
150             (sizeof (struct ddi_parent_private_data) +
151             sizeof (struct intrspec)), KM_SLEEP);
152         pdptr->par_intr = (struct intrspec *)(pdptr + 1);
153         pdptr->par_nintr = 1;
154         ddi_set_parent_data(dip, pdptr);
155 }

157 /*
158  * pci_get_priority:
159  *      Figure out the priority of the device
160  */
161 static int
162 pci_get_priority(dev_info_t *dip, ddi_intr_handle_impl_t *hdlp, int *pri)
163 {
164         struct intrspec *ispec;

166         DDI_INTR_NEXDBG((CE_CONT, "pci_get_priority: dip = 0x%p, hdlp = %p\n",
167             (void *)dip, (void *)hdlp));

169         if ((ispec = (struct intrspec *)pci_intx_get_ispec(dip, dip,
170             hdlp->ih_inum)) == NULL) {
171                 if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type)) {
172                         *pri = pci_class_to_pil(dip);
173                         pci_common_set_parent_private_data(hdlp->ih_dip);
174                         ispec = (struct intrspec *)pci_intx_get_ispec(dip, dip,
175                             hdlp->ih_inum);
176                         return (DDI_SUCCESS);
177                 }
178                 return (DDI_FAILURE);
179         }

181         *pri = ispec->intrspec_pri;
182         return (DDI_SUCCESS);
183 }



187 static int pcieb_intr_pri_counter = 0;

189 /*
190  * pci_common_intr_ops: bus_intr_op() function for interrupt support
191  */
```



```
192 int
193 pci_common_intr_ops(dev_info_t *pdip, dev_info_t *rdip, ddi_intr_op_t intr_op,
194     ddi_intr_handle_impl_t *hdlp, void *result)
195 {
196         int                     priority = 0;
197         int                     psm_status = 0;
198         int                     pci_status = 0;
199         int                     pci_rval, psm_rval = PSM_FAILURE;
200         int                     types = 0;
201         int                     pciepci = 0;
202         int                     i, j, count;
203         int                     rv;
204         int                     behavior;
205         int                     cap_ptr;
206         boolean_t               did_pci_config_setup = B_FALSE;
207         boolean_t               did_intr_vec_alloc = B_FALSE;
208         boolean_t               did_msi_cap_set = B_FALSE;
209 #endif /* ! codereview */
210         uint16_t                msi_cap_base, msix_cap_base, cap_ctrl;
211         char                    *prop;
212         ddi_intrspec_t          isp;
213         struct intrspec         *ispec;
214         ddi_intr_handle_impl_t  tmp_hdl;
215         ddi_intr_msix_t         *msix_p;
216         ihdl_plat_t             *ihdl_plat_datap;
217         ddi_intr_handle_t       *h_array;
218         ddi_acc_handle_t        handle;
219         apic_get_intr_t         intrinfo;

221         DDI_INTR_NEXDBG((CE_CONT,
222             "pci_common_intr_ops: pdip 0x%p, rdip 0x%p, op %x handle 0x%p\n",
223             (void *)pdip, (void *)rdip, intr_op, (void *)hdlp));

225         /* Process the request */
226         switch (intr_op) {
227         case DDI_INTROP_SUPPORTED_TYPES:
228                 /*
229                  * First we determine the interrupt types supported by the
230                  * device itself, then we filter them through what the OS
231                  * and system supports.  We determine system-level
232                  * interrupt type support for anything other than fixed intrs
233                  * through the psm_intr_ops vector
234                  */
235                 rv = DDI_FAILURE;

237                 /* Fixed supported by default */
238                 types = DDI_INTR_TYPE_FIXED;

240                 if (psm_intr_ops == NULL) {
241                         *(int *)result = types;
242                         return (DDI_SUCCESS);
243                 }
244                 if (pci_config_setup(rdip, &handle) != DDI_SUCCESS)
245                         return (DDI_FAILURE);

247                 /* Sanity test cap control values if found */

249                 if (PCI_CAP_LOCATE(handle, PCI_CAP_ID_MSI, &msi_cap_base) ==
250                     DDI_SUCCESS) {
251                         cap_ctrl = PCI_CAP_GET16(handle, 0, msi_cap_base,
252                             PCI_MSI_CTRL);
253                         if (cap_ctrl == PCI_CAP_EINVAL16)
254                                 goto SUPPORTED_TYPES_OUT;

256                         types |= DDI_INTR_TYPE_MSI;
257                 }
```

```
259                 if (PCI_CAP_LOCATE(handle, PCI_CAP_ID_MSI_X, &msix_cap_base) ==
260                     DDI_SUCCESS) {
261                         cap_ctrl = PCI_CAP_GET16(handle, 0, msix_cap_base,
262                             PCI_MSIX_CTRL);
263                         if (cap_ctrl == PCI_CAP_EINVAL16)
264                                 goto SUPPORTED_TYPES_OUT;
266                         types |= DDI_INTR_TYPE_MSIX;
267                 }
269                 /*
270                  * Filter device-level types through system-level support
271                  */
272                 tmp_hdl.ih_type = types;
273                 if ((*psm_intr_ops)(rdip, &tmp_hdl, PSM_INTR_OP_CHECK_MSI,
274                     &types) != PSM_SUCCESS)
275                         goto SUPPORTED_TYPES_OUT;
277                 DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
278                     "rdip: 0x%p supported types: 0x%x\n", (void *)rdip,
279                     types));
281                 /*
282                  * Export any MSI/MSI-X cap locations via properties
283                  */
284                 if (types & DDI_INTR_TYPE_MSI) {
285                         if (ndi_prop_update_int(DDI_DEV_T_NONE, rdip,
286                             "pci-msi-capid-pointer", (int)msi_cap_base) !=
287                             DDI_PROP_SUCCESS)
288                                 goto SUPPORTED_TYPES_OUT;
289                 }
290                 if (types & DDI_INTR_TYPE_MSIX) {
291                         if (ndi_prop_update_int(DDI_DEV_T_NONE, rdip,
292                             "pci-msix-capid-pointer", (int)msix_cap_base) !=
293                             DDI_PROP_SUCCESS)
294                                 goto SUPPORTED_TYPES_OUT;
295                 }
297                 rv = DDI_SUCCESS;
299 SUPPORTED_TYPES_OUT:
300                 *(int *)result = types;
301                 pci_config_teardown(&handle);
302                 return (rv);
304         case DDI_INTROP_NAVAIL:
305         case DDI_INTROP_NINTRS:
306                 if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type)) {
307                         if (pci_msi_get_nintrs(hdlp->ih_dip, hdlp->ih_type,
308                             result) != DDI_SUCCESS)
309                                 return (DDI_FAILURE);
310                 } else {
311                         *(int *)result = i_ddi_get_intx_nintrs(hdlp->ih_dip);
312                         if (*(int *)result == 0)
313                                 return (DDI_FAILURE);
314                 }
315                 break;
316         case DDI_INTROP_ALLOC:
318                 /*
319                  * FIXED type
320                  */
321                 if (hdlp->ih_type == DDI_INTR_TYPE_FIXED)
322                         return (pci_alloc_intr_fixed(pdip, rdip, hdlp, result));
323                 /*
```



```
324                  * MSI or MSIX (figure out number of vectors available)
325                  */
326                 if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type) &&
327                     (psm_intr_ops != NULL) &&
328                     (pci_get_priority(rdip, hdlp, &priority) == DDI_SUCCESS)) {
329                         /*
330                          * Following check is a special case for 'pcieb'.
331                          * This makes sure vectors with the right priority
332                          * are allocated for pcieb during ALLOC time.
333                          */
334                         if (strcmp(ddi_driver_name(rdip), "pcieb") == 0) {
335                                 hdlp->ih_pri =
336                                     (pcieb_intr_pri_counter % 2) ? 4 : 7;
337                                 pciepci = 1;
338                         } else
339                                 hdlp->ih_pri = priority;
340                         behavior = (int)(uintptr_t)hdlp->ih_scratch2;
342                         /*
343                          * Cache in the config handle and cap_ptr
344                          */
345                         if (i_ddi_get_pci_config_handle(rdip) == NULL) {
346                                 if (pci_config_setup(rdip, &handle) !=
347                                     DDI_SUCCESS)
348                                         return (DDI_FAILURE);
349                                 i_ddi_set_pci_config_handle(rdip, handle);
350                                 did_pci_config_setup = B_TRUE;
351 #endif /* ! codereview */
352                         }
354                         prop = NULL;
355                         cap_ptr = 0;
356                         if (hdlp->ih_type == DDI_INTR_TYPE_MSI)
357                                 prop = "pci-msi-capid-pointer";
358                         else if (hdlp->ih_type == DDI_INTR_TYPE_MSIX)
359                                 prop = "pci-msix-capid-pointer";
361                         /*
362                          * Enforce the calling of DDI_INTROP_SUPPORTED_TYPES
363                          * for MSI(X) before allocation
364                          */
365                         if (prop != NULL) {
366                                 cap_ptr = ddi_prop_get_int(DDI_DEV_T_ANY, rdip,
367                                     DDI_PROP_DONTPASS, prop, 0);
368                                 if (cap_ptr == 0) {
369                                         DDI_INTR_NEXDBG((CE_CONT,
370                                             "pci_common_intr_ops: rdip: 0x%p "
371                                             "attempted MSI(X) alloc without "
372                                             "cap property\n", (void *)rdip));
373                                         return (DDI_FAILURE);
374                                 }
375                         }
376                         i_ddi_set_msi_msix_cap_ptr(rdip, cap_ptr);
377                         did_msi_cap_set = B_TRUE;
378 #endif /* ! codereview */
380                         /*
381                          * Allocate interrupt vectors
382                          */
383                         (void) (*psm_intr_ops)(rdip, hdlp,
384                             PSM_INTR_OP_ALLOC_VECTORS, result);
386                         if (*(int *)result == 0) {
387                                 rv = DDI_INTR_NOTFOUND;
388                                 goto HANDLE_ALLOC_FAILURE;
389                         }
```

```
390                         did_intr_vec_alloc = B_TRUE;
 24                         if (*(int *)result == 0)
 25                                 return (DDI_INTR_NOTFOUND);

392                         /* verify behavior flag and take appropriate action */
393                         if ((behavior == DDI_INTR_ALLOC_STRICT) &&
394                             (*(int *)result < hdlp->ih_scratch1)) {
395                                 DDI_INTR_NEXDBG((CE_CONT,
396                                     "pci_common_intr_ops: behavior %x, "
397                                     "couldn't get enough intrs\n", behavior));
398                                 hdlp->ih_scratch1 = *(int *)result;
399                                 rv = DDI_EAGAIN;
400                                 goto HANDLE_ALLOC_FAILURE;
 34                                 (void) (*psm_intr_ops)(rdip, hdlp,
 35                                     PSM_INTR_OP_FREE_VECTORS, NULL);
 36                                 return (DDI_EAGAIN);
401                         }

403                         if (hdlp->ih_type == DDI_INTR_TYPE_MSIX) {
404                                 if (!(msix_p = i_ddi_get_msix(hdlp->ih_dip))) {
405                                         msix_p = pci_msix_init(hdlp->ih_dip);
406                                         if (msix_p) {
407                                                 i_ddi_set_msix(hdlp->ih_dip,
408                                                     msix_p);
409                                         } else {
410                                                 DDI_INTR_NEXDBG((CE_CONT,
411                                                     "pci_common_intr_ops: MSI-X"
412                                                     "table initilization failed"
413                                                     ", rdip 0x%p inum 0x%x\n",
414                                                     (void *)rdip,
415                                                     hdlp->ih_inum));

417                                                 rv = DDI_FAILURE;
418                                                 goto HANDLE_ALLOC_FAILURE;
 53                                                 (void) (*psm_intr_ops)(rdip,
 54                                                     hdlp,
 55                                                     PSM_INTR_OP_FREE_VECTORS,
 56                                                     NULL);

 58                                                 return (DDI_FAILURE);
419                                         }
420                                 }
421                         }

423                         if (pciepci) {
424                                 /* update priority in ispec */
425                                 isp = pci_intx_get_ispec(pdip, rdip,
426                                     (int)hdlp->ih_inum);
427                                 ispec = (struct intrspec *)isp;
428                                 if (ispec)
429                                         ispec->intrspec_pri = hdlp->ih_pri;
430                                 ++pcieb_intr_pri_counter;
431                         }

433                 } else
434                         return (DDI_FAILURE);
435                 break;

437 HANDLE_ALLOC_FAILURE:
438                 if (did_intr_vec_alloc == B_TRUE)
439                         (void) (*psm_intr_ops)(rdip, hdlp,
440                             PSM_INTR_OP_FREE_VECTORS, NULL);
441                 if (did_msi_cap_set == B_TRUE)
442                         i_ddi_set_msi_msix_cap_ptr(rdip, 0);
443                 if (did_pci_config_setup == B_TRUE) {
444                         (void) pci_config_teardown(&handle);
```



```
445                         i_ddi_set_pci_config_handle(rdip, NULL);
446                 }
447                 return (rv);

449 #endif /* ! codereview */
450         case DDI_INTROP_FREE:
451                 if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type) &&
452                     (psm_intr_ops != NULL)) {
453                         if (i_ddi_intr_get_current_nintrs(hdlp->ih_dip) - 1 ==
454                             0) {
455                                 if (handle = i_ddi_get_pci_config_handle(
456                                     rdip)) {
457                                         (void) pci_config_teardown(&handle);
458                                         i_ddi_set_pci_config_handle(rdip, NULL);
459                                 }
460                                 if (cap_ptr = i_ddi_get_msi_msix_cap_ptr(rdip))
461                                         i_ddi_set_msi_msix_cap_ptr(rdip, 0);
462                         }

464                         (void) (*psm_intr_ops)(rdip, hdlp,
465                             PSM_INTR_OP_FREE_VECTORS, NULL);

467                         if (hdlp->ih_type == DDI_INTR_TYPE_MSIX) {
468                                 msix_p = i_ddi_get_msix(hdlp->ih_dip);
469                                 if (msix_p &&
470                                     (i_ddi_intr_get_current_nintrs(
471                                     hdlp->ih_dip) - 1) == 0) {
472                                         pci_msix_fini(msix_p);
473                                         i_ddi_set_msix(hdlp->ih_dip, NULL);
474                                 }
475                         }
476                 } else if (hdlp->ih_type == DDI_INTR_TYPE_FIXED) {
477                         return (pci_free_intr_fixed(pdip, rdip, hdlp));
478                 } else
479                         return (DDI_FAILURE);
480                 break;
481         case DDI_INTROP_GETPRI:
482                 /* Get the priority */
483                 if (pci_get_priority(rdip, hdlp, &priority) != DDI_SUCCESS)
484                         return (DDI_FAILURE);
485                 DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
486                     "priority = 0x%x\n", priority));
487                 *(int *)result = priority;
488                 break;
489         case DDI_INTROP_SETPRI:
490                 /* Validate the interrupt priority passed */
491                 if (*(int *)result > LOCK_LEVEL)
492                         return (DDI_FAILURE);

494                 /* Ensure that PSM is all initialized */
495                 if (psm_intr_ops == NULL)
496                         return (DDI_FAILURE);

498                 isp = pci_intx_get_ispec(pdip, rdip, (int)hdlp->ih_inum);
499                 ispec = (struct intrspec *)isp;
500                 if (ispec == NULL)
501                         return (DDI_FAILURE);

503                 /* For fixed interrupts */
504                 if (hdlp->ih_type == DDI_INTR_TYPE_FIXED) {
505                         /* if interrupt is shared, return failure */
506                         ((ihdl_plat_t *)hdlp->ih_private)->ip_ispecp = ispec;
507                         psm_rval = (*psm_intr_ops)(rdip, hdlp,
508                             PSM_INTR_OP_GET_SHARED, &psm_status);
509                         /*
510                          * For fixed interrupts, the irq may not have been
```

```
511                      * allocated when SET_PRI is called, and the above
512                      * GET_SHARED op may return PSM_FAILURE. This is not
513                      * a real error and is ignored below.
514                      */
515                     if ((psm_rval != PSM_FAILURE) && (psm_status == 1)) {
516                             DDI_INTR_NEXDBG((CE_CONT,
517                                 "pci_common_intr_ops: "
518                                 "dip 0x%p cannot setpri, psm_rval=%d,"
519                                 "psm_status=%d\n", (void *)rdip, psm_rval,
520                                 psm_status));
521                             return (DDI_FAILURE);
522                     }
523             }

525             /* Change the priority */
526             if ((*psm_intr_ops)(rdip, hdlp, PSM_INTR_OP_SET_PRI, result) ==
527                 PSM_FAILURE)
528                     return (DDI_FAILURE);

530             /* update ispec */
531             ispec->intrspec_pri = *(int *)result;
532             break;
533     case DDI_INTROP_ADDISR:
534             /* update ispec */
535             isp = pci_intx_get_ispec(pdip, rdip, (int)hdlp->ih_inum);
536             ispec = (struct intrspec *)isp;
537             if (ispec) {
538                     ispec->intrspec_func = hdlp->ih_cb_func;
539                     ihdl_plat_datap = (ihdl_plat_t *)hdlp->ih_private;
540                     pci_kstat_create(&ihdl_plat_datap->ip_ksp, pdip, hdlp);
541             }
542             break;
543     case DDI_INTROP_REMISR:
544             /* Get the interrupt structure pointer */
545             isp = pci_intx_get_ispec(pdip, rdip, (int)hdlp->ih_inum);
546             ispec = (struct intrspec *)isp;
547             if (ispec) {
548                     ispec->intrspec_func = (uint_t (*)()) 0;
549                     ihdl_plat_datap = (ihdl_plat_t *)hdlp->ih_private;
550                     if (ihdl_plat_datap->ip_ksp != NULL)
551                             pci_kstat_delete(ihdl_plat_datap->ip_ksp);
552             }
553             break;
554     case DDI_INTROP_GETCAP:
555             /*
556              * First check the config space and/or
557              * MSI capability register(s)
558              */
559             if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type))
560                     pci_rval = pci_msi_get_cap(rdip, hdlp->ih_type,
561                         &pci_status);
562             else if (hdlp->ih_type == DDI_INTR_TYPE_FIXED)
563                     pci_rval = pci_intx_get_cap(rdip, &pci_status);

565             /* next check with PSM module */
566             if (psm_intr_ops != NULL)
567                     psm_rval = (*psm_intr_ops)(rdip, hdlp,
568                         PSM_INTR_OP_GET_CAP, &psm_status);

570             DDI_INTR_NEXDBG((CE_CONT, "pci: GETCAP returned psm_rval = %x, "
571                 "psm_status = %x, pci_rval = %x, pci_status = %x\n",
572                 psm_rval, psm_status, pci_rval, pci_status));

574             if (psm_rval == PSM_FAILURE && pci_rval == DDI_FAILURE) {
575                     *(int *)result = 0;
576                     return (DDI_FAILURE);
```



```
577             }

579             if (psm_rval == PSM_SUCCESS)
580                     *(int *)result = psm_status;

582             if (pci_rval == DDI_SUCCESS)
583                     *(int *)result |= pci_status;

585             DDI_INTR_NEXDBG((CE_CONT, "pci: GETCAP returned = %x\n",
586                 *(int *)result));
587             break;
588     case DDI_INTROP_SETCAP:
589             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
590                 "SETCAP cap=0x%x\n", *(int *)result));
591             if (psm_intr_ops == NULL)
592                     return (DDI_FAILURE);

594             if ((*psm_intr_ops)(rdip, hdlp, PSM_INTR_OP_SET_CAP, result)) {
595                     DDI_INTR_NEXDBG((CE_CONT, "GETCAP: psm_intr_ops"
596                         " returned failure\n"));
597                     return (DDI_FAILURE);
598             }
599             break;
600     case DDI_INTROP_ENABLE:
601             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: ENABLE\n"));
602             if (psm_intr_ops == NULL)
603                     return (DDI_FAILURE);

605             if (pci_enable_intr(pdip, rdip, hdlp, hdlp->ih_inum) !=
606                 DDI_SUCCESS)
607                     return (DDI_FAILURE);

609             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: ENABLE "
610                 "vector=0x%x\n", hdlp->ih_vector));
611             break;
612     case DDI_INTROP_DISABLE:
613             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: DISABLE\n"));
614             if (psm_intr_ops == NULL)
615                     return (DDI_FAILURE);

617             pci_disable_intr(pdip, rdip, hdlp, hdlp->ih_inum);
618             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: DISABLE "
619                 "vector = %x\n", hdlp->ih_vector));
620             break;
621     case DDI_INTROP_BLOCKENABLE:
622             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
623                 "BLOCKENABLE\n"));
624             if (hdlp->ih_type != DDI_INTR_TYPE_MSI) {
625                     DDI_INTR_NEXDBG((CE_CONT, "BLOCKENABLE: not MSI\n"));
626                     return (DDI_FAILURE);
627             }

629             /* Check if psm_intr_ops is NULL? */
630             if (psm_intr_ops == NULL)
631                     return (DDI_FAILURE);

633             count = hdlp->ih_scratch1;
634             h_array = (ddi_intr_handle_t *)hdlp->ih_scratch2;
635             for (i = 0; i < count; i++) {
636                     hdlp = (ddi_intr_handle_impl_t *)h_array[i];
637                     if (pci_enable_intr(pdip, rdip, hdlp,
638                         hdlp->ih_inum) != DDI_SUCCESS) {
639                             DDI_INTR_NEXDBG((CE_CONT, "BLOCKENABLE: "
640                                 "pci_enable_intr failed for %d\n", i));
641                             for (j = 0; j < i; j++) {
642                                     hdlp = (ddi_intr_handle_impl_t *)
```

```
643                                     h_array[j];
644                                 pci_disable_intr(pdip, rdip, hdlp,
645                                     hdlp->ih_inum);
646                             }
647                             return (DDI_FAILURE);
648                     }
649                     DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
650                         "BLOCKENABLE inum %x done\n", hdlp->ih_inum));
651             }
652             break;
653     case DDI_INTROP_BLOCKDISABLE:
654             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
655                 "BLOCKDISABLE\n"));
656             if (hdlp->ih_type != DDI_INTR_TYPE_MSI) {
657                     DDI_INTR_NEXDBG((CE_CONT, "BLOCKDISABLE: not MSI\n"));
658                     return (DDI_FAILURE);
659             }

661             /* Check if psm_intr_ops is present */
662             if (psm_intr_ops == NULL)
663                     return (DDI_FAILURE);

665             count = hdlp->ih_scratch1;
666             h_array = (ddi_intr_handle_t *)hdlp->ih_scratch2;
667             for (i = 0; i < count; i++) {
668                     hdlp = (ddi_intr_handle_impl_t *)h_array[i];
669                     pci_disable_intr(pdip, rdip, hdlp, hdlp->ih_inum);
670                     DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: "
671                         "BLOCKDISABLE inum %x done\n", hdlp->ih_inum));
672             }
673             break;
674     case DDI_INTROP_SETMASK:
675     case DDI_INTROP_CLRMASK:
676             /*
677              * First handle in the config space
678              */
679             if (intr_op == DDI_INTROP_SETMASK) {
680                     if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type))
681                             pci_status = pci_msi_set_mask(rdip,
682                                 hdlp->ih_type, hdlp->ih_inum);
683                     else if (hdlp->ih_type == DDI_INTR_TYPE_FIXED)
684                             pci_status = pci_intx_set_mask(rdip);
685             } else {
686                     if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type))
687                             pci_status = pci_msi_clr_mask(rdip,
688                                 hdlp->ih_type, hdlp->ih_inum);
689                     else if (hdlp->ih_type == DDI_INTR_TYPE_FIXED)
690                             pci_status = pci_intx_clr_mask(rdip);
691             }

693             /* For MSI/X; no need to check with PSM module */
694             if (hdlp->ih_type != DDI_INTR_TYPE_FIXED)
695                     return (pci_status);

697             /* For fixed interrupts only: handle config space first */
698             if (hdlp->ih_type == DDI_INTR_TYPE_FIXED &&
699                 pci_status == DDI_SUCCESS)
700                     break;

702             /* For fixed interrupts only: confer with PSM module next */
703             if (psm_intr_ops != NULL) {
704                     /* If interrupt is shared; do nothing */
705                     psm_rval = (*psm_intr_ops)(rdip, hdlp,
706                         PSM_INTR_OP_GET_SHARED, &psm_status);

708                     if (psm_rval == PSM_FAILURE || psm_status == 1)
```



```
709                             return (pci_status);

711                     /* Now, PSM module should try to set/clear the mask */
712                     if (intr_op == DDI_INTROP_SETMASK)
713                             psm_rval = (*psm_intr_ops)(rdip, hdlp,
714                                 PSM_INTR_OP_SET_MASK, NULL);
715                     else
716                             psm_rval = (*psm_intr_ops)(rdip, hdlp,
717                                 PSM_INTR_OP_CLEAR_MASK, NULL);
718             }
719             return ((psm_rval == PSM_FAILURE) ? DDI_FAILURE : DDI_SUCCESS);
720     case DDI_INTROP_GETPENDING:
721             /*
722              * First check the config space and/or
723              * MSI capability register(s)
724              */
725             if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type))
726                     pci_rval = pci_msi_get_pending(rdip, hdlp->ih_type,
727                         hdlp->ih_inum, &pci_status);
728             else if (hdlp->ih_type == DDI_INTR_TYPE_FIXED)
729                     pci_rval = pci_intx_get_pending(rdip, &pci_status);

731             /* On failure; next try with PSM module */
732             if (pci_rval != DDI_SUCCESS && psm_intr_ops != NULL)
733                     psm_rval = (*psm_intr_ops)(rdip, hdlp,
734                         PSM_INTR_OP_GET_PENDING, &psm_status);

736             DDI_INTR_NEXDBG((CE_CONT, "pci: GETPENDING returned "
737                 "psm_rval = %x, psm_status = %x, pci_rval = %x, "
738                 "pci_status = %x\n", psm_rval, psm_status, pci_rval,
739                 pci_status));
740             if (psm_rval == PSM_FAILURE && pci_rval == DDI_FAILURE) {
741                     *(int *)result = 0;
742                     return (DDI_FAILURE);
743             }

745             if (psm_rval != PSM_FAILURE)
746                     *(int *)result = psm_status;
747             else if (pci_rval != DDI_FAILURE)
748                     *(int *)result = pci_status;
749             DDI_INTR_NEXDBG((CE_CONT, "pci: GETPENDING returned = %x\n",
750                 *(int *)result));
751             break;
752     case DDI_INTROP_GETTARGET:
753             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: GETTARGET\n"));

755             bcopy(hdlp, &tmp_hdl, sizeof (ddi_intr_handle_impl_t));
756             tmp_hdl.ih_private = (void *)&intrinfo;
757             intrinfo.avgi_req_flags = PSMGI_INTRBY_DEFAULT;
758             intrinfo.avgi_req_flags |= PSMGI_REQ_CPUID;

760             if ((*psm_intr_ops)(rdip, &tmp_hdl, PSM_INTR_OP_GET_INTR,
761                 NULL) == PSM_FAILURE)
762                     return (DDI_FAILURE);

764             *(int *)result = intrinfo.avgi_cpu_id;
765             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: GETTARGET "
766                 "vector = 0x%x, cpu = 0x%x\n", hdlp->ih_vector,
767                 *(int *)result));
768             break;
769     case DDI_INTROP_SETTARGET:
770             DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: SETTARGET\n"));

772             bcopy(hdlp, &tmp_hdl, sizeof (ddi_intr_handle_impl_t));
773             tmp_hdl.ih_private = (void *)(uintptr_t)*(int *)result;
774             tmp_hdl.ih_flags = PSMGI_INTRBY_DEFAULT;
```

```
776                 if ((*psm_intr_ops)(rdip, &tmp_hdl, PSM_INTR_OP_SET_CPU,
777                     &psm_status) == PSM_FAILURE)
778                         return (DDI_FAILURE);
779 
780                 hdlp->ih_vector = tmp_hdl.ih_vector;
781                 DDI_INTR_NEXDBG((CE_CONT, "pci_common_intr_ops: SETTARGET "
782                     "vector = 0x%x\n", hdlp->ih_vector));
783                 break;
784         case DDI_INTROP_GETPOOL:
785                 /*
786                  * For MSI/X interrupts use global IRM pool if available.
787                  */
788                 if (apix_irm_pool_p && DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type)) {
789                         *(ddi_irm_pool_t **)result = apix_irm_pool_p;
790                         return (DDI_SUCCESS);
791                 }
792                 return (DDI_ENOTSUP);
793         default:
794                 return (i_ddi_intr_ops(pdip, rdip, intr_op, hdlp, result));
795         }
796 
797         return (DDI_SUCCESS);
798 }
799 
800 /*
801  * Allocate a vector for FIXED type interrupt.
802  */
803 int
804 pci_alloc_intr_fixed(dev_info_t *pdip, dev_info_t *rdip,
805     ddi_intr_handle_impl_t *hdlp, void *result)
806 {
807         struct intrspec         *ispec;
808         ddi_intr_handle_impl_t  info_hdl;
809         int                     ret;
810         int                     free_phdl = 0;
811         int                     pci_rval;
812         int                     pci_status = 0;
813         apic_get_type_t         type_info;
814 
815         if (psm_intr_ops == NULL)
816                 return (DDI_FAILURE);
817 
818         /* Figure out if this device supports MASKING */
819         pci_rval = pci_intx_get_cap(rdip, &pci_status);
820         if (pci_rval == DDI_SUCCESS && pci_status)
821                 hdlp->ih_cap |= pci_status;
822 
823         /*
824          * If the PSM module is "APIX" then pass the request for
825          * allocating the vector now.
826          */
827         bzero(&info_hdl, sizeof (ddi_intr_handle_impl_t));
828         info_hdl.ih_private = &type_info;
829         if ((*psm_intr_ops)(NULL, &info_hdl, PSM_INTR_OP_APIC_TYPE, NULL) ==
830             PSM_SUCCESS && strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
831                 ispec = (struct intrspec *)pci_intx_get_ispec(pdip, rdip,
832                     (int)hdlp->ih_inum);
833                 if (ispec == NULL)
834                         return (DDI_FAILURE);
835                 if (hdlp->ih_private == NULL) { /* allocate phdl structure */
836                         free_phdl = 1;
837                         i_ddi_alloc_intr_phdl(hdlp);
838                 }
839                 ((ihdl_plat_t *)hdlp->ih_private)->ip_ispecp = ispec;
840                 ret = (*psm_intr_ops)(rdip, hdlp,
```



```
841                     PSM_INTR_OP_ALLOC_VECTORS, result);
842                 if (free_phdl) { /* free up the phdl structure */
843                         free_phdl = 0;
844                         i_ddi_free_intr_phdl(hdlp);
845                         hdlp->ih_private = NULL;
846                 }
847         } else {
848                 /*
849                  * No APIX module; fall back to the old scheme where the
850                  * interrupt vector is allocated during ddi_enable_intr() call.
851                  */
852                 *(int *)result = 1;
853                 ret = DDI_SUCCESS;
854         }
855 
856         return (ret);
857 }
858 
859 /*
860  * Free up the vector for FIXED (legacy) type interrupt.
861  */
862 static int
863 pci_free_intr_fixed(dev_info_t *pdip, dev_info_t *rdip,
864     ddi_intr_handle_impl_t *hdlp)
865 {
866         struct intrspec                 *ispec;
867         ddi_intr_handle_impl_t          info_hdl;
868         int                             ret;
869         apic_get_type_t                 type_info;
870 
871         if (psm_intr_ops == NULL)
872                 return (DDI_FAILURE);
873 
874         /*
875          * If the PSM module is "APIX" then pass the request to it
876          * to free up the vector now.
877          */
878         bzero(&info_hdl, sizeof (ddi_intr_handle_impl_t));
879         info_hdl.ih_private = &type_info;
880         if ((*psm_intr_ops)(NULL, &info_hdl, PSM_INTR_OP_APIC_TYPE, NULL) ==
881             PSM_SUCCESS && strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
882                 ispec = (struct intrspec *)pci_intx_get_ispec(pdip, rdip,
883                     (int)hdlp->ih_inum);
884                 if (ispec == NULL)
885                         return (DDI_FAILURE);
886                 ((ihdl_plat_t *)hdlp->ih_private)->ip_ispecp = ispec;
887                 ret = (*psm_intr_ops)(rdip, hdlp,
888                     PSM_INTR_OP_FREE_VECTORS, NULL);
889         } else {
890                 /*
891                  * No APIX module; fall back to the old scheme where
892                  * the interrupt vector was already freed during
893                  * ddi_disable_intr() call.
894                  */
895                 ret = DDI_SUCCESS;
896         }
897 
898         return (ret);
899 }
900 
901 int
902 pci_get_intr_from_vecirq(apic_get_intr_t *intrinfo_p,
903     int vecirq, boolean_t is_irq)
904 {
905         ddi_intr_handle_impl_t  get_info_ii_hdl;
```

```
907         if (is_irq)
908                 intrinfo_p->avgi_req_flags |= PSMGI_INTRBY_IRQ;
909
910         /*
911          * For this locally-declared and used handle, ih_private will contain a
912          * pointer to apic_get_intr_t, not an ihdl_plat_t as used for
913          * global interrupt handling.
914          */
915         get_info_ii_hdl.ih_private = intrinfo_p;
916         get_info_ii_hdl.ih_vector = vecirq;
917
918         if ((*psm_intr_ops)(NULL, &get_info_ii_hdl,
919             PSM_INTR_OP_GET_INTR, NULL) == PSM_FAILURE)
920                 return (DDI_FAILURE);
921
922         return (DDI_SUCCESS);
923 }
924
925
926 int
927 pci_get_cpu_from_vecirq(int vecirq, boolean_t is_irq)
928 {
929         int rval;
930         apic_get_intr_t intrinfo;
931
932         intrinfo.avgi_req_flags = PSMGI_REQ_CPUID;
933         rval = pci_get_intr_from_vecirq(&intrinfo, vecirq, is_irq);
934
935         if (rval == DDI_SUCCESS)
936                 return (intrinfo.avgi_cpu_id);
937         else
938                 return (-1);
939 }
940
941
942 static int
943 pci_enable_intr(dev_info_t *pdip, dev_info_t *rdip,
944     ddi_intr_handle_impl_t *hdlp, uint32_t inum)
945 {
946         struct intrspec *ispec;
947         int             irq;
948         ihdl_plat_t     *ihdl_plat_datap = (ihdl_plat_t *)hdlp->ih_private;
949
950         DDI_INTR_NEXDBG((CE_CONT, "pci_enable_intr: hdlp %p inum %x\n",
951             (void *)hdlp, inum));
952
953         /* Translate the interrupt if needed */
954         ispec = (struct intrspec *)pci_intx_get_ispec(pdip, rdip, (int)inum);
955         if (ispec == NULL)
956                 return (DDI_FAILURE);
957         if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type)) {
958                 ispec->intrspec_vec = inum;
959                 ispec->intrspec_pri = hdlp->ih_pri;
960         }
961         ihdl_plat_datap->ip_ispecp = ispec;
962
963         /* translate the interrupt if needed */
964         if ((*psm_intr_ops)(rdip, hdlp, PSM_INTR_OP_XLATE_VECTOR, &irq) ==
965             PSM_FAILURE)
966                 return (DDI_FAILURE);
967         DDI_INTR_NEXDBG((CE_CONT, "pci_enable_intr: priority=%x irq=%x\n",
968             hdlp->ih_pri, irq));
969
970         /* Add the interrupt handler */
971         if (!add_avintr((void *)hdlp, hdlp->ih_pri, hdlp->ih_cb_func,
972             DEVI(rdip)->devi_name, irq, hdlp->ih_cb_arg1,
```



```
973             hdlp->ih_cb_arg2, &ihdl_plat_datap->ip_ticks, rdip))
974                 return (DDI_FAILURE);
975
976         hdlp->ih_vector = irq;
977
978         return (DDI_SUCCESS);
979 }
980
981
982 static void
983 pci_disable_intr(dev_info_t *pdip, dev_info_t *rdip,
984     ddi_intr_handle_impl_t *hdlp, uint32_t inum)
985 {
986         int             irq;
987         struct intrspec *ispec;
988         ihdl_plat_t     *ihdl_plat_datap = (ihdl_plat_t *)hdlp->ih_private;
989
990         DDI_INTR_NEXDBG((CE_CONT, "pci_disable_intr: \n"));
991         ispec = (struct intrspec *)pci_intx_get_ispec(pdip, rdip, (int)inum);
992         if (ispec == NULL)
993                 return;
994         if (DDI_INTR_IS_MSI_OR_MSIX(hdlp->ih_type)) {
995                 ispec->intrspec_vec = inum;
996                 ispec->intrspec_pri = hdlp->ih_pri;
997         }
998         ihdl_plat_datap->ip_ispecp = ispec;
999
1000         /* translate the interrupt if needed */
1001         (void) (*psm_intr_ops)(rdip, hdlp, PSM_INTR_OP_XLATE_VECTOR, &irq);
1002
1003         /* Disable the interrupt handler */
1004         rem_avintr((void *)hdlp, hdlp->ih_pri, hdlp->ih_cb_func, irq);
1005         ihdl_plat_datap->ip_ispecp = NULL;
1006 }
1007
1008 /*
1009  * Miscellaneous library function
1010  */
1011 int
1012 pci_common_get_reg_prop(dev_info_t *dip, pci_regspec_t *pci_rp)
1013 {
1014         int             i;
1015         int             number;
1016         int             assigned_addr_len;
1017         uint_t          phys_hi = pci_rp->pci_phys_hi;
1018         pci_regspec_t   *assigned_addr;
1019
1020         if (((phys_hi & PCI_REG_ADDR_M) == PCI_ADDR_CONFIG) ||
1021             (phys_hi & PCI_RELOCAT_B))
1022                 return (DDI_SUCCESS);
1023
1024         /*
1025          * the "reg" property specifies relocatable, get and interpret the
1026          * "assigned-addresses" property.
1027          */
1028         if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
1029             "assigned-addresses", (int **)&assigned_addr,
1030             (uint_t *)&assigned_addr_len) != DDI_PROP_SUCCESS)
1031                 return (DDI_FAILURE);
1032
1033         /*
1034          * Scan the "assigned-addresses" for one that matches the specified
1035          * "reg" property entry.
1036          */
1037         phys_hi &= PCI_CONF_ADDR_MASK;
1038         number = assigned_addr_len / (sizeof (pci_regspec_t) / sizeof (int));
```

```
1039         for (i = 0; i < number; i++) {
1040                 if ((assigned_addr[i].pci_phys_hi & PCI_CONF_ADDR_MASK) ==
1041                     phys_hi) {
1042                         pci_rp->pci_phys_mid = assigned_addr[i].pci_phys_mid;
1043                         pci_rp->pci_phys_low = assigned_addr[i].pci_phys_low;
1044                         ddi_prop_free(assigned_addr);
1045                         return (DDI_SUCCESS);
1046                 }
1047         }

1049         ddi_prop_free(assigned_addr);
1050         return (DDI_FAILURE);
1051 }


1054 /*
1055  * To handle PCI tool ioctls
1056  */

1058 /*ARGSUSED*/
1059 int
1060 pci_common_ioctl(dev_info_t *dip, dev_t dev, int cmd, intptr_t arg,
1061     int mode, cred_t *credp, int *rvalp)
1062 {
1063         minor_t minor = getminor(dev);
1064         int     rv = ENOTTY;

1066         switch (PCI_MINOR_NUM_TO_PCI_DEVNUM(minor)) {
1067         case PCI_TOOL_REG_MINOR_NUM:

1069                 switch (cmd) {
1070                 case PCITOOL_DEVICE_SET_REG:
1071                 case PCITOOL_DEVICE_GET_REG:

1073                         /* Require full privileges. */
1074                         if (secpolicy_kmdb(credp))
1075                                 rv = EPERM;
1076                         else
1077                                 rv = pcitool_dev_reg_ops(dip, (void *)arg,
1078                                     cmd, mode);
1079                         break;

1081                 case PCITOOL_NEXUS_SET_REG:
1082                 case PCITOOL_NEXUS_GET_REG:

1084                         /* Require full privileges. */
1085                         if (secpolicy_kmdb(credp))
1086                                 rv = EPERM;
1087                         else
1088                                 rv = pcitool_bus_reg_ops(dip, (void *)arg,
1089                                     cmd, mode);
1090                         break;
1091                 }
1092                 break;

1094         case PCI_TOOL_INTR_MINOR_NUM:

1096                 switch (cmd) {
1097                 case PCITOOL_DEVICE_SET_INTR:

1099                         /* Require PRIV_SYS_RES_CONFIG, same as psradm */
1100                         if (secpolicy_ponline(credp)) {
1101                                 rv = EPERM;
1102                                 break;
1103                         }
```



```
1105                 /*FALLTHRU*/
1106                 /* These require no special privileges. */
1107                 case PCITOOL_DEVICE_GET_INTR:
1108                 case PCITOOL_SYSTEM_INTR_INFO:
1109                         rv = pcitool_intr_admn(dip, (void *)arg, cmd, mode);
1110                         break;
1111                 }
1112                 break;

1114         default:
1115                 break;
1116         }

1118         return (rv);
1119 }


1122 int
1123 pci_common_ctlops_poke(peekpoke_ctlops_t *in_args)
1124 {
1125         size_t size = in_args->size;
1126         uintptr_t dev_addr = in_args->dev_addr;
1127         uintptr_t host_addr = in_args->host_addr;
1128         ddi_acc_impl_t *hp = (ddi_acc_impl_t *)in_args->handle;
1129         ddi_acc_hdl_t *hdlp = (ddi_acc_hdl_t *)in_args->handle;
1130         size_t repcount = in_args->repcount;
1131         uint_t flags = in_args->flags;
1132         int err = DDI_SUCCESS;

1134         /*
1135          * if no handle then this is a poke. We have to return failure here
1136          * as we have no way of knowing whether this is a MEM or IO space access
1137          */
1138         if (in_args->handle == NULL)
1139                 return (DDI_FAILURE);

1141         /*
1142          * rest of this function is actually for cautious puts
1143          */
1144         for (; repcount; repcount--) {
1145                 if (hp->ahi_acc_attr == DDI_ACCATTR_CONFIG_SPACE) {
1146                         switch (size) {
1147                         case sizeof (uint8_t):
1148                                 pci_config_wr8(hp, (uint8_t *)dev_addr,
1149                                     *(uint8_t *)host_addr);
1150                                 break;
1151                         case sizeof (uint16_t):
1152                                 pci_config_wr16(hp, (uint16_t *)dev_addr,
1153                                     *(uint16_t *)host_addr);
1154                                 break;
1155                         case sizeof (uint32_t):
1156                                 pci_config_wr32(hp, (uint32_t *)dev_addr,
1157                                     *(uint32_t *)host_addr);
1158                                 break;
1159                         case sizeof (uint64_t):
1160                                 pci_config_wr64(hp, (uint64_t *)dev_addr,
1161                                     *(uint64_t *)host_addr);
1162                                 break;
1163                         default:
1164                                 err = DDI_FAILURE;
1165                                 break;
1166                         }
1167                 } else if (hp->ahi_acc_attr & DDI_ACCATTR_IO_SPACE) {
1168                         if (hdlp->ah_acc.devacc_attr_endian_flags ==
1169                             DDI_STRUCTURE_BE_ACC) {
1170                                 switch (size) {
```

```
1171                                case sizeof (uint8_t):
1172                                        i_ddi_io_put8(hp,
1173                                            (uint8_t *)dev_addr,
1174                                            *(uint8_t *)host_addr);
1175                                        break;
1176                                case sizeof (uint16_t):
1177                                        i_ddi_io_swap_put16(hp,
1178                                            (uint16_t *)dev_addr,
1179                                            *(uint16_t *)host_addr);
1180                                        break;
1181                                case sizeof (uint32_t):
1182                                        i_ddi_io_swap_put32(hp,
1183                                            (uint32_t *)dev_addr,
1184                                            *(uint32_t *)host_addr);
1185                                        break;
1186                                /*
1187                                 * note the 64-bit case is a dummy
1188                                 * function - so no need to swap
1189                                 */
1190                                case sizeof (uint64_t):
1191                                        i_ddi_io_put64(hp,
1192                                            (uint64_t *)dev_addr,
1193                                            *(uint64_t *)host_addr);
1194                                        break;
1195                                default:
1196                                        err = DDI_FAILURE;
1197                                        break;
1198                                }
1199                        } else {
1200                                switch (size) {
1201                                case sizeof (uint8_t):
1202                                        i_ddi_io_put8(hp,
1203                                            (uint8_t *)dev_addr,
1204                                            *(uint8_t *)host_addr);
1205                                        break;
1206                                case sizeof (uint16_t):
1207                                        i_ddi_io_put16(hp,
1208                                            (uint16_t *)dev_addr,
1209                                            *(uint16_t *)host_addr);
1210                                        break;
1211                                case sizeof (uint32_t):
1212                                        i_ddi_io_put32(hp,
1213                                            (uint32_t *)dev_addr,
1214                                            *(uint32_t *)host_addr);
1215                                        break;
1216                                case sizeof (uint64_t):
1217                                        i_ddi_io_put64(hp,
1218                                            (uint64_t *)dev_addr,
1219                                            *(uint64_t *)host_addr);
1220                                        break;
1221                                default:
1222                                        err = DDI_FAILURE;
1223                                        break;
1224                                }
1225                        }
1226                } else {
1227                        if (hdlp->ah_acc.devacc_attr_endian_flags ==
1228                            DDI_STRUCTURE_BE_ACC) {
1229                                switch (size) {
1230                                case sizeof (uint8_t):
1231                                        *(uint8_t *)dev_addr =
1232                                            *(uint8_t *)host_addr;
1233                                        break;
1234                                case sizeof (uint16_t):
1235                                        *(uint16_t *)dev_addr =
1236                                            ddi_swap16(*(uint16_t *)host_addr);
```



```
1237                                        break;
1238                                case sizeof (uint32_t):
1239                                        *(uint32_t *)dev_addr =
1240                                            ddi_swap32(*(uint32_t *)host_addr);
1241                                        break;
1242                                case sizeof (uint64_t):
1243                                        *(uint64_t *)dev_addr =
1244                                            ddi_swap64(*(uint64_t *)host_addr);
1245                                        break;
1246                                default:
1247                                        err = DDI_FAILURE;
1248                                        break;
1249                                }
1250                        } else {
1251                                switch (size) {
1252                                case sizeof (uint8_t):
1253                                        *(uint8_t *)dev_addr =
1254                                            *(uint8_t *)host_addr;
1255                                        break;
1256                                case sizeof (uint16_t):
1257                                        *(uint16_t *)dev_addr =
1258                                            *(uint16_t *)host_addr;
1259                                        break;
1260                                case sizeof (uint32_t):
1261                                        *(uint32_t *)dev_addr =
1262                                            *(uint32_t *)host_addr;
1263                                        break;
1264                                case sizeof (uint64_t):
1265                                        *(uint64_t *)dev_addr =
1266                                            *(uint64_t *)host_addr;
1267                                        break;
1268                                default:
1269                                        err = DDI_FAILURE;
1270                                        break;
1271                                }
1272                        }
1273                }
1274                host_addr += size;
1275                if (flags == DDI_DEV_AUTOINCR)
1276                        dev_addr += size;
1277        }
1278        return (err);
1279 }


1282 int
1283 pci_fm_acc_setup(ddi_acc_hdl_t *hp, off_t offset, off_t len)
1284 {
1285        ddi_acc_impl_t  *ap = (ddi_acc_impl_t *)hp->ah_platform_private;

1287        /* endian-ness check */
1288        if (hp->ah_acc.devacc_attr_endian_flags == DDI_STRUCTURE_BE_ACC)
1289                return (DDI_FAILURE);

1291        /*
1292         * range check
1293         */
1294        if ((offset >= PCI_CONF_HDR_SIZE) ||
1295            (len > PCI_CONF_HDR_SIZE) ||
1296            (offset + len > PCI_CONF_HDR_SIZE))
1297                return (DDI_FAILURE);

1299        ap->ahi_acc_attr |= DDI_ACCATTR_CONFIG_SPACE;
1300        /*
1301         * always use cautious mechanism for config space gets
1302         */
```

```
1303         ap->ahi_get8 = i_ddi_caut_get8;
1304         ap->ahi_get16 = i_ddi_caut_get16;
1305         ap->ahi_get32 = i_ddi_caut_get32;
1306         ap->ahi_get64 = i_ddi_caut_get64;
1307         ap->ahi_rep_get8 = i_ddi_caut_rep_get8;
1308         ap->ahi_rep_get16 = i_ddi_caut_rep_get16;
1309         ap->ahi_rep_get32 = i_ddi_caut_rep_get32;
1310         ap->ahi_rep_get64 = i_ddi_caut_rep_get64;
1311         if (hp->ah_acc.devacc_attr_access == DDI_CAUTIOUS_ACC) {
1312                 ap->ahi_put8 = i_ddi_caut_put8;
1313                 ap->ahi_put16 = i_ddi_caut_put16;
1314                 ap->ahi_put32 = i_ddi_caut_put32;
1315                 ap->ahi_put64 = i_ddi_caut_put64;
1316                 ap->ahi_rep_put8 = i_ddi_caut_rep_put8;
1317                 ap->ahi_rep_put16 = i_ddi_caut_rep_put16;
1318                 ap->ahi_rep_put32 = i_ddi_caut_rep_put32;
1319                 ap->ahi_rep_put64 = i_ddi_caut_rep_put64;
1320         } else {
1321                 ap->ahi_put8 = pci_config_wr8;
1322                 ap->ahi_put16 = pci_config_wr16;
1323                 ap->ahi_put32 = pci_config_wr32;
1324                 ap->ahi_put64 = pci_config_wr64;
1325                 ap->ahi_rep_put8 = pci_config_rep_wr8;
1326                 ap->ahi_rep_put16 = pci_config_rep_wr16;
1327                 ap->ahi_rep_put32 = pci_config_rep_wr32;
1328                 ap->ahi_rep_put64 = pci_config_rep_wr64;
1329         }

1331         /* Initialize to default check/notify functions */
1332         ap->ahi_fault_check = i_ddi_acc_fault_check;
1333         ap->ahi_fault_notify = i_ddi_acc_fault_notify;
1334         ap->ahi_fault = 0;
1335         impl_acc_err_init(hp);
1336         return (DDI_SUCCESS);
1337 }


1340 int
1341 pci_common_ctlops_peek(peekpoke_ctlops_t *in_args)
1342 {
1343         size_t size = in_args->size;
1344         uintptr_t dev_addr = in_args->dev_addr;
1345         uintptr_t host_addr = in_args->host_addr;
1346         ddi_acc_impl_t *hp = (ddi_acc_impl_t *)in_args->handle;
1347         ddi_acc_hdl_t *hdlp = (ddi_acc_hdl_t *)in_args->handle;
1348         size_t repcount = in_args->repcount;
1349         uint_t flags = in_args->flags;
1350         int err = DDI_SUCCESS;

1352         /*
1353          * if no handle then this is a peek. We have to return failure here
1354          * as we have no way of knowing whether this is a MEM or IO space access
1355          */
1356         if (in_args->handle == NULL)
1357                 return (DDI_FAILURE);

1359         for (; repcount; repcount--) {
1360                 if (hp->ahi_acc_attr == DDI_ACCATTR_CONFIG_SPACE) {
1361                         switch (size) {
1362                         case sizeof (uint8_t):
1363                                 *(uint8_t *)host_addr = pci_config_rd8(hp,
1364                                     (uint8_t *)dev_addr);
1365                                 break;
1366                         case sizeof (uint16_t):
1367                                 *(uint16_t *)host_addr = pci_config_rd16(hp,
1368                                     (uint16_t *)dev_addr);
```



```
1369                                 break;
1370                         case sizeof (uint32_t):
1371                                 *(uint32_t *)host_addr = pci_config_rd32(hp,
1372                                     (uint32_t *)dev_addr);
1373                                 break;
1374                         case sizeof (uint64_t):
1375                                 *(uint64_t *)host_addr = pci_config_rd64(hp,
1376                                     (uint64_t *)dev_addr);
1377                                 break;
1378                         default:
1379                                 err = DDI_FAILURE;
1380                                 break;
1381                         }
1382                 } else if (hp->ahi_acc_attr & DDI_ACCATTR_IO_SPACE) {
1383                         if (hdlp->ah_acc.devacc_attr_endian_flags ==
1384                             DDI_STRUCTURE_BE_ACC) {
1385                                 switch (size) {
1386                                 case sizeof (uint8_t):
1387                                         *(uint8_t *)host_addr =
1388                                             i_ddi_io_get8(hp,
1389                                             (uint8_t *)dev_addr);
1390                                         break;
1391                                 case sizeof (uint16_t):
1392                                         *(uint16_t *)host_addr =
1393                                             i_ddi_io_swap_get16(hp,
1394                                             (uint16_t *)dev_addr);
1395                                         break;
1396                                 case sizeof (uint32_t):
1397                                         *(uint32_t *)host_addr =
1398                                             i_ddi_io_swap_get32(hp,
1399                                             (uint32_t *)dev_addr);
1400                                         break;
1401                                 /*
1402                                  * note the 64-bit case is a dummy
1403                                  * function - so no need to swap
1404                                  */
1405                                 case sizeof (uint64_t):
1406                                         *(uint64_t *)host_addr =
1407                                             i_ddi_io_get64(hp,
1408                                             (uint64_t *)dev_addr);
1409                                         break;
1410                                 default:
1411                                         err = DDI_FAILURE;
1412                                         break;
1413                                 }
1414                         } else {
1415                                 switch (size) {
1416                                 case sizeof (uint8_t):
1417                                         *(uint8_t *)host_addr =
1418                                             i_ddi_io_get8(hp,
1419                                             (uint8_t *)dev_addr);
1420                                         break;
1421                                 case sizeof (uint16_t):
1422                                         *(uint16_t *)host_addr =
1423                                             i_ddi_io_get16(hp,
1424                                             (uint16_t *)dev_addr);
1425                                         break;
1426                                 case sizeof (uint32_t):
1427                                         *(uint32_t *)host_addr =
1428                                             i_ddi_io_get32(hp,
1429                                             (uint32_t *)dev_addr);
1430                                         break;
1431                                 case sizeof (uint64_t):
1432                                         *(uint64_t *)host_addr =
1433                                             i_ddi_io_get64(hp,
1434                                             (uint64_t *)dev_addr);
```

```
1435                                    break;
1436                            default:
1437                                    err = DDI_FAILURE;
1438                                    break;
1439                            }
1440                    }
1441            } else {
1442                    if (hdlp->ah_acc.devacc_attr_endian_flags ==
1443                        DDI_STRUCTURE_BE_ACC) {
1444                            switch (in_args->size) {
1445                            case sizeof (uint8_t):
1446                                    *(uint8_t *)host_addr =
1447                                        *(uint8_t *)dev_addr;
1448                                    break;
1449                            case sizeof (uint16_t):
1450                                    *(uint16_t *)host_addr =
1451                                        ddi_swap16(*(uint16_t *)dev_addr);
1452                                    break;
1453                            case sizeof (uint32_t):
1454                                    *(uint32_t *)host_addr =
1455                                        ddi_swap32(*(uint32_t *)dev_addr);
1456                                    break;
1457                            case sizeof (uint64_t):
1458                                    *(uint64_t *)host_addr =
1459                                        ddi_swap64(*(uint64_t *)dev_addr);
1460                                    break;
1461                            default:
1462                                    err = DDI_FAILURE;
1463                                    break;
1464                            }
1465                    } else {
1466                            switch (in_args->size) {
1467                            case sizeof (uint8_t):
1468                                    *(uint8_t *)host_addr =
1469                                        *(uint8_t *)dev_addr;
1470                                    break;
1471                            case sizeof (uint16_t):
1472                                    *(uint16_t *)host_addr =
1473                                        *(uint16_t *)dev_addr;
1474                                    break;
1475                            case sizeof (uint32_t):
1476                                    *(uint32_t *)host_addr =
1477                                        *(uint32_t *)dev_addr;
1478                                    break;
1479                            case sizeof (uint64_t):
1480                                    *(uint64_t *)host_addr =
1481                                        *(uint64_t *)dev_addr;
1482                                    break;
1483                            default:
1484                                    err = DDI_FAILURE;
1485                                    break;
1486                            }
1487                    }
1488            }
1489            host_addr += size;
1490            if (flags == DDI_DEV_AUTOINCR)
1491                    dev_addr += size;
1492        }
1493        return (err);
1494 }

1496 /*ARGSUSED*/
1497 int
1498 pci_common_peekpoke(dev_info_t *dip, dev_info_t *rdip,
1499         ddi_ctl_enum_t ctlop, void *arg, void *result)
1500 {
```



```
1501         if (ctlop == DDI_CTLOPS_PEEK)
1502                 return (pci_common_ctlops_peek((peekpoke_ctlops_t *)arg));
1503         else
1504                 return (pci_common_ctlops_poke((peekpoke_ctlops_t *)arg));
1505 }

1507 /*
1508  * These are the get and put functions to be shared with drivers. The
1509  * mutex locking is done inside the functions referenced, rather than
1510  * here, and is thus shared across PCI child drivers and any other
1511  * consumers of PCI config space (such as the ACPI subsystem).
1512  *
1513  * The configuration space addresses come in as pointers.  This is fine on
1514  * a 32-bit system, where the VM space and configuration space are the same
1515  * size.  It's not such a good idea on a 64-bit system, where memory
1516  * addresses are twice as large as configuration space addresses.  At some
1517  * point in the call tree we need to take a stand and say "you are 32-bit
1518  * from this time forth", and this seems like a nice self-contained place.
1519  */

1521 uint8_t
1522 pci_config_rd8(ddi_acc_impl_t *hdlp, uint8_t *addr)
1523 {
1524         pci_acc_cfblk_t *cfp;
1525         uint8_t rval;
1526         int reg;

1528         ASSERT64(((uintptr_t)addr >> 32) == 0);

1530         reg = (int)(uintptr_t)addr;

1532         cfp = (pci_acc_cfblk_t *)&hdlp->ahi_common.ah_bus_private;

1534         rval = (*pci_getb_func)(cfp->c_busnum, cfp->c_devnum, cfp->c_funcnum,
1535             reg);

1537         return (rval);
1538 }

1540 void
1541 pci_config_rep_rd8(ddi_acc_impl_t *hdlp, uint8_t *host_addr,
1542         uint8_t *dev_addr, size_t repcount, uint_t flags)
1543 {
1544         uint8_t *h, *d;

1546         h = host_addr;
1547         d = dev_addr;

1549         if (flags == DDI_DEV_AUTOINCR)
1550                 for (; repcount; repcount--)
1551                         *h++ = pci_config_rd8(hdlp, d++);
1552         else
1553                 for (; repcount; repcount--)
1554                         *h++ = pci_config_rd8(hdlp, d);
1555 }

1557 uint16_t
1558 pci_config_rd16(ddi_acc_impl_t *hdlp, uint16_t *addr)
1559 {
1560         pci_acc_cfblk_t *cfp;
1561         uint16_t rval;
1562         int reg;

1564         ASSERT64(((uintptr_t)addr >> 32) == 0);

1566         reg = (int)(uintptr_t)addr;
```

```
1568         cfp = (pci_acc_cfblk_t *)&hdlp->ahi_common.ah_bus_private;

1570         rval = (*pci_getw_func)(cfp->c_busnum, cfp->c_devnum, cfp->c_funcnum,
1571             reg);

1573         return (rval);
1574 }

1576 void
1577 pci_config_rep_rd16(ddi_acc_impl_t *hdlp, uint16_t *host_addr,
1578         uint16_t *dev_addr, size_t repcount, uint_t flags)
1579 {
1580         uint16_t *h, *d;

1582         h = host_addr;
1583         d = dev_addr;

1585         if (flags == DDI_DEV_AUTOINCR)
1586                 for (; repcount; repcount--)
1587                         *h++ = pci_config_rd16(hdlp, d++);
1588         else
1589                 for (; repcount; repcount--)
1590                         *h++ = pci_config_rd16(hdlp, d);
1591 }

1593 uint32_t
1594 pci_config_rd32(ddi_acc_impl_t *hdlp, uint32_t *addr)
1595 {
1596         pci_acc_cfblk_t *cfp;
1597         uint32_t rval;
1598         int reg;

1600         ASSERT64(((uintptr_t)addr >> 32) == 0);

1602         reg = (int)(uintptr_t)addr;

1604         cfp = (pci_acc_cfblk_t *)&hdlp->ahi_common.ah_bus_private;

1606         rval = (*pci_getl_func)(cfp->c_busnum, cfp->c_devnum,
1607             cfp->c_funcnum, reg);

1609         return (rval);
1610 }

1612 void
1613 pci_config_rep_rd32(ddi_acc_impl_t *hdlp, uint32_t *host_addr,
1614         uint32_t *dev_addr, size_t repcount, uint_t flags)
1615 {
1616         uint32_t *h, *d;

1618         h = host_addr;
1619         d = dev_addr;

1621         if (flags == DDI_DEV_AUTOINCR)
1622                 for (; repcount; repcount--)
1623                         *h++ = pci_config_rd32(hdlp, d++);
1624         else
1625                 for (; repcount; repcount--)
1626                         *h++ = pci_config_rd32(hdlp, d);
1627 }


1630 void
1631 pci_config_wr8(ddi_acc_impl_t *hdlp, uint8_t *addr, uint8_t value)
1632 {
```



```
1633         pci_acc_cfblk_t *cfp;
1634         int reg;

1636         ASSERT64(((uintptr_t)addr >> 32) == 0);

1638         reg = (int)(uintptr_t)addr;

1640         cfp = (pci_acc_cfblk_t *)&hdlp->ahi_common.ah_bus_private;

1642         (*pci_putb_func)(cfp->c_busnum, cfp->c_devnum,
1643             cfp->c_funcnum, reg, value);
1644 }

1646 void
1647 pci_config_rep_wr8(ddi_acc_impl_t *hdlp, uint8_t *host_addr,
1648         uint8_t *dev_addr, size_t repcount, uint_t flags)
1649 {
1650         uint8_t *h, *d;

1652         h = host_addr;
1653         d = dev_addr;

1655         if (flags == DDI_DEV_AUTOINCR)
1656                 for (; repcount; repcount--)
1657                         pci_config_wr8(hdlp, d++, *h++);
1658         else
1659                 for (; repcount; repcount--)
1660                         pci_config_wr8(hdlp, d, *h++);
1661 }

1663 void
1664 pci_config_wr16(ddi_acc_impl_t *hdlp, uint16_t *addr, uint16_t value)
1665 {
1666         pci_acc_cfblk_t *cfp;
1667         int reg;

1669         ASSERT64(((uintptr_t)addr >> 32) == 0);

1671         reg = (int)(uintptr_t)addr;

1673         cfp = (pci_acc_cfblk_t *)&hdlp->ahi_common.ah_bus_private;

1675         (*pci_putw_func)(cfp->c_busnum, cfp->c_devnum,
1676             cfp->c_funcnum, reg, value);
1677 }

1679 void
1680 pci_config_rep_wr16(ddi_acc_impl_t *hdlp, uint16_t *host_addr,
1681         uint16_t *dev_addr, size_t repcount, uint_t flags)
1682 {
1683         uint16_t *h, *d;

1685         h = host_addr;
1686         d = dev_addr;

1688         if (flags == DDI_DEV_AUTOINCR)
1689                 for (; repcount; repcount--)
1690                         pci_config_wr16(hdlp, d++, *h++);
1691         else
1692                 for (; repcount; repcount--)
1693                         pci_config_wr16(hdlp, d, *h++);
1694 }

1696 void
1697 pci_config_wr32(ddi_acc_impl_t *hdlp, uint32_t *addr, uint32_t value)
1698 {
```

```
1699         pci_acc_cfblk_t *cfp;
1700         int reg;

1702         ASSERT64(((uintptr_t)addr >> 32) == 0);

1704         reg = (int)(uintptr_t)addr;

1706         cfp = (pci_acc_cfblk_t *)&hdlp->ahi_common.ah_bus_private;

1708         (*pci_putl_func)(cfp->c_busnum, cfp->c_devnum,
1709             cfp->c_funcnum, reg, value);
1710 }

1712 void
1713 pci_config_rep_wr32(ddi_acc_impl_t *hdlp, uint32_t *host_addr,
1714         uint32_t *dev_addr, size_t repcount, uint_t flags)
1715 {
1716         uint32_t *h, *d;

1718         h = host_addr;
1719         d = dev_addr;

1721         if (flags == DDI_DEV_AUTOINCR)
1722                 for (; repcount; repcount--)
1723                         pci_config_wr32(hdlp, d++, *h++);
1724         else
1725                 for (; repcount; repcount--)
1726                         pci_config_wr32(hdlp, d, *h++);
1727 }

1729 uint64_t
1730 pci_config_rd64(ddi_acc_impl_t *hdlp, uint64_t *addr)
1731 {
1732         uint32_t lw_val;
1733         uint32_t hi_val;
1734         uint32_t *dp;
1735         uint64_t val;

1737         dp = (uint32_t *)addr;
1738         lw_val = pci_config_rd32(hdlp, dp);
1739         dp++;
1740         hi_val = pci_config_rd32(hdlp, dp);
1741         val = ((uint64_t)hi_val << 32) | lw_val;
1742         return (val);
1743 }

1745 void
1746 pci_config_wr64(ddi_acc_impl_t *hdlp, uint64_t *addr, uint64_t value)
1747 {
1748         uint32_t lw_val;
1749         uint32_t hi_val;
1750         uint32_t *dp;

1752         dp = (uint32_t *)addr;
1753         lw_val = (uint32_t)(value & 0xffffffff);
1754         hi_val = (uint32_t)(value >> 32);
1755         pci_config_wr32(hdlp, dp, lw_val);
1756         dp++;
1757         pci_config_wr32(hdlp, dp, hi_val);
1758 }

1760 void
1761 pci_config_rep_rd64(ddi_acc_impl_t *hdlp, uint64_t *host_addr,
1762         uint64_t *dev_addr, size_t repcount, uint_t flags)
1763 {
1764         if (flags == DDI_DEV_AUTOINCR) {
```



```
1765                 for (; repcount; repcount--)
1766                         *host_addr++ = pci_config_rd64(hdlp, dev_addr++);
1767         } else {
1768                 for (; repcount; repcount--)
1769                         *host_addr++ = pci_config_rd64(hdlp, dev_addr);
1770         }
1771 }

1773 void
1774 pci_config_rep_wr64(ddi_acc_impl_t *hdlp, uint64_t *host_addr,
1775         uint64_t *dev_addr, size_t repcount, uint_t flags)
1776 {
1777         if (flags == DDI_DEV_AUTOINCR) {
1778                 for (; repcount; repcount--)
1779                         pci_config_wr64(hdlp, host_addr++, *dev_addr++);
1780         } else {
1781                 for (; repcount; repcount--)
1782                         pci_config_wr64(hdlp, host_addr++, *dev_addr);
1783         }
1784 }
```