****
438259 Thu Oct 23 10:42:09 2014
new/usr/src/uts/common/dtrace/dtrace.c
5255 uts shouldn't open-code ISP2
****
_____unchanged_portion_omitted_

```
12909 /*
12910  * The dof_hdr_t passed to dtrace_dof_slurp() should be a partially validated
12911  * header:  it should be at the front of a memory region that is at least
12912  * sizeof (dof_hdr_t) in size -- and then at least dof_hdr.dofh_loadsz in
12913  * size.  It need not be validated in any other way.
12914  */
12915 static int
12916 dtrace_dof_slurp(dof_hdr_t *dof, dtrace_vstate_t *vstate, cred_t *cr,
12917     dtrace_enabling_t **enabp, uint64_t ubase, int noprobes)
12918 {
12919         uint64_t len = dof->dofh_loadsz, seclen;
12920         uintptr_t daddr = (uintptr_t)dof;
12921         dtrace_ecbdesc_t *ep;
12922         dtrace_enabling_t *enab;
12923         uint_t i;

12925         ASSERT(MUTEX_HELD(&dtrace_lock));
12926         ASSERT(dof->dofh_loadsz >= sizeof (dof_hdr_t));

12928         /*
12929          * Check the DOF header identification bytes.  In addition to checking
12930          * valid settings, we also verify that unused bits/bytes are zeroed so
12931          * we can use them later without fear of regressing existing binaries.
12932          */
12933         if (bcmp(&dof->dofh_ident[DOF_ID_MAG0],
12934             DOF_MAG_STRING, DOF_MAG_STRLEN) != 0) {
12935                 dtrace_dof_error(dof, "DOF magic string mismatch");
12936                 return (-1);
12937         }

12939         if (dof->dofh_ident[DOF_ID_MODEL] != DOF_MODEL_ILP32 &&
12940             dof->dofh_ident[DOF_ID_MODEL] != DOF_MODEL_LP64) {
12941                 dtrace_dof_error(dof, "DOF has invalid data model");
12942                 return (-1);
12943         }

12945         if (dof->dofh_ident[DOF_ID_ENCODING] != DOF_ENCODE_NATIVE) {
12946                 dtrace_dof_error(dof, "DOF encoding mismatch");
12947                 return (-1);
12948         }

12950         if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
12951             dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_2) {
12952                 dtrace_dof_error(dof, "DOF version mismatch");
12953                 return (-1);
12954         }

12956         if (dof->dofh_ident[DOF_ID_DIFVERS] != DIF_VERSION_2) {
12957                 dtrace_dof_error(dof, "DOF uses unsupported instruction set");
12958                 return (-1);
12959         }

12961         if (dof->dofh_ident[DOF_ID_DIFIREG] > DIF_DIR_NREGS) {
12962                 dtrace_dof_error(dof, "DOF uses too many integer registers");
12963                 return (-1);
12964         }

12966         if (dof->dofh_ident[DOF_ID_DIFTREG] > DIF_DTR_NREGS) {
12967                 dtrace_dof_error(dof, "DOF uses too many tuple registers");
```



```
12968                 return (-1);
12969         }

12971         for (i = DOF_ID_PAD; i < DOF_ID_SIZE; i++) {
12972                 if (dof->dofh_ident[i] != 0) {
12973                         dtrace_dof_error(dof, "DOF has invalid ident byte set");
12974                         return (-1);
12975                 }
12976         }

12978         if (dof->dofh_flags & ~DOF_FL_VALID) {
12979                 dtrace_dof_error(dof, "DOF has invalid flag bits set");
12980                 return (-1);
12981         }

12983         if (dof->dofh_secsize == 0) {
12984                 dtrace_dof_error(dof, "zero section header size");
12985                 return (-1);
12986         }

12988         /*
12989          * Check that the section headers don't exceed the amount of DOF
12990          * data.  Note that we cast the section size and number of sections
12991          * to uint64_t's to prevent possible overflow in the multiplication.
12992          */
12993         seclen = (uint64_t)dof->dofh_secnum * (uint64_t)dof->dofh_secsize;

12995         if (dof->dofh_secoff > len || seclen > len ||
12996             dof->dofh_secoff + seclen > len) {
12997                 dtrace_dof_error(dof, "truncated section headers");
12998                 return (-1);
12999         }

13001         if (!IS_P2ALIGNED(dof->dofh_secoff, sizeof (uint64_t))) {
13002                 dtrace_dof_error(dof, "misaligned section headers");
13003                 return (-1);
13004         }

13006         if (!IS_P2ALIGNED(dof->dofh_secsize, sizeof (uint64_t))) {
13007                 dtrace_dof_error(dof, "misaligned section size");
13008                 return (-1);
13009         }

13011         /*
13012          * Take an initial pass through the section headers to be sure that
13013          * the headers don't have stray offsets.  If the 'noprobes' flag is
13014          * set, do not permit sections relating to providers, probes, or args.
13015          */
13016         for (i = 0; i < dof->dofh_secnum; i++) {
13017                 dof_sec_t *sec = (dof_sec_t *)(daddr +
13018                     (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

13020                 if (noprobes) {
13021                         switch (sec->dofs_type) {
13022                         case DOF_SECT_PROVIDER:
13023                         case DOF_SECT_PROBES:
13024                         case DOF_SECT_PRARGS:
13025                         case DOF_SECT_PROFFS:
13026                                 dtrace_dof_error(dof, "illegal sections "
13027                                     "for enabling");
13028                                 return (-1);
13029                         }
13030                 }

13032                 if (DOF_SEC_ISLOADABLE(sec->dofs_type) &&
13033                     !(sec->dofs_flags & DOF_SECF_LOAD)) {
```

```
13034                           dtrace_dof_error(dof, "loadable section with load "
13035                               "flag unset");
13036                           return (-1);
13037                   }

13039                   if (!(sec->dofs_flags & DOF_SECF_LOAD))
13040                           continue; /* just ignore non-loadable sections */

13042                   if (!ISP2(sec->dofs_align)) {
13042                   if (sec->dofs_align & (sec->dofs_align - 1)) {
13043                           dtrace_dof_error(dof, "bad section alignment");
13044                           return (-1);
13045                   }

13047                   if (sec->dofs_offset & (sec->dofs_align - 1)) {
13048                           dtrace_dof_error(dof, "misaligned section");
13049                           return (-1);
13050                   }

13052                   if (sec->dofs_offset > len || sec->dofs_size > len ||
13053                       sec->dofs_offset + sec->dofs_size > len) {
13054                           dtrace_dof_error(dof, "corrupt section header");
13055                           return (-1);
13056                   }

13058                   if (sec->dofs_type == DOF_SECT_STRTAB && *((char *)daddr +
13059                       sec->dofs_offset + sec->dofs_size - 1) != '\0') {
13060                           dtrace_dof_error(dof, "non-terminating string table");
13061                           return (-1);
13062                   }
13063           }

13065           /*
13066            * Take a second pass through the sections and locate and perform any
13067            * relocations that are present.  We do this after the first pass to
13068            * be sure that all sections have had their headers validated.
13069            */
13070           for (i = 0; i < dof->dofh_secnum; i++) {
13071                   dof_sec_t *sec = (dof_sec_t *)(daddr +
13072                       (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

13074                   if (!(sec->dofs_flags & DOF_SECF_LOAD))
13075                           continue; /* skip sections that are not loadable */

13077                   switch (sec->dofs_type) {
13078                   case DOF_SECT_URELHDR:
13079                           if (dtrace_dof_relocate(dof, sec, ubase) != 0)
13080                                   return (-1);
13081                           break;
13082                   }
13083           }

13085           if ((enab = *enabp) == NULL)
13086                   enab = *enabp = dtrace_enabling_create(vstate);

13088           for (i = 0; i < dof->dofh_secnum; i++) {
13089                   dof_sec_t *sec = (dof_sec_t *)(daddr +
13090                       (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

13092                   if (sec->dofs_type != DOF_SECT_ECBDESC)
13093                           continue;

13095                   if ((ep = dtrace_dof_ecbdesc(dof, sec, vstate, cr)) == NULL) {
13096                           dtrace_enabling_destroy(enab);
13097                           *enabp = NULL;
13098                           return (-1);
```



```
13099                   }

13101                   dtrace_enabling_add(enab, ep);
13102           }

13104           return (0);
13105   }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   62703 Thu Oct 23 10:42:09 2014
new/usr/src/uts/common/dtrace/fasttrap.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

2116 static int
2117 fasttrap_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
2118 {
2119         ulong_t nent;

2121         switch (cmd) {
2122         case DDI_ATTACH:
2123                 break;
2124         case DDI_RESUME:
2125                 return (DDI_SUCCESS);
2126         default:
2127                 return (DDI_FAILURE);
2128         }

2130         if (ddi_create_minor_node(devi, "fasttrap", S_IFCHR, 0,
2131             DDI_PSEUDO, NULL) == DDI_FAILURE) {
2132                 ddi_remove_minor_node(devi, NULL);
2133                 return (DDI_FAILURE);
2134         }

2136         ddi_report_dev(devi);
2137         fasttrap_devi = devi;

2139         /*
2140          * Install our hooks into fork(2), exec(2), and exit(2).
2141          */
2142         dtrace_fasttrap_fork_ptr = &fasttrap_fork;
2143         dtrace_fasttrap_exit_ptr = &fasttrap_exec_exit;
2144         dtrace_fasttrap_exec_ptr = &fasttrap_exec_exit;

2146         fasttrap_max = ddi_getprop(DDI_DEV_T_ANY, devi, DDI_PROP_DONTPASS,
2147             "fasttrap-max-probes", FASTTRAP_MAX_DEFAULT);
2148         fasttrap_total = 0;

2150         /*
2151          * Conjure up the tracepoints hashtable...
2152          */
2153         nent = ddi_getprop(DDI_DEV_T_ANY, devi, DDI_PROP_DONTPASS,
2154             "fasttrap-hash-size", FASTTRAP_TPOINTS_DEFAULT_SIZE);

2156         if (nent == 0 || nent > 0x1000000)
2157                 nent = FASTTRAP_TPOINTS_DEFAULT_SIZE;

2159         if (ISP2(nent))
2159         if ((nent & (nent - 1)) == 0)
2160                 fasttrap_tpoints.fth_nent = nent;
2161         else
2162                 fasttrap_tpoints.fth_nent = 1 << fasttrap_highbit(nent);
2163         ASSERT(fasttrap_tpoints.fth_nent > 0);
2164         fasttrap_tpoints.fth_mask = fasttrap_tpoints.fth_nent - 1;
2165         fasttrap_tpoints.fth_table = kmem_zalloc(fasttrap_tpoints.fth_nent *
2166             sizeof (fasttrap_bucket_t), KM_SLEEP);

2168         /*
2169          * ... and the providers hash table...
2170          */
2171         nent = FASTTRAP_PROVIDERS_DEFAULT_SIZE;
2172         if (ISP2(nent))
2172         if ((nent & (nent - 1)) == 0)
```



```
2173                 fasttrap_provs.fth_nent = nent;
2174         else
2175                 fasttrap_provs.fth_nent = 1 << fasttrap_highbit(nent);
2176         ASSERT(fasttrap_provs.fth_nent > 0);
2177         fasttrap_provs.fth_mask = fasttrap_provs.fth_nent - 1;
2178         fasttrap_provs.fth_table = kmem_zalloc(fasttrap_provs.fth_nent *
2179             sizeof (fasttrap_bucket_t), KM_SLEEP);

2181         /*
2182          * ... and the procs hash table.
2183          */
2184         nent = FASTTRAP_PROCS_DEFAULT_SIZE;
2185         if (ISP2(nent))
2185         if ((nent & (nent - 1)) == 0)
2186                 fasttrap_procs.fth_nent = nent;
2187         else
2188                 fasttrap_procs.fth_nent = 1 << fasttrap_highbit(nent);
2189         ASSERT(fasttrap_procs.fth_nent > 0);
2190         fasttrap_procs.fth_mask = fasttrap_procs.fth_nent - 1;
2191         fasttrap_procs.fth_table = kmem_zalloc(fasttrap_procs.fth_nent *
2192             sizeof (fasttrap_bucket_t), KM_SLEEP);

2194         (void) dtrace_meta_register("fasttrap", &fasttrap_mops, NULL,
2195             &fasttrap_meta_id);

2197         return (DDI_SUCCESS);
2198 }
______unchanged_portion_omitted_
```

```
**********************************************************
   27077 Thu Oct 23 10:42:10 2014
new/usr/src/uts/common/fs/dcfs/dc_vnops.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 919 /*
 920  * Return shadow vnode with the given vp as its subordinate
 921  */
 922 struct vnode *
 923 decompvp(struct vnode *vp, cred_t *cred, caller_context_t *ctp)
 924 {
 925         struct dcnode *dp, *ndp;
 926         struct comphdr thdr, *hdr;
 927         struct kmem_cache **cpp;
 928         struct vattr vattr;
 929         size_t hdrsize, bsize;
 930         int error;

 932         /*
 933          * See if we have an existing shadow
 934          * If none, we have to manufacture one
 935          */
 936         mutex_enter(&dctable_lock);
 937         dp = dcfind(vp);
 938         mutex_exit(&dctable_lock);
 939         if (dp != NULL)
 940                 return (DCTOV(dp));

 942         /*
 943          * Make sure it's a valid compressed file
 944          */
 945         hdr = &thdr;
 946         error = vn_rdwr(UIO_READ, vp, (caddr_t)hdr, sizeof (struct comphdr), 0,
 947             UIO_SYSSPACE, 0, 0, cred, NULL);
 948         if (error || hdr->ch_magic != CH_MAGIC_ZLIB ||
 949             hdr->ch_version != CH_VERSION || hdr->ch_algorithm != CH_ALG_ZLIB ||
 950             hdr->ch_fsize == 0 || hdr->ch_blksize < PAGESIZE ||
 951             hdr->ch_blksize > ptob(DCCACHESIZE) || !ISP2(hdr->ch_blksize))
 951             hdr->ch_blksize > ptob(DCCACHESIZE) ||
 952             (hdr->ch_blksize & (hdr->ch_blksize - 1)) != 0)
 952                 return (NULL);

 954         /* get underlying file size */
 955         if (VOP_GETATTR(vp, &vattr, 0, cred, ctp) != 0)
 956                 return (NULL);

 958         /*
 959          * Re-read entire header
 960          */
 961         hdrsize = hdr->ch_blkmap[0] + sizeof (uint64_t);
 962         hdr = kmem_alloc(hdrsize, KM_SLEEP);
 963         error = vn_rdwr(UIO_READ, vp, (caddr_t)hdr, hdrsize, 0, UIO_SYSSPACE,
 964             0, 0, cred, NULL);
 965         if (error) {
 966                 kmem_free(hdr, hdrsize);
 967                 return (NULL);
 968         }

 970         /*
 971          * add extra blkmap entry to make dc_getblock()'s
 972          * life easier
 973          */
 974         bsize = hdr->ch_blksize;
 975         hdr->ch_blkmap[((hdr->ch_fsize-1) / bsize) + 1] = vattr.va_size;
```



```
 977         ndp = dcnode_alloc();
 978         ndp->dc_subvp = vp;
 979         VN_HOLD(vp);
 980         ndp->dc_hdr = hdr;
 981         ndp->dc_hdrsize = hdrsize;

 983         /*
 984          * Allocate kmem cache if none there already
 985          */
 986         ndp->dc_zmax = ZMAXBUF(bsize);
 987         cpp = &dcbuf_cache[btop(bsize)];
 988         mutex_enter(&dccache_lock);
 989         if (*cpp == NULL)
 990                 *cpp = kmem_cache_create("dcbuf_cache", ndp->dc_zmax, 0, NULL,
 991                     NULL, NULL, NULL, NULL, 0);
 992         mutex_exit(&dccache_lock);
 993         ndp->dc_bufcache = *cpp;

 995         /*
 996          * Recheck table in case someone else created shadow
 997          * while we were blocked above.
 998          */
 999         mutex_enter(&dctable_lock);
1000         dp = dcfind(vp);
1001         if (dp != NULL) {
1002                 mutex_exit(&dctable_lock);
1003                 dcnode_recycle(ndp);
1004                 kmem_cache_free(dcnode_cache, ndp);
1005                 return (DCTOV(dp));
1006         }
1007         dcinsert(ndp);
1008         mutex_exit(&dctable_lock);

1010         return (DCTOV(ndp));
1011 }
______unchanged_portion_omitted_
```

```
**********************************************************
   92891 Thu Oct 23 10:42:10 2014
new/usr/src/uts/common/fs/zfs/zio.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2011, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/spa_impl.h>
#include <sys/vdev_impl.h>
#include <sys/zio_impl.h>
#include <sys/zio_compress.h>
#include <sys/zio_checksum.h>
#include <sys/dmu_objset.h>
#include <sys/arc.h>
#include <sys/ddt.h>
#include <sys/blkptr.h>
#include <sys/zfeature.h>

/*
 * ==========================================================================
 * I/O type descriptions
 * ==========================================================================
 */
const char *zio_type_name[ZIO_TYPES] = {
        "zio_null", "zio_read", "zio_write", "zio_free", "zio_claim",
        "zio_ioctl"
};

/*
 * ==========================================================================
 * I/O kmem caches
 * ==========================================================================
 */
kmem_cache_t *zio_cache;
kmem_cache_t *zio_link_cache;
kmem_cache_t *zio_buf_cache[SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT];
```



```
kmem_cache_t *zio_data_buf_cache[SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT];

#ifdef _KERNEL
extern vmem_t *zio_alloc_arena;
#endif

/*
 * The following actions directly effect the spa's sync-to-convergence logic.
 * The values below define the sync pass when we start performing the action.
 * Care should be taken when changing these values as they directly impact
 * spa_sync() performance. Tuning these values may introduce subtle performance
 * pathologies and should only be done in the context of performance analysis.
 * These tunables will eventually be removed and replaced with #defines once
 * enough analysis has been done to determine optimal values.
 *
 * The 'zfs_sync_pass_deferred_free' pass must be greater than 1 to ensure that
 * regular blocks are not deferred.
 */
int zfs_sync_pass_deferred_free = 2; /* defer frees starting in this pass */
int zfs_sync_pass_dont_compress = 5; /* don't compress starting in this pass */
int zfs_sync_pass_rewrite = 2; /* rewrite new bps starting in this pass */

/*
 * An allocating zio is one that either currently has the DVA allocate
 * stage set or will have it later in its lifetime.
 */
#define IO_IS_ALLOCATING(zio) ((zio)->io_orig_pipeline & ZIO_STAGE_DVA_ALLOCATE)

boolean_t       zio_requeue_io_start_cut_in_line = B_TRUE;

#ifdef ZFS_DEBUG
int zio_buf_debug_limit = 16384;
#else
int zio_buf_debug_limit = 0;
#endif

void
zio_init(void)
{
        size_t c;
        vmem_t *data_alloc_arena = NULL;

#ifdef _KERNEL
        data_alloc_arena = zio_alloc_arena;
#endif
        zio_cache = kmem_cache_create("zio_cache",
            sizeof (zio_t), 0, NULL, NULL, NULL, NULL, NULL, 0);
        zio_link_cache = kmem_cache_create("zio_link_cache",
            sizeof (zio_link_t), 0, NULL, NULL, NULL, NULL, NULL, 0);

        /*
         * For small buffers, we want a cache for each multiple of
         * SPA_MINBLOCKSIZE.  For medium-size buffers, we want a cache
         * for each quarter-power of 2.  For large buffers, we want
         * a cache for each multiple of PAGESIZE.
         */
        for (c = 0; c < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT; c++) {
                size_t size = (c + 1) << SPA_MINBLOCKSHIFT;
                size_t p2 = size;
                size_t align = 0;
                size_t cflags = (size > zio_buf_debug_limit) ? KMC_NODEBUG : 0;

                while (!ISP2(p2))
                while (p2 & (p2 - 1))
                        p2 &= p2 - 1;
```

```
127 #ifndef _KERNEL
128                 /*
129                  * If we are using watchpoints, put each buffer on its own page,
130                  * to eliminate the performance overhead of trapping to the
131                  * kernel when modifying a non-watched buffer that shares the
132                  * page with a watched buffer.
133                  */
134                 if (arc_watch && !IS_P2ALIGNED(size, PAGESIZE))
135                         continue;
136 #endif
137                 if (size <= 4 * SPA_MINBLOCKSIZE) {
138                         align = SPA_MINBLOCKSIZE;
139                 } else if (IS_P2ALIGNED(size, PAGESIZE)) {
140                         align = PAGESIZE;
141                 } else if (IS_P2ALIGNED(size, p2 >> 2)) {
142                         align = p2 >> 2;
143                 }

145                 if (align != 0) {
146                         char name[36];
147                         (void) sprintf(name, "zio_buf_%lu", (ulong_t)size);
148                         zio_buf_cache[c] = kmem_cache_create(name, size,
149                             align, NULL, NULL, NULL, NULL, NULL, cflags);

151                         /*
152                          * Since zio_data bufs do not appear in crash dumps, we
153                          * pass KMC_NOTOUCH so that no allocator metadata is
154                          * stored with the buffers.
155                          */
156                         (void) sprintf(name, "zio_data_buf_%lu", (ulong_t)size);
157                         zio_data_buf_cache[c] = kmem_cache_create(name, size,
158                             align, NULL, NULL, NULL, NULL, data_alloc_arena,
159                             cflags | KMC_NOTOUCH);
160                 }
161         }

163         while (--c != 0) {
164                 ASSERT(zio_buf_cache[c] != NULL);
165                 if (zio_buf_cache[c - 1] == NULL)
166                         zio_buf_cache[c - 1] = zio_buf_cache[c];

168                 ASSERT(zio_data_buf_cache[c] != NULL);
169                 if (zio_data_buf_cache[c - 1] == NULL)
170                         zio_data_buf_cache[c - 1] = zio_data_buf_cache[c];
171         }

173         zio_inject_init();
174 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   59184 Thu Oct 23 10:42:10 2014
new/usr/src/uts/common/inet/ilb/ilb.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/kmem.h>
#include <sys/ksynch.h>
#include <sys/systm.h>
#include <sys/socket.h>
#include <sys/disp.h>
#include <sys/taskq.h>
#include <sys/cmn_err.h>
#include <sys/strsun.h>
#include <sys/sdt.h>
#include <sys/atomic.h>
#include <netinet/in.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/tcp.h>
#include <inet/udp_impl.h>
#include <inet/kstatcom.h>

#include <inet/ilb_ip.h>
#include "ilb_alg.h"
#include "ilb_nat.h"
#include "ilb_conn.h"

/* ILB kmem cache flag */
int ilb_kmem_flags = 0;

/*
 * The default size for the different hash tables.  Global for all stacks.
 * But each stack has its own table, just that their sizes are the same.
 */
static size_t ilb_rule_hash_size = 2048;

static size_t ilb_conn_hash_size = 262144;
```



```
static size_t ilb_sticky_hash_size = 262144;

/* This should be a prime number. */
static size_t ilb_nat_src_hash_size = 97;

/* Default NAT cache entry expiry time. */
static uint32_t ilb_conn_tcp_expiry = 120;
static uint32_t ilb_conn_udp_expiry = 60;

/* Default sticky entry expiry time. */
static uint32_t ilb_sticky_expiry = 60;

/* addr is assumed to be a uint8_t * to an ipaddr_t. */
#define ILB_RULE_HASH(addr, hash_size) \
        ((*((addr) + 3) * 29791 + *((addr) + 2) * 961 + *((addr) + 1) * 31 + \
        *(addr)) & ((hash_size) - 1))

/*
 * Note on ILB delayed processing
 *
 * To avoid in line removal on some of the data structures, such as rules,
 * servers and ilb_conn_hash entries, ILB delays such processing to a taskq.
 * There are three types of ILB taskq:
 *
 * 1. rule handling: created at stack initialialization time, ilb_stack_init()
 * 2. conn hash handling: created at conn hash initialization time,
 *                        ilb_conn_hash_init()
 * 3. sticky hash handling: created at sticky hash initialization time,
 *                          ilb_sticky_hash_init()
 *
 * The rule taskq is for processing rule and server removal.  When a user
 * land rule/server removal request comes in, a taskq is dispatched after
 * removing the rule/server from all related hashes.  This taskq will wait
 * until all references to the rule/server are gone before removing it.
 * So the user land thread requesting the removal does not need to wait
 * for the removal completion.
 *
 * The conn hash/sticky hash taskq is for processing ilb_conn_hash and
 * ilb_sticky_hash table entry removal.  There are ilb_conn_timer_size timers
 * and ilb_sticky_timer_size timers running for ilb_conn_hash and
 * ilb_sticky_hash cleanup respectively.   Each timer is responsible for one
 * portion (same size) of the hash table.  When a timer fires, it dispatches
 * a conn hash taskq to clean up its portion of the table.  This avoids in
 * line processing of the removal.
 *
 * There is another delayed processing, the clean up of NAT source address
 * table.  We just use the timer to directly handle it instead of using
 * a taskq.  The reason is that the table is small so it is OK to use the
 * timer.
 */

/* ILB rule taskq constants. */
#define ILB_RULE_TASKQ_NUM_THR  20

/* Argument passed to ILB rule taskq routines. */
typedef struct {
        ilb_stack_t     *ilbs;
        ilb_rule_t      *rule;
} ilb_rule_tq_t;

/* kstat handling routines. */
static kstat_t *ilb_kstat_g_init(netstackid_t, ilb_stack_t *);
static void ilb_kstat_g_fini(netstackid_t, ilb_stack_t *);
static kstat_t *ilb_rule_kstat_init(netstackid_t, ilb_rule_t *);
static kstat_t *ilb_server_kstat_init(netstackid_t, ilb_rule_t *,
    ilb_server_t *);
```

```
/* Rule hash handling routines. */
static void ilb_rule_hash_init(ilb_stack_t *);
static void ilb_rule_hash_fini(ilb_stack_t *);
static void ilb_rule_hash_add(ilb_stack_t *, ilb_rule_t *, const in6_addr_t *);
static void ilb_rule_hash_del(ilb_rule_t *);
static ilb_rule_t *ilb_rule_hash(ilb_stack_t *, int, int, in6_addr_t *,
    in_port_t, zoneid_t, uint32_t, boolean_t *);

static void ilb_rule_g_add(ilb_stack_t *, ilb_rule_t *);
static void ilb_rule_g_del(ilb_stack_t *, ilb_rule_t *);
static void ilb_del_rule_common(ilb_stack_t *, ilb_rule_t *);
static ilb_rule_t *ilb_find_rule_locked(ilb_stack_t *, zoneid_t, const char *,
    int *);
static boolean_t ilb_match_rule(ilb_stack_t *, zoneid_t, const char *, int,
    int, in_port_t, in_port_t, const in6_addr_t *);

/* Back end server handling routines. */
static void ilb_server_free(ilb_server_t *);

/* Network stack handling routines. */
static void *ilb_stack_init(netstackid_t, netstack_t *);
static void ilb_stack_shutdown(netstackid_t, void *);
static void ilb_stack_fini(netstackid_t, void *);

/* Sticky connection handling routines. */
static void ilb_rule_sticky_init(ilb_rule_t *);
static void ilb_rule_sticky_fini(ilb_rule_t *);

/* Handy macro to check for unspecified address. */
#define IS_ADDR_UNSPEC(addr)                                    \
        (IN6_IS_ADDR_V4MAPPED(addr) ? IN6_IS_ADDR_V4MAPPED_ANY(addr) :  \
            IN6_IS_ADDR_UNSPECIFIED(addr))

/*
 * Global kstat instance counter.  When a rule is created, its kstat instance
 * number is assigned by ilb_kstat_instance and ilb_kstat_instance is
 * incremented.
 */
static uint_t ilb_kstat_instance = 0;

/*
 * The ILB global kstat has name ILB_G_KS_NAME and class name ILB_G_KS_CNAME.
 * A rule's kstat has ILB_RULE_KS_CNAME class name.
 */
#define ILB_G_KS_NAME           "global"
#define ILB_G_KS_CNAME          "kstat"
#define ILB_RULE_KS_CNAME       "rulestat"

static kstat_t *
ilb_kstat_g_init(netstackid_t stackid, ilb_stack_t *ilbs)
{
        kstat_t *ksp;
        ilb_g_kstat_t template = {
                { "num_rules",          KSTAT_DATA_UINT64, 0 },
                { "ip_frag_in",         KSTAT_DATA_UINT64, 0 },
                { "ip_frag_dropped",    KSTAT_DATA_UINT64, 0 }
        };

        ksp = kstat_create_netstack(ILB_KSTAT_MOD_NAME, 0, ILB_G_KS_NAME,
            ILB_G_KS_CNAME, KSTAT_TYPE_NAMED, NUM_OF_FIELDS(ilb_g_kstat_t),
            KSTAT_FLAG_VIRTUAL, stackid);
        if (ksp == NULL)
                return (NULL);
        bcopy(&template, ilbs->ilbs_kstat, sizeof (template));
        ksp->ks_data = ilbs->ilbs_kstat;
```



```
        ksp->ks_private = (void *)(uintptr_t)stackid;

        kstat_install(ksp);
        return (ksp);
}

static void
ilb_kstat_g_fini(netstackid_t stackid, ilb_stack_t *ilbs)
{
        if (ilbs->ilbs_ksp != NULL) {
                ASSERT(stackid == (netstackid_t)(uintptr_t)
                    ilbs->ilbs_ksp->ks_private);
                kstat_delete_netstack(ilbs->ilbs_ksp, stackid);
                ilbs->ilbs_ksp = NULL;
        }
}

static kstat_t *
ilb_rule_kstat_init(netstackid_t stackid, ilb_rule_t *rule)
{
        kstat_t *ksp;
        ilb_rule_kstat_t template = {
                { "num_servers",                KSTAT_DATA_UINT64, 0 },
                { "bytes_not_processed",        KSTAT_DATA_UINT64, 0 },
                { "pkt_not_processed",          KSTAT_DATA_UINT64, 0 },
                { "bytes_dropped",              KSTAT_DATA_UINT64, 0 },
                { "pkt_dropped",                KSTAT_DATA_UINT64, 0 },
                { "nomem_bytes_dropped",        KSTAT_DATA_UINT64, 0 },
                { "nomem_pkt_dropped",          KSTAT_DATA_UINT64, 0 },
                { "noport_bytes_dropped",       KSTAT_DATA_UINT64, 0 },
                { "noport_pkt_dropped",         KSTAT_DATA_UINT64, 0 },
                { "icmp_echo_processed",        KSTAT_DATA_UINT64, 0 },
                { "icmp_dropped",               KSTAT_DATA_UINT64, 0 },
                { "icmp_too_big_processed",     KSTAT_DATA_UINT64, 0 },
                { "icmp_too_big_dropped",       KSTAT_DATA_UINT64, 0 }
        };

        ksp = kstat_create_netstack(ILB_KSTAT_MOD_NAME, rule->ir_ks_instance,
            rule->ir_name, ILB_RULE_KS_CNAME, KSTAT_TYPE_NAMED,
            NUM_OF_FIELDS(ilb_rule_kstat_t), KSTAT_FLAG_VIRTUAL, stackid);
        if (ksp == NULL)
                return (NULL);

        bcopy(&template, &rule->ir_kstat, sizeof (template));
        ksp->ks_data = &rule->ir_kstat;
        ksp->ks_private = (void *)(uintptr_t)stackid;

        kstat_install(ksp);
        return (ksp);
}

static kstat_t *
ilb_server_kstat_init(netstackid_t stackid, ilb_rule_t *rule,
    ilb_server_t *server)
{
        kstat_t *ksp;
        ilb_server_kstat_t template = {
                { "bytes_processed",    KSTAT_DATA_UINT64, 0 },
                { "pkt_processed",      KSTAT_DATA_UINT64, 0 },
                { "ip_address",         KSTAT_DATA_STRING, 0 }
        };
        char cname_buf[KSTAT_STRLEN];

        /* 7 is "-sstat" */
        ASSERT(strlen(rule->ir_name) + 7 < KSTAT_STRLEN);
        (void) sprintf(cname_buf, "%s-sstat", rule->ir_name);
```

```
260         ksp = kstat_create_netstack(ILB_KSTAT_MOD_NAME, rule->ir_ks_instance,
261             server->iser_name, cname_buf, KSTAT_TYPE_NAMED,
262             NUM_OF_FIELDS(ilb_server_kstat_t), KSTAT_FLAG_VIRTUAL, stackid);
263         if (ksp == NULL)
264                 return (NULL);

266         bcopy(&template, &server->iser_kstat, sizeof (template));
267         ksp->ks_data = &server->iser_kstat;
268         ksp->ks_private = (void *)(uintptr_t)stackid;

270         kstat_named_setstr(&server->iser_kstat.ip_address,
271             server->iser_ip_addr);
272         /* We never change the IP address */
273         ksp->ks_data_size += strlen(server->iser_ip_addr) + 1;

275         kstat_install(ksp);
276         return (ksp);
277 }

279 /* Initialize the rule hash table. */
280 static void
281 ilb_rule_hash_init(ilb_stack_t *ilbs)
282 {
283         int i;

285         /*
286          * If ilbs->ilbs_rule_hash_size is not a power of 2, bump it up to
287          * the next power of 2.
288          */
289         if (!ISP2(ilbs->ilbs_rule_hash_size)) {
 27         if (ilbs->ilbs_rule_hash_size & (ilbs->ilbs_rule_hash_size - 1)) {
290                 for (i = 0; i < 31; i++) {
291                         if (ilbs->ilbs_rule_hash_size < (1 << i))
292                                 break;
293                 }
294                 ilbs->ilbs_rule_hash_size = 1 << i;
295         }
296         ilbs->ilbs_g_hash = kmem_zalloc(sizeof (ilb_hash_t) *
297             ilbs->ilbs_rule_hash_size, KM_SLEEP);
298         for (i = 0; i < ilbs->ilbs_rule_hash_size; i++) {
299                 mutex_init(&ilbs->ilbs_g_hash[i].ilb_hash_lock, NULL,
300                     MUTEX_DEFAULT, NULL);
301         }
302 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   44075 Thu Oct 23 10:42:10 2014
new/usr/src/uts/common/inet/ilb/ilb_conn.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/time.h>
#include <sys/taskq.h>
#include <sys/cmn_err.h>
#include <sys/sdt.h>
#include <sys/atomic.h>
#include <netinet/in.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/tcp.h>
#include <inet/udp_impl.h>
#include <inet/ilb.h>

#include "ilb_stack.h"
#include "ilb_impl.h"
#include "ilb_conn.h"
#include "ilb_nat.h"

/*
 * Timer struct for ilb_conn_t and ilb_sticky_t garbage collection
 *
 * start: starting index into the hash table to do gc
 * end: ending index into the hash table to do gc
 * ilbs: pointer to the ilb_stack_t of the IP stack
 * tid_lock: mutex to protect the timer id.
 * tid: timer id of the timer
 */
typedef struct ilb_timer_s {
        uint32_t        start;
        uint32_t        end;
        ilb_stack_t     *ilbs;
        kmutex_t        tid_lock;
```



```
        timeout_id_t    tid;
} ilb_timer_t;

/* Hash macro for finding the index to the conn hash table */
#define ILB_CONN_HASH(saddr, sport, daddr, dport, hash_size)    \
        (((*((saddr) + 3) ^ *((daddr) + 3)) * 50653 +           \
        (*((saddr) + 2) ^ *((daddr) + 2)) * 1369 +              \
        (*((saddr) + 1) ^ *((daddr) + 1)) * 37 +                \
        (*(saddr) ^ *(daddr)) + (sport) * 37 + (dport)) &       \
        ((hash_size) - 1))

/* Kmem cache for the conn hash entry */
static struct kmem_cache *ilb_conn_cache = NULL;

/*
 * There are 60 timers running to do conn cache garbage collection.  Each
 * gc thread is responsible for 1/60 of the conn hash table.
 */
static int ilb_conn_timer_size = 60;

/* Each of the above gc timers wake up every 15s to do the gc. */
static int ilb_conn_cache_timeout = 15;

#define ILB_STICKY_HASH(saddr, rule, hash_size)                 \
        (((*((saddr) + 3) ^ ((rule) >> 24)) * 29791 +           \
        (*((saddr) + 2) ^ ((rule) >> 16)) * 961 +               \
        (*((saddr) + 1) ^ ((rule) >> 8)) * 31 +                 \
        (*(saddr) ^ (rule))) & ((hash_size) - 1))

static struct kmem_cache *ilb_sticky_cache = NULL;

/*
 * There are 60 timers running to do sticky cache garbage collection.  Each
 * gc thread is responsible for 1/60 of the sticky hash table.
 */
static int ilb_sticky_timer_size = 60;

/* Each of the above gc timers wake up every 15s to do the gc. */
static int ilb_sticky_timeout = 15;

#define ILB_STICKY_REFRELE(s)                   \
{                                               \
        mutex_enter(&(s)->hash->sticky_lock);   \
        (s)->refcnt--;                          \
        (s)->atime = ddi_get_lbolt64();         \
        mutex_exit(&s->hash->sticky_lock);      \
}


static void
ilb_conn_cache_init(void)
{
        ilb_conn_cache = kmem_cache_create("ilb_conn_cache",
            sizeof (ilb_conn_t), 0, NULL, NULL, NULL, NULL, NULL,
            ilb_kmem_flags);
}

void
ilb_conn_cache_fini(void)
{
        if (ilb_conn_cache != NULL) {
                kmem_cache_destroy(ilb_conn_cache);
                ilb_conn_cache = NULL;
        }
}
```

```
128 static void
129 ilb_conn_remove_common(ilb_conn_t *connp, boolean_t c2s)
130 {
131         ilb_conn_hash_t *hash;
132         ilb_conn_t **next, **prev;
133         ilb_conn_t **next_prev, **prev_next;
134
135         if (c2s) {
136                 hash = connp->conn_c2s_hash;
137                 ASSERT(MUTEX_HELD(&hash->ilb_conn_hash_lock));
138                 next = &connp->conn_c2s_next;
139                 prev = &connp->conn_c2s_prev;
140                 if (*next != NULL)
141                         next_prev = &(*next)->conn_c2s_prev;
142                 if (*prev != NULL)
143                         prev_next = &(*prev)->conn_c2s_next;
144         } else {
145                 hash = connp->conn_s2c_hash;
146                 ASSERT(MUTEX_HELD(&hash->ilb_conn_hash_lock));
147                 next = &connp->conn_s2c_next;
148                 prev = &connp->conn_s2c_prev;
149                 if (*next != NULL)
150                         next_prev = &(*next)->conn_s2c_prev;
151                 if (*prev != NULL)
152                         prev_next = &(*prev)->conn_s2c_next;
153         }
154
155         if (hash->ilb_connp == connp) {
156                 hash->ilb_connp = *next;
157                 if (*next != NULL)
158                         *next_prev = NULL;
159         } else {
160                 if (*prev != NULL)
161                         *prev_next = *next;
162                 if (*next != NULL)
163                         *next_prev = *prev;
164         }
165         ASSERT(hash->ilb_conn_cnt > 0);
166         hash->ilb_conn_cnt--;
167
168         *next = NULL;
169         *prev = NULL;
170 }
171
172 static void
173 ilb_conn_remove(ilb_conn_t *connp)
174 {
175         ASSERT(MUTEX_HELD(&connp->conn_c2s_hash->ilb_conn_hash_lock));
176         ilb_conn_remove_common(connp, B_TRUE);
177         ASSERT(MUTEX_HELD(&connp->conn_s2c_hash->ilb_conn_hash_lock));
178         ilb_conn_remove_common(connp, B_FALSE);
179
180         if (connp->conn_rule_cache.topo == ILB_TOPO_IMPL_NAT) {
181                 in_port_t port;
182
183                 port = ntohs(connp->conn_rule_cache.info.nat_sport);
184                 vmem_free(connp->conn_rule_cache.info.src_ent->nse_port_arena,
185                     (void *)(uintptr_t)port, 1);
186         }
187
188         if (connp->conn_sticky != NULL)
189                 ILB_STICKY_REFRELE(connp->conn_sticky);
190         ILB_SERVER_REFRELE(connp->conn_server);
191         kmem_cache_free(ilb_conn_cache, connp);
192 }
```



```
194 /*
195  * Routine to do periodic garbage collection of conn hash entries.  When
196  * a conn hash timer fires, it dispatches a taskq to call this function
197  * to do the gc.  Note that each taskq is responisble for a portion of
198  * the table.  The portion is stored in timer->start, timer->end.
199  */
200 static void
201 ilb_conn_cleanup(void *arg)
202 {
203         ilb_timer_t *timer = (ilb_timer_t *)arg;
204         uint32_t i;
205         ilb_stack_t *ilbs;
206         ilb_conn_hash_t *c2s_hash, *s2c_hash;
207         ilb_conn_t *connp, *nxt_connp;
208         int64_t now;
209         int64_t expiry;
210         boolean_t die_now;
211
212         ilbs = timer->ilbs;
213         c2s_hash = ilbs->ilbs_c2s_conn_hash;
214         ASSERT(c2s_hash != NULL);
215
216         now = ddi_get_lbolt64();
217         for (i = timer->start; i < timer->end; i++) {
218                 mutex_enter(&c2s_hash[i].ilb_conn_hash_lock);
219                 if ((connp = c2s_hash[i].ilb_connp) == NULL) {
220                         ASSERT(c2s_hash[i].ilb_conn_cnt == 0);
221                         mutex_exit(&c2s_hash[i].ilb_conn_hash_lock);
222                         continue;
223                 }
224                 do {
225                         ASSERT(c2s_hash[i].ilb_conn_cnt > 0);
226                         ASSERT(connp->conn_c2s_hash == &c2s_hash[i]);
227                         nxt_connp = connp->conn_c2s_next;
228                         expiry = now - SEC_TO_TICK(connp->conn_expiry);
229                         if (connp->conn_server->iser_die_time != 0 &&
230                             connp->conn_server->iser_die_time < now)
231                                 die_now = B_TRUE;
232                         else
233                                 die_now = B_FALSE;
234                         s2c_hash = connp->conn_s2c_hash;
235                         mutex_enter(&s2c_hash->ilb_conn_hash_lock);
236
237                         if (connp->conn_gc || die_now ||
238                             (connp->conn_c2s_atime < expiry &&
239                             connp->conn_s2c_atime < expiry)) {
240                                 /* Need to update the nat list cur_connp */
241                                 if (connp == ilbs->ilbs_conn_list_connp) {
242                                         ilbs->ilbs_conn_list_connp =
243                                             connp->conn_c2s_next;
244                                 }
245                                 ilb_conn_remove(connp);
246                                 goto nxt_connp;
247                         }
248
249                         if (connp->conn_l4 != IPPROTO_TCP)
250                                 goto nxt_connp;
251
252                         /* Update and check TCP related conn info */
253                         if (connp->conn_c2s_tcp_fin_sent &&
254                             SEQ_GT(connp->conn_s2c_tcp_ack,
255                             connp->conn_c2s_tcp_fss)) {
256                                 connp->conn_c2s_tcp_fin_acked = B_TRUE;
257                         }
258                         if (connp->conn_s2c_tcp_fin_sent &&
259                             SEQ_GT(connp->conn_c2s_tcp_ack,
```

```
                            connp->conn_s2c_tcp_fss)) {
                                connp->conn_s2c_tcp_fin_acked = B_TRUE;
                        }
                        if (connp->conn_c2s_tcp_fin_acked &&
                            connp->conn_s2c_tcp_fin_acked) {
                                ilb_conn_remove(connp);
                        }
nxt_connp:
                        mutex_exit(&s2c_hash->ilb_conn_hash_lock);
                        connp = nxt_connp;
                } while (connp != NULL);
                mutex_exit(&c2s_hash[i].ilb_conn_hash_lock);
        }
}

/* Conn hash timer routine.  It dispatches a taskq and restart the timer */
static void
ilb_conn_timer(void *arg)
{
        ilb_timer_t *timer = (ilb_timer_t *)arg;

        (void) taskq_dispatch(timer->ilbs->ilbs_conn_taskq, ilb_conn_cleanup,
            arg, TQ_SLEEP);
        mutex_enter(&timer->tid_lock);
        if (timer->tid == 0) {
                mutex_exit(&timer->tid_lock);
        } else {
                timer->tid = timeout(ilb_conn_timer, arg,
                    SEC_TO_TICK(ilb_conn_cache_timeout));
                mutex_exit(&timer->tid_lock);
        }
}

void
ilb_conn_hash_init(ilb_stack_t *ilbs)
{
        extern pri_t minclsyspri;
        int i, part;
        ilb_timer_t *tm;
        char tq_name[TASKQ_NAMELEN];

        /*
         * If ilbs->ilbs_conn_hash_size is not a power of 2, bump it up to
         * the next power of 2.
         */
        if (!ISP2(ilbs->ilbs_conn_hash_size)) {
        if (ilbs->ilbs_conn_hash_size & (ilbs->ilbs_conn_hash_size - 1)) {
                for (i = 0; i < 31; i++) {
                        if (ilbs->ilbs_conn_hash_size < (1 << i))
                                break;
                }
                ilbs->ilbs_conn_hash_size = 1 << i;
        }

        /*
         * Can sleep since this should be called when a rule is being added,
         * hence we are not in interrupt context.
         */
        ilbs->ilbs_c2s_conn_hash = kmem_zalloc(sizeof (ilb_conn_hash_t) *
            ilbs->ilbs_conn_hash_size, KM_SLEEP);
        ilbs->ilbs_s2c_conn_hash = kmem_zalloc(sizeof (ilb_conn_hash_t) *
            ilbs->ilbs_conn_hash_size, KM_SLEEP);

        for (i = 0; i < ilbs->ilbs_conn_hash_size; i++) {
                mutex_init(&ilbs->ilbs_c2s_conn_hash[i].ilb_conn_hash_lock,
                    NULL, MUTEX_DEFAULT, NULL);
```



```
        }
        for (i = 0; i < ilbs->ilbs_conn_hash_size; i++) {
                mutex_init(&ilbs->ilbs_s2c_conn_hash[i].ilb_conn_hash_lock,
                    NULL, MUTEX_DEFAULT, NULL);
        }

        if (ilb_conn_cache == NULL)
                ilb_conn_cache_init();

        (void) snprintf(tq_name, sizeof (tq_name), "ilb_conn_taskq_%p",
            (void *)ilbs->ilbs_netstack);
        ASSERT(ilbs->ilbs_conn_taskq == NULL);
        ilbs->ilbs_conn_taskq = taskq_create(tq_name,
            ilb_conn_timer_size * 2, minclsyspri, ilb_conn_timer_size,
            ilb_conn_timer_size * 2, TASKQ_PREPOPULATE|TASKQ_DYNAMIC);

        ASSERT(ilbs->ilbs_conn_timer_list == NULL);
        ilbs->ilbs_conn_timer_list = kmem_zalloc(sizeof (ilb_timer_t) *
            ilb_conn_timer_size, KM_SLEEP);

        /*
         * The hash table is divided in equal partition for those timers
         * to do garbage collection.
         */
        part = ilbs->ilbs_conn_hash_size / ilb_conn_timer_size + 1;
        for (i = 0; i < ilb_conn_timer_size; i++) {
                tm = ilbs->ilbs_conn_timer_list + i;
                tm->start = i * part;
                tm->end = i * part + part;
                if (tm->end > ilbs->ilbs_conn_hash_size)
                        tm->end = ilbs->ilbs_conn_hash_size;
                tm->ilbs = ilbs;
                mutex_init(&tm->tid_lock, NULL, MUTEX_DEFAULT, NULL);
                /* Spread out the starting execution time of all the timers. */
                tm->tid = timeout(ilb_conn_timer, tm,
                    SEC_TO_TICK(ilb_conn_cache_timeout + i));
        }
}
_____unchanged_portion_omitted_

void
ilb_sticky_hash_init(ilb_stack_t *ilbs)
{
        extern pri_t minclsyspri;
        int i, part;
        char tq_name[TASKQ_NAMELEN];
        ilb_timer_t *tm;

        if (!ISP2(ilbs->ilbs_sticky_hash_size)) {
        if (ilbs->ilbs_sticky_hash_size & (ilbs->ilbs_sticky_hash_size - 1)) {
                for (i = 0; i < 31; i++) {
                        if (ilbs->ilbs_sticky_hash_size < (1 << i))
                                break;
                }
                ilbs->ilbs_sticky_hash_size = 1 << i;
        }

        ilbs->ilbs_sticky_hash = kmem_zalloc(sizeof (ilb_sticky_hash_t) *
            ilbs->ilbs_sticky_hash_size, KM_SLEEP);
        for (i = 0; i < ilbs->ilbs_sticky_hash_size; i++) {
                mutex_init(&ilbs->ilbs_sticky_hash[i].sticky_lock, NULL,
                    MUTEX_DEFAULT, NULL);
                list_create(&ilbs->ilbs_sticky_hash[i].sticky_head,
                    sizeof (ilb_sticky_t),
                    offsetof(ilb_sticky_t, list));
        }
```

```
1382         if (ilb_sticky_cache == NULL)
1383                 ilb_sticky_cache_init();

1385         (void) snprintf(tq_name, sizeof (tq_name), "ilb_sticky_taskq_%p",
1386             (void *)ilbs->ilbs_netstack);
1387         ASSERT(ilbs->ilbs_sticky_taskq == NULL);
1388         ilbs->ilbs_sticky_taskq = taskq_create(tq_name,
1389             ilb_sticky_timer_size * 2, minclsyspri, ilb_sticky_timer_size,
1390             ilb_sticky_timer_size * 2, TASKQ_PREPOPULATE|TASKQ_DYNAMIC);

1392         ASSERT(ilbs->ilbs_sticky_timer_list == NULL);
1393         ilbs->ilbs_sticky_timer_list = kmem_zalloc(sizeof (ilb_timer_t) *
1394             ilb_sticky_timer_size, KM_SLEEP);
1395         part = ilbs->ilbs_sticky_hash_size / ilb_sticky_timer_size + 1;
1396         for (i = 0; i < ilb_sticky_timer_size; i++) {
1397                 tm = ilbs->ilbs_sticky_timer_list + i;
1398                 tm->start = i * part;
1399                 tm->end = i * part + part;
1400                 if (tm->end > ilbs->ilbs_sticky_hash_size)
1401                         tm->end = ilbs->ilbs_sticky_hash_size;
1402                 tm->ilbs = ilbs;
1403                 mutex_init(&tm->tid_lock, NULL, MUTEX_DEFAULT, NULL);
1404                 /* Spread out the starting execution time of all the timers. */
1405                 tm->tid = timeout(ilb_sticky_timer, tm,
1406                     SEC_TO_TICK(ilb_sticky_timeout + i));
1407         }
1408 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   23905 Thu Oct 23 10:42:10 2014
new/usr/src/uts/common/inet/sctp/sctp_hash.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/socket.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/tsol/tndb.h>
#include <sys/tsol/tnet.h>

#include <netinet/in.h>
#include <netinet/ip6.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/ipclassifier.h>
#include <inet/ipsec_impl.h>
#include <inet/ipp_common.h>
#include <inet/sctp_ip.h>

#include "sctp_impl.h"
#include "sctp_addr.h"

/* Default association hash size.  The size must be a power of 2. */
#define SCTP_CONN_HASH_SIZE     8192

uint_t          sctp_conn_hash_size = SCTP_CONN_HASH_SIZE; /* /etc/system */

/*
 * Cluster networking hook for traversing current assoc list.
 * This routine is used to extract the current list of live associations
 * which must continue to to be dispatched to this node.
 */
int cl_sctp_walk_list(int (*cl_callback)(cl_sctp_info_t *, void *), void *,
    boolean_t);
static int cl_sctp_walk_list_stack(int (*cl_callback)(cl_sctp_info_t *,
    void *), void *arg, boolean_t cansleep, sctp_stack_t *sctps);
```



```
void
sctp_hash_init(sctp_stack_t *sctps)
{
        int i;

        /* Start with /etc/system value */
        sctps->sctps_conn_hash_size = sctp_conn_hash_size;

        if (!ISP2(sctps->sctps_conn_hash_size)) {
        if (sctps->sctps_conn_hash_size & (sctps->sctps_conn_hash_size - 1)) {
                /* Not a power of two. Round up to nearest power of two */
                for (i = 0; i < 31; i++) {
                        if (sctps->sctps_conn_hash_size < (1 << i))
                                break;
                }
                sctps->sctps_conn_hash_size = 1 << i;
        }
        if (sctps->sctps_conn_hash_size < SCTP_CONN_HASH_SIZE) {
                sctps->sctps_conn_hash_size = SCTP_CONN_HASH_SIZE;
                cmn_err(CE_CONT, "using sctp_conn_hash_size = %u\n",
                    sctps->sctps_conn_hash_size);
        }
        sctps->sctps_conn_fanout =
            (sctp_tf_t *)kmem_zalloc(sctps->sctps_conn_hash_size *
            sizeof (sctp_tf_t), KM_SLEEP);
        for (i = 0; i < sctps->sctps_conn_hash_size; i++) {
                mutex_init(&sctps->sctps_conn_fanout[i].tf_lock, NULL,
                    MUTEX_DEFAULT, NULL);
        }
        sctps->sctps_listen_fanout = kmem_zalloc(SCTP_LISTEN_FANOUT_SIZE *
            sizeof (sctp_tf_t), KM_SLEEP);
        for (i = 0; i < SCTP_LISTEN_FANOUT_SIZE; i++) {
                mutex_init(&sctps->sctps_listen_fanout[i].tf_lock, NULL,
                    MUTEX_DEFAULT, NULL);
        }
        sctps->sctps_bind_fanout = kmem_zalloc(SCTP_BIND_FANOUT_SIZE *
            sizeof (sctp_tf_t), KM_SLEEP);
        for (i = 0; i < SCTP_BIND_FANOUT_SIZE; i++) {
                mutex_init(&sctps->sctps_bind_fanout[i].tf_lock, NULL,
                    MUTEX_DEFAULT, NULL);
        }
}
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  173367 Thu Oct 23 10:42:11 2014
new/usr/src/uts/common/inet/udp/udp.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2014, OmniTI Computer Consulting, Inc. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#include <sys/strsun.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/timod.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/strsubr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/kmem.h>
#include <sys/cred_impl.h>
#include <sys/policy.h>
#include <sys/priv.h>
#include <sys/ucred.h>
#include <sys/zone.h>

#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/sockio.h>
#include <sys/vtrace.h>
#include <sys/sdt.h>
#include <sys/debug.h>
#include <sys/isa_defs.h>
#include <sys/random.h>
#include <netinet/in.h>
#include <netinet/ip6.h>
#include <netinet/icmp6.h>
#include <netinet/udp.h>
```



```
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip_impl.h>
#include <inet/ipsec_impl.h>
#include <inet/ip6.h>
#include <inet/ip_ire.h>
#include <inet/ip_if.h>
#include <inet/ip_multi.h>
#include <inet/ip_ndp.h>
#include <inet/proto_set.h>
#include <inet/mib2.h>
#include <inet/optcom.h>
#include <inet/snmpcom.h>
#include <inet/kstatcom.h>
#include <inet/ipclassifier.h>
#include <sys/squeue_impl.h>
#include <inet/ipnet.h>
#include <sys/ethernet.h>

#include <sys/tsol/label.h>
#include <sys/tsol/tnet.h>
#include <rpc/pmap_prot.h>

#include <inet/udp_impl.h>

/*
 * Synchronization notes:
 *
 * UDP is MT and uses the usual kernel synchronization primitives. There are 2
 * locks, the fanout lock (uf_lock) and conn_lock. conn_lock
 * protects the contents of the udp_t. uf_lock protects the address and the
 * fanout information.
 * The lock order is conn_lock -> uf_lock.
 *
 * The fanout lock uf_lock:
 * When a UDP endpoint is bound to a local port, it is inserted into
 * a bind hash list.  The list consists of an array of udp_fanout_t buckets.
 * The size of the array is controlled by the udp_bind_fanout_size variable.
 * This variable can be changed in /etc/system if the default value is
 * not large enough.  Each bind hash bucket is protected by a per bucket
 * lock.  It protects the udp_bind_hash and udp_ptpbhn fields in the udp_t
 * structure and a few other fields in the udp_t. A UDP endpoint is removed
 * from the bind hash list only when it is being unbound or being closed.
 * The per bucket lock also protects a UDP endpoint's state changes.
 *
 * Plumbing notes:
 * UDP is always a device driver. For compatibility with mibopen() code
 * it is possible to I_PUSH "udp", but that results in pushing a passthrough
 * dummy module.
 *
 * The above implies that we don't support any intermediate module to
 * reside in between /dev/ip and udp -- in fact, we never supported such
 * scenario in the past as the inter-layer communication semantics have
 * always been private.
 */

/* For /etc/system control */
uint_t udp_bind_fanout_size = UDP_BIND_FANOUT_SIZE;

static void     udp_addr_req(queue_t *q, mblk_t *mp);
static void     udp_tpi_bind(queue_t *q, mblk_t *mp);
static void     udp_bind_hash_insert(udp_fanout_t *uf, udp_t *udp);
static void     udp_bind_hash_remove(udp_t *udp, boolean_t caller_holds_lock);
static int      udp_build_hdr_template(conn_t *, const in6_addr_t *,
    const in6_addr_t *, in_port_t, uint32_t);
```

```
128 static void     udp_capability_req(queue_t *q, mblk_t *mp);
129 static int      udp_tpi_close(queue_t *q, int flags);
130 static void     udp_close_free(conn_t *);
131 static void     udp_tpi_connect(queue_t *q, mblk_t *mp);
132 static void     udp_tpi_disconnect(queue_t *q, mblk_t *mp);
133 static void     udp_err_ack(queue_t *q, mblk_t *mp, t_scalar_t t_error,
134     int sys_error);
135 static void     udp_err_ack_prim(queue_t *q, mblk_t *mp, t_scalar_t primitive,
136     t_scalar_t tlierr, int sys_error);
137 static int      udp_extra_priv_ports_get(queue_t *q, mblk_t *mp, caddr_t cp,
138                     cred_t *cr);
139 static int      udp_extra_priv_ports_add(queue_t *q, mblk_t *mp,
140                     char *value, caddr_t cp, cred_t *cr);
141 static int      udp_extra_priv_ports_del(queue_t *q, mblk_t *mp,
142                     char *value, caddr_t cp, cred_t *cr);
143 static void     udp_icmp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
144 static void     udp_icmp_error_ipv6(conn_t *connp, mblk_t *mp,
145     ip_recv_attr_t *ira);
146 static void     udp_info_req(queue_t *q, mblk_t *mp);
147 static void     udp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
148 static void     udp_lrput(queue_t *, mblk_t *);
149 static void     udp_lwput(queue_t *, mblk_t *);
150 static int      udp_open(queue_t *q, dev_t *devp, int flag, int sflag,
151                     cred_t *credp, boolean_t isv6);
152 static int      udp_openv4(queue_t *q, dev_t *devp, int flag, int sflag,
153                     cred_t *credp);
154 static int      udp_openv6(queue_t *q, dev_t *devp, int flag, int sflag,
155                     cred_t *credp);
156 static boolean_t udp_opt_allow_udr_set(t_scalar_t level, t_scalar_t name);
157 int             udp_opt_set(conn_t *connp, uint_t optset_context,
158                     int level, int name, uint_t inlen,
159                     uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
160                     void *thisdg_attrs, cred_t *cr);
161 int             udp_opt_get(conn_t *connp, int level, int name,
162                     uchar_t *ptr);
163 static int      udp_output_connected(conn_t *connp, mblk_t *mp, cred_t *cr,
164                     pid_t pid);
165 static int      udp_output_lastdst(conn_t *connp, mblk_t *mp, cred_t *cr,
166     pid_t pid, ip_xmit_attr_t *ixa);
167 static int      udp_output_newdst(conn_t *connp, mblk_t *data_mp, sin_t *sin,
168                     sin6_t *sin6, ushort_t ipversion, cred_t *cr, pid_t,
169                     ip_xmit_attr_t *ixa);
170 static mblk_t   *udp_prepend_hdr(conn_t *, ip_xmit_attr_t *, const ip_pkt_t *,
171     const in6_addr_t *, const in6_addr_t *, in_port_t, uint32_t, mblk_t *,
172     int *);
173 static mblk_t   *udp_prepend_header_template(conn_t *, ip_xmit_attr_t *,
174     mblk_t *, const in6_addr_t *, in_port_t, uint32_t, int *);
175 static void     udp_ud_err(queue_t *q, mblk_t *mp, t_scalar_t err);
176 static void     udp_ud_err_connected(conn_t *, t_scalar_t);
177 static void     udp_tpi_unbind(queue_t *q, mblk_t *mp);
178 static in_port_t udp_update_next_port(udp_t *udp, in_port_t port,
179     boolean_t random);
180 static void     udp_wput_other(queue_t *q, mblk_t *mp);
181 static void     udp_wput_iocdata(queue_t *q, mblk_t *mp);
182 static void     udp_wput_fallback(queue_t *q, mblk_t *mp);
183 static size_t   udp_set_rcv_hiwat(udp_t *udp, size_t size);

185 static void     *udp_stack_init(netstackid_t stackid, netstack_t *ns);
186 static void     udp_stack_fini(netstackid_t stackid, void *arg);

188 /* Common routines for TPI and socket module */
189 static void     udp_ulp_recv(conn_t *, mblk_t *, uint_t, ip_recv_attr_t *);

191 /* Common routine for TPI and socket module */
192 static conn_t   *udp_do_open(cred_t *, boolean_t, int, int *);
193 static void     udp_do_close(conn_t *);
```



```
194 static int      udp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
195     boolean_t);
196 static int      udp_do_unbind(conn_t *);

198 int             udp_getsockname(sock_lower_handle_t,
199     struct sockaddr *, socklen_t *, cred_t *);
200 int             udp_getpeername(sock_lower_handle_t,
201     struct sockaddr *, socklen_t *, cred_t *);
202 static int      udp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
203     cred_t *, pid_t);

205 #pragma inline(udp_output_connected, udp_output_newdst, udp_output_lastdst)

207 /*
208  * Checks if the given destination addr/port is allowed out.
209  * If allowed, registers the (dest_addr/port, node_ID) mapping at Cluster.
210  * Called for each connect() and for sendto()/sendmsg() to a different
211  * destination.
212  * For connect(), called in udp_connect().
213  * For sendto()/sendmsg(), called in udp_output_newdst().
214  *
215  * This macro assumes that the cl_inet_connect2 hook is not NULL.
216  * Please check this before calling this macro.
217  *
218  * void
219  * CL_INET_UDP_CONNECT(conn_t cp, udp_t *udp, boolean_t is_outgoing,
220  *     in6_addr_t *faddrp, in_port_t (or uint16_t) fport, int err);
221  */
222 #define CL_INET_UDP_CONNECT(cp, is_outgoing, faddrp, fport, err) {      \
223         (err) = 0;                                                      \
224         /*                                                              \
225          * Running in cluster mode - check and register active          \
226          * "connection" information                                     \
227          */                                                             \
228         if ((cp)->conn_ipversion == IPV4_VERSION)                       \
229                 (err) = (*cl_inet_connect2)(                            \
230                     (cp)->conn_netstack->netstack_stackid,              \
231                     IPPROTO_UDP, is_outgoing, AF_INET,                  \
232                     (uint8_t *)&((cp)->conn_laddr_v4),                  \
233                     (cp)->conn_lport,                                   \
234                     (uint8_t *)&(V4_PART_OF_V6(*faddrp)),               \
235                     (in_port_t)(fport), NULL);                          \
236         else                                                            \
237                 (err) = (*cl_inet_connect2)(                            \
238                     (cp)->conn_netstack->netstack_stackid,              \
239                     IPPROTO_UDP, is_outgoing, AF_INET6,                 \
240                     (uint8_t *)&((cp)->conn_laddr_v6),                  \
241                     (cp)->conn_lport,                                   \
242                     (uint8_t *)(faddrp), (in_port_t)(fport), NULL);     \
243 }

245 static struct module_info udp_mod_info =  {
246         UDP_MOD_ID, UDP_MOD_NAME, 1, INFPSZ, UDP_RECV_HIWATER, UDP_RECV_LOWATER
247 };

249 /*
250  * Entry points for UDP as a device.
251  * We have separate open functions for the /dev/udp and /dev/udp6 devices.
252  */
253 static struct qinit udp_rinitv4 = {
254         NULL, NULL, udp_openv4, udp_tpi_close, NULL, &udp_mod_info, NULL
255 };

257 static struct qinit udp_rinitv6 = {
258         NULL, NULL, udp_openv6, udp_tpi_close, NULL, &udp_mod_info, NULL
259 };
```

```
static struct qinit udp_winit = {
        (pfi_t)udp_wput, (pfi_t)ip_wsrv, NULL, NULL, NULL, &udp_mod_info
};

/* UDP entry point during fallback */
struct qinit udp_fallback_sock_winit = {
        (pfi_t)udp_wput_fallback, NULL, NULL, NULL, NULL, &udp_mod_info
};

/*
 * UDP needs to handle I_LINK and I_PLINK since ifconfig
 * likes to use it as a place to hang the various streams.
 */
static struct qinit udp_lrinit = {
        (pfi_t)udp_lrput, NULL, udp_openv4, udp_tpi_close, NULL, &udp_mod_info
};

static struct qinit udp_lwinit = {
        (pfi_t)udp_lwput, NULL, udp_openv4, udp_tpi_close, NULL, &udp_mod_info
};

/* For AF_INET aka /dev/udp */
struct streamtab udpinfov4 = {
        &udp_rinitv4, &udp_winit, &udp_lrinit, &udp_lwinit
};

/* For AF_INET6 aka /dev/udp6 */
struct streamtab udpinfov6 = {
        &udp_rinitv6, &udp_winit, &udp_lrinit, &udp_lwinit
};

#define UDP_MAXPACKET_IPV4 (IP_MAXPACKET - UDPH_SIZE - IP_SIMPLE_HDR_LENGTH)

/* Default structure copied into T_INFO_ACK messages */
static struct T_info_ack udp_g_t_info_ack_ipv4 = {
        T_INFO_ACK,
        UDP_MAXPACKET_IPV4,     /* TSDU_size. Excl. headers */
        T_INVALID,      /* ETSU_size.  udp does not support expedited data. */
        T_INVALID,      /* CDATA_size. udp does not support connect data. */
        T_INVALID,      /* DDATA_size. udp does not support disconnect data. */
        sizeof (sin_t), /* ADDR_size. */
        0,              /* OPT_size - not initialized here */
        UDP_MAXPACKET_IPV4,     /* TIDU_size.  Excl. headers */
        T_CLTS,         /* SERV_type.  udp supports connection-less. */
        TS_UNBND,       /* CURRENT_state.  This is set from udp_state. */
        (XPG4_1|SENDZERO) /* PROVIDER_flag */
};

#define UDP_MAXPACKET_IPV6 (IP_MAXPACKET - UDPH_SIZE - IPV6_HDR_LEN)

static  struct T_info_ack udp_g_t_info_ack_ipv6 = {
        T_INFO_ACK,
        UDP_MAXPACKET_IPV6,     /* TSDU_size.  Excl. headers */
        T_INVALID,      /* ETSU_size.  udp does not support expedited data. */
        T_INVALID,      /* CDATA_size. udp does not support connect data. */
        T_INVALID,      /* DDATA_size. udp does not support disconnect data. */
        sizeof (sin6_t), /* ADDR_size. */
        0,              /* OPT_size - not initialized here */
        UDP_MAXPACKET_IPV6,     /* TIDU_size. Excl. headers */
        T_CLTS,         /* SERV_type.  udp supports connection-less. */
        TS_UNBND,       /* CURRENT_state.  This is set from udp_state. */
        (XPG4_1|SENDZERO) /* PROVIDER_flag */
};

/*
```



```
 * UDP tunables related declarations. Definitions are in udp_tunables.c
 */
extern mod_prop_info_t udp_propinfo_tbl[];
extern int udp_propinfo_count;

/* Setable in /etc/system */
/* If set to 0, pick ephemeral port sequentially; otherwise randomly. */
uint32_t udp_random_anon_port = 1;

/*
 * Hook functions to enable cluster networking.
 * On non-clustered systems these vectors must always be NULL
 */

void (*cl_inet_bind)(netstackid_t stack_id, uchar_t protocol,
    sa_family_t addr_family, uint8_t *laddrp, in_port_t lport,
    void *args) = NULL;
void (*cl_inet_unbind)(netstackid_t stack_id, uint8_t protocol,
    sa_family_t addr_family, uint8_t *laddrp, in_port_t lport,
    void *args) = NULL;

typedef union T_primitives *t_primp_t;

/*
 * Return the next anonymous port in the privileged port range for
 * bind checking.
 *
 * Trusted Extension (TX) notes: TX allows administrator to mark or
 * reserve ports as Multilevel ports (MLP). MLP has special function
 * on TX systems. Once a port is made MLP, it's not available as
 * ordinary port. This creates "holes" in the port name space. It
 * may be necessary to skip the "holes" find a suitable anon port.
 */
static in_port_t
udp_get_next_priv_port(udp_t *udp)
{
        static in_port_t next_priv_port = IPPORT_RESERVED - 1;
        in_port_t nextport;
        boolean_t restart = B_FALSE;
        udp_stack_t *us = udp->udp_us;

retry:
        if (next_priv_port < us->us_min_anonpriv_port ||
            next_priv_port >= IPPORT_RESERVED) {
                next_priv_port = IPPORT_RESERVED - 1;
                if (restart)
                        return (0);
                restart = B_TRUE;
        }

        if (is_system_labeled() &&
            (nextport = tsol_next_port(crgetzone(udp->udp_connp->conn_cred),
            next_priv_port, IPPROTO_UDP, B_FALSE)) != 0) {
                next_priv_port = nextport;
                goto retry;
        }

        return (next_priv_port--);
}

/*
 * Hash list removal routine for udp_t structures.
 */
static void
udp_bind_hash_remove(udp_t *udp, boolean_t caller_holds_lock)
{
```

```
392         udp_t           *udpnext;
393         kmutex_t        *lockp;
394         udp_stack_t     *us = udp->udp_us;
395         conn_t          *connp = udp->udp_connp;

397         if (udp->udp_ptpbhn == NULL)
398                 return;

400         /*
401          * Extract the lock pointer in case there are concurrent
402          * hash_remove's for this instance.
403          */
404         ASSERT(connp->conn_lport != 0);
405         if (!caller_holds_lock) {
406                 lockp = &us->us_bind_fanout[UDP_BIND_HASH(connp->conn_lport,
407                     us->us_bind_fanout_size)].uf_lock;
408                 ASSERT(lockp != NULL);
409                 mutex_enter(lockp);
410         }
411         if (udp->udp_ptpbhn != NULL) {
412                 udpnext = udp->udp_bind_hash;
413                 if (udpnext != NULL) {
414                         udpnext->udp_ptpbhn = udp->udp_ptpbhn;
415                         udp->udp_bind_hash = NULL;
416                 }
417                 *udp->udp_ptpbhn = udpnext;
418                 udp->udp_ptpbhn = NULL;
419         }
420         if (!caller_holds_lock) {
421                 mutex_exit(lockp);
422         }
423 }

425 static void
426 udp_bind_hash_insert(udp_fanout_t *uf, udp_t *udp)
427 {
428         conn_t  *connp = udp->udp_connp;
429         udp_t   **udpp;
430         udp_t   *udpnext;
431         conn_t  *connext;

433         ASSERT(MUTEX_HELD(&uf->uf_lock));
434         ASSERT(udp->udp_ptpbhn == NULL);
435         udpp = &uf->uf_udp;
436         udpnext = udpp[0];
437         if (udpnext != NULL) {
438                 /*
439                  * If the new udp bound to the INADDR_ANY address
440                  * and the first one in the list is not bound to
441                  * INADDR_ANY we skip all entries until we find the
442                  * first one bound to INADDR_ANY.
443                  * This makes sure that applications binding to a
444                  * specific address get preference over those binding to
445                  * INADDR_ANY.
446                  */
447                 connext = udpnext->udp_connp;
448                 if (V6_OR_V4_INADDR_ANY(connp->conn_bound_addr_v6) &&
449                     !V6_OR_V4_INADDR_ANY(connext->conn_bound_addr_v6)) {
450                         while ((udpnext = udpp[0]) != NULL &&
451                             !V6_OR_V4_INADDR_ANY(connext->conn_bound_addr_v6)) {
452                                 udpp = &(udpnext->udp_bind_hash);
453                         }
454                         if (udpnext != NULL)
455                                 udpnext->udp_ptpbhn = &udp->udp_bind_hash;
456                 } else {
457                         udpnext->udp_ptpbhn = &udp->udp_bind_hash;
```



```
458                 }
459         }
460         udp->udp_bind_hash = udpnext;
461         udp->udp_ptpbhn = udpp;
462         udpp[0] = udp;
463 }

465 /*
466  * This routine is called to handle each O_T_BIND_REQ/T_BIND_REQ message
467  * passed to udp_wput.
468  * It associates a port number and local address with the stream.
469  * It calls IP to verify the local IP address, and calls IP to insert
470  * the conn_t in the fanout table.
471  * If everything is ok it then sends the T_BIND_ACK back up.
472  *
473  * Note that UDP over IPv4 and IPv6 sockets can use the same port number
474  * without setting SO_REUSEADDR. This is needed so that they
475  * can be viewed as two independent transport protocols.
476  * However, anonymouns ports are allocated from the same range to avoid
477  * duplicating the us->us_next_port_to_try.
478  */
479 static void
480 udp_tpi_bind(queue_t *q, mblk_t *mp)
481 {
482         sin_t           *sin;
483         sin6_t          *sin6;
484         mblk_t          *mp1;
485         struct T_bind_req *tbr;
486         conn_t          *connp;
487         udp_t           *udp;
488         int             error;
489         struct sockaddr *sa;
490         cred_t          *cr;

492         /*
493          * All Solaris components should pass a db_credp
494          * for this TPI message, hence we ASSERT.
495          * But in case there is some other M_PROTO that looks
496          * like a TPI message sent by some other kernel
497          * component, we check and return an error.
498          */
499         cr = msg_getcred(mp, NULL);
500         ASSERT(cr != NULL);
501         if (cr == NULL) {
502                 udp_err_ack(q, mp, TSYSERR, EINVAL);
503                 return;
504         }

506         connp = Q_TO_CONN(q);
507         udp = connp->conn_udp;
508         if ((mp->b_wptr - mp->b_rptr) < sizeof (*tbr)) {
509                 (void) mi_strlog(q, 1, SL_ERROR|SL_TRACE,
510                     "udp_bind: bad req, len %u",
511                     (uint_t)(mp->b_wptr - mp->b_rptr));
512                 udp_err_ack(q, mp, TPROTO, 0);
513                 return;
514         }
515         if (udp->udp_state != TS_UNBND) {
516                 (void) mi_strlog(q, 1, SL_ERROR|SL_TRACE,
517                     "udp_bind: bad state, %u", udp->udp_state);
518                 udp_err_ack(q, mp, TOUTSTATE, 0);
519                 return;
520         }
521         /*
522          * Reallocate the message to make sure we have enough room for an
523          * address.
```

```
524          */
525         mp1 = reallocb(mp, sizeof (struct T_bind_ack) + sizeof (sin6_t), 1);
526         if (mp1 == NULL) {
527                 udp_err_ack(q, mp, TSYSERR, ENOMEM);
528                 return;
529         }
531         mp = mp1;
533         /* Reset the message type in preparation for shipping it back. */
534         DB_TYPE(mp) = M_PCPROTO;
536         tbr = (struct T_bind_req *)mp->b_rptr;
537         switch (tbr->ADDR_length) {
538         case 0:                 /* Request for a generic port */
539                 tbr->ADDR_offset = sizeof (struct T_bind_req);
540                 if (connp->conn_family == AF_INET) {
541                         tbr->ADDR_length = sizeof (sin_t);
542                         sin = (sin_t *)&tbr[1];
543                         *sin = sin_null;
544                         sin->sin_family = AF_INET;
545                         mp->b_wptr = (uchar_t *)&sin[1];
546                         sa = (struct sockaddr *)sin;
547                 } else {
548                         ASSERT(connp->conn_family == AF_INET6);
549                         tbr->ADDR_length = sizeof (sin6_t);
550                         sin6 = (sin6_t *)&tbr[1];
551                         *sin6 = sin6_null;
552                         sin6->sin6_family = AF_INET6;
553                         mp->b_wptr = (uchar_t *)&sin6[1];
554                         sa = (struct sockaddr *)sin6;
555                 }
556                 break;
558         case sizeof (sin_t):    /* Complete IPv4 address */
559                 sa = (struct sockaddr *)mi_offset_param(mp, tbr->ADDR_offset,
560                     sizeof (sin_t));
561                 if (sa == NULL || !OK_32PTR((char *)sa)) {
562                         udp_err_ack(q, mp, TSYSERR, EINVAL);
563                         return;
564                 }
565                 if (connp->conn_family != AF_INET ||
566                     sa->sa_family != AF_INET) {
567                         udp_err_ack(q, mp, TSYSERR, EAFNOSUPPORT);
568                         return;
569                 }
570                 break;
572         case sizeof (sin6_t):   /* complete IPv6 address */
573                 sa = (struct sockaddr *)mi_offset_param(mp, tbr->ADDR_offset,
574                     sizeof (sin6_t));
575                 if (sa == NULL || !OK_32PTR((char *)sa)) {
576                         udp_err_ack(q, mp, TSYSERR, EINVAL);
577                         return;
578                 }
579                 if (connp->conn_family != AF_INET6 ||
580                     sa->sa_family != AF_INET6) {
581                         udp_err_ack(q, mp, TSYSERR, EAFNOSUPPORT);
582                         return;
583                 }
584                 break;
586         default:                /* Invalid request */
587                 (void) mi_strlog(q, 1, SL_ERROR|SL_TRACE,
588                     "udp_bind: bad ADDR_length length %u", tbr->ADDR_length);
589                 udp_err_ack(q, mp, TBADADDR, 0);
```



```
590                 return;
591         }
593         error = udp_do_bind(connp, sa, tbr->ADDR_length, cr,
594             tbr->PRIM_type != O_T_BIND_REQ);
596         if (error != 0) {
597                 if (error > 0) {
598                         udp_err_ack(q, mp, TSYSERR, error);
599                 } else {
600                         udp_err_ack(q, mp, -error, 0);
601                 }
602         } else {
603                 tbr->PRIM_type = T_BIND_ACK;
604                 qreply(q, mp);
605         }
606 }
608 /*
609  * This routine handles each T_CONN_REQ message passed to udp.  It
610  * associates a default destination address with the stream.
611  *
612  * After various error checks are completed, udp_connect() lays
613  * the target address and port into the composite header template.
614  * Then we ask IP for information, including a source address if we didn't
615  * already have one. Finally we send up the T_OK_ACK reply message.
616  */
617 static void
618 udp_tpi_connect(queue_t *q, mblk_t *mp)
619 {
620         conn_t  *connp = Q_TO_CONN(q);
621         int     error;
622         socklen_t       len;
623         struct sockaddr         *sa;
624         struct T_conn_req       *tcr;
625         cred_t          *cr;
626         pid_t           pid;
627         /*
628          * All Solaris components should pass a db_credp
629          * for this TPI message, hence we ASSERT.
630          * But in case there is some other M_PROTO that looks
631          * like a TPI message sent by some other kernel
632          * component, we check and return an error.
633          */
634         cr = msg_getcred(mp, &pid);
635         ASSERT(cr != NULL);
636         if (cr == NULL) {
637                 udp_err_ack(q, mp, TSYSERR, EINVAL);
638                 return;
639         }
641         tcr = (struct T_conn_req *)mp->b_rptr;
643         /* A bit of sanity checking */
644         if ((mp->b_wptr - mp->b_rptr) < sizeof (struct T_conn_req)) {
645                 udp_err_ack(q, mp, TPROTO, 0);
646                 return;
647         }
649         if (tcr->OPT_length != 0) {
650                 udp_err_ack(q, mp, TBADOPT, 0);
651                 return;
652         }
654         /*
655          * Determine packet type based on type of address passed in
```

```
656          * the request should contain an IPv4 or IPv6 address.
657          * Make sure that address family matches the type of
658          * family of the address passed down.
659          */
660         len = tcr->DEST_length;
661         switch (tcr->DEST_length) {
662         default:
663                 udp_err_ack(q, mp, TBADADDR, 0);
664                 return;
666         case sizeof (sin_t):
667                 sa = (struct sockaddr *)mi_offset_param(mp, tcr->DEST_offset,
668                     sizeof (sin_t));
669                 break;
671         case sizeof (sin6_t):
672                 sa = (struct sockaddr *)mi_offset_param(mp, tcr->DEST_offset,
673                     sizeof (sin6_t));
674                 break;
675         }
677         error = proto_verify_ip_addr(connp->conn_family, sa, len);
678         if (error != 0) {
679                 udp_err_ack(q, mp, TSYSERR, error);
680                 return;
681         }
683         error = udp_do_connect(connp, sa, len, cr, pid);
684         if (error != 0) {
685                 if (error < 0)
686                         udp_err_ack(q, mp, -error, 0);
687                 else
688                         udp_err_ack(q, mp, TSYSERR, error);
689         } else {
690                 mblk_t  *mp1;
691                 /*
692                  * We have to send a connection confirmation to
693                  * keep TLI happy.
694                  */
695                 if (connp->conn_family == AF_INET) {
696                         mp1 = mi_tpi_conn_con(NULL, (char *)sa,
697                             sizeof (sin_t), NULL, 0);
698                 } else {
699                         mp1 = mi_tpi_conn_con(NULL, (char *)sa,
700                             sizeof (sin6_t), NULL, 0);
701                 }
702                 if (mp1 == NULL) {
703                         udp_err_ack(q, mp, TSYSERR, ENOMEM);
704                         return;
705                 }
707                 /*
708                  * Send ok_ack for T_CONN_REQ
709                  */
710                 mp = mi_tpi_ok_ack_alloc(mp);
711                 if (mp == NULL) {
712                         /* Unable to reuse the T_CONN_REQ for the ack. */
713                         udp_err_ack_prim(q, mp1, T_CONN_REQ, TSYSERR, ENOMEM);
714                         return;
715                 }
717                 putnext(connp->conn_rq, mp);
718                 putnext(connp->conn_rq, mp1);
719         }
720 }
```



```
722 static int
723 udp_tpi_close(queue_t *q, int flags)
724 {
725         conn_t  *connp;
727         if (flags & SO_FALLBACK) {
728                 /*
729                  * stream is being closed while in fallback
730                  * simply free the resources that were allocated
731                  */
732                 inet_minor_free(WR(q)->q_ptr, (dev_t)(RD(q)->q_ptr));
733                 qprocsoff(q);
734                 goto done;
735         }
737         connp = Q_TO_CONN(q);
738         udp_do_close(connp);
739 done:
740         q->q_ptr = WR(q)->q_ptr = NULL;
741         return (0);
742 }
744 static void
745 udp_close_free(conn_t *connp)
746 {
747         udp_t *udp = connp->conn_udp;
749         /* If there are any options associated with the stream, free them. */
750         if (udp->udp_recv_ipp.ipp_fields != 0)
751                 ip_pkt_free(&udp->udp_recv_ipp);
753         /*
754          * Clear any fields which the kmem_cache constructor clears.
755          * Only udp_connp needs to be preserved.
756          * TBD: We should make this more efficient to avoid clearing
757          * everything.
758          */
759         ASSERT(udp->udp_connp == connp);
760         bzero(udp, sizeof (udp_t));
761         udp->udp_connp = connp;
762 }
764 static int
765 udp_do_disconnect(conn_t *connp)
766 {
767         udp_t   *udp;
768         udp_fanout_t *udpf;
769         udp_stack_t *us;
770         int     error;
772         udp = connp->conn_udp;
773         us = udp->udp_us;
774         mutex_enter(&connp->conn_lock);
775         if (udp->udp_state != TS_DATA_XFER) {
776                 mutex_exit(&connp->conn_lock);
777                 return (-TOUTSTATE);
778         }
779         udpf = &us->us_bind_fanout[UDP_BIND_HASH(connp->conn_lport,
780             us->us_bind_fanout_size)];
781         mutex_enter(&udpf->uf_lock);
782         if (connp->conn_mcbc_bind)
783                 connp->conn_saddr_v6 = ipv6_all_zeros;
784         else
785                 connp->conn_saddr_v6 = connp->conn_bound_addr_v6;
786         connp->conn_laddr_v6 = connp->conn_bound_addr_v6;
787         connp->conn_faddr_v6 = ipv6_all_zeros;
```

```
788         connp->conn_fport = 0;
789         udp->udp_state = TS_IDLE;
790         mutex_exit(&udpf->uf_lock);

792         /* Remove any remnants of mapped address binding */
793         if (connp->conn_family == AF_INET6)
794                 connp->conn_ipversion = IPV6_VERSION;

796         connp->conn_v6lastdst = ipv6_all_zeros;
797         error = udp_build_hdr_template(connp, &connp->conn_saddr_v6,
798             &connp->conn_faddr_v6, connp->conn_fport, connp->conn_flowinfo);
799         mutex_exit(&connp->conn_lock);
800         if (error != 0)
801                 return (error);

803         /*
804          * Tell IP to remove the full binding and revert
805          * to the local address binding.
806          */
807         return (ip_laddr_fanout_insert(connp));
808 }

810 static void
811 udp_tpi_disconnect(queue_t *q, mblk_t *mp)
812 {
813         conn_t  *connp = Q_TO_CONN(q);
814         int     error;

816         /*
817          * Allocate the largest primitive we need to send back
818          * T_error_ack is > than T_ok_ack
819          */
820         mp = reallocb(mp, sizeof (struct T_error_ack), 1);
821         if (mp == NULL) {
822                 /* Unable to reuse the T_DISCON_REQ for the ack. */
823                 udp_err_ack_prim(q, mp, T_DISCON_REQ, TSYSERR, ENOMEM);
824                 return;
825         }

827         error = udp_do_disconnect(connp);

829         if (error != 0) {
830                 if (error < 0) {
831                         udp_err_ack(q, mp, -error, 0);
832                 } else {
833                         udp_err_ack(q, mp, TSYSERR, error);
834                 }
835         } else {
836                 mp = mi_tpi_ok_ack_alloc(mp);
837                 ASSERT(mp != NULL);
838                 qreply(q, mp);
839         }
840 }

842 int
843 udp_disconnect(conn_t *connp)
844 {
845         int error;

847         connp->conn_dgram_errind = B_FALSE;
848         error = udp_do_disconnect(connp);
849         if (error < 0)
850                 error = proto_tlitosyserr(-error);

852         return (error);
853 }
```



```
855 /* This routine creates a T_ERROR_ACK message and passes it upstream. */
856 static void
857 udp_err_ack(queue_t *q, mblk_t *mp, t_scalar_t t_error, int sys_error)
858 {
859         if ((mp = mi_tpi_err_ack_alloc(mp, t_error, sys_error)) != NULL)
860                 qreply(q, mp);
861 }

863 /* Shorthand to generate and send TPI error acks to our client */
864 static void
865 udp_err_ack_prim(queue_t *q, mblk_t *mp, t_scalar_t primitive,
866     t_scalar_t t_error, int sys_error)
867 {
868         struct T_error_ack      *teackp;

870         if ((mp = tpi_ack_alloc(mp, sizeof (struct T_error_ack),
871             M_PCPROTO, T_ERROR_ACK)) != NULL) {
872                 teackp = (struct T_error_ack *)mp->b_rptr;
873                 teackp->ERROR_prim = primitive;
874                 teackp->TLI_error = t_error;
875                 teackp->UNIX_error = sys_error;
876                 qreply(q, mp);
877         }
878 }

880 /* At minimum we need 4 bytes of UDP header */
881 #define ICMP_MIN_UDP_HDR        4

883 /*
884  * udp_icmp_input is called as conn_recvicmp to process ICMP messages.
885  * Generates the appropriate T_UDERROR_IND for permanent (non-transient) errors.
886  * Assumes that IP has pulled up everything up to and including the ICMP header.
887  */
888 /* ARGSUSED2 */
889 static void
890 udp_icmp_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
891 {
892         conn_t          *connp = (conn_t *)arg1;
893         icmph_t         *icmph;
894         ipha_t          *ipha;
895         int             iph_hdr_length;
896         udpha_t         *udpha;
897         sin_t           sin;
898         sin6_t          sin6;
899         mblk_t          *mp1;
900         int             error = 0;
901         udp_t           *udp = connp->conn_udp;

903         ipha = (ipha_t *)mp->b_rptr;

905         ASSERT(OK_32PTR(mp->b_rptr));

907         if (IPH_HDR_VERSION(ipha) != IPV4_VERSION) {
908                 ASSERT(IPH_HDR_VERSION(ipha) == IPV6_VERSION);
909                 udp_icmp_error_ipv6(connp, mp, ira);
910                 return;
911         }
912         ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);

914         /* Skip past the outer IP and ICMP headers */
915         ASSERT(IPH_HDR_LENGTH(ipha) == ira->ira_ip_hdr_length);
916         iph_hdr_length = ira->ira_ip_hdr_length;
917         icmph = (icmph_t *)&mp->b_rptr[iph_hdr_length];
918         ipha = (ipha_t *)&icmph[1];     /* Inner IP header */
```

```
920         /* Skip past the inner IP and find the ULP header */
921         iph_hdr_length = IPH_HDR_LENGTH(ipha);
922         udpha = (udpha_t *)((char *)ipha + iph_hdr_length);
923
924         switch (icmph->icmph_type) {
925         case ICMP_DEST_UNREACHABLE:
926                 switch (icmph->icmph_code) {
927                 case ICMP_FRAGMENTATION_NEEDED: {
928                         ipha_t          *ipha;
929                         ip_xmit_attr_t  *ixa;
930                         /*
931                          * IP has already adjusted the path MTU.
932                          * But we need to adjust DF for IPv4.
933                          */
934                         if (connp->conn_ipversion != IPV4_VERSION)
935                                 break;
936
937                         ixa = conn_get_ixa(connp, B_FALSE);
938                         if (ixa == NULL || ixa->ixa_ire == NULL) {
939                                 /*
940                                  * Some other thread holds conn_ixa. We will
941                                  * redo this on the next ICMP too big.
942                                  */
943                                 if (ixa != NULL)
944                                         ixa_refrele(ixa);
945                                 break;
946                         }
947                         (void) ip_get_pmtu(ixa);
948
949                         mutex_enter(&connp->conn_lock);
950                         ipha = (ipha_t *)connp->conn_ht_iphc;
951                         if (ixa->ixa_flags & IXAF_PMTU_IPV4_DF) {
952                                 ipha->ipha_fragment_offset_and_flags |=
953                                     IPH_DF_HTONS;
954                         } else {
955                                 ipha->ipha_fragment_offset_and_flags &=
956                                     ~IPH_DF_HTONS;
957                         }
958                         mutex_exit(&connp->conn_lock);
959                         ixa_refrele(ixa);
960                         break;
961                 }
962                 case ICMP_PORT_UNREACHABLE:
963                 case ICMP_PROTOCOL_UNREACHABLE:
964                         error = ECONNREFUSED;
965                         break;
966                 default:
967                         /* Transient errors */
968                         break;
969                 }
970                 break;
971         default:
972                 /* Transient errors */
973                 break;
974         }
975         if (error == 0) {
976                 freemsg(mp);
977                 return;
978         }
979
980         /*
981          * Deliver T_UDERROR_IND when the application has asked for it.
982          * The socket layer enables this automatically when connected.
983          */
984         if (!connp->conn_dgram_errind) {
985                 freemsg(mp);
```



```
986                 return;
987         }
988
989         switch (connp->conn_family) {
990         case AF_INET:
991                 sin = sin_null;
992                 sin.sin_family = AF_INET;
993                 sin.sin_addr.s_addr = ipha->ipha_dst;
994                 sin.sin_port = udpha->uha_dst_port;
995                 if (IPCL_IS_NONSTR(connp)) {
996                         mutex_enter(&connp->conn_lock);
997                         if (udp->udp_state == TS_DATA_XFER) {
998                                 if (sin.sin_port == connp->conn_fport &&
999                                     sin.sin_addr.s_addr ==
1000                                    connp->conn_faddr_v4) {
1001                                        mutex_exit(&connp->conn_lock);
1002                                        (*connp->conn_upcalls->su_set_error)
1003                                            (connp->conn_upper_handle, error);
1004                                        goto done;
1005                                }
1006                        } else {
1007                                udp->udp_delayed_error = error;
1008                                *((sin_t *)&udp->udp_delayed_addr) = sin;
1009                        }
1010                        mutex_exit(&connp->conn_lock);
1011                } else {
1012                        mp1 = mi_tpi_uderror_ind((char *)&sin, sizeof (sin_t),
1013                            NULL, 0, error);
1014                        if (mp1 != NULL)
1015                                putnext(connp->conn_rq, mp1);
1016                }
1017                break;
1018        case AF_INET6:
1019                sin6 = sin6_null;
1020                sin6.sin6_family = AF_INET6;
1021                IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &sin6.sin6_addr);
1022                sin6.sin6_port = udpha->uha_dst_port;
1023                if (IPCL_IS_NONSTR(connp)) {
1024                        mutex_enter(&connp->conn_lock);
1025                        if (udp->udp_state == TS_DATA_XFER) {
1026                                if (sin6.sin6_port == connp->conn_fport &&
1027                                    IN6_ARE_ADDR_EQUAL(&sin6.sin6_addr,
1028                                    &connp->conn_faddr_v6)) {
1029                                        mutex_exit(&connp->conn_lock);
1030                                        (*connp->conn_upcalls->su_set_error)
1031                                            (connp->conn_upper_handle, error);
1032                                        goto done;
1033                                }
1034                        } else {
1035                                udp->udp_delayed_error = error;
1036                                *((sin6_t *)&udp->udp_delayed_addr) = sin6;
1037                        }
1038                        mutex_exit(&connp->conn_lock);
1039                } else {
1040                        mp1 = mi_tpi_uderror_ind((char *)&sin6, sizeof (sin6_t),
1041                            NULL, 0, error);
1042                        if (mp1 != NULL)
1043                                putnext(connp->conn_rq, mp1);
1044                }
1045                break;
1046        }
1047 done:
1048        freemsg(mp);
1049 }
1050
1051 /*
```

```
 * udp_icmp_error_ipv6 is called by udp_icmp_error to process ICMP for IPv6.
 * Generates the appropriate T_UDERROR_IND for permanent (non-transient) errors.
 * Assumes that IP has pulled up all the extension headers as well as the
 * ICMPv6 header.
 */
static void
udp_icmp_error_ipv6(conn_t *connp, mblk_t *mp, ip_recv_attr_t *ira)
{
        icmp6_t         *icmp6;
        ip6_t           *ip6h, *outer_ip6h;
        uint16_t        iph_hdr_length;
        uint8_t         *nexthdrp;
        udpha_t         *udpha;
        sin6_t          sin6;
        mblk_t          *mp1;
        int             error = 0;
        udp_t           *udp = connp->conn_udp;
        udp_stack_t     *us = udp->udp_us;

        outer_ip6h = (ip6_t *)mp->b_rptr;
#ifdef DEBUG
        if (outer_ip6h->ip6_nxt != IPPROTO_ICMPV6)
                iph_hdr_length = ip_hdr_length_v6(mp, outer_ip6h);
        else
                iph_hdr_length = IPV6_HDR_LEN;
        ASSERT(iph_hdr_length == ira->ira_ip_hdr_length);
#endif
        /* Skip past the outer IP and ICMP headers */
        iph_hdr_length = ira->ira_ip_hdr_length;
        icmp6 = (icmp6_t *)&mp->b_rptr[iph_hdr_length];

        /* Skip past the inner IP and find the ULP header */
        ip6h = (ip6_t *)&icmp6[1];      /* Inner IP header */
        if (!ip_hdr_length_nexthdr_v6(mp, ip6h, &iph_hdr_length, &nexthdrp)) {
                freemsg(mp);
                return;
        }
        udpha = (udpha_t *)((char *)ip6h + iph_hdr_length);

        switch (icmp6->icmp6_type) {
        case ICMP6_DST_UNREACH:
                switch (icmp6->icmp6_code) {
                case ICMP6_DST_UNREACH_NOPORT:
                        error = ECONNREFUSED;
                        break;
                case ICMP6_DST_UNREACH_ADMIN:
                case ICMP6_DST_UNREACH_NOROUTE:
                case ICMP6_DST_UNREACH_BEYONDSCOPE:
                case ICMP6_DST_UNREACH_ADDR:
                        /* Transient errors */
                        break;
                default:
                        break;
                }
                break;
        case ICMP6_PACKET_TOO_BIG: {
                struct T_unitdata_ind   *tudi;
                struct T_opthdr         *toh;
                size_t                  udi_size;
                mblk_t                  *newmp;
                t_scalar_t              opt_length = sizeof (struct T_opthdr) +
                    sizeof (struct ip6_mtuinfo);
                sin6_t                  *sin6;
                struct ip6_mtuinfo      *mtuinfo;

                /*
```



```
                 * If the application has requested to receive path mtu
                 * information, send up an empty message containing an
                 * IPV6_PATHMTU ancillary data item.
                 */
                if (!connp->conn_ipv6_recvpathmtu)
                        break;

                udi_size = sizeof (struct T_unitdata_ind) + sizeof (sin6_t) +
                    opt_length;
                if ((newmp = allocb(udi_size, BPRI_MED)) == NULL) {
                        UDPS_BUMP_MIB(us, udpInErrors);
                        break;
                }

                /*
                 * newmp->b_cont is left to NULL on purpose.  This is an
                 * empty message containing only ancillary data.
                 */
                newmp->b_datap->db_type = M_PROTO;
                tudi = (struct T_unitdata_ind *)newmp->b_rptr;
                newmp->b_wptr = (uchar_t *)tudi + udi_size;
                tudi->PRIM_type = T_UNITDATA_IND;
                tudi->SRC_length = sizeof (sin6_t);
                tudi->SRC_offset = sizeof (struct T_unitdata_ind);
                tudi->OPT_offset = tudi->SRC_offset + sizeof (sin6_t);
                tudi->OPT_length = opt_length;

                sin6 = (sin6_t *)&tudi[1];
                bzero(sin6, sizeof (sin6_t));
                sin6->sin6_family = AF_INET6;
                sin6->sin6_addr = connp->conn_faddr_v6;

                toh = (struct T_opthdr *)&sin6[1];
                toh->level = IPPROTO_IPV6;
                toh->name = IPV6_PATHMTU;
                toh->len = opt_length;
                toh->status = 0;

                mtuinfo = (struct ip6_mtuinfo *)&toh[1];
                bzero(mtuinfo, sizeof (struct ip6_mtuinfo));
                mtuinfo->ip6m_addr.sin6_family = AF_INET6;
                mtuinfo->ip6m_addr.sin6_addr = ip6h->ip6_dst;
                mtuinfo->ip6m_mtu = icmp6->icmp6_mtu;
                /*
                 * We've consumed everything we need from the original
                 * message.  Free it, then send our empty message.
                 */
                freemsg(mp);
                udp_ulp_recv(connp, newmp, msgdsize(newmp), ira);
                return;
        }
        case ICMP6_TIME_EXCEEDED:
                /* Transient errors */
                break;
        case ICMP6_PARAM_PROB:
                /* If this corresponds to an ICMP_PROTOCOL_UNREACHABLE */
                if (icmp6->icmp6_code == ICMP6_PARAMPROB_NEXTHEADER &&
                    (uchar_t *)ip6h + icmp6->icmp6_pptr ==
                    (uchar_t *)nexthdrp) {
                        error = ECONNREFUSED;
                        break;
                }
                break;
        }
        if (error == 0) {
                freemsg(mp);
```

```
1184                return;
1185        }

1187        /*
1188         * Deliver T_UDERROR_IND when the application has asked for it.
1189         * The socket layer enables this automatically when connected.
1190         */
1191        if (!connp->conn_dgram_errind) {
1192                freemsg(mp);
1193                return;
1194        }

1196        sin6 = sin6_null;
1197        sin6.sin6_family = AF_INET6;
1198        sin6.sin6_addr = ip6h->ip6_dst;
1199        sin6.sin6_port = udpha->uha_dst_port;
1200        sin6.sin6_flowinfo = ip6h->ip6_vcf & ~IPV6_VERS_AND_FLOW_MASK;

1202        if (IPCL_IS_NONSTR(connp)) {
1203                mutex_enter(&connp->conn_lock);
1204                if (udp->udp_state == TS_DATA_XFER) {
1205                        if (sin6.sin6_port == connp->conn_fport &&
1206                            IN6_ARE_ADDR_EQUAL(&sin6.sin6_addr,
1207                            &connp->conn_faddr_v6)) {
1208                                mutex_exit(&connp->conn_lock);
1209                                (*connp->conn_upcalls->su_set_error)
1210                                    (connp->conn_upper_handle, error);
1211                                goto done;
1212                        }
1213                } else {
1214                        udp->udp_delayed_error = error;
1215                        *((sin6_t *)&udp->udp_delayed_addr) = sin6;
1216                }
1217                mutex_exit(&connp->conn_lock);
1218        } else {
1219                mp1 = mi_tpi_uderror_ind((char *)&sin6, sizeof (sin6_t),
1220                    NULL, 0, error);
1221                if (mp1 != NULL)
1222                        putnext(connp->conn_rq, mp1);
1223        }
1224 done:
1225        freemsg(mp);
1226 }

1228 /*
1229  * This routine responds to T_ADDR_REQ messages.  It is called by udp_wput.
1230  * The local address is filled in if endpoint is bound. The remote address
1231  * is filled in if remote address has been precified ("connected endpoint")
1232  * (The concept of connected CLTS sockets is alien to published TPI
1233  *  but we support it anyway).
1234  */
1235 static void
1236 udp_addr_req(queue_t *q, mblk_t *mp)
1237 {
1238        struct sockaddr *sa;
1239        mblk_t  *ackmp;
1240        struct T_addr_ack *taa;
1241        udp_t   *udp = Q_TO_UDP(q);
1242        conn_t  *connp = udp->udp_connp;
1243        uint_t  addrlen;

1245        /* Make it large enough for worst case */
1246        ackmp = reallocb(mp, sizeof (struct T_addr_ack) +
1247            2 * sizeof (sin6_t), 1);
1248        if (ackmp == NULL) {
1249                udp_err_ack(q, mp, TSYSERR, ENOMEM);
```



```
1250                return;
1251        }
1252        taa = (struct T_addr_ack *)ackmp->b_rptr;

1254        bzero(taa, sizeof (struct T_addr_ack));
1255        ackmp->b_wptr = (uchar_t *)&taa[1];

1257        taa->PRIM_type = T_ADDR_ACK;
1258        ackmp->b_datap->db_type = M_PCPROTO;

1260        if (connp->conn_family == AF_INET)
1261                addrlen = sizeof (sin_t);
1262        else
1263                addrlen = sizeof (sin6_t);

1265        mutex_enter(&connp->conn_lock);
1266        /*
1267         * Note: Following code assumes 32 bit alignment of basic
1268         * data structures like sin_t and struct T_addr_ack.
1269         */
1270        if (udp->udp_state != TS_UNBND) {
1271                /*
1272                 * Fill in local address first
1273                 */
1274                taa->LOCADDR_offset = sizeof (*taa);
1275                taa->LOCADDR_length = addrlen;
1276                sa = (struct sockaddr *)&taa[1];
1277                (void) conn_getsockname(connp, sa, &addrlen);
1278                ackmp->b_wptr += addrlen;
1279        }
1280        if (udp->udp_state == TS_DATA_XFER) {
1281                /*
1282                 * connected, fill remote address too
1283                 */
1284                taa->REMADDR_length = addrlen;
1285                /* assumed 32-bit alignment */
1286                taa->REMADDR_offset = taa->LOCADDR_offset + taa->LOCADDR_length;
1287                sa = (struct sockaddr *)(ackmp->b_rptr + taa->REMADDR_offset);
1288                (void) conn_getpeername(connp, sa, &addrlen);
1289                ackmp->b_wptr += addrlen;
1290        }
1291        mutex_exit(&connp->conn_lock);
1292        ASSERT(ackmp->b_wptr <= ackmp->b_datap->db_lim);
1293        qreply(q, ackmp);
1294 }

1296 static void
1297 udp_copy_info(struct T_info_ack *tap, udp_t *udp)
1298 {
1299        conn_t          *connp = udp->udp_connp;

1301        if (connp->conn_family == AF_INET) {
1302                *tap = udp_g_t_info_ack_ipv4;
1303        } else {
1304                *tap = udp_g_t_info_ack_ipv6;
1305        }
1306        tap->CURRENT_state = udp->udp_state;
1307        tap->OPT_size = udp_max_optsize;
1308 }

1310 static void
1311 udp_do_capability_ack(udp_t *udp, struct T_capability_ack *tcap,
1312     t_uscalar_t cap_bits1)
1313 {
1314        tcap->CAP_bits1 = 0;
```

```
1316         if (cap_bits1 & TC1_INFO) {
1317                 udp_copy_info(&tcap->INFO_ack, udp);
1318                 tcap->CAP_bits1 |= TC1_INFO;
1319         }
1320 }

1322 /*
1323  * This routine responds to T_CAPABILITY_REQ messages.  It is called by
1324  * udp_wput.  Much of the T_CAPABILITY_ACK information is copied from
1325  * udp_g_t_info_ack.  The current state of the stream is copied from
1326  * udp_state.
1327  */
1328 static void
1329 udp_capability_req(queue_t *q, mblk_t *mp)
1330 {
1331         t_uscalar_t             cap_bits1;
1332         struct T_capability_ack *tcap;
1333         udp_t   *udp = Q_TO_UDP(q);

1335         cap_bits1 = ((struct T_capability_req *)mp->b_rptr)->CAP_bits1;

1337         mp = tpi_ack_alloc(mp, sizeof (struct T_capability_ack),
1338             mp->b_datap->db_type, T_CAPABILITY_ACK);
1339         if (!mp)
1340                 return;

1342         tcap = (struct T_capability_ack *)mp->b_rptr;
1343         udp_do_capability_ack(udp, tcap, cap_bits1);

1345         qreply(q, mp);
1346 }

1348 /*
1349  * This routine responds to T_INFO_REQ messages.  It is called by udp_wput.
1350  * Most of the T_INFO_ACK information is copied from udp_g_t_info_ack.
1351  * The current state of the stream is copied from udp_state.
1352  */
1353 static void
1354 udp_info_req(queue_t *q, mblk_t *mp)
1355 {
1356         udp_t *udp = Q_TO_UDP(q);

1358         /* Create a T_INFO_ACK message. */
1359         mp = tpi_ack_alloc(mp, sizeof (struct T_info_ack), M_PCPROTO,
1360             T_INFO_ACK);
1361         if (!mp)
1362                 return;
1363         udp_copy_info((struct T_info_ack *)mp->b_rptr, udp);
1364         qreply(q, mp);
1365 }

1367 /* For /dev/udp aka AF_INET open */
1368 static int
1369 udp_openv4(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
1370 {
1371         return (udp_open(q, devp, flag, sflag, credp, B_FALSE));
1372 }

1374 /* For /dev/udp6 aka AF_INET6 open */
1375 static int
1376 udp_openv6(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
1377 {
1378         return (udp_open(q, devp, flag, sflag, credp, B_TRUE));
1379 }

1381 /*
```



```
1382  * This is the open routine for udp.  It allocates a udp_t structure for
1383  * the stream and, on the first open of the module, creates an ND table.
1384  */
1385 static int
1386 udp_open(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp,
1387     boolean_t isv6)
1388 {
1389         udp_t           *udp;
1390         conn_t          *connp;
1391         dev_t           conn_dev;
1392         vmem_t          *minor_arena;
1393         int             err;

1395         /* If the stream is already open, return immediately. */
1396         if (q->q_ptr != NULL)
1397                 return (0);

1399         if (sflag == MODOPEN)
1400                 return (EINVAL);

1402         if ((ip_minor_arena_la != NULL) && (flag & SO_SOCKSTR) &&
1403             ((conn_dev = inet_minor_alloc(ip_minor_arena_la)) != 0)) {
1404                 minor_arena = ip_minor_arena_la;
1405         } else {
1406                 /*
1407                  * Either minor numbers in the large arena were exhausted
1408                  * or a non socket application is doing the open.
1409                  * Try to allocate from the small arena.
1410                  */
1411                 if ((conn_dev = inet_minor_alloc(ip_minor_arena_sa)) == 0)
1412                         return (EBUSY);

1414                 minor_arena = ip_minor_arena_sa;
1415         }

1417         if (flag & SO_FALLBACK) {
1418                 /*
1419                  * Non streams socket needs a stream to fallback to
1420                  */
1421                 RD(q)->q_ptr = (void *)conn_dev;
1422                 WR(q)->q_qinfo = &udp_fallback_sock_winit;
1423                 WR(q)->q_ptr = (void *)minor_arena;
1424                 qprocson(q);
1425                 return (0);
1426         }

1428         connp = udp_do_open(credp, isv6, KM_SLEEP, &err);
1429         if (connp == NULL) {
1430                 inet_minor_free(minor_arena, conn_dev);
1431                 return (err);
1432         }
1433         udp = connp->conn_udp;

1435         *devp = makedevice(getemajor(*devp), (minor_t)conn_dev);
1436         connp->conn_dev = conn_dev;
1437         connp->conn_minor_arena = minor_arena;

1439         /*
1440          * Initialize the udp_t structure for this stream.
1441          */
1442         q->q_ptr = connp;
1443         WR(q)->q_ptr = connp;
1444         connp->conn_rq = q;
1445         connp->conn_wq = WR(q);

1447         /*
```

```
1448          * Since this conn_t/udp_t is not yet visible to anybody else we don't
1449          * need to lock anything.
1450          */
1451         ASSERT(connp->conn_proto == IPPROTO_UDP);
1452         ASSERT(connp->conn_udp == udp);
1453         ASSERT(udp->udp_connp == connp);

1455         if (flag & SO_SOCKSTR) {
1456                 udp->udp_issocket = B_TRUE;
1457         }

1459         WR(q)->q_hiwat = connp->conn_sndbuf;
1460         WR(q)->q_lowat = connp->conn_sndlowat;

1462         qprocson(q);

1464         /* Set the Stream head write offset and high watermark. */
1465         (void) proto_set_tx_wroff(q, connp, connp->conn_wroff);
1466         (void) proto_set_rx_hiwat(q, connp,
1467             udp_set_rcv_hiwat(udp, connp->conn_rcvbuf));

1469         mutex_enter(&connp->conn_lock);
1470         connp->conn_state_flags &= ~CONN_INCIPIENT;
1471         mutex_exit(&connp->conn_lock);
1472         return (0);
1473 }

1475 /*
1476  * Which UDP options OK to set through T_UNITDATA_REQ...
1477  */
1478 /* ARGSUSED */
1479 static boolean_t
1480 udp_opt_allow_udr_set(t_scalar_t level, t_scalar_t name)
1481 {
1482         return (B_TRUE);
1483 }

1485 /*
1486  * This routine gets default values of certain options whose default
1487  * values are maintained by protcol specific code
1488  */
1489 int
1490 udp_opt_default(queue_t *q, t_scalar_t level, t_scalar_t name, uchar_t *ptr)
1491 {
1492         udp_t           *udp = Q_TO_UDP(q);
1493         udp_stack_t *us = udp->udp_us;
1494         int *i1 = (int *)ptr;

1496         switch (level) {
1497         case IPPROTO_IP:
1498                 switch (name) {
1499                 case IP_MULTICAST_TTL:
1500                         *ptr = (uchar_t)IP_DEFAULT_MULTICAST_TTL;
1501                         return (sizeof (uchar_t));
1502                 case IP_MULTICAST_LOOP:
1503                         *ptr = (uchar_t)IP_DEFAULT_MULTICAST_LOOP;
1504                         return (sizeof (uchar_t));
1505                 }
1506                 break;
1507         case IPPROTO_IPV6:
1508                 switch (name) {
1509                 case IPV6_MULTICAST_HOPS:
1510                         *i1 = IP_DEFAULT_MULTICAST_TTL;
1511                         return (sizeof (int));
1512                 case IPV6_MULTICAST_LOOP:
1513                         *i1 = IP_DEFAULT_MULTICAST_LOOP;
```



```
1514                         return (sizeof (int));
1515                 case IPV6_UNICAST_HOPS:
1516                         *i1 = us->us_ipv6_hoplimit;
1517                         return (sizeof (int));
1518                 }
1519                 break;
1520         }
1521         return (-1);
1522 }

1524 /*
1525  * This routine retrieves the current status of socket options.
1526  * It returns the size of the option retrieved, or -1.
1527  */
1528 int
1529 udp_opt_get(conn_t *connp, t_scalar_t level, t_scalar_t name,
1530     uchar_t *ptr)
1531 {
1532         int             *i1 = (int *)ptr;
1533         udp_t           *udp = connp->conn_udp;
1534         int             len;
1535         conn_opt_arg_t  coas;
1536         int             retval;

1538         coas.coa_connp = connp;
1539         coas.coa_ixa = connp->conn_ixa;
1540         coas.coa_ipp = &connp->conn_xmit_ipp;
1541         coas.coa_ancillary = B_FALSE;
1542         coas.coa_changed = 0;

1544         /*
1545          * We assume that the optcom framework has checked for the set
1546          * of levels and names that are supported, hence we don't worry
1547          * about rejecting based on that.
1548          * First check for UDP specific handling, then pass to common routine.
1549          */
1550         switch (level) {
1551         case IPPROTO_IP:
1552                 /*
1553                  * Only allow IPv4 option processing on IPv4 sockets.
1554                  */
1555                 if (connp->conn_family != AF_INET)
1556                         return (-1);

1558                 switch (name) {
1559                 case IP_OPTIONS:
1560                 case T_IP_OPTIONS:
1561                         mutex_enter(&connp->conn_lock);
1562                         if (!(udp->udp_recv_ipp.ipp_fields &
1563                             IPPF_IPV4_OPTIONS)) {
1564                                 mutex_exit(&connp->conn_lock);
1565                                 return (0);
1566                         }

1568                         len = udp->udp_recv_ipp.ipp_ipv4_options_len;
1569                         ASSERT(len != 0);
1570                         bcopy(udp->udp_recv_ipp.ipp_ipv4_options, ptr, len);
1571                         mutex_exit(&connp->conn_lock);
1572                         return (len);
1573                 }
1574                 break;
1575         case IPPROTO_UDP:
1576                 switch (name) {
1577                 case UDP_NAT_T_ENDPOINT:
1578                         mutex_enter(&connp->conn_lock);
1579                         *i1 = udp->udp_nat_t_endpoint;
```

```
1580                        mutex_exit(&connp->conn_lock);
1581                        return (sizeof (int));
1582                case UDP_RCVHDR:
1583                        mutex_enter(&connp->conn_lock);
1584                        *i1 = udp->udp_rcvhdr ? 1 : 0;
1585                        mutex_exit(&connp->conn_lock);
1586                        return (sizeof (int));
1587                }
1588        }
1589        mutex_enter(&connp->conn_lock);
1590        retval = conn_opt_get(&coas, level, name, ptr);
1591        mutex_exit(&connp->conn_lock);
1592        return (retval);
1593 }

1595 /*
1596  * This routine retrieves the current status of socket options.
1597  * It returns the size of the option retrieved, or -1.
1598  */
1599 int
1600 udp_tpi_opt_get(queue_t *q, t_scalar_t level, t_scalar_t name, uchar_t *ptr)
1601 {
1602        conn_t          *connp = Q_TO_CONN(q);
1603        int             err;

1605        err = udp_opt_get(connp, level, name, ptr);
1606        return (err);
1607 }

1609 /*
1610  * This routine sets socket options.
1611  */
1612 int
1613 udp_do_opt_set(conn_opt_arg_t *coa, int level, int name,
1614     uint_t inlen, uchar_t *invalp, cred_t *cr, boolean_t checkonly)
1615 {
1616        conn_t          *connp = coa->coa_connp;
1617        ip_xmit_attr_t  *ixa = coa->coa_ixa;
1618        udp_t           *udp = connp->conn_udp;
1619        udp_stack_t     *us = udp->udp_us;
1620        int             *i1 = (int *)invalp;
1621        boolean_t       onoff = (*i1 == 0) ? 0 : 1;
1622        int             error;

1624        ASSERT(MUTEX_NOT_HELD(&coa->coa_connp->conn_lock));
1625        /*
1626         * First do UDP specific sanity checks and handle UDP specific
1627         * options. Note that some IPPROTO_UDP options are handled
1628         * by conn_opt_set.
1629         */
1630        switch (level) {
1631        case SOL_SOCKET:
1632                switch (name) {
1633                case SO_SNDBUF:
1634                        if (*i1 > us->us_max_buf) {
1635                                return (ENOBUFS);
1636                        }
1637                        break;
1638                case SO_RCVBUF:
1639                        if (*i1 > us->us_max_buf) {
1640                                return (ENOBUFS);
1641                        }
1642                        break;

1644                case SCM_UCRED: {
1645                        struct ucred_s *ucr;
```



```
1646                        cred_t *newcr;
1647                        ts_label_t *tsl;

1649                        /*
1650                         * Only sockets that have proper privileges and are
1651                         * bound to MLPs will have any other value here, so
1652                         * this implicitly tests for privilege to set label.
1653                         */
1654                        if (connp->conn_mlp_type == mlptSingle)
1655                                break;

1657                        ucr = (struct ucred_s *)invalp;
1658                        if (inlen < sizeof (*ucr) + sizeof (bslabel_t) ||
1659                            ucr->uc_labeloff < sizeof (*ucr) ||
1660                            ucr->uc_labeloff + sizeof (bslabel_t) > inlen)
1661                                return (EINVAL);
1662                        if (!checkonly) {
1663                                /*
1664                                 * Set ixa_tsl to the new label.
1665                                 * We assume that crgetzoneid doesn't change
1666                                 * as part of the SCM_UCRED.
1667                                 */
1668                                ASSERT(cr != NULL);
1669                                if ((tsl = crgetlabel(cr)) == NULL)
1670                                        return (EINVAL);
1671                                newcr = copycred_from_bslabel(cr, UCLABEL(ucr),
1672                                    tsl->tsl_doi, KM_NOSLEEP);
1673                                if (newcr == NULL)
1674                                        return (ENOSR);
1675                                ASSERT(newcr->cr_label != NULL);
1676                                /*
1677                                 * Move the hold on the cr_label to ixa_tsl by
1678                                 * setting cr_label to NULL. Then release newcr.
1679                                 */
1680                                ip_xmit_attr_replace_tsl(ixa, newcr->cr_label);
1681                                ixa->ixa_flags |= IXAF_UCRED_TSL;
1682                                newcr->cr_label = NULL;
1683                                crfree(newcr);
1684                                coa->coa_changed |= COA_HEADER_CHANGED;
1685                                coa->coa_changed |= COA_WROFF_CHANGED;
1686                        }
1687                        /* Fully handled this option. */
1688                        return (0);
1689                }
1690                }
1691                break;
1692        case IPPROTO_UDP:
1693                switch (name) {
1694                case UDP_NAT_T_ENDPOINT:
1695                        if ((error = secpolicy_ip_config(cr, B_FALSE)) != 0) {
1696                                return (error);
1697                        }

1699                        /*
1700                         * Use conn_family instead so we can avoid ambiguitites
1701                         * with AF_INET6 sockets that may switch from IPv4
1702                         * to IPv6.
1703                         */
1704                        if (connp->conn_family != AF_INET) {
1705                                return (EAFNOSUPPORT);
1706                        }

1708                        if (!checkonly) {
1709                                mutex_enter(&connp->conn_lock);
1710                                udp->udp_nat_t_endpoint = onoff;
1711                                mutex_exit(&connp->conn_lock);
```

```
1712                                coa->coa_changed |= COA_HEADER_CHANGED;
1713                                coa->coa_changed |= COA_WROFF_CHANGED;
1714                        }
1715                        /* Fully handled this option. */
1716                        return (0);
1717                case UDP_RCVHDR:
1718                        mutex_enter(&connp->conn_lock);
1719                        udp->udp_rcvhdr = onoff;
1720                        mutex_exit(&connp->conn_lock);
1721                        return (0);
1722                }
1723                break;
1724        }
1725        error = conn_opt_set(coa, level, name, inlen, invalp,
1726            checkonly, cr);
1727        return (error);
1728 }

1730 /*
1731  * This routine sets socket options.
1732  */
1733 int
1734 udp_opt_set(conn_t *connp, uint_t optset_context, int level,
1735     int name, uint_t inlen, uchar_t *invalp, uint_t *outlenp,
1736     uchar_t *outvalp, void *thisdg_attrs, cred_t *cr)
1737 {
1738        udp_t           *udp = connp->conn_udp;
1739        int             err;
1740        conn_opt_arg_t  coas, *coa;
1741        boolean_t       checkonly;
1742        udp_stack_t     *us = udp->udp_us;

1744        switch (optset_context) {
1745        case SETFN_OPTCOM_CHECKONLY:
1746                checkonly = B_TRUE;
1747                /*
1748                 * Note: Implies T_CHECK semantics for T_OPTCOM_REQ
1749                 * inlen != 0 implies value supplied and
1750                 *      we have to "pretend" to set it.
1751                 * inlen == 0 implies that there is no
1752                 *      value part in T_CHECK request and just validation
1753                 * done elsewhere should be enough, we just return here.
1754                 */
1755                if (inlen == 0) {
1756                        *outlenp = 0;
1757                        return (0);
1758                }
1759                break;
1760        case SETFN_OPTCOM_NEGOTIATE:
1761                checkonly = B_FALSE;
1762                break;
1763        case SETFN_UD_NEGOTIATE:
1764        case SETFN_CONN_NEGOTIATE:
1765                checkonly = B_FALSE;
1766                /*
1767                 * Negotiating local and "association-related" options
1768                 * through T_UNITDATA_REQ.
1769                 *
1770                 * Following routine can filter out ones we do not
1771                 * want to be "set" this way.
1772                 */
1773                if (!udp_opt_allow_udr_set(level, name)) {
1774                        *outlenp = 0;
1775                        return (EINVAL);
1776                }
1777                break;
```



```
1778        default:
1779                /*
1780                 * We should never get here
1781                 */
1782                *outlenp = 0;
1783                return (EINVAL);
1784        }

1786        ASSERT((optset_context != SETFN_OPTCOM_CHECKONLY) ||
1787            (optset_context == SETFN_OPTCOM_CHECKONLY && inlen != 0));

1789        if (thisdg_attrs != NULL) {
1790                /* Options from T_UNITDATA_REQ */
1791                coa = (conn_opt_arg_t *)thisdg_attrs;
1792                ASSERT(coa->coa_connp == connp);
1793                ASSERT(coa->coa_ixa != NULL);
1794                ASSERT(coa->coa_ipp != NULL);
1795                ASSERT(coa->coa_ancillary);
1796        } else {
1797                coa = &coas;
1798                coas.coa_connp = connp;
1799                /* Get a reference on conn_ixa to prevent concurrent mods */
1800                coas.coa_ixa = conn_get_ixa(connp, B_TRUE);
1801                if (coas.coa_ixa == NULL) {
1802                        *outlenp = 0;
1803                        return (ENOMEM);
1804                }
1805                coas.coa_ipp = &connp->conn_xmit_ipp;
1806                coas.coa_ancillary = B_FALSE;
1807                coas.coa_changed = 0;
1808        }

1810        err = udp_do_opt_set(coa, level, name, inlen, invalp,
1811            cr, checkonly);
1812        if (err != 0) {
1813 errout:
1814                if (!coa->coa_ancillary)
1815                        ixa_refrele(coa->coa_ixa);
1816                *outlenp = 0;
1817                return (err);
1818        }
1819        /* Handle DHCPINIT here outside of lock */
1820        if (level == IPPROTO_IP && name == IP_DHCPINIT_IF) {
1821                uint_t  ifindex;
1822                ill_t   *ill;

1824                ifindex = *(uint_t *)invalp;
1825                if (ifindex == 0) {
1826                        ill = NULL;
1827                } else {
1828                        ill = ill_lookup_on_ifindex(ifindex, B_FALSE,
1829                            coa->coa_ixa->ixa_ipst);
1830                        if (ill == NULL) {
1831                                err = ENXIO;
1832                                goto errout;
1833                        }

1835                        mutex_enter(&ill->ill_lock);
1836                        if (ill->ill_state_flags & ILL_CONDEMNED) {
1837                                mutex_exit(&ill->ill_lock);
1838                                ill_refrele(ill);
1839                                err = ENXIO;
1840                                goto errout;
1841                        }
1842                        if (IS_VNI(ill)) {
1843                                mutex_exit(&ill->ill_lock);
```

```
1844                                ill_refrele(ill);
1845                                err = EINVAL;
1846                                goto errout;
1847                        }
1848                }
1849                mutex_enter(&connp->conn_lock);

1851                if (connp->conn_dhcpinit_ill != NULL) {
1852                        /*
1853                         * We've locked the conn so conn_cleanup_ill()
1854                         * cannot clear conn_dhcpinit_ill -- so it's
1855                         * safe to access the ill.
1856                         */
1857                        ill_t *oill = connp->conn_dhcpinit_ill;

1859                        ASSERT(oill->ill_dhcpinit != 0);
1860                        atomic_dec_32(&oill->ill_dhcpinit);
1861                        ill_set_inputfn(connp->conn_dhcpinit_ill);
1862                        connp->conn_dhcpinit_ill = NULL;
1863                }

1865                if (ill != NULL) {
1866                        connp->conn_dhcpinit_ill = ill;
1867                        atomic_inc_32(&ill->ill_dhcpinit);
1868                        ill_set_inputfn(ill);
1869                        mutex_exit(&connp->conn_lock);
1870                        mutex_exit(&ill->ill_lock);
1871                        ill_refrele(ill);
1872                } else {
1873                        mutex_exit(&connp->conn_lock);
1874                }
1875        }

1877        /*
1878         * Common case of OK return with outval same as inval.
1879         */
1880        if (invalp != outvalp) {
1881                /* don't trust bcopy for identical src/dst */
1882                (void) bcopy(invalp, outvalp, inlen);
1883        }
1884        *outlenp = inlen;

1886        /*
1887         * If this was not ancillary data, then we rebuild the headers,
1888         * update the IRE/NCE, and IPsec as needed.
1889         * Since the label depends on the destination we go through
1890         * ip_set_destination first.
1891         */
1892        if (coa->coa_ancillary) {
1893                return (0);
1894        }

1896        if (coa->coa_changed & COA_ROUTE_CHANGED) {
1897                in6_addr_t saddr, faddr, nexthop;
1898                in_port_t fport;

1900                /*
1901                 * We clear lastdst to make sure we pick up the change
1902                 * next time sending.
1903                 * If we are connected we re-cache the information.
1904                 * We ignore errors to preserve BSD behavior.
1905                 * Note that we don't redo IPsec policy lookup here
1906                 * since the final destination (or source) didn't change.
1907                 */
1908                mutex_enter(&connp->conn_lock);
1909                connp->conn_v6lastdst = ipv6_all_zeros;
```



```
1911                ip_attr_nexthop(coa->coa_ipp, coa->coa_ixa,
1912                    &connp->conn_faddr_v6, &nexthop);
1913                saddr = connp->conn_saddr_v6;
1914                faddr = connp->conn_faddr_v6;
1915                fport = connp->conn_fport;
1916                mutex_exit(&connp->conn_lock);

1918                if (!IN6_IS_ADDR_UNSPECIFIED(&faddr) &&
1919                    !IN6_IS_ADDR_V4MAPPED_ANY(&faddr)) {
1920                        (void) ip_attr_connect(connp, coa->coa_ixa,
1921                            &saddr, &faddr, &nexthop, fport, NULL, NULL,
1922                            IPDF_ALLOW_MCBC | IPDF_VERIFY_DST);
1923                }
1924        }

1926        ixa_refrele(coa->coa_ixa);

1928        if (coa->coa_changed & COA_HEADER_CHANGED) {
1929                /*
1930                 * Rebuild the header template if we are connected.
1931                 * Otherwise clear conn_v6lastdst so we rebuild the header
1932                 * in the data path.
1933                 */
1934                mutex_enter(&connp->conn_lock);
1935                if (!IN6_IS_ADDR_UNSPECIFIED(&connp->conn_faddr_v6) &&
1936                    !IN6_IS_ADDR_V4MAPPED_ANY(&connp->conn_faddr_v6)) {
1937                        err = udp_build_hdr_template(connp,
1938                            &connp->conn_saddr_v6, &connp->conn_faddr_v6,
1939                            connp->conn_fport, connp->conn_flowinfo);
1940                        if (err != 0) {
1941                                mutex_exit(&connp->conn_lock);
1942                                return (err);
1943                        }
1944                } else {
1945                        connp->conn_v6lastdst = ipv6_all_zeros;
1946                }
1947                mutex_exit(&connp->conn_lock);
1948        }
1949        if (coa->coa_changed & COA_RCVBUF_CHANGED) {
1950                (void) proto_set_rx_hiwat(connp->conn_rq, connp,
1951                    connp->conn_rcvbuf);
1952        }
1953        if ((coa->coa_changed & COA_SNDBUF_CHANGED) && !IPCL_IS_NONSTR(connp)) {
1954                connp->conn_wq->q_hiwat = connp->conn_sndbuf;
1955        }
1956        if (coa->coa_changed & COA_WROFF_CHANGED) {
1957                /* Increase wroff if needed */
1958                uint_t wroff;

1960                mutex_enter(&connp->conn_lock);
1961                wroff = connp->conn_ht_iphc_allocated + us->us_wroff_extra;
1962                if (udp->udp_nat_t_endpoint)
1963                        wroff += sizeof (uint32_t);
1964                if (wroff > connp->conn_wroff) {
1965                        connp->conn_wroff = wroff;
1966                        mutex_exit(&connp->conn_lock);
1967                        (void) proto_set_tx_wroff(connp->conn_rq, connp, wroff);
1968                } else {
1969                        mutex_exit(&connp->conn_lock);
1970                }
1971        }
1972        return (err);
1973 }

1975 /* This routine sets socket options. */
```

```
1976 int
1977 udp_tpi_opt_set(queue_t *q, uint_t optset_context, int level, int name,
1978     uint_t inlen, uchar_t *invalp, uint_t *outlenp, uchar_t *outvalp,
1979     void *thisdg_attrs, cred_t *cr)
1980 {
1981         conn_t  *connp = Q_TO_CONN(q);
1982         int error;

1984         error = udp_opt_set(connp, optset_context, level, name, inlen, invalp,
1985             outlenp, outvalp, thisdg_attrs, cr);
1986         return (error);
1987 }

1989 /*
1990  * Setup IP and UDP headers.
1991  * Returns NULL on allocation failure, in which case data_mp is freed.
1992  */
1993 mblk_t *
1994 udp_prepend_hdr(conn_t *connp, ip_xmit_attr_t *ixa, const ip_pkt_t *ipp,
1995     const in6_addr_t *v6src, const in6_addr_t *v6dst, in_port_t dstport,
1996     uint32_t flowinfo, mblk_t *data_mp, int *errorp)
1997 {
1998         mblk_t          *mp;
1999         udpha_t         *udpha;
2000         udp_stack_t     *us = connp->conn_netstack->netstack_udp;
2001         uint_t          data_len;
2002         uint32_t        cksum;
2003         udp_t           *udp = connp->conn_udp;
2004         boolean_t       insert_spi = udp->udp_nat_t_endpoint;
2005         uint_t          ulp_hdr_len;

2007         data_len = msgdsize(data_mp);
2008         ulp_hdr_len = UDPH_SIZE;
2009         if (insert_spi)
2010                 ulp_hdr_len += sizeof (uint32_t);

2012         mp = conn_prepend_hdr(ixa, ipp, v6src, v6dst, IPPROTO_UDP, flowinfo,
2013             ulp_hdr_len, data_mp, data_len, us->us_wroff_extra, &cksum, errorp);
2014         if (mp == NULL) {
2015                 ASSERT(*errorp != 0);
2016                 return (NULL);
2017         }

2019         data_len += ulp_hdr_len;
2020         ixa->ixa_pktlen = data_len + ixa->ixa_ip_hdr_length;

2022         udpha = (udpha_t *)(mp->b_rptr + ixa->ixa_ip_hdr_length);
2023         udpha->uha_src_port = connp->conn_lport;
2024         udpha->uha_dst_port = dstport;
2025         udpha->uha_checksum = 0;
2026         udpha->uha_length = htons(data_len);

2028         /*
2029          * If there was a routing option/header then conn_prepend_hdr
2030          * has massaged it and placed the pseudo-header checksum difference
2031          * in the cksum argument.
2032          *
2033          * Setup header length and prepare for ULP checksum done in IP.
2034          *
2035          * We make it easy for IP to include our pseudo header
2036          * by putting our length in uha_checksum.
2037          * The IP source, destination, and length have already been set by
2038          * conn_prepend_hdr.
2039          */
2040         cksum += data_len;
2041         cksum = (cksum >> 16) + (cksum & 0xFFFF);
```



```
2042         ASSERT(cksum < 0x10000);

2044         if (ixa->ixa_flags & IXAF_IS_IPV4) {
2045                 ipha_t  *ipha = (ipha_t *)mp->b_rptr;

2047                 ASSERT(ntohs(ipha->ipha_length) == ixa->ixa_pktlen);

2049                 /* IP does the checksum if uha_checksum is non-zero */
2050                 if (us->us_do_checksum) {
2051                         if (cksum == 0)
2052                                 udpha->uha_checksum = 0xffff;
2053                         else
2054                                 udpha->uha_checksum = htons(cksum);
2055                 } else {
2056                         udpha->uha_checksum = 0;
2057                 }
2058         } else {
2059                 ip6_t *ip6h = (ip6_t *)mp->b_rptr;

2061                 ASSERT(ntohs(ip6h->ip6_plen) + IPV6_HDR_LEN == ixa->ixa_pktlen);
2062                 if (cksum == 0)
2063                         udpha->uha_checksum = 0xffff;
2064                 else
2065                         udpha->uha_checksum = htons(cksum);
2066         }

2068         /* Insert all-0s SPI now. */
2069         if (insert_spi)
2070                 *((uint32_t *)(udpha + 1)) = 0;

2072         return (mp);
2073 }

2075 static int
2076 udp_build_hdr_template(conn_t *connp, const in6_addr_t *v6src,
2077     const in6_addr_t *v6dst, in_port_t dstport, uint32_t flowinfo)
2078 {
2079         udpha_t         *udpha;
2080         int             error;

2082         ASSERT(MUTEX_HELD(&connp->conn_lock));
2083         /*
2084          * We clear lastdst to make sure we don't use the lastdst path
2085          * next time sending since we might not have set v6dst yet.
2086          */
2087         connp->conn_v6lastdst = ipv6_all_zeros;

2089         error = conn_build_hdr_template(connp, UDPH_SIZE, 0, v6src, v6dst,
2090             flowinfo);
2091         if (error != 0)
2092                 return (error);

2094         /*
2095          * Any routing header/option has been massaged. The checksum difference
2096          * is stored in conn_sum.
2097          */
2098         udpha = (udpha_t *)connp->conn_ht_ulp;
2099         udpha->uha_src_port = connp->conn_lport;
2100         udpha->uha_dst_port = dstport;
2101         udpha->uha_checksum = 0;
2102         udpha->uha_length = htons(UDPH_SIZE);   /* Filled in later */
2103         return (0);
2104 }

2106 static mblk_t *
2107 udp_queue_fallback(udp_t *udp, mblk_t *mp)
```

```
2108 {
2109         ASSERT(MUTEX_HELD(&udp->udp_recv_lock));
2110         if (IPCL_IS_NONSTR(udp->udp_connp)) {
2111                 /*
2112                  * fallback has started but messages have not been moved yet
2113                  */
2114                 if (udp->udp_fallback_queue_head == NULL) {
2115                         ASSERT(udp->udp_fallback_queue_tail == NULL);
2116                         udp->udp_fallback_queue_head = mp;
2117                         udp->udp_fallback_queue_tail = mp;
2118                 } else {
2119                         ASSERT(udp->udp_fallback_queue_tail != NULL);
2120                         udp->udp_fallback_queue_tail->b_next = mp;
2121                         udp->udp_fallback_queue_tail = mp;
2122                 }
2123                 return (NULL);
2124         } else {
2125                 /*
2126                  * Fallback completed, let the caller putnext() the mblk.
2127                  */
2128                 return (mp);
2129         }
2130 }

2132 /*
2133  * Deliver data to ULP. In case we have a socket, and it's falling back to
2134  * TPI, then we'll queue the mp for later processing.
2135  */
2136 static void
2137 udp_ulp_recv(conn_t *connp, mblk_t *mp, uint_t len, ip_recv_attr_t *ira)
2138 {
2139         if (IPCL_IS_NONSTR(connp)) {
2140                 udp_t *udp = connp->conn_udp;
2141                 int error;

2143                 ASSERT(len == msgdsize(mp));
2144                 if ((*connp->conn_upcalls->su_recv)
2145                     (connp->conn_upper_handle, mp, len, 0, &error, NULL) < 0) {
2146                         mutex_enter(&udp->udp_recv_lock);
2147                         if (error == ENOSPC) {
2148                                 /*
2149                                  * let's confirm while holding the lock
2150                                  */
2151                                 if ((*connp->conn_upcalls->su_recv)
2152                                     (connp->conn_upper_handle, NULL, 0, 0,
2153                                     &error, NULL) < 0) {
2154                                         ASSERT(error == ENOSPC);
2155                                         if (error == ENOSPC) {
2156                                                 connp->conn_flow_cntrld =
2157                                                     B_TRUE;
2158                                         }
2159                                 }
2160                                 mutex_exit(&udp->udp_recv_lock);
2161                         } else {
2162                                 ASSERT(error == EOPNOTSUPP);
2163                                 mp = udp_queue_fallback(udp, mp);
2164                                 mutex_exit(&udp->udp_recv_lock);
2165                                 if (mp != NULL)
2166                                         putnext(connp->conn_rq, mp);
2167                         }
2168                 }
2169                 ASSERT(MUTEX_NOT_HELD(&udp->udp_recv_lock));
2170         } else {
2171                 if (is_system_labeled()) {
2172                         ASSERT(ira->ira_cred != NULL);
2173                         /*
```



```
2174                          * Provide for protocols above UDP such as RPC
2175                          * NOPID leaves db_cpid unchanged.
2176                          */
2177                         mblk_setcred(mp, ira->ira_cred, NOPID);
2178                 }

2180                 putnext(connp->conn_rq, mp);
2181         }
2182 }

2184 /*
2185  * This is the inbound data path.
2186  * IP has already pulled up the IP plus UDP headers and verified alignment
2187  * etc.
2188  */
2189 /* ARGSUSED2 */
2190 static void
2191 udp_input(void *arg1, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
2192 {
2193         conn_t                  *connp = (conn_t *)arg1;
2194         struct T_unitdata_ind   *tudi;
2195         uchar_t                 *rptr;          /* Pointer to IP header */
2196         int                     hdr_length;     /* Length of IP+UDP headers */
2197         int                     udi_size;       /* Size of T_unitdata_ind */
2198         int                     pkt_len;
2199         udp_t                   *udp;
2200         udpha_t                 *udpha;
2201         ip_pkt_t                ipps;
2202         ip6_t                   *ip6h;
2203         mblk_t                  *mp1;
2204         uint32_t                udp_ipv4_options_len;
2205         crb_t                   recv_ancillary;
2206         udp_stack_t             *us;

2208         ASSERT(connp->conn_flags & IPCL_UDPCONN);

2210         udp = connp->conn_udp;
2211         us = udp->udp_us;
2212         rptr = mp->b_rptr;

2214         ASSERT(DB_TYPE(mp) == M_DATA);
2215         ASSERT(OK_32PTR(rptr));
2216         ASSERT(ira->ira_pktlen == msgdsize(mp));
2217         pkt_len = ira->ira_pktlen;

2219         /*
2220          * Get a snapshot of these and allow other threads to change
2221          * them after that. We need the same recv_ancillary when determining
2222          * the size as when adding the ancillary data items.
2223          */
2224         mutex_enter(&connp->conn_lock);
2225         udp_ipv4_options_len = udp->udp_recv_ipp.ipp_ipv4_options_len;
2226         recv_ancillary = connp->conn_recv_ancillary;
2227         mutex_exit(&connp->conn_lock);

2229         hdr_length = ira->ira_ip_hdr_length;

2231         /*
2232          * IP inspected the UDP header thus all of it must be in the mblk.
2233          * UDP length check is performed for IPv6 packets and IPv4 packets
2234          * to check if the size of the packet as specified
2235          * by the UDP header is the same as the length derived from the IP
2236          * header.
2237          */
2238         udpha = (udpha_t *)(rptr + hdr_length);
2239         if (pkt_len != ntohs(udpha->uha_length) + hdr_length)
```

```
2240                goto tossit;

2242        hdr_length += UDPH_SIZE;
2243        ASSERT(MBLKL(mp) >= hdr_length);        /* IP did a pullup */

2245        /* Initialize regardless of IP version */
2246        ipps.ipp_fields = 0;

2248        if (((ira->ira_flags & IRAF_IPV4_OPTIONS) ||
2249            udp_ipv4_options_len > 0) &&
2250            connp->conn_family == AF_INET) {
2251                int     err;

2253                /*
2254                 * Record/update udp_recv_ipp with the lock
2255                 * held. Not needed for AF_INET6 sockets
2256                 * since they don't support a getsockopt of IP_OPTIONS.
2257                 */
2258                mutex_enter(&connp->conn_lock);
2259                err = ip_find_hdr_v4((ipha_t *)rptr, &udp->udp_recv_ipp,
2260                    B_TRUE);
2261                if (err != 0) {
2262                        /* Allocation failed. Drop packet */
2263                        mutex_exit(&connp->conn_lock);
2264                        freemsg(mp);
2265                        UDPS_BUMP_MIB(us, udpInErrors);
2266                        return;
2267                }
2268                mutex_exit(&connp->conn_lock);
2269        }

2271        if (recv_ancillary.crb_all != 0) {
2272                /*
2273                 * Record packet information in the ip_pkt_t
2274                 */
2275                if (ira->ira_flags & IRAF_IS_IPV4) {
2276                        ASSERT(IPH_HDR_VERSION(rptr) == IPV4_VERSION);
2277                        ASSERT(MBLKL(mp) >= sizeof (ipha_t));
2278                        ASSERT(((ipha_t *)rptr)->ipha_protocol == IPPROTO_UDP);
2279                        ASSERT(ira->ira_ip_hdr_length == IPH_HDR_LENGTH(rptr));

2281                        (void) ip_find_hdr_v4((ipha_t *)rptr, &ipps, B_FALSE);
2282                } else {
2283                        uint8_t nexthdrp;

2285                        ASSERT(IPH_HDR_VERSION(rptr) == IPV6_VERSION);
2286                        /*
2287                         * IPv6 packets can only be received by applications
2288                         * that are prepared to receive IPv6 addresses.
2289                         * The IP fanout must ensure this.
2290                         */
2291                        ASSERT(connp->conn_family == AF_INET6);

2293                        ip6h = (ip6_t *)rptr;

2295                        /* We don't care about the length, but need the ipp */
2296                        hdr_length = ip_find_hdr_v6(mp, ip6h, B_TRUE, &ipps,
2297                            &nexthdrp);
2298                        ASSERT(hdr_length == ira->ira_ip_hdr_length);
2299                        /* Restore */
2300                        hdr_length = ira->ira_ip_hdr_length + UDPH_SIZE;
2301                        ASSERT(nexthdrp == IPPROTO_UDP);
2302                }
2303        }

2305        /*
```



```
2306         * This is the inbound data path.  Packets are passed upstream as
2307         * T_UNITDATA_IND messages.
2308         */
2309        if (connp->conn_family == AF_INET) {
2310                sin_t *sin;

2312                ASSERT(IPH_HDR_VERSION((ipha_t *)rptr) == IPV4_VERSION);

2314                /*
2315                 * Normally only send up the source address.
2316                 * If any ancillary data items are wanted we add those.
2317                 */
2318                udi_size = sizeof (struct T_unitdata_ind) + sizeof (sin_t);
2319                if (recv_ancillary.crb_all != 0) {
2320                        udi_size += conn_recvancillary_size(connp,
2321                            recv_ancillary, ira, mp, &ipps);
2322                }

2324                /* Allocate a message block for the T_UNITDATA_IND structure. */
2325                mp1 = allocb(udi_size, BPRI_MED);
2326                if (mp1 == NULL) {
2327                        freemsg(mp);
2328                        UDPS_BUMP_MIB(us, udpInErrors);
2329                        return;
2330                }
2331                mp1->b_cont = mp;
2332                mp1->b_datap->db_type = M_PROTO;
2333                tudi = (struct T_unitdata_ind *)mp1->b_rptr;
2334                mp1->b_wptr = (uchar_t *)tudi + udi_size;
2335                tudi->PRIM_type = T_UNITDATA_IND;
2336                tudi->SRC_length = sizeof (sin_t);
2337                tudi->SRC_offset = sizeof (struct T_unitdata_ind);
2338                tudi->OPT_offset = sizeof (struct T_unitdata_ind) +
2339                    sizeof (sin_t);
2340                udi_size -= (sizeof (struct T_unitdata_ind) + sizeof (sin_t));
2341                tudi->OPT_length = udi_size;
2342                sin = (sin_t *)&tudi[1];
2343                sin->sin_addr.s_addr = ((ipha_t *)rptr)->ipha_src;
2344                sin->sin_port = udpha->uha_src_port;
2345                sin->sin_family = connp->conn_family;
2346                *(uint32_t *)&sin->sin_zero[0] = 0;
2347                *(uint32_t *)&sin->sin_zero[4] = 0;

2349                /*
2350                 * Add options if IP_RECVDSTADDR, IP_RECVIF, IP_RECVSLLA or
2351                 * IP_RECVTTL has been set.
2352                 */
2353                if (udi_size != 0) {
2354                        conn_recvancillary_add(connp, recv_ancillary, ira,
2355                            &ipps, (uchar_t *)&sin[1], udi_size);
2356                }
2357        } else {
2358                sin6_t *sin6;

2360                /*
2361                 * Handle both IPv4 and IPv6 packets for IPv6 sockets.
2362                 *
2363                 * Normally we only send up the address. If receiving of any
2364                 * optional receive side information is enabled, we also send
2365                 * that up as options.
2366                 */
2367                udi_size = sizeof (struct T_unitdata_ind) + sizeof (sin6_t);

2369                if (recv_ancillary.crb_all != 0) {
2370                        udi_size += conn_recvancillary_size(connp,
2371                            recv_ancillary, ira, mp, &ipps);
```

```
2372                }

2374                mp1 = allocb(udi_size, BPRI_MED);
2375                if (mp1 == NULL) {
2376                        freemsg(mp);
2377                        UDPS_BUMP_MIB(us, udpInErrors);
2378                        return;
2379                }
2380                mp1->b_cont = mp;
2381                mp1->b_datap->db_type = M_PROTO;
2382                tudi = (struct T_unitdata_ind *)mp1->b_rptr;
2383                mp1->b_wptr = (uchar_t *)tudi + udi_size;
2384                tudi->PRIM_type = T_UNITDATA_IND;
2385                tudi->SRC_length = sizeof (sin6_t);
2386                tudi->SRC_offset = sizeof (struct T_unitdata_ind);
2387                tudi->OPT_offset = sizeof (struct T_unitdata_ind) +
2388                    sizeof (sin6_t);
2389                udi_size -= (sizeof (struct T_unitdata_ind) + sizeof (sin6_t));
2390                tudi->OPT_length = udi_size;
2391                sin6 = (sin6_t *)&tudi[1];
2392                if (ira->ira_flags & IRAF_IS_IPV4) {
2393                        in6_addr_t v6dst;

2395                        IN6_IPADDR_TO_V4MAPPED(((ipha_t *)rptr)->ipha_src,
2396                            &sin6->sin6_addr);
2397                        IN6_IPADDR_TO_V4MAPPED(((ipha_t *)rptr)->ipha_dst,
2398                            &v6dst);
2399                        sin6->sin6_flowinfo = 0;
2400                        sin6->sin6_scope_id = 0;
2401                        sin6->__sin6_src_id = ip_srcid_find_addr(&v6dst,
2402                            IPCL_ZONEID(connp), us->us_netstack);
2403                } else {
2404                        ip6h = (ip6_t *)rptr;

2406                        sin6->sin6_addr = ip6h->ip6_src;
2407                        /* No sin6_flowinfo per API */
2408                        sin6->sin6_flowinfo = 0;
2409                        /* For link-scope pass up scope id */
2410                        if (IN6_IS_ADDR_LINKSCOPE(&ip6h->ip6_src))
2411                                sin6->sin6_scope_id = ira->ira_ruifindex;
2412                        else
2413                                sin6->sin6_scope_id = 0;
2414                        sin6->__sin6_src_id = ip_srcid_find_addr(
2415                            &ip6h->ip6_dst, IPCL_ZONEID(connp),
2416                            us->us_netstack);
2417                }
2418                sin6->sin6_port = udpha->uha_src_port;
2419                sin6->sin6_family = connp->conn_family;

2421                if (udi_size != 0) {
2422                        conn_recvancillary_add(connp, recv_ancillary, ira,
2423                            &ipps, (uchar_t *)&sin6[1], udi_size);
2424                }
2425        }

2427        /*
2428         * DTrace this UDP input as udp:::receive (this is for IPv4, IPv6 and
2429         * loopback traffic).
2430         */
2431        DTRACE_UDP5(receive, mblk_t *, NULL, ip_xmit_attr_t *, connp->conn_ixa,
2432            void_ip_t *, rptr, udp_t *, udp, udpha_t *, udpha);

2434        /* Walk past the headers unless IP_RECVHDR was set. */
2435        if (!udp->udp_rcvhdr) {
2436                mp->b_rptr = rptr + hdr_length;
2437                pkt_len -= hdr_length;
```



```
2438        }

2440        UDPS_BUMP_MIB(us, udpHCInDatagrams);
2441        udp_ulp_recv(connp, mp1, pkt_len, ira);
2442        return;

2444 tossit:
2445        freemsg(mp);
2446        UDPS_BUMP_MIB(us, udpInErrors);
2447 }

2449 /*
2450  * This routine creates a T_UDERROR_IND message and passes it upstream.
2451  * The address and options are copied from the T_UNITDATA_REQ message
2452  * passed in mp.  This message is freed.
2453  */
2454 static void
2455 udp_ud_err(queue_t *q, mblk_t *mp, t_scalar_t err)
2456 {
2457        struct T_unitdata_req *tudr;
2458        mblk_t  *mp1;
2459        uchar_t *destaddr;
2460        t_scalar_t destlen;
2461        uchar_t *optaddr;
2462        t_scalar_t optlen;

2464        if ((mp->b_wptr < mp->b_rptr) ||
2465            (MBLKL(mp)) < sizeof (struct T_unitdata_req)) {
2466                goto done;
2467        }
2468        tudr = (struct T_unitdata_req *)mp->b_rptr;
2469        destaddr = mp->b_rptr + tudr->DEST_offset;
2470        if (destaddr < mp->b_rptr || destaddr >= mp->b_wptr ||
2471            destaddr + tudr->DEST_length < mp->b_rptr ||
2472            destaddr + tudr->DEST_length > mp->b_wptr) {
2473                goto done;
2474        }
2475        optaddr = mp->b_rptr + tudr->OPT_offset;
2476        if (optaddr < mp->b_rptr || optaddr >= mp->b_wptr ||
2477            optaddr + tudr->OPT_length < mp->b_rptr ||
2478            optaddr + tudr->OPT_length > mp->b_wptr) {
2479                goto done;
2480        }
2481        destlen = tudr->DEST_length;
2482        optlen = tudr->OPT_length;

2484        mp1 = mi_tpi_uderror_ind((char *)destaddr, destlen,
2485            (char *)optaddr, optlen, err);
2486        if (mp1 != NULL)
2487                qreply(q, mp1);

2489 done:
2490        freemsg(mp);
2491 }

2493 /*
2494  * This routine removes a port number association from a stream.  It
2495  * is called by udp_wput to handle T_UNBIND_REQ messages.
2496  */
2497 static void
2498 udp_tpi_unbind(queue_t *q, mblk_t *mp)
2499 {
2500        conn_t  *connp = Q_TO_CONN(q);
2501        int     error;

2503        error = udp_do_unbind(connp);
```

```
2504         if (error) {
2505                 if (error < 0)
2506                         udp_err_ack(q, mp, -error, 0);
2507                 else
2508                         udp_err_ack(q, mp, TSYSERR, error);
2509                 return;
2510         }

2512         mp = mi_tpi_ok_ack_alloc(mp);
2513         ASSERT(mp != NULL);
2514         ASSERT(((struct T_ok_ack *)mp->b_rptr)->PRIM_type == T_OK_ACK);
2515         qreply(q, mp);
2516 }

2518 /*
2519  * Don't let port fall into the privileged range.
2520  * Since the extra privileged ports can be arbitrary we also
2521  * ensure that we exclude those from consideration.
2522  * us->us_epriv_ports is not sorted thus we loop over it until
2523  * there are no changes.
2524  */
2525 static in_port_t
2526 udp_update_next_port(udp_t *udp, in_port_t port, boolean_t random)
2527 {
2528         int i, bump;
2529         in_port_t nextport;
2530         boolean_t restart = B_FALSE;
2531         udp_stack_t *us = udp->udp_us;

2533         if (random && udp_random_anon_port != 0) {
2534                 (void) random_get_pseudo_bytes((uint8_t *)&port,
2535                     sizeof (in_port_t));
2536                 /*
2537                  * Unless changed by a sys admin, the smallest anon port
2538                  * is 32768 and the largest anon port is 65535.  It is
2539                  * very likely (50%) for the random port to be smaller
2540                  * than the smallest anon port.  When that happens,
2541                  * add port % (anon port range) to the smallest anon
2542                  * port to get the random port.  It should fall into the
2543                  * valid anon port range.
2544                  */
2545                 if ((port < us->us_smallest_anon_port) ||
2546                     (port > us->us_largest_anon_port)) {
2547                         if (us->us_smallest_anon_port ==
2548                             us->us_largest_anon_port) {
2549                                 bump = 0;
2550                         } else {
2551                                 bump = port % (us->us_largest_anon_port -
2552                                     us->us_smallest_anon_port);
2553                         }

2555                         port = us->us_smallest_anon_port + bump;
2556                 }
2557         }

2559 retry:
2560         if (port < us->us_smallest_anon_port)
2561                 port = us->us_smallest_anon_port;

2563         if (port > us->us_largest_anon_port) {
2564                 port = us->us_smallest_anon_port;
2565                 if (restart)
2566                         return (0);
2567                 restart = B_TRUE;
2568         }
```



```
2570         if (port < us->us_smallest_nonpriv_port)
2571                 port = us->us_smallest_nonpriv_port;

2573         for (i = 0; i < us->us_num_epriv_ports; i++) {
2574                 if (port == us->us_epriv_ports[i]) {
2575                         port++;
2576                         /*
2577                          * Make sure that the port is in the
2578                          * valid range.
2579                          */
2580                         goto retry;
2581                 }
2582         }

2584         if (is_system_labeled() &&
2585             (nextport = tsol_next_port(crgetzone(udp->udp_connp->conn_cred),
2586             port, IPPROTO_UDP, B_TRUE)) != 0) {
2587                 port = nextport;
2588                 goto retry;
2589         }

2591         return (port);
2592 }

2594 /*
2595  * Handle T_UNITDATA_REQ with options. Both IPv4 and IPv6
2596  * Either tudr_mp or msg is set. If tudr_mp we take ancillary data from
2597  * the TPI options, otherwise we take them from msg_control.
2598  * If both sin and sin6 is set it is a connected socket and we use conn_faddr.
2599  * Always consumes mp; never consumes tudr_mp.
2600  */
2601 static int
2602 udp_output_ancillary(conn_t *connp, sin_t *sin, sin6_t *sin6, mblk_t *mp,
2603     mblk_t *tudr_mp, struct nmsghdr *msg, cred_t *cr, pid_t pid)
2604 {
2605         udp_t           *udp = connp->conn_udp;
2606         udp_stack_t     *us = udp->udp_us;
2607         int             error;
2608         ip_xmit_attr_t  *ixa;
2609         ip_pkt_t        *ipp;
2610         in6_addr_t      v6src;
2611         in6_addr_t      v6dst;
2612         in6_addr_t      v6nexthop;
2613         in_port_t       dstport;
2614         uint32_t        flowinfo;
2615         uint_t          srcid;
2616         int             is_absreq_failure = 0;
2617         conn_opt_arg_t  coas, *coa;

2619         ASSERT(tudr_mp != NULL || msg != NULL);

2621         /*
2622          * Get ixa before checking state to handle a disconnect race.
2623          *
2624          * We need an exclusive copy of conn_ixa since the ancillary data
2625          * options might modify it. That copy has no pointers hence we
2626          * need to set them up once we've parsed the ancillary data.
2627          */
2628         ixa = conn_get_ixa_exclusive(connp);
2629         if (ixa == NULL) {
2630                 UDPS_BUMP_MIB(us, udpOutErrors);
2631                 freemsg(mp);
2632                 return (ENOMEM);
2633         }
2634         ASSERT(cr != NULL);
2635         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
```

```
2636         ixa->ixa_cred = cr;
2637         ixa->ixa_cpid = pid;
2638         if (is_system_labeled()) {
2639                 /* We need to restart with a label based on the cred */
2640                 ip_xmit_attr_restore_tsl(ixa, ixa->ixa_cred);
2641         }

2643         /* In case previous destination was multicast or multirt */
2644         ip_attr_newdst(ixa);

2646         /* Get a copy of conn_xmit_ipp since the options might change it */
2647         ipp = kmem_zalloc(sizeof (*ipp), KM_NOSLEEP);
2648         if (ipp == NULL) {
2649                 ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
2650                 ixa->ixa_cred = connp->conn_cred;       /* Restore */
2651                 ixa->ixa_cpid = connp->conn_cpid;
2652                 ixa_refrele(ixa);
2653                 UDPS_BUMP_MIB(us, udpOutErrors);
2654                 freemsg(mp);
2655                 return (ENOMEM);
2656         }
2657         mutex_enter(&connp->conn_lock);
2658         error = ip_pkt_copy(&connp->conn_xmit_ipp, ipp, KM_NOSLEEP);
2659         mutex_exit(&connp->conn_lock);
2660         if (error != 0) {
2661                 UDPS_BUMP_MIB(us, udpOutErrors);
2662                 freemsg(mp);
2663                 goto done;
2664         }

2666         /*
2667          * Parse the options and update ixa and ipp as a result.
2668          * Note that ixa_tsl can be updated if SCM_UCRED.
2669          * ixa_refrele/ixa_inactivate will release any reference on ixa_tsl.
2670          */

2672         coa = &coas;
2673         coa->coa_connp = connp;
2674         coa->coa_ixa = ixa;
2675         coa->coa_ipp = ipp;
2676         coa->coa_ancillary = B_TRUE;
2677         coa->coa_changed = 0;

2679         if (msg != NULL) {
2680                 error = process_auxiliary_options(connp, msg->msg_control,
2681                     msg->msg_controllen, coa, &udp_opt_obj, udp_opt_set, cr);
2682         } else {
2683                 struct T_unitdata_req *tudr;

2685                 tudr = (struct T_unitdata_req *)tudr_mp->b_rptr;
2686                 ASSERT(tudr->PRIM_type == T_UNITDATA_REQ);
2687                 error = tpi_optcom_buf(connp->conn_wq, tudr_mp,
2688                     &tudr->OPT_length, tudr->OPT_offset, cr, &udp_opt_obj,
2689                     coa, &is_absreq_failure);
2690         }
2691         if (error != 0) {
2692                 /*
2693                  * Note: No special action needed in this
2694                  * module for "is_absreq_failure"
2695                  */
2696                 freemsg(mp);
2697                 UDPS_BUMP_MIB(us, udpOutErrors);
2698                 goto done;
2699         }
2700         ASSERT(is_absreq_failure == 0);
```



```
2702         mutex_enter(&connp->conn_lock);
2703         /*
2704          * If laddr is unspecified then we look at sin6_src_id.
2705          * We will give precedence to a source address set with IPV6_PKTINFO
2706          * (aka IPPF_ADDR) but that is handled in build_hdrs. However, we don't
2707          * want ip_attr_connect to select a source (since it can fail) when
2708          * IPV6_PKTINFO is specified.
2709          * If this doesn't result in a source address then we get a source
2710          * from ip_attr_connect() below.
2711          */
2712         v6src = connp->conn_saddr_v6;
2713         if (sin != NULL) {
2714                 IN6_IPADDR_TO_V4MAPPED(sin->sin_addr.s_addr, &v6dst);
2715                 dstport = sin->sin_port;
2716                 flowinfo = 0;
2717                 ixa->ixa_flags &= ~IXAF_SCOPEID_SET;
2718                 ixa->ixa_flags |= IXAF_IS_IPV4;
2719         } else if (sin6 != NULL) {
2720                 boolean_t v4mapped;

2722                 v6dst = sin6->sin6_addr;
2723                 dstport = sin6->sin6_port;
2724                 flowinfo = sin6->sin6_flowinfo;
2725                 srcid = sin6->__sin6_src_id;
2726                 if (IN6_IS_ADDR_LINKSCOPE(&v6dst) && sin6->sin6_scope_id != 0) {
2727                         ixa->ixa_scopeid = sin6->sin6_scope_id;
2728                         ixa->ixa_flags |= IXAF_SCOPEID_SET;
2729                 } else {
2730                         ixa->ixa_flags &= ~IXAF_SCOPEID_SET;
2731                 }
2732                 v4mapped = IN6_IS_ADDR_V4MAPPED(&v6dst);
2733                 if (v4mapped)
2734                         ixa->ixa_flags |= IXAF_IS_IPV4;
2735                 else
2736                         ixa->ixa_flags &= ~IXAF_IS_IPV4;
2737                 if (srcid != 0 && IN6_IS_ADDR_UNSPECIFIED(&v6src)) {
2738                         if (!ip_srcid_find_id(srcid, &v6src, IPCL_ZONEID(connp),
2739                             v4mapped, connp->conn_netstack)) {
2740                                 /* Mismatch - v4mapped/v6 specified by srcid. */
2741                                 mutex_exit(&connp->conn_lock);
2742                                 error = EADDRNOTAVAIL;
2743                                 goto failed;    /* Does freemsg() and mib. */
2744                         }
2745                 }
2746         } else {
2747                 /* Connected case */
2748                 v6dst = connp->conn_faddr_v6;
2749                 dstport = connp->conn_fport;
2750                 flowinfo = connp->conn_flowinfo;
2751         }
2752         mutex_exit(&connp->conn_lock);

2754         /* Handle IP_PKTINFO/IPV6_PKTINFO setting source address. */
2755         if (ipp->ipp_fields & IPPF_ADDR) {
2756                 if (ixa->ixa_flags & IXAF_IS_IPV4) {
2757                         if (IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
2758                                 v6src = ipp->ipp_addr;
2759                 } else {
2760                         if (!IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
2761                                 v6src = ipp->ipp_addr;
2762                 }
2763         }

2765         ip_attr_nexthop(ipp, ixa, &v6dst, &v6nexthop);
2766         error = ip_attr_connect(connp, ixa, &v6src, &v6dst, &v6nexthop, dstport,
2767             &v6src, NULL, IPDF_ALLOW_MCBC | IPDF_VERIFY_DST | IPDF_IPSEC);
```

```
2769         switch (error) {
2770         case 0:
2771                 break;
2772         case EADDRNOTAVAIL:
2773                 /*
2774                  * IXAF_VERIFY_SOURCE tells us to pick a better source.
2775                  * Don't have the application see that errno
2776                  */
2777                 error = ENETUNREACH;
2778                 goto failed;
2779         case ENETDOWN:
2780                 /*
2781                  * Have !ipif_addr_ready address; drop packet silently
2782                  * until we can get applications to not send until we
2783                  * are ready.
2784                  */
2785                 error = 0;
2786                 goto failed;
2787         case EHOSTUNREACH:
2788         case ENETUNREACH:
2789                 if (ixa->ixa_ire != NULL) {
2790                         /*
2791                          * Let conn_ip_output/ire_send_noroute return
2792                          * the error and send any local ICMP error.
2793                          */
2794                         error = 0;
2795                         break;
2796                 }
2797                 /* FALLTHRU */
2798         default:
2799         failed:
2800                 freemsg(mp);
2801                 UDPS_BUMP_MIB(us, udpOutErrors);
2802                 goto done;
2803         }

2805         /*
2806          * We might be going to a different destination than last time,
2807          * thus check that TX allows the communication and compute any
2808          * needed label.
2809          *
2810          * TSOL Note: We have an exclusive ipp and ixa for this thread so we
2811          * don't have to worry about concurrent threads.
2812          */
2813         if (is_system_labeled()) {
2814                 /* Using UDP MLP requires SCM_UCRED from user */
2815                 if (connp->conn_mlp_type != mlptSingle &&
2816                     !((ixa->ixa_flags & IXAF_UCRED_TSL))) {
2817                         UDPS_BUMP_MIB(us, udpOutErrors);
2818                         error = ECONNREFUSED;
2819                         freemsg(mp);
2820                         goto done;
2821                 }
2822                 /*
2823                  * Check whether Trusted Solaris policy allows communication
2824                  * with this host, and pretend that the destination is
2825                  * unreachable if not.
2826                  * Compute any needed label and place it in ipp_label_v4/v6.
2827                  *
2828                  * Later conn_build_hdr_template/conn_prepend_hdr takes
2829                  * ipp_label_v4/v6 to form the packet.
2830                  *
2831                  * Tsol note: We have ipp structure local to this thread so
2832                  * no locking is needed.
2833                  */
```



```
2834                 error = conn_update_label(connp, ixa, &v6dst, ipp);
2835                 if (error != 0) {
2836                         freemsg(mp);
2837                         UDPS_BUMP_MIB(us, udpOutErrors);
2838                         goto done;
2839                 }
2840         }
2841         mp = udp_prepend_hdr(connp, ixa, ipp, &v6src, &v6dst, dstport,
2842             flowinfo, mp, &error);
2843         if (mp == NULL) {
2844                 ASSERT(error != 0);
2845                 UDPS_BUMP_MIB(us, udpOutErrors);
2846                 goto done;
2847         }
2848         if (ixa->ixa_pktlen > IP_MAXPACKET) {
2849                 error = EMSGSIZE;
2850                 UDPS_BUMP_MIB(us, udpOutErrors);
2851                 freemsg(mp);
2852                 goto done;
2853         }
2854         /* We're done.  Pass the packet to ip. */
2855         UDPS_BUMP_MIB(us, udpHCOutDatagrams);

2857         DTRACE_UDP5(send, mblk_t *, NULL, ip_xmit_attr_t *, ixa,
2858             void_ip_t *, mp->b_rptr, udp_t *, udp, udpha_t *,
2859             &mp->b_rptr[ixa->ixa_ip_hdr_length]);

2861         error = conn_ip_output(mp, ixa);
2862         /* No udpOutErrors if an error since IP increases its error counter */
2863         switch (error) {
2864         case 0:
2865                 break;
2866         case EWOULDBLOCK:
2867                 (void) ixa_check_drain_insert(connp, ixa);
2868                 error = 0;
2869                 break;
2870         case EADDRNOTAVAIL:
2871                 /*
2872                  * IXAF_VERIFY_SOURCE tells us to pick a better source.
2873                  * Don't have the application see that errno
2874                  */
2875                 error = ENETUNREACH;
2876                 /* FALLTHRU */
2877         default:
2878                 mutex_enter(&connp->conn_lock);
2879                 /*
2880                  * Clear the source and v6lastdst so we call ip_attr_connect
2881                  * for the next packet and try to pick a better source.
2882                  */
2883                 if (connp->conn_mcbc_bind)
2884                         connp->conn_saddr_v6 = ipv6_all_zeros;
2885                 else
2886                         connp->conn_saddr_v6 = connp->conn_bound_addr_v6;
2887                 connp->conn_v6lastdst = ipv6_all_zeros;
2888                 mutex_exit(&connp->conn_lock);
2889                 break;
2890         }
2891 done:
2892         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
2893         ixa->ixa_cred = connp->conn_cred;       /* Restore */
2894         ixa->ixa_cpid = connp->conn_cpid;
2895         ixa_refrele(ixa);
2896         ip_pkt_free(ipp);
2897         kmem_free(ipp, sizeof (*ipp));
2898         return (error);
2899 }
```

```
/*
 * Handle sending an M_DATA for a connected socket.
 * Handles both IPv4 and IPv6.
 */
static int
udp_output_connected(conn_t *connp, mblk_t *mp, cred_t *cr, pid_t pid)
{
        udp_t           *udp = connp->conn_udp;
        udp_stack_t     *us = udp->udp_us;
        int             error;
        ip_xmit_attr_t  *ixa;

        /*
         * If no other thread is using conn_ixa this just gets a reference to
         * conn_ixa. Otherwise we get a safe copy of conn_ixa.
         */
        ixa = conn_get_ixa(connp, B_FALSE);
        if (ixa == NULL) {
                UDPS_BUMP_MIB(us, udpOutErrors);
                freemsg(mp);
                return (ENOMEM);
        }

        ASSERT(cr != NULL);
        ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
        ixa->ixa_cred = cr;
        ixa->ixa_cpid = pid;

        mutex_enter(&connp->conn_lock);
        mp = udp_prepend_header_template(connp, ixa, mp, &connp->conn_saddr_v6,
            connp->conn_fport, connp->conn_flowinfo, &error);

        if (mp == NULL) {
                ASSERT(error != 0);
                mutex_exit(&connp->conn_lock);
                ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
                ixa->ixa_cred = connp->conn_cred;       /* Restore */
                ixa->ixa_cpid = connp->conn_cpid;
                ixa_refrele(ixa);
                UDPS_BUMP_MIB(us, udpOutErrors);
                freemsg(mp);
                return (error);
        }

        /*
         * In case we got a safe copy of conn_ixa, or if opt_set made us a new
         * safe copy, then we need to fill in any pointers in it.
         */
        if (ixa->ixa_ire == NULL) {
                in6_addr_t      faddr, saddr;
                in6_addr_t      nexthop;
                in_port_t       fport;

                saddr = connp->conn_saddr_v6;
                faddr = connp->conn_faddr_v6;
                fport = connp->conn_fport;
                ip_attr_nexthop(&connp->conn_xmit_ipp, ixa, &faddr, &nexthop);
                mutex_exit(&connp->conn_lock);

                error = ip_attr_connect(connp, ixa, &saddr, &faddr, &nexthop,
                    fport, NULL, NULL, IPDF_ALLOW_MCBC | IPDF_VERIFY_DST |
                    IPDF_IPSEC);
                switch (error) {
                case 0:
                        break;
```



```
                case EADDRNOTAVAIL:
                        /*
                         * IXAF_VERIFY_SOURCE tells us to pick a better source.
                         * Don't have the application see that errno
                         */
                        error = ENETUNREACH;
                        goto failed;
                case ENETDOWN:
                        /*
                         * Have !ipif_addr_ready address; drop packet silently
                         * until we can get applications to not send until we
                         * are ready.
                         */
                        error = 0;
                        goto failed;
                case EHOSTUNREACH:
                case ENETUNREACH:
                        if (ixa->ixa_ire != NULL) {
                                /*
                                 * Let conn_ip_output/ire_send_noroute return
                                 * the error and send any local ICMP error.
                                 */
                                error = 0;
                                break;
                        }
                        /* FALLTHRU */
                default:
                failed:
                        ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
                        ixa->ixa_cred = connp->conn_cred;       /* Restore */
                        ixa->ixa_cpid = connp->conn_cpid;
                        ixa_refrele(ixa);
                        freemsg(mp);
                        UDPS_BUMP_MIB(us, udpOutErrors);
                        return (error);
                }
        } else {
                /* Done with conn_t */
                mutex_exit(&connp->conn_lock);
        }
        ASSERT(ixa->ixa_ire != NULL);

        /* We're done.  Pass the packet to ip. */
        UDPS_BUMP_MIB(us, udpHCOutDatagrams);

        DTRACE_UDP5(send, mblk_t *, NULL, ip_xmit_attr_t *, ixa,
            void_ip_t *, mp->b_rptr, udp_t *, udp, udpha_t *,
            &mp->b_rptr[ixa->ixa_ip_hdr_length]);

        error = conn_ip_output(mp, ixa);
        /* No udpOutErrors if an error since IP increases its error counter */
        switch (error) {
        case 0:
                break;
        case EWOULDBLOCK:
                (void) ixa_check_drain_insert(connp, ixa);
                error = 0;
                break;
        case EADDRNOTAVAIL:
                /*
                 * IXAF_VERIFY_SOURCE tells us to pick a better source.
                 * Don't have the application see that errno
                 */
                error = ENETUNREACH;
                break;
        }
```

```
3032         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3033         ixa->ixa_cred = connp->conn_cred;       /* Restore */
3034         ixa->ixa_cpid = connp->conn_cpid;
3035         ixa_refrele(ixa);
3036         return (error);
3037 }

3039 /*
3040  * Handle sending an M_DATA to the last destination.
3041  * Handles both IPv4 and IPv6.
3042  *
3043  * NOTE: The caller must hold conn_lock and we drop it here.
3044  */
3045 static int
3046 udp_output_lastdst(conn_t *connp, mblk_t *mp, cred_t *cr, pid_t pid,
3047     ip_xmit_attr_t *ixa)
3048 {
3049         udp_t           *udp = connp->conn_udp;
3050         udp_stack_t     *us = udp->udp_us;
3051         int             error;

3053         ASSERT(MUTEX_HELD(&connp->conn_lock));
3054         ASSERT(ixa != NULL);

3056         ASSERT(cr != NULL);
3057         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3058         ixa->ixa_cred = cr;
3059         ixa->ixa_cpid = pid;

3061         mp = udp_prepend_header_template(connp, ixa, mp, &connp->conn_v6lastsrc,
3062             connp->conn_lastdstport, connp->conn_lastflowinfo, &error);

3064         if (mp == NULL) {
3065                 ASSERT(error != 0);
3066                 mutex_exit(&connp->conn_lock);
3067                 ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3068                 ixa->ixa_cred = connp->conn_cred;       /* Restore */
3069                 ixa->ixa_cpid = connp->conn_cpid;
3070                 ixa_refrele(ixa);
3071                 UDPS_BUMP_MIB(us, udpOutErrors);
3072                 freemsg(mp);
3073                 return (error);
3074         }

3076         /*
3077          * In case we got a safe copy of conn_ixa, or if opt_set made us a new
3078          * safe copy, then we need to fill in any pointers in it.
3079          */
3080         if (ixa->ixa_ire == NULL) {
3081                 in6_addr_t      lastdst, lastsrc;
3082                 in6_addr_t      nexthop;
3083                 in_port_t       lastport;

3085                 lastsrc = connp->conn_v6lastsrc;
3086                 lastdst = connp->conn_v6lastdst;
3087                 lastport = connp->conn_lastdstport;
3088                 ip_attr_nexthop(&connp->conn_xmit_ipp, ixa, &lastdst, &nexthop);
3089                 mutex_exit(&connp->conn_lock);

3091                 error = ip_attr_connect(connp, ixa, &lastsrc, &lastdst,
3092                     &nexthop, lastport, NULL, NULL, IPDF_ALLOW_MCBC |
3093                     IPDF_VERIFY_DST | IPDF_IPSEC);
3094                 switch (error) {
3095                 case 0:
3096                         break;
3097                 case EADDRNOTAVAIL:
```



```
3098                         /*
3099                          * IXAF_VERIFY_SOURCE tells us to pick a better source.
3100                          * Don't have the application see that errno
3101                          */
3102                         error = ENETUNREACH;
3103                         goto failed;
3104                 case ENETDOWN:
3105                         /*
3106                          * Have !ipif_addr_ready address; drop packet silently
3107                          * until we can get applications to not send until we
3108                          * are ready.
3109                          */
3110                         error = 0;
3111                         goto failed;
3112                 case EHOSTUNREACH:
3113                 case ENETUNREACH:
3114                         if (ixa->ixa_ire != NULL) {
3115                                 /*
3116                                  * Let conn_ip_output/ire_send_noroute return
3117                                  * the error and send any local ICMP error.
3118                                  */
3119                                 error = 0;
3120                                 break;
3121                         }
3122                         /* FALLTHRU */
3123                 default:
3124                 failed:
3125                         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3126                         ixa->ixa_cred = connp->conn_cred;       /* Restore */
3127                         ixa->ixa_cpid = connp->conn_cpid;
3128                         ixa_refrele(ixa);
3129                         freemsg(mp);
3130                         UDPS_BUMP_MIB(us, udpOutErrors);
3131                         return (error);
3132                 }
3133         } else {
3134                 /* Done with conn_t */
3135                 mutex_exit(&connp->conn_lock);
3136         }

3138         /* We're done.  Pass the packet to ip. */
3139         UDPS_BUMP_MIB(us, udpHCOutDatagrams);

3141         DTRACE_UDP5(send, mblk_t *, NULL, ip_xmit_attr_t *, ixa,
3142             void_ip_t *, mp->b_rptr, udp_t *, udp, udpha_t *,
3143             &mp->b_rptr[ixa->ixa_ip_hdr_length]);

3145         error = conn_ip_output(mp, ixa);
3146         /* No udpOutErrors if an error since IP increases its error counter */
3147         switch (error) {
3148         case 0:
3149                 break;
3150         case EWOULDBLOCK:
3151                 (void) ixa_check_drain_insert(connp, ixa);
3152                 error = 0;
3153                 break;
3154         case EADDRNOTAVAIL:
3155                 /*
3156                  * IXAF_VERIFY_SOURCE tells us to pick a better source.
3157                  * Don't have the application see that errno
3158                  */
3159                 error = ENETUNREACH;
3160                 /* FALLTHRU */
3161         default:
3162                 mutex_enter(&connp->conn_lock);
3163                 /*
```

```
3164                 * Clear the source and v6lastdst so we call ip_attr_connect
3165                 * for the next packet and try to pick a better source.
3166                 */
3167                if (connp->conn_mcbc_bind)
3168                        connp->conn_saddr_v6 = ipv6_all_zeros;
3169                else
3170                        connp->conn_saddr_v6 = connp->conn_bound_addr_v6;
3171                connp->conn_v6lastdst = ipv6_all_zeros;
3172                mutex_exit(&connp->conn_lock);
3173                break;
3174        }
3175        ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3176        ixa->ixa_cred = connp->conn_cred;       /* Restore */
3177        ixa->ixa_cpid = connp->conn_cpid;
3178        ixa_refrele(ixa);
3179        return (error);
3180 }


3183 /*
3184  * Prepend the header template and then fill in the source and
3185  * flowinfo. The caller needs to handle the destination address since
3186  * it's setting is different if rthdr or source route.
3187  *
3188  * Returns NULL is allocation failed or if the packet would exceed IP_MAXPACKET.
3189  * When it returns NULL it sets errorp.
3190  */
3191 static mblk_t *
3192 udp_prepend_header_template(conn_t *connp, ip_xmit_attr_t *ixa, mblk_t *mp,
3193     const in6_addr_t *v6src, in_port_t dstport, uint32_t flowinfo, int *errorp)
3194 {
3195        udp_t           *udp = connp->conn_udp;
3196        udp_stack_t     *us = udp->udp_us;
3197        boolean_t       insert_spi = udp->udp_nat_t_endpoint;
3198        uint_t          pktlen;
3199        uint_t          alloclen;
3200        uint_t          copylen;
3201        uint8_t         *iph;
3202        uint_t          ip_hdr_length;
3203        udpha_t         *udpha;
3204        uint32_t        cksum;
3205        ip_pkt_t        *ipp;

3207        ASSERT(MUTEX_HELD(&connp->conn_lock));

3209        /*
3210         * Copy the header template and leave space for an SPI
3211         */
3212        copylen = connp->conn_ht_iphc_len;
3213        alloclen = copylen + (insert_spi ? sizeof (uint32_t) : 0);
3214        pktlen = alloclen + msgdsize(mp);
3215        if (pktlen > IP_MAXPACKET) {
3216                freemsg(mp);
3217                *errorp = EMSGSIZE;
3218                return (NULL);
3219        }
3220        ixa->ixa_pktlen = pktlen;

3222        /* check/fix buffer config, setup pointers into it */
3223        iph = mp->b_rptr - alloclen;
3224        if (DB_REF(mp) != 1 || iph < DB_BASE(mp) || !OK_32PTR(iph)) {
3225                mblk_t *mp1;

3227                mp1 = allocb(alloclen + us->us_wroff_extra, BPRI_MED);
3228                if (mp1 == NULL) {
3229                        freemsg(mp);
```



```
3230                        *errorp = ENOMEM;
3231                        return (NULL);
3232                }
3233                mp1->b_wptr = DB_LIM(mp1);
3234                mp1->b_cont = mp;
3235                mp = mp1;
3236                iph = (mp->b_wptr - alloclen);
3237        }
3238        mp->b_rptr = iph;
3239        bcopy(connp->conn_ht_iphc, iph, copylen);
3240        ip_hdr_length = (uint_t)(connp->conn_ht_ulp - connp->conn_ht_iphc);

3242        ixa->ixa_ip_hdr_length = ip_hdr_length;
3243        udpha = (udpha_t *)(iph + ip_hdr_length);

3245        /*
3246         * Setup header length and prepare for ULP checksum done in IP.
3247         * udp_build_hdr_template has already massaged any routing header
3248         * and placed the result in conn_sum.
3249         *
3250         * We make it easy for IP to include our pseudo header
3251         * by putting our length in uha_checksum.
3252         */
3253        cksum = pktlen - ip_hdr_length;
3254        udpha->uha_length = htons(cksum);

3256        cksum += connp->conn_sum;
3257        cksum = (cksum >> 16) + (cksum & 0xFFFF);
3258        ASSERT(cksum < 0x10000);

3260        ipp = &connp->conn_xmit_ipp;
3261        if (ixa->ixa_flags & IXAF_IS_IPV4) {
3262                ipha_t  *ipha = (ipha_t *)iph;

3264                ipha->ipha_length = htons((uint16_t)pktlen);

3266                /* IP does the checksum if uha_checksum is non-zero */
3267                if (us->us_do_checksum)
3268                        udpha->uha_checksum = htons(cksum);

3270                /* if IP_PKTINFO specified an addres it wins over bind() */
3271                if ((ipp->ipp_fields & IPPF_ADDR) &&
3272                    IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr)) {
3273                        ASSERT(ipp->ipp_addr_v4 != INADDR_ANY);
3274                        ipha->ipha_src = ipp->ipp_addr_v4;
3275                } else {
3276                        IN6_V4MAPPED_TO_IPADDR(v6src, ipha->ipha_src);
3277                }
3278        } else {
3279                ip6_t *ip6h = (ip6_t *)iph;

3281                ip6h->ip6_plen =  htons((uint16_t)(pktlen - IPV6_HDR_LEN));
3282                udpha->uha_checksum = htons(cksum);

3284                /* if IP_PKTINFO specified an addres it wins over bind() */
3285                if ((ipp->ipp_fields & IPPF_ADDR) &&
3286                    !IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr)) {
3287                        ASSERT(!IN6_IS_ADDR_UNSPECIFIED(&ipp->ipp_addr));
3288                        ip6h->ip6_src = ipp->ipp_addr;
3289                } else {
3290                        ip6h->ip6_src = *v6src;
3291                }
3292                ip6h->ip6_vcf =
3293                    (IPV6_DEFAULT_VERS_AND_FLOW & IPV6_VERS_AND_FLOW_MASK) |
3294                    (flowinfo & ~IPV6_VERS_AND_FLOW_MASK);
3295                if (ipp->ipp_fields & IPPF_TCLASS) {
```

```
3296                        /* Overrides the class part of flowinfo */
3297                        ip6h->ip6_vcf = IPV6_TCLASS_FLOW(ip6h->ip6_vcf,
3298                            ipp->ipp_tclass);
3299                }
3300        }

3302        /* Insert all-0s SPI now. */
3303        if (insert_spi)
3304                *((uint32_t *)(udpha + 1)) = 0;

3306        udpha->uha_dst_port = dstport;
3307        return (mp);
3308 }

3310 /*
3311  * Send a T_UDERR_IND in response to an M_DATA
3312  */
3313 static void
3314 udp_ud_err_connected(conn_t *connp, t_scalar_t error)
3315 {
3316        struct sockaddr_storage ss;
3317        sin_t           *sin;
3318        sin6_t          *sin6;
3319        struct sockaddr *addr;
3320        socklen_t       addrlen;
3321        mblk_t          *mp1;

3323        mutex_enter(&connp->conn_lock);
3324        /* Initialize addr and addrlen as if they're passed in */
3325        if (connp->conn_family == AF_INET) {
3326                sin = (sin_t *)&ss;
3327                *sin = sin_null;
3328                sin->sin_family = AF_INET;
3329                sin->sin_port = connp->conn_fport;
3330                sin->sin_addr.s_addr = connp->conn_faddr_v4;
3331                addr = (struct sockaddr *)sin;
3332                addrlen = sizeof (*sin);
3333        } else {
3334                sin6 = (sin6_t *)&ss;
3335                *sin6 = sin6_null;
3336                sin6->sin6_family = AF_INET6;
3337                sin6->sin6_port = connp->conn_fport;
3338                sin6->sin6_flowinfo = connp->conn_flowinfo;
3339                sin6->sin6_addr = connp->conn_faddr_v6;
3340                if (IN6_IS_ADDR_LINKSCOPE(&connp->conn_faddr_v6) &&
3341                    (connp->conn_ixa->ixa_flags & IXAF_SCOPEID_SET)) {
3342                        sin6->sin6_scope_id = connp->conn_ixa->ixa_scopeid;
3343                } else {
3344                        sin6->sin6_scope_id = 0;
3345                }
3346                sin6->__sin6_src_id = 0;
3347                addr = (struct sockaddr *)sin6;
3348                addrlen = sizeof (*sin6);
3349        }
3350        mutex_exit(&connp->conn_lock);

3352        mp1 = mi_tpi_uderror_ind((char *)addr, addrlen, NULL, 0, error);
3353        if (mp1 != NULL)
3354                putnext(connp->conn_rq, mp1);
3355 }

3357 /*
3358  * This routine handles all messages passed downstream.  It either
3359  * consumes the message or passes it downstream; it never queues a
3360  * a message.
3361  *
```



```
3362  * Also entry point for sockfs when udp is in "direct sockfs" mode.  This mode
3363  * is valid when we are directly beneath the stream head, and thus sockfs
3364  * is able to bypass STREAMS and directly call us, passing along the sockaddr
3365  * structure without the cumbersome T_UNITDATA_REQ interface for the case of
3366  * connected endpoints.
3367  */
3368 void
3369 udp_wput(queue_t *q, mblk_t *mp)
3370 {
3371        sin6_t          *sin6;
3372        sin_t           *sin = NULL;
3373        uint_t          srcid;
3374        conn_t          *connp = Q_TO_CONN(q);
3375        udp_t           *udp = connp->conn_udp;
3376        int             error = 0;
3377        struct sockaddr *addr = NULL;
3378        socklen_t       addrlen;
3379        udp_stack_t     *us = udp->udp_us;
3380        struct T_unitdata_req *tudr;
3381        mblk_t          *data_mp;
3382        ushort_t        ipversion;
3383        cred_t          *cr;
3384        pid_t           pid;

3386        /*
3387         * We directly handle several cases here: T_UNITDATA_REQ message
3388         * coming down as M_PROTO/M_PCPROTO and M_DATA messages for connected
3389         * socket.
3390         */
3391        switch (DB_TYPE(mp)) {
3392        case M_DATA:
3393                if (!udp->udp_issocket || udp->udp_state != TS_DATA_XFER) {
3394                        /* Not connected; address is required */
3395                        UDPS_BUMP_MIB(us, udpOutErrors);
3396                        UDP_DBGSTAT(us, udp_data_notconn);
3397                        UDP_STAT(us, udp_out_err_notconn);
3398                        freemsg(mp);
3399                        return;
3400                }
3401                /*
3402                 * All Solaris components should pass a db_credp
3403                 * for this message, hence we ASSERT.
3404                 * On production kernels we return an error to be robust against
3405                 * random streams modules sitting on top of us.
3406                 */
3407                cr = msg_getcred(mp, &pid);
3408                ASSERT(cr != NULL);
3409                if (cr == NULL) {
3410                        UDPS_BUMP_MIB(us, udpOutErrors);
3411                        freemsg(mp);
3412                        return;
3413                }
3414                ASSERT(udp->udp_issocket);
3415                UDP_DBGSTAT(us, udp_data_conn);
3416                error = udp_output_connected(connp, mp, cr, pid);
3417                if (error != 0) {
3418                        UDP_STAT(us, udp_out_err_output);
3419                        if (connp->conn_rq != NULL)
3420                                udp_ud_err_connected(connp, (t_scalar_t)error);
3421 #ifdef DEBUG
3422                        printf("udp_output_connected returned %d\n", error);
3423 #endif
3424                }
3425                return;

3427        case M_PROTO:
```

```
3428         case M_PCPROTO:
3429                 tudr = (struct T_unitdata_req *)mp->b_rptr;
3430                 if (MBLKL(mp) < sizeof (*tudr) ||
3431                     ((t_primp_t)mp->b_rptr)->type != T_UNITDATA_REQ) {
3432                         udp_wput_other(q, mp);
3433                         return;
3434                 }
3435                 break;

3437         default:
3438                 udp_wput_other(q, mp);
3439                 return;
3440         }

3442         /* Handle valid T_UNITDATA_REQ here */
3443         data_mp = mp->b_cont;
3444         if (data_mp == NULL) {
3445                 error = EPROTO;
3446                 goto ud_error2;
3447         }
3448         mp->b_cont = NULL;

3450         if (!MBLKIN(mp, 0, tudr->DEST_offset + tudr->DEST_length)) {
3451                 error = EADDRNOTAVAIL;
3452                 goto ud_error2;
3453         }

3455         /*
3456          * All Solaris components should pass a db_credp
3457          * for this TPI message, hence we should ASSERT.
3458          * However, RPC (svc_clts_ksend) does this odd thing where it
3459          * passes the options from a T_UNITDATA_IND unchanged in a
3460          * T_UNITDATA_REQ. While that is the right thing to do for
3461          * some options, SCM_UCRED being the key one, this also makes it
3462          * pass down IP_RECVDSTADDR. Hence we can't ASSERT here.
3463          */
3464         cr = msg_getcred(mp, &pid);
3465         if (cr == NULL) {
3466                 cr = connp->conn_cred;
3467                 pid = connp->conn_cpid;
3468         }

3470         /*
3471          * If a port has not been bound to the stream, fail.
3472          * This is not a problem when sockfs is directly
3473          * above us, because it will ensure that the socket
3474          * is first bound before allowing data to be sent.
3475          */
3476         if (udp->udp_state == TS_UNBND) {
3477                 error = EPROTO;
3478                 goto ud_error2;
3479         }
3480         addr = (struct sockaddr *)&mp->b_rptr[tudr->DEST_offset];
3481         addrlen = tudr->DEST_length;

3483         switch (connp->conn_family) {
3484         case AF_INET6:
3485                 sin6 = (sin6_t *)addr;
3486                 if (!OK_32PTR((char *)sin6) || (addrlen != sizeof (sin6_t)) ||
3487                     (sin6->sin6_family != AF_INET6)) {
3488                         error = EADDRNOTAVAIL;
3489                         goto ud_error2;
3490                 }

3492                 srcid = sin6->__sin6_src_id;
3493                 if (!IN6_IS_ADDR_V4MAPPED(&sin6->sin6_addr)) {
```



```
3494                         /*
3495                          * Destination is a non-IPv4-compatible IPv6 address.
3496                          * Send out an IPv6 format packet.
3497                          */

3499                         /*
3500                          * If the local address is a mapped address return
3501                          * an error.
3502                          * It would be possible to send an IPv6 packet but the
3503                          * response would never make it back to the application
3504                          * since it is bound to a mapped address.
3505                          */
3506                         if (IN6_IS_ADDR_V4MAPPED(&connp->conn_saddr_v6)) {
3507                                 error = EADDRNOTAVAIL;
3508                                 goto ud_error2;
3509                         }

3511                         UDP_DBGSTAT(us, udp_out_ipv6);

3513                         if (IN6_IS_ADDR_UNSPECIFIED(&sin6->sin6_addr))
3514                                 sin6->sin6_addr = ipv6_loopback;
3515                         ipversion = IPV6_VERSION;
3516                 } else {
3517                         if (connp->conn_ipv6_v6only) {
3518                                 error = EADDRNOTAVAIL;
3519                                 goto ud_error2;
3520                         }

3522                         /*
3523                          * If the local address is not zero or a mapped address
3524                          * return an error.  It would be possible to send an
3525                          * IPv4 packet but the response would never make it
3526                          * back to the application since it is bound to a
3527                          * non-mapped address.
3528                          */
3529                         if (!IN6_IS_ADDR_V4MAPPED(&connp->conn_saddr_v6) &&
3530                             !IN6_IS_ADDR_UNSPECIFIED(&connp->conn_saddr_v6)) {
3531                                 error = EADDRNOTAVAIL;
3532                                 goto ud_error2;
3533                         }
3534                         UDP_DBGSTAT(us, udp_out_mapped);

3536                         if (V4_PART_OF_V6(sin6->sin6_addr) == INADDR_ANY) {
3537                                 V4_PART_OF_V6(sin6->sin6_addr) =
3538                                     htonl(INADDR_LOOPBACK);
3539                         }
3540                         ipversion = IPV4_VERSION;
3541                 }

3543                 if (tudr->OPT_length != 0) {
3544                         /*
3545                          * If we are connected then the destination needs to be
3546                          * the same as the connected one.
3547                          */
3548                         if (udp->udp_state == TS_DATA_XFER &&
3549                             !conn_same_as_last_v6(connp, sin6)) {
3550                                 error = EISCONN;
3551                                 goto ud_error2;
3552                         }
3553                         UDP_STAT(us, udp_out_opt);
3554                         error = udp_output_ancillary(connp, NULL, sin6,
3555                             data_mp, mp, NULL, cr, pid);
3556                 } else {
3557                         ip_xmit_attr_t *ixa;

3559                         /*
```

```
3560                         * We have to allocate an ip_xmit_attr_t before we grab
3561                         * conn_lock and we need to hold conn_lock once we've
3562                         * checked conn_same_as_last_v6 to handle concurrent
3563                         * send* calls on a socket.
3564                         */
3565                        ixa = conn_get_ixa(connp, B_FALSE);
3566                        if (ixa == NULL) {
3567                                error = ENOMEM;
3568                                goto ud_error2;
3569                        }
3570                        mutex_enter(&connp->conn_lock);

3572                        if (conn_same_as_last_v6(connp, sin6) &&
3573                            connp->conn_lastsrcid == srcid &&
3574                            ipsec_outbound_policy_current(ixa)) {
3575                                UDP_DBGSTAT(us, udp_out_lastdst);
3576                                /* udp_output_lastdst drops conn_lock */
3577                                error = udp_output_lastdst(connp, data_mp, cr,
3578                                    pid, ixa);
3579                        } else {
3580                                UDP_DBGSTAT(us, udp_out_diffdst);
3581                                /* udp_output_newdst drops conn_lock */
3582                                error = udp_output_newdst(connp, data_mp, NULL,
3583                                    sin6, ipversion, cr, pid, ixa);
3584                        }
3585                        ASSERT(MUTEX_NOT_HELD(&connp->conn_lock));
3586                }
3587                if (error == 0) {
3588                        freeb(mp);
3589                        return;
3590                }
3591                break;

3593        case AF_INET:
3594                sin = (sin_t *)addr;
3595                if ((!OK_32PTR((char *)sin) || addrlen != sizeof (sin_t)) ||
3596                    (sin->sin_family != AF_INET)) {
3597                        error = EADDRNOTAVAIL;
3598                        goto ud_error2;
3599                }
3600                UDP_DBGSTAT(us, udp_out_ipv4);
3601                if (sin->sin_addr.s_addr == INADDR_ANY)
3602                        sin->sin_addr.s_addr = htonl(INADDR_LOOPBACK);
3603                ipversion = IPV4_VERSION;

3605                srcid = 0;
3606                if (tudr->OPT_length != 0) {
3607                        /*
3608                         * If we are connected then the destination needs to be
3609                         * the same as the connected one.
3610                         */
3611                        if (udp->udp_state == TS_DATA_XFER &&
3612                            !conn_same_as_last_v4(connp, sin)) {
3613                                error = EISCONN;
3614                                goto ud_error2;
3615                        }
3616                        UDP_STAT(us, udp_out_opt);
3617                        error = udp_output_ancillary(connp, sin, NULL,
3618                            data_mp, mp, NULL, cr, pid);
3619                } else {
3620                        ip_xmit_attr_t *ixa;

3622                        /*
3623                         * We have to allocate an ip_xmit_attr_t before we grab
3624                         * conn_lock and we need to hold conn_lock once we've
3625                         * checked conn_same_as_last_v4 to handle concurrent
```



```
3626                         * send* calls on a socket.
3627                         */
3628                        ixa = conn_get_ixa(connp, B_FALSE);
3629                        if (ixa == NULL) {
3630                                error = ENOMEM;
3631                                goto ud_error2;
3632                        }
3633                        mutex_enter(&connp->conn_lock);

3635                        if (conn_same_as_last_v4(connp, sin) &&
3636                            ipsec_outbound_policy_current(ixa)) {
3637                                UDP_DBGSTAT(us, udp_out_lastdst);
3638                                /* udp_output_lastdst drops conn_lock */
3639                                error = udp_output_lastdst(connp, data_mp, cr,
3640                                    pid, ixa);
3641                        } else {
3642                                UDP_DBGSTAT(us, udp_out_diffdst);
3643                                /* udp_output_newdst drops conn_lock */
3644                                error = udp_output_newdst(connp, data_mp, sin,
3645                                    NULL, ipversion, cr, pid, ixa);
3646                        }
3647                        ASSERT(MUTEX_NOT_HELD(&connp->conn_lock));
3648                }
3649                if (error == 0) {
3650                        freeb(mp);
3651                        return;
3652                }
3653                break;
3654        }
3655        UDP_STAT(us, udp_out_err_output);
3656        ASSERT(mp != NULL);
3657        /* mp is freed by the following routine */
3658        udp_ud_err(q, mp, (t_scalar_t)error);
3659        return;

3661 ud_error2:
3662        UDPS_BUMP_MIB(us, udpOutErrors);
3663        freemsg(data_mp);
3664        UDP_STAT(us, udp_out_err_output);
3665        ASSERT(mp != NULL);
3666        /* mp is freed by the following routine */
3667        udp_ud_err(q, mp, (t_scalar_t)error);
3668 }

3670 /*
3671  * Handle the case of the IP address, port, flow label being different
3672  * for both IPv4 and IPv6.
3673  *
3674  * NOTE: The caller must hold conn_lock and we drop it here.
3675  */
3676 static int
3677 udp_output_newdst(conn_t *connp, mblk_t *data_mp, sin_t *sin, sin6_t *sin6,
3678     ushort_t ipversion, cred_t *cr, pid_t pid, ip_xmit_attr_t *ixa)
3679 {
3680        uint_t          srcid;
3681        uint32_t        flowinfo;
3682        udp_t           *udp = connp->conn_udp;
3683        int             error = 0;
3684        ip_xmit_attr_t  *oldixa;
3685        udp_stack_t     *us = udp->udp_us;
3686        in6_addr_t      v6src;
3687        in6_addr_t      v6dst;
3688        in6_addr_t      v6nexthop;
3689        in_port_t       dstport;

3691        ASSERT(MUTEX_HELD(&connp->conn_lock));
```

```
3692         ASSERT(ixa != NULL);
3693         /*
3694          * We hold conn_lock across all the use and modifications of
3695          * the conn_lastdst, conn_ixa, and conn_xmit_ipp to ensure that they
3696          * stay consistent.
3697          */

3699         ASSERT(cr != NULL);
3700         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3701         ixa->ixa_cred = cr;
3702         ixa->ixa_cpid = pid;
3703         if (is_system_labeled()) {
3704                 /* We need to restart with a label based on the cred */
3705                 ip_xmit_attr_restore_tsl(ixa, ixa->ixa_cred);
3706         }

3708         /*
3709          * If we are connected then the destination needs to be the
3710          * same as the connected one, which is not the case here since we
3711          * checked for that above.
3712          */
3713         if (udp->udp_state == TS_DATA_XFER) {
3714                 mutex_exit(&connp->conn_lock);
3715                 error = EISCONN;
3716                 goto ud_error;
3717         }

3719         /* In case previous destination was multicast or multirt */
3720         ip_attr_newdst(ixa);

3722         /*
3723          * If laddr is unspecified then we look at sin6_src_id.
3724          * We will give precedence to a source address set with IPV6_PKTINFO
3725          * (aka IPPF_ADDR) but that is handled in build_hdrs. However, we don't
3726          * want ip_attr_connect to select a source (since it can fail) when
3727          * IPV6_PKTINFO is specified.
3728          * If this doesn't result in a source address then we get a source
3729          * from ip_attr_connect() below.
3730          */
3731         v6src = connp->conn_saddr_v6;
3732         if (sin != NULL) {
3733                 IN6_IPADDR_TO_V4MAPPED(sin->sin_addr.s_addr, &v6dst);
3734                 dstport = sin->sin_port;
3735                 flowinfo = 0;
3736                 /* Don't bother with ip_srcid_find_id(), but indicate anyway. */
3737                 srcid = 0;
3738                 ixa->ixa_flags &= ~IXAF_SCOPEID_SET;
3739                 ixa->ixa_flags |= IXAF_IS_IPV4;
3740         } else {
3741                 boolean_t v4mapped;

3743                 v6dst = sin6->sin6_addr;
3744                 dstport = sin6->sin6_port;
3745                 flowinfo = sin6->sin6_flowinfo;
3746                 srcid = sin6->__sin6_src_id;
3747                 if (IN6_IS_ADDR_LINKSCOPE(&v6dst) && sin6->sin6_scope_id != 0) {
3748                         ixa->ixa_scopeid = sin6->sin6_scope_id;
3749                         ixa->ixa_flags |= IXAF_SCOPEID_SET;
3750                 } else {
3751                         ixa->ixa_flags &= ~IXAF_SCOPEID_SET;
3752                 }
3753                 v4mapped = IN6_IS_ADDR_V4MAPPED(&v6dst);
3754                 if (v4mapped)
3755                         ixa->ixa_flags |= IXAF_IS_IPV4;
3756                 else
3757                         ixa->ixa_flags &= ~IXAF_IS_IPV4;
```



```
3758                 if (srcid != 0 && IN6_IS_ADDR_UNSPECIFIED(&v6src)) {
3759                         if (!ip_srcid_find_id(srcid, &v6src, IPCL_ZONEID(connp),
3760                             v4mapped, connp->conn_netstack)) {
3761                                 /* Mismatched v4mapped/v6 specified by srcid. */
3762                                 mutex_exit(&connp->conn_lock);
3763                                 error = EADDRNOTAVAIL;
3764                                 goto ud_error;
3765                         }
3766                 }
3767         }
3768         /* Handle IP_PKTINFO/IPV6_PKTINFO setting source address. */
3769         if (connp->conn_xmit_ipp.ipp_fields & IPPF_ADDR) {
3770                 ip_pkt_t *ipp = &connp->conn_xmit_ipp;

3772                 if (ixa->ixa_flags & IXAF_IS_IPV4) {
3773                         if (IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
3774                                 v6src = ipp->ipp_addr;
3775                 } else {
3776                         if (!IN6_IS_ADDR_V4MAPPED(&ipp->ipp_addr))
3777                                 v6src = ipp->ipp_addr;
3778                 }
3779         }

3781         ip_attr_nexthop(&connp->conn_xmit_ipp, ixa, &v6dst, &v6nexthop);
3782         mutex_exit(&connp->conn_lock);

3784         error = ip_attr_connect(connp, ixa, &v6src, &v6dst, &v6nexthop, dstport,
3785             &v6src, NULL, IPDF_ALLOW_MCBC | IPDF_VERIFY_DST | IPDF_IPSEC);
3786         switch (error) {
3787         case 0:
3788                 break;
3789         case EADDRNOTAVAIL:
3790                 /*
3791                  * IXAF_VERIFY_SOURCE tells us to pick a better source.
3792                  * Don't have the application see that errno
3793                  */
3794                 error = ENETUNREACH;
3795                 goto failed;
3796         case ENETDOWN:
3797                 /*
3798                  * Have !ipif_addr_ready address; drop packet silently
3799                  * until we can get applications to not send until we
3800                  * are ready.
3801                  */
3802                 error = 0;
3803                 goto failed;
3804         case EHOSTUNREACH:
3805         case ENETUNREACH:
3806                 if (ixa->ixa_ire != NULL) {
3807                         /*
3808                          * Let conn_ip_output/ire_send_noroute return
3809                          * the error and send any local ICMP error.
3810                          */
3811                         error = 0;
3812                         break;
3813                 }
3814                 /* FALLTHRU */
3815         failed:
3816         default:
3817                 goto ud_error;
3818         }


3821         /*
3822          * Cluster note: we let the cluster hook know that we are sending to a
3823          * new address and/or port.
```

```
3824          */
3825         if (cl_inet_connect2 != NULL) {
3826                 CL_INET_UDP_CONNECT(connp, B_TRUE, &v6dst, dstport, error);
3827                 if (error != 0) {
3828                         error = EHOSTUNREACH;
3829                         goto ud_error;
3830                 }
3831         }

3833         mutex_enter(&connp->conn_lock);
3834         /*
3835          * While we dropped the lock some other thread might have connected
3836          * this socket. If so we bail out with EISCONN to ensure that the
3837          * connecting thread is the one that updates conn_ixa, conn_ht_*
3838          * and conn_*last*.
3839          */
3840         if (udp->udp_state == TS_DATA_XFER) {
3841                 mutex_exit(&connp->conn_lock);
3842                 error = EISCONN;
3843                 goto ud_error;
3844         }

3846         /*
3847          * We need to rebuild the headers if
3848          *  - we are labeling packets (could be different for different
3849          *    destinations)
3850          *  - we have a source route (or routing header) since we need to
3851          *    massage that to get the pseudo-header checksum
3852          *  - the IP version is different than the last time
3853          *  - a socket option with COA_HEADER_CHANGED has been set which
3854          *    set conn_v6lastdst to zero.
3855          *
3856          * Otherwise the prepend function will just update the src, dst,
3857          * dstport, and flow label.
3858          */
3859         if (is_system_labeled()) {
3860                 /* TX MLP requires SCM_UCRED and don't have that here */
3861                 if (connp->conn_mlp_type != mlptSingle) {
3862                         mutex_exit(&connp->conn_lock);
3863                         error = ECONNREFUSED;
3864                         goto ud_error;
3865                 }
3866                 /*
3867                  * Check whether Trusted Solaris policy allows communication
3868                  * with this host, and pretend that the destination is
3869                  * unreachable if not.
3870                  * Compute any needed label and place it in ipp_label_v4/v6.
3871                  *
3872                  * Later conn_build_hdr_template/conn_prepend_hdr takes
3873                  * ipp_label_v4/v6 to form the packet.
3874                  *
3875                  * Tsol note: Since we hold conn_lock we know no other
3876                  * thread manipulates conn_xmit_ipp.
3877                  */
3878                 error = conn_update_label(connp, ixa, &v6dst,
3879                     &connp->conn_xmit_ipp);
3880                 if (error != 0) {
3881                         mutex_exit(&connp->conn_lock);
3882                         goto ud_error;
3883                 }
3884                 /* Rebuild the header template */
3885                 error = udp_build_hdr_template(connp, &v6src, &v6dst, dstport,
3886                     flowinfo);
3887                 if (error != 0) {
3888                         mutex_exit(&connp->conn_lock);
3889                         goto ud_error;
```



```
3890                 }
3891         } else if ((connp->conn_xmit_ipp.ipp_fields &
3892             (IPPF_IPV4_OPTIONS|IPPF_RTHDR)) ||
3893             ipversion != connp->conn_lastipversion ||
3894             IN6_IS_ADDR_UNSPECIFIED(&connp->conn_v6lastdst)) {
3895                 /* Rebuild the header template */
3896                 error = udp_build_hdr_template(connp, &v6src, &v6dst, dstport,
3897                     flowinfo);
3898                 if (error != 0) {
3899                         mutex_exit(&connp->conn_lock);
3900                         goto ud_error;
3901                 }
3902         } else {
3903                 /* Simply update the destination address if no source route */
3904                 if (ixa->ixa_flags & IXAF_IS_IPV4) {
3905                         ipha_t  *ipha = (ipha_t *)connp->conn_ht_iphc;

3907                         IN6_V4MAPPED_TO_IPADDR(&v6dst, ipha->ipha_dst);
3908                         if (ixa->ixa_flags & IXAF_PMTU_IPV4_DF) {
3909                                 ipha->ipha_fragment_offset_and_flags |=
3910                                     IPH_DF_HTONS;
3911                         } else {
3912                                 ipha->ipha_fragment_offset_and_flags &=
3913                                     ~IPH_DF_HTONS;
3914                         }
3915                 } else {
3916                         ip6_t *ip6h = (ip6_t *)connp->conn_ht_iphc;
3917                         ip6h->ip6_dst = v6dst;
3918                 }
3919         }

3921         /*
3922          * Remember the dst/dstport etc which corresponds to the built header
3923          * template and conn_ixa.
3924          */
3925         oldixa = conn_replace_ixa(connp, ixa);
3926         connp->conn_v6lastdst = v6dst;
3927         connp->conn_lastipversion = ipversion;
3928         connp->conn_lastdstport = dstport;
3929         connp->conn_lastflowinfo = flowinfo;
3930         connp->conn_lastscopeid = ixa->ixa_scopeid;
3931         connp->conn_lastsrcid = srcid;
3932         /* Also remember a source to use together with lastdst */
3933         connp->conn_v6lastsrc = v6src;

3935         data_mp = udp_prepend_header_template(connp, ixa, data_mp, &v6src,
3936             dstport, flowinfo, &error);

3938         /* Done with conn_t */
3939         mutex_exit(&connp->conn_lock);
3940         ixa_refrele(oldixa);

3942         if (data_mp == NULL) {
3943                 ASSERT(error != 0);
3944                 goto ud_error;
3945         }

3947         /* We're done.  Pass the packet to ip. */
3948         UDPS_BUMP_MIB(us, udpHCOutDatagrams);

3950         DTRACE_UDP5(send, mblk_t *, NULL, ip_xmit_attr_t *, ixa,
3951             void_ip_t *, data_mp->b_rptr, udp_t *, udp, udpha_t *,
3952             &data_mp->b_rptr[ixa->ixa_ip_hdr_length]);

3954         error = conn_ip_output(data_mp, ixa);
3955         /* No udpOutErrors if an error since IP increases its error counter */
```

```
3956         switch (error) {
3957         case 0:
3958                 break;
3959         case EWOULDBLOCK:
3960                 (void) ixa_check_drain_insert(connp, ixa);
3961                 error = 0;
3962                 break;
3963         case EADDRNOTAVAIL:
3964                 /*
3965                  * IXAF_VERIFY_SOURCE tells us to pick a better source.
3966                  * Don't have the application see that errno
3967                  */
3968                 error = ENETUNREACH;
3969                 /* FALLTHRU */
3970         default:
3971                 mutex_enter(&connp->conn_lock);
3972                 /*
3973                  * Clear the source and v6lastdst so we call ip_attr_connect
3974                  * for the next packet and try to pick a better source.
3975                  */
3976                 if (connp->conn_mcbc_bind)
3977                         connp->conn_saddr_v6 = ipv6_all_zeros;
3978                 else
3979                         connp->conn_saddr_v6 = connp->conn_bound_addr_v6;
3980                 connp->conn_v6lastdst = ipv6_all_zeros;
3981                 mutex_exit(&connp->conn_lock);
3982                 break;
3983         }
3984         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3985         ixa->ixa_cred = connp->conn_cred;       /* Restore */
3986         ixa->ixa_cpid = connp->conn_cpid;
3987         ixa_refrele(ixa);
3988         return (error);

3990 ud_error:
3991         ASSERT(!(ixa->ixa_free_flags & IXA_FREE_CRED));
3992         ixa->ixa_cred = connp->conn_cred;       /* Restore */
3993         ixa->ixa_cpid = connp->conn_cpid;
3994         ixa_refrele(ixa);

3996         freemsg(data_mp);
3997         UDPS_BUMP_MIB(us, udpOutErrors);
3998         UDP_STAT(us, udp_out_err_output);
3999         return (error);
4000 }

4002 /* ARGSUSED */
4003 static void
4004 udp_wput_fallback(queue_t *wq, mblk_t *mp)
4005 {
4006 #ifdef DEBUG
4007         cmn_err(CE_CONT, "udp_wput_fallback: Message in fallback \n");
4008 #endif
4009         freemsg(mp);
4010 }


4013 /*
4014  * Handle special out-of-band ioctl requests (see PSARC/2008/265).
4015  */
4016 static void
4017 udp_wput_cmdblk(queue_t *q, mblk_t *mp)
4018 {
4019         void    *data;
4020         mblk_t  *datamp = mp->b_cont;
4021         conn_t  *connp = Q_TO_CONN(q);
```



```
4022         udp_t   *udp = connp->conn_udp;
4023         cmdblk_t *cmdp = (cmdblk_t *)mp->b_rptr;

4025         if (datamp == NULL || MBLKL(datamp) < cmdp->cb_len) {
4026                 cmdp->cb_error = EPROTO;
4027                 qreply(q, mp);
4028                 return;
4029         }
4030         data = datamp->b_rptr;

4032         mutex_enter(&connp->conn_lock);
4033         switch (cmdp->cb_cmd) {
4034         case TI_GETPEERNAME:
4035                 if (udp->udp_state != TS_DATA_XFER)
4036                         cmdp->cb_error = ENOTCONN;
4037                 else
4038                         cmdp->cb_error = conn_getpeername(connp, data,
4039                             &cmdp->cb_len);
4040                 break;
4041         case TI_GETMYNAME:
4042                 cmdp->cb_error = conn_getsockname(connp, data, &cmdp->cb_len);
4043                 break;
4044         default:
4045                 cmdp->cb_error = EINVAL;
4046                 break;
4047         }
4048         mutex_exit(&connp->conn_lock);

4050         qreply(q, mp);
4051 }

4053 static void
4054 udp_use_pure_tpi(udp_t *udp)
4055 {
4056         conn_t  *connp = udp->udp_connp;

4058         mutex_enter(&connp->conn_lock);
4059         udp->udp_issocket = B_FALSE;
4060         mutex_exit(&connp->conn_lock);
4061         UDP_STAT(udp->udp_us, udp_sock_fallback);
4062 }

4064 static void
4065 udp_wput_other(queue_t *q, mblk_t *mp)
4066 {
4067         uchar_t *rptr = mp->b_rptr;
4068         struct iocblk *iocp;
4069         conn_t  *connp = Q_TO_CONN(q);
4070         udp_t   *udp = connp->conn_udp;
4071         cred_t  *cr;

4073         switch (mp->b_datap->db_type) {
4074         case M_CMD:
4075                 udp_wput_cmdblk(q, mp);
4076                 return;

4078         case M_PROTO:
4079         case M_PCPROTO:
4080                 if (mp->b_wptr - rptr < sizeof (t_scalar_t)) {
4081                         /*
4082                          * If the message does not contain a PRIM_type,
4083                          * throw it away.
4084                          */
4085                         freemsg(mp);
4086                         return;
4087                 }
```

```
4088                switch (((t_primp_t)rptr)->type) {
4089                case T_ADDR_REQ:
4090                        udp_addr_req(q, mp);
4091                        return;
4092                case O_T_BIND_REQ:
4093                case T_BIND_REQ:
4094                        udp_tpi_bind(q, mp);
4095                        return;
4096                case T_CONN_REQ:
4097                        udp_tpi_connect(q, mp);
4098                        return;
4099                case T_CAPABILITY_REQ:
4100                        udp_capability_req(q, mp);
4101                        return;
4102                case T_INFO_REQ:
4103                        udp_info_req(q, mp);
4104                        return;
4105                case T_UNITDATA_REQ:
4106                        /*
4107                         * If a T_UNITDATA_REQ gets here, the address must
4108                         * be bad.  Valid T_UNITDATA_REQs are handled
4109                         * in udp_wput.
4110                         */
4111                        udp_ud_err(q, mp, EADDRNOTAVAIL);
4112                        return;
4113                case T_UNBIND_REQ:
4114                        udp_tpi_unbind(q, mp);
4115                        return;
4116                case T_SVR4_OPTMGMT_REQ:
4117                        /*
4118                         * All Solaris components should pass a db_credp
4119                         * for this TPI message, hence we ASSERT.
4120                         * But in case there is some other M_PROTO that looks
4121                         * like a TPI message sent by some other kernel
4122                         * component, we check and return an error.
4123                         */
4124                        cr = msg_getcred(mp, NULL);
4125                        ASSERT(cr != NULL);
4126                        if (cr == NULL) {
4127                                udp_err_ack(q, mp, TSYSERR, EINVAL);
4128                                return;
4129                        }
4130                        if (!snmpcom_req(q, mp, udp_snmp_set, ip_snmp_get,
4131                            cr)) {
4132                                svr4_optcom_req(q, mp, cr, &udp_opt_obj);
4133                        }
4134                        return;

4136                case T_OPTMGMT_REQ:
4137                        /*
4138                         * All Solaris components should pass a db_credp
4139                         * for this TPI message, hence we ASSERT.
4140                         * But in case there is some other M_PROTO that looks
4141                         * like a TPI message sent by some other kernel
4142                         * component, we check and return an error.
4143                         */
4144                        cr = msg_getcred(mp, NULL);
4145                        ASSERT(cr != NULL);
4146                        if (cr == NULL) {
4147                                udp_err_ack(q, mp, TSYSERR, EINVAL);
4148                                return;
4149                        }
4150                        tpi_optcom_req(q, mp, cr, &udp_opt_obj);
4151                        return;

4153                case T_DISCON_REQ:
```



```
4154                        udp_tpi_disconnect(q, mp);
4155                        return;

4157                /* The following TPI message is not supported by udp. */
4158                case O_T_CONN_RES:
4159                case T_CONN_RES:
4160                        udp_err_ack(q, mp, TNOTSUPPORT, 0);
4161                        return;

4163                /* The following 3 TPI requests are illegal for udp. */
4164                case T_DATA_REQ:
4165                case T_EXDATA_REQ:
4166                case T_ORDREL_REQ:
4167                        udp_err_ack(q, mp, TNOTSUPPORT, 0);
4168                        return;
4169                default:
4170                        break;
4171                }
4172                break;
4173        case M_FLUSH:
4174                if (*rptr & FLUSHW)
4175                        flushq(q, FLUSHDATA);
4176                break;
4177        case M_IOCTL:
4178                iocp = (struct iocblk *)mp->b_rptr;
4179                switch (iocp->ioc_cmd) {
4180                case TI_GETPEERNAME:
4181                        if (udp->udp_state != TS_DATA_XFER) {
4182                                /*
4183                                 * If a default destination address has not
4184                                 * been associated with the stream, then we
4185                                 * don't know the peer's name.
4186                                 */
4187                                iocp->ioc_error = ENOTCONN;
4188                                iocp->ioc_count = 0;
4189                                mp->b_datap->db_type = M_IOCACK;
4190                                qreply(q, mp);
4191                                return;
4192                        }
4193                        /* FALLTHRU */
4194                case TI_GETMYNAME:
4195                        /*
4196                         * For TI_GETPEERNAME and TI_GETMYNAME, we first
4197                         * need to copyin the user's strbuf structure.
4198                         * Processing will continue in the M_IOCDATA case
4199                         * below.
4200                         */
4201                        mi_copyin(q, mp, NULL,
4202                            SIZEOF_STRUCT(strbuf, iocp->ioc_flag));
4203                        return;
4204                case _SIOCSOCKFALLBACK:
4205                        /*
4206                         * Either sockmod is about to be popped and the
4207                         * socket would now be treated as a plain stream,
4208                         * or a module is about to be pushed so we have
4209                         * to follow pure TPI semantics.
4210                         */
4211                        if (!udp->udp_issocket) {
4212                                DB_TYPE(mp) = M_IOCNAK;
4213                                iocp->ioc_error = EINVAL;
4214                        } else {
4215                                udp_use_pure_tpi(udp);

4217                                DB_TYPE(mp) = M_IOCACK;
4218                                iocp->ioc_error = 0;
4219                        }
```

```
4220                        iocp->ioc_count = 0;
4221                        iocp->ioc_rval = 0;
4222                        qreply(q, mp);
4223                        return;
4224                default:
4225                        break;
4226                }
4227                break;
4228        case M_IOCDATA:
4229                udp_wput_iocdata(q, mp);
4230                return;
4231        default:
4232                /* Unrecognized messages are passed through without change. */
4233                break;
4234        }
4235        ip_wput_nondata(q, mp);
4236 }

4238 /*
4239  * udp_wput_iocdata is called by udp_wput_other to handle all M_IOCDATA
4240  * messages.
4241  */
4242 static void
4243 udp_wput_iocdata(queue_t *q, mblk_t *mp)
4244 {
4245        mblk_t          *mp1;
4246        struct  iocblk *iocp = (struct iocblk *)mp->b_rptr;
4247        STRUCT_HANDLE(strbuf, sb);
4248        uint_t          addrlen;
4249        conn_t          *connp = Q_TO_CONN(q);
4250        udp_t           *udp = connp->conn_udp;

4252        /* Make sure it is one of ours. */
4253        switch (iocp->ioc_cmd) {
4254        case TI_GETMYNAME:
4255        case TI_GETPEERNAME:
4256                break;
4257        default:
4258                ip_wput_nondata(q, mp);
4259                return;
4260        }

4262        switch (mi_copy_state(q, mp, &mp1)) {
4263        case -1:
4264                return;
4265        case MI_COPY_CASE(MI_COPY_IN, 1):
4266                break;
4267        case MI_COPY_CASE(MI_COPY_OUT, 1):
4268                /*
4269                 * The address has been copied out, so now
4270                 * copyout the strbuf.
4271                 */
4272                mi_copyout(q, mp);
4273                return;
4274        case MI_COPY_CASE(MI_COPY_OUT, 2):
4275                /*
4276                 * The address and strbuf have been copied out.
4277                 * We're done, so just acknowledge the original
4278                 * M_IOCTL.
4279                 */
4280                mi_copy_done(q, mp, 0);
4281                return;
4282        default:
4283                /*
4284                 * Something strange has happened, so acknowledge
4285                 * the original M_IOCTL with an EPROTO error.
```



```
4286                 */
4287                mi_copy_done(q, mp, EPROTO);
4288                return;
4289        }

4291        /*
4292         * Now we have the strbuf structure for TI_GETMYNAME
4293         * and TI_GETPEERNAME.  Next we copyout the requested
4294         * address and then we'll copyout the strbuf.
4295         */
4296        STRUCT_SET_HANDLE(sb, iocp->ioc_flag, (void *)mp1->b_rptr);

4298        if (connp->conn_family == AF_INET)
4299                addrlen = sizeof (sin_t);
4300        else
4301                addrlen = sizeof (sin6_t);

4303        if (STRUCT_FGET(sb, maxlen) < addrlen) {
4304                mi_copy_done(q, mp, EINVAL);
4305                return;
4306        }

4308        switch (iocp->ioc_cmd) {
4309        case TI_GETMYNAME:
4310                break;
4311        case TI_GETPEERNAME:
4312                if (udp->udp_state != TS_DATA_XFER) {
4313                        mi_copy_done(q, mp, ENOTCONN);
4314                        return;
4315                }
4316                break;
4317        }
4318        mp1 = mi_copyout_alloc(q, mp, STRUCT_FGETP(sb, buf), addrlen, B_TRUE);
4319        if (!mp1)
4320                return;

4322        STRUCT_FSET(sb, len, addrlen);
4323        switch (((struct iocblk *)mp->b_rptr)->ioc_cmd) {
4324        case TI_GETMYNAME:
4325                (void) conn_getsockname(connp, (struct sockaddr *)mp1->b_wptr,
4326                    &addrlen);
4327                break;
4328        case TI_GETPEERNAME:
4329                (void) conn_getpeername(connp, (struct sockaddr *)mp1->b_wptr,
4330                    &addrlen);
4331                break;
4332        }
4333        mp1->b_wptr += addrlen;
4334        /* Copy out the address */
4335        mi_copyout(q, mp);
4336 }

4338 void
4339 udp_ddi_g_init(void)
4340 {
4341        udp_max_optsize = optcom_max_optsize(udp_opt_obj.odb_opt_des_arr,
4342            udp_opt_obj.odb_opt_arr_cnt);

4344        /*
4345         * We want to be informed each time a stack is created or
4346         * destroyed in the kernel, so we can maintain the
4347         * set of udp_stack_t's.
4348         */
4349        netstack_register(NS_UDP, udp_stack_init, NULL, udp_stack_fini);
4350 }
```

```
4352 void
4353 udp_ddi_g_destroy(void)
4354 {
4355         netstack_unregister(NS_UDP);
4356 }

4358 #define INET_NAME       "ip"

4360 /*
4361  * Initialize the UDP stack instance.
4362  */
4363 static void *
4364 udp_stack_init(netstackid_t stackid, netstack_t *ns)
4365 {
4366         udp_stack_t     *us;
4367         int             i;
4368         int             error = 0;
4369         major_t         major;
4370         size_t          arrsz;

4372         us = (udp_stack_t *)kmem_zalloc(sizeof (*us), KM_SLEEP);
4373         us->us_netstack = ns;

4375         mutex_init(&us->us_epriv_port_lock, NULL, MUTEX_DEFAULT, NULL);
4376         us->us_num_epriv_ports = UDP_NUM_EPRIV_PORTS;
4377         us->us_epriv_ports[0] = ULP_DEF_EPRIV_PORT1;
4378         us->us_epriv_ports[1] = ULP_DEF_EPRIV_PORT2;

4380         /*
4381          * The smallest anonymous port in the priviledged port range which UDP
4382          * looks for free port.  Use in the option UDP_ANONPRIVBIND.
4383          */
4384         us->us_min_anonpriv_port = 512;

4386         us->us_bind_fanout_size = udp_bind_fanout_size;

4388         /* Roundup variable that might have been modified in /etc/system */
4389         if (!ISP2(us->us_bind_fanout_size)) {
  28         if (us->us_bind_fanout_size & (us->us_bind_fanout_size - 1)) {
4390                 /* Not a power of two. Round up to nearest power of two */
4391                 for (i = 0; i < 31; i++) {
4392                         if (us->us_bind_fanout_size < (1 << i))
4393                                 break;
4394                 }
4395                 us->us_bind_fanout_size = 1 << i;
4396         }
4397         us->us_bind_fanout = kmem_zalloc(us->us_bind_fanout_size *
4398             sizeof (udp_fanout_t), KM_SLEEP);
4399         for (i = 0; i < us->us_bind_fanout_size; i++) {
4400                 mutex_init(&us->us_bind_fanout[i].uf_lock, NULL, MUTEX_DEFAULT,
4401                     NULL);
4402         }

4404         arrsz = udp_propinfo_count * sizeof (mod_prop_info_t);
4405         us->us_propinfo_tbl = (mod_prop_info_t *)kmem_alloc(arrsz,
4406             KM_SLEEP);
4407         bcopy(udp_propinfo_tbl, us->us_propinfo_tbl, arrsz);

4409         /* Allocate the per netstack stats */
4410         mutex_enter(&cpu_lock);
4411         us->us_sc_cnt = MAX(ncpus, boot_ncpus);
4412         mutex_exit(&cpu_lock);
4413         us->us_sc = kmem_zalloc(max_ncpus  * sizeof (udp_stats_cpu_t *),
4414             KM_SLEEP);
4415         for (i = 0; i < us->us_sc_cnt; i++) {
4416                 us->us_sc[i] = kmem_zalloc(sizeof (udp_stats_cpu_t),
```



```
4417                     KM_SLEEP);
4418         }

4420         us->us_kstat = udp_kstat2_init(stackid);
4421         us->us_mibkp = udp_kstat_init(stackid);

4423         major = mod_name_to_major(INET_NAME);
4424         error = ldi_ident_from_major(major, &us->us_ldi_ident);
4425         ASSERT(error == 0);
4426         return (us);
4427 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  115705 Thu Oct 23 10:42:11 2014
new/usr/src/uts/common/io/1394/t1394.c
5255 uts shouldn't open-code ISP2
**********************************************************
/*
 * 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 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * t1394.c
 *    1394 Target Driver Interface
 *    This file contains all of the 1394 Software Framework routines called
 *    by target drivers
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/disp.h>
#include <sys/tnf_probe.h>

#include <sys/1394/t1394.h>
#include <sys/1394/s1394.h>
#include <sys/1394/h1394.h>
#include <sys/1394/ieee1394.h>

static int s1394_allow_detach = 0;

/*
 * Function:    t1394_attach()
 * Input(s):    dip                     The dip given to the target driver
 *                                          in it's attach() routine
 *              version                 The version of the target driver -
 *                                          T1394_VERSION_V1
 *              flags                   The flags parameter is unused (for now)
 *
 * Output(s):   attachinfo              Used to pass info back to target,
 *                                          including bus generation, local
```



```
 *                                          node ID, dma attribute, etc.
 *              t1394_hdl               The target "handle" to be used for
 *                                          all subsequent calls into the
 *                                          1394 Software Framework
 *
 * Description: t1394_attach() registers the target (based on its dip) with
 *              the 1394 Software Framework.  It returns the bus_generation,
 *              local_nodeID, iblock_cookie and other useful information to
 *              the target, as well as a handle (t1394_hdl) that will be used
 *              in all subsequent calls into this framework.
 */
/* ARGSUSED */
int
t1394_attach(dev_info_t *dip, int version, uint_t flags,
    t1394_attachinfo_t *attachinfo, t1394_handle_t *t1394_hdl)
{
        s1394_hal_t     *hal;
        s1394_target_t  *target;
        uint_t          dev;
        uint_t          curr;
        uint_t          unit_dir;
        int             hp_node = 0;

        ASSERT(t1394_hdl != NULL);
        ASSERT(attachinfo != NULL);

        TNF_PROBE_0_DEBUG(t1394_attach_enter, S1394_TNF_SL_HOTPLUG_STACK, "");

        *t1394_hdl = NULL;

        if (version != T1394_VERSION_V1) {
                TNF_PROBE_1(t1394_attach_error, S1394_TNF_SL_HOTPLUG_ERROR, "",
                    tnf_string, msg, "Invalid version");
                TNF_PROBE_0_DEBUG(t1394_attach_exit,
                    S1394_TNF_SL_HOTPLUG_STACK, "");
                return (DDI_FAILURE);
        }

        hal = s1394_dip_to_hal(ddi_get_parent(dip));
        if (hal == NULL) {
                TNF_PROBE_1(t1394_attach_error, S1394_TNF_SL_HOTPLUG_ERROR, "",
                    tnf_string, msg, "No parent dip found for target");
                TNF_PROBE_0_DEBUG(t1394_attach_exit,
                    S1394_TNF_SL_HOTPLUG_STACK, "");
                return (DDI_FAILURE);
        }

        ASSERT(MUTEX_NOT_HELD(&hal->topology_tree_mutex));

        hp_node = ddi_prop_exists(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
            "hp-node");

        /* Allocate space for s1394_target_t */
        target = kmem_zalloc(sizeof (s1394_target_t), KM_SLEEP);

        mutex_enter(&hal->topology_tree_mutex);

        target->target_version = version;

        /* Copy in the params */
        target->target_dip = dip;
        target->on_hal     = hal;

        /* Place the target on the appropriate node */
        target->on_node = NULL;
```

```
126         rw_enter(&target->on_hal->target_list_rwlock, RW_WRITER);
127         if (hp_node != 0) {
128                 s1394_add_target_to_node(target);
129                 /*
130                  * on_node can be NULL if the node got unplugged
131                  * while the target driver is in its attach routine.
132                  */
133                 if (target->on_node == NULL) {
134                         s1394_remove_target_from_node(target);
135                         rw_exit(&target->on_hal->target_list_rwlock);
136                         mutex_exit(&hal->topology_tree_mutex);
137                         kmem_free(target, sizeof (s1394_target_t));
138                         TNF_PROBE_1(t1394_attach_error,
139                             S1394_TNF_SL_HOTPLUG_ERROR, "", tnf_string, msg,
140                             "on_node == NULL");
141                         TNF_PROBE_0_DEBUG(t1394_attach_exit,
142                             S1394_TNF_SL_HOTPLUG_STACK, "");
143                         return (DDI_FAILURE);
144                 }
145
146                 target->target_state = S1394_TARG_HP_NODE;
147                 if (S1394_NODE_BUS_PWR_CONSUMER(target->on_node) == B_TRUE)
148                         target->target_state |= S1394_TARG_BUS_PWR_CONSUMER;
149         }
150
151         /* Return the current generation */
152         attachinfo->localinfo.bus_generation = target->on_hal->generation_count;
153
154         /* Fill in hal node id */
155         attachinfo->localinfo.local_nodeID = target->on_hal->node_id;
156
157         /* Give the target driver the iblock_cookie */
158         attachinfo->iblock_cookie = target->on_hal->halinfo.hw_interrupt;
159
160         /* Give the target driver the attributes */
161         attachinfo->acc_attr    = target->on_hal->halinfo.acc_attr;
162         attachinfo->dma_attr    = target->on_hal->halinfo.dma_attr;
163
164         unit_dir = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
165                 DDI_PROP_DONTPASS, "unit-dir-offset", 0);
166         target->unit_dir = unit_dir;
167
168         /* By default, disable all physical AR requests */
169         target->physical_arreq_enabled = 0;
170
171
172         /* Get dev_max_payload & current_max_payload */
173         s1394_get_maxpayload(target, &dev, &curr);
174         target->dev_max_payload         = dev;
175         target->current_max_payload     = curr;
176
177         /* Add into linked list */
178         if ((target->on_hal->target_head == NULL) &&
179             (target->on_hal->target_tail == NULL)) {
180                 target->on_hal->target_head = target;
181                 target->on_hal->target_tail = target;
182         } else {
183                 target->on_hal->target_tail->target_next = target;
184                 target->target_prev = target->on_hal->target_tail;
185                 target->on_hal->target_tail = target;
186         }
187         rw_exit(&target->on_hal->target_list_rwlock);
188
189         /* Fill in services layer private info */
190         *t1394_hdl = (t1394_handle_t)target;
```



```
192         mutex_exit(&hal->topology_tree_mutex);
193
194         TNF_PROBE_0_DEBUG(t1394_attach_exit, S1394_TNF_SL_HOTPLUG_STACK, "");
195         return (DDI_SUCCESS);
196 }
197
198 /*
199  * Function:    t1394_detach()
200  * Input(s):    t1394_hdl               The target "handle" returned by
201  *                                          t1394_attach()
202  *              flags                   The flags parameter is unused (for now)
203  *
204  * Output(s):   DDI_SUCCESS             Target successfully detached
205  *              DDI_FAILURE             Target failed to detach
206  *
207  * Description: t1394_detach() unregisters the target from the 1394 Software
208  *              Framework.  t1394_detach() can fail if the target has any
209  *              allocated commands that haven't been freed.
210  */
211 /* ARGSUSED */
212 int
213 t1394_detach(t1394_handle_t *t1394_hdl, uint_t flags)
214 {
215         s1394_target_t  *target;
216         uint_t          num_cmds;
217
218         TNF_PROBE_0_DEBUG(t1394_detach_enter, S1394_TNF_SL_HOTPLUG_STACK, "");
219
220         ASSERT(t1394_hdl != NULL);
221
222         target = (s1394_target_t *)(*t1394_hdl);
223
224         ASSERT(target->on_hal);
225
226         mutex_enter(&target->on_hal->topology_tree_mutex);
227         rw_enter(&target->on_hal->target_list_rwlock, RW_WRITER);
228
229         /* How many cmds has this target allocated? */
230         num_cmds = target->target_num_cmds;
231
232         if (num_cmds != 0) {
233                 rw_exit(&target->on_hal->target_list_rwlock);
234                 mutex_exit(&target->on_hal->topology_tree_mutex);
235                 TNF_PROBE_1(t1394_detach_error, S1394_TNF_SL_HOTPLUG_ERROR, "",
236                     tnf_string, msg, "Must free all commands before detach()");
237                 TNF_PROBE_0_DEBUG(t1394_detach_exit,
238                     S1394_TNF_SL_HOTPLUG_STACK, "");
239                 return (DDI_FAILURE);
240         }
241
242         /*
243          * Remove from linked lists. Topology tree is already locked
244          * so that the node won't go away while we are looking at it.
245          */
246         if ((target->on_hal->target_head == target) &&
247             (target->on_hal->target_tail == target)) {
248                 target->on_hal->target_head = NULL;
249                 target->on_hal->target_tail = NULL;
250         } else {
251                 if (target->target_prev)
252                         target->target_prev->target_next = target->target_next;
253                 if (target->target_next)
254                         target->target_next->target_prev = target->target_prev;
255                 if (target->on_hal->target_head == target)
256                         target->on_hal->target_head = target->target_next;
257                 if (target->on_hal->target_tail == target)
```

```
258                         target->on_hal->target_tail = target->target_prev;
259         }
260
261         s1394_remove_target_from_node(target);
262         rw_exit(&target->on_hal->target_list_rwlock);
263
264         mutex_exit(&target->on_hal->topology_tree_mutex);
265
266         /* Free memory */
267         kmem_free(target, sizeof (s1394_target_t));
268
269         *t1394_hdl = NULL;
270
271         TNF_PROBE_0_DEBUG(t1394_detach_exit, S1394_TNF_SL_HOTPLUG_STACK, "");
272         return (DDI_SUCCESS);
273 }
274
275 /*
276  * Function:    t1394_alloc_cmd()
277  * Input(s):    t1394_hdl               The target "handle" returned by
278  *                                          t1394_attach()
279  *              flags                   The flags parameter is described below
280  *
281  * Output(s):   cmdp                    Pointer to the newly allocated command
282  *
283  * Description: t1394_alloc_cmd() allocates a command for use with the
284  *              t1394_read(), t1394_write(), or t1394_lock() interfaces
285  *              of the 1394 Software Framework.  By default, t1394_alloc_cmd()
286  *              may sleep while allocating memory for the command structure.
287  *              If this is undesirable, the target may set the
288  *              T1394_ALLOC_CMD_NOSLEEP bit in the flags parameter.  Also,
289  *              this call may fail because a target driver has already
290  *              allocated MAX_NUMBER_ALLOC_CMDS commands.
291  */
292 int
293 t1394_alloc_cmd(t1394_handle_t t1394_hdl, uint_t flags, cmd1394_cmd_t **cmdp)
294 {
295         s1394_hal_t      *hal;
296         s1394_target_t   *target;
297         s1394_cmd_priv_t *s_priv;
298         uint_t           num_cmds;
299
300         TNF_PROBE_0_DEBUG(t1394_alloc_cmd_enter, S1394_TNF_SL_ATREQ_STACK, "");
301
302         ASSERT(t1394_hdl != NULL);
303
304         target = (s1394_target_t *)t1394_hdl;
305
306         /* Find the HAL this target resides on */
307         hal = target->on_hal;
308
309         rw_enter(&hal->target_list_rwlock, RW_WRITER);
310
311         /* How many cmds has this target allocated? */
312         num_cmds = target->target_num_cmds;
313
314         if (num_cmds >= MAX_NUMBER_ALLOC_CMDS) {
315                 rw_exit(&hal->target_list_rwlock);
316                 TNF_PROBE_1(t1394_alloc_cmd_error, S1394_TNF_SL_ATREQ_ERROR,
317                     "", tnf_string, msg, "Attempted to alloc > "
318                     "MAX_NUMBER_ALLOC_CMDS");
319                 TNF_PROBE_0_DEBUG(t1394_alloc_cmd_exit,
320                     S1394_TNF_SL_ATREQ_STACK, "");
321                 /* kstats - cmd alloc failures */
322                 hal->hal_kstats->cmd_alloc_fail++;
323                 return (DDI_FAILURE);
```



```
324         }
325
326         /* Increment the number of cmds this target has allocated? */
327         target->target_num_cmds = num_cmds + 1;
328
329         if (s1394_alloc_cmd(hal, flags, cmdp) != DDI_SUCCESS) {
330                 target->target_num_cmds = num_cmds;     /* Undo increment */
331                 rw_exit(&hal->target_list_rwlock);
332                 TNF_PROBE_1(t1394_alloc_cmd_error, S1394_TNF_SL_ATREQ_ERROR, "",
333                     tnf_string, msg, "Failed to allocate command structure");
334                 TNF_PROBE_0_DEBUG(t1394_alloc_cmd_exit,
335                     S1394_TNF_SL_ATREQ_STACK, "");
336                 /* kstats - cmd alloc failures */
337                 hal->hal_kstats->cmd_alloc_fail++;
338                 return (DDI_FAILURE);
339         }
340
341         rw_exit(&hal->target_list_rwlock);
342
343         /* Get the Services Layer private area */
344         s_priv = S1394_GET_CMD_PRIV(*cmdp);
345
346         /* Initialize the command's blocking mutex */
347         mutex_init(&s_priv->blocking_mutex, NULL, MUTEX_DRIVER,
348             hal->halinfo.hw_interrupt);
349
350         /* Initialize the command's blocking condition variable */
351         cv_init(&s_priv->blocking_cv, NULL, CV_DRIVER, NULL);
352
353         TNF_PROBE_0_DEBUG(t1394_alloc_cmd_exit, S1394_TNF_SL_ATREQ_STACK, "");
354         return (DDI_SUCCESS);
355 }
356
357 /*
358  * Function:    t1394_free_cmd()
359  * Input(s):    t1394_hdl               The target "handle" returned by
360  *                                          t1394_attach()
361  *              flags                   The flags parameter is unused (for now)
362  *              cmdp                    Pointer to the command to be freed
363  *
364  * Output(s):   DDI_SUCCESS             Target successfully freed command
365  *              DDI_FAILURE             Target failed to free command
366  *
367  * Description: t1394_free_cmd() attempts to free a command that has previously
368  *              been allocated by the target driver.  It is possible for
369  *              t1394_free_cmd() to fail because the command is currently
370  *              in-use by the 1394 Software Framework.
371  */
372 /* ARGSUSED */
373 int
374 t1394_free_cmd(t1394_handle_t t1394_hdl, uint_t flags, cmd1394_cmd_t **cmdp)
375 {
376         s1394_hal_t      *hal;
377         s1394_target_t   *target;
378         s1394_cmd_priv_t *s_priv;
379         uint_t           num_cmds;
380
381         TNF_PROBE_0_DEBUG(t1394_free_cmd_enter, S1394_TNF_SL_ATREQ_STACK, "");
382
383         ASSERT(t1394_hdl != NULL);
384
385         target = (s1394_target_t *)t1394_hdl;
386
387         /* Find the HAL this target resides on */
388         hal = target->on_hal;
```

```
390         rw_enter(&hal->target_list_rwlock, RW_WRITER);

392         /* How many cmds has this target allocated? */
393         num_cmds = target->target_num_cmds;

395         if (num_cmds == 0) {
396                 rw_exit(&hal->target_list_rwlock);
397                 TNF_PROBE_2(t1394_free_cmd_error, S1394_TNF_SL_ATREQ_ERROR, "",
398                     tnf_string, msg, "No commands left to be freed "
399                     "(num_cmds <= 0)", tnf_uint, num_cmds, num_cmds);
400                 TNF_PROBE_0_DEBUG(t1394_free_cmd_exit,
401                     S1394_TNF_SL_ATREQ_STACK, "");
402                 ASSERT(num_cmds != 0);
403                 return (DDI_FAILURE);
404         }

406         /* Get the Services Layer private area */
407         s_priv = S1394_GET_CMD_PRIV(*cmdp);

409         /* Check that command isn't in use */
410         if (s_priv->cmd_in_use == B_TRUE) {
411                 rw_exit(&hal->target_list_rwlock);
412                 TNF_PROBE_1(t1394_free_cmd_error, S1394_TNF_SL_ATREQ_ERROR, "",
413                     tnf_string, msg, "Attempted to free an in-use command");
414                 TNF_PROBE_0_DEBUG(t1394_free_cmd_exit,
415                     S1394_TNF_SL_ATREQ_STACK, "");
416                 ASSERT(s_priv->cmd_in_use == B_FALSE);
417                 return (DDI_FAILURE);
418         }

420         /* Decrement the number of cmds this target has allocated */
421         target->target_num_cmds--;

423         rw_exit(&hal->target_list_rwlock);

425         /* Destroy the command's blocking condition variable */
426         cv_destroy(&s_priv->blocking_cv);

428         /* Destroy the command's blocking mutex */
429         mutex_destroy(&s_priv->blocking_mutex);

431         kmem_cache_free(hal->hal_kmem_cachep, *cmdp);

433         /* Command pointer is set to NULL before returning */
434         *cmdp = NULL;

436         /* kstats - number of cmd frees */
437         hal->hal_kstats->cmd_free++;

439         TNF_PROBE_0_DEBUG(t1394_free_cmd_exit, S1394_TNF_SL_ATREQ_STACK, "");
440         return (DDI_SUCCESS);
441 }

443 /*
444  * Function:    t1394_read()
445  * Input(s):    t1394_hdl               The target "handle" returned by
446  *                                          t1394_attach()
447  *              cmd                     Pointer to the command to send
448  *
449  * Output(s):   DDI_SUCCESS             Target successful sent the command
450  *              DDI_FAILURE             Target failed to send command
451  *
452  * Description: t1394_read() attempts to send an asynchronous read request
453  *              onto the 1394 bus.
454  */
455 int
```



```
456 t1394_read(t1394_handle_t t1394_hdl, cmd1394_cmd_t *cmd)
457 {
458         s1394_hal_t       *to_hal;
459         s1394_target_t    *target;
460         s1394_cmd_priv_t  *s_priv;
461         s1394_hal_state_t state;
462         int               ret;
463         int               err;

465         TNF_PROBE_0_DEBUG(t1394_read_enter, S1394_TNF_SL_ATREQ_STACK, "");

467         ASSERT(t1394_hdl != NULL);
468         ASSERT(cmd != NULL);

470         /* Get the Services Layer private area */
471         s_priv = S1394_GET_CMD_PRIV(cmd);

473         /* Is this command currently in use? */
474         if (s_priv->cmd_in_use == B_TRUE) {
475                 TNF_PROBE_1(t1394_read_error, S1394_TNF_SL_ATREQ_ERROR, "",
476                     tnf_string, msg, "Attempted to resend an in-use command");
477                 TNF_PROBE_0_DEBUG(t1394_read_exit, S1394_TNF_SL_ATREQ_STACK,
478                     "");
479                 ASSERT(s_priv->cmd_in_use == B_FALSE);
480                 return (DDI_FAILURE);
481         }

483         target = (s1394_target_t *)t1394_hdl;

485         /* Set-up the destination of the command */
486         to_hal = target->on_hal;

488         /* No status (default) */
489         cmd->cmd_result = CMD1394_NOSTATUS;

491         /* Check for proper command type */
492         if ((cmd->cmd_type != CMD1394_ASYNCH_RD_QUAD) &&
493             (cmd->cmd_type != CMD1394_ASYNCH_RD_BLOCK)) {
494                 cmd->cmd_result = CMD1394_EINVALID_COMMAND;
495                 TNF_PROBE_1(t1394_read_error, S1394_TNF_SL_ATREQ_ERROR, "",
496                     tnf_string, msg, "Invalid command type specified");
497                 TNF_PROBE_0_DEBUG(t1394_read_exit,
498                     S1394_TNF_SL_ATREQ_STACK, "");
499                 return (DDI_FAILURE);
500         }

502         /* Is this a blocking command on interrupt stack? */
503         if ((cmd->cmd_options & CMD1394_BLOCKING) &&
504             (servicing_interrupt())) {
505                 cmd->cmd_result = CMD1394_EINVALID_CONTEXT;
506                 TNF_PROBE_1(t1394_read_error, S1394_TNF_SL_ATREQ_ERROR, "",
507                     tnf_string, msg, "Tried to use CMD1394_BLOCKING in "
508                     "intr context");
509                 TNF_PROBE_0_DEBUG(t1394_read_exit,
510                     S1394_TNF_SL_ATREQ_STACK, "");
511                 return (DDI_FAILURE);
512         }

514         mutex_enter(&to_hal->topology_tree_mutex);
515         state = to_hal->hal_state;
516         if (state != S1394_HAL_NORMAL) {
517                 ret = s1394_HAL_asynch_error(to_hal, cmd, state);
518                 if (ret != CMD1394_CMDSUCCESS) {
519                         cmd->cmd_result = ret;
520                         mutex_exit(&to_hal->topology_tree_mutex);
521                         return (DDI_FAILURE);
```

```
522                 }
523         }

525         ret = s1394_setup_asynch_command(to_hal, target, cmd,
526             S1394_CMD_READ, &err);

528         /* Command has now been put onto the queue! */
529         if (ret != DDI_SUCCESS) {
530                 /* Copy error code into result */
531                 cmd->cmd_result = err;
532                 mutex_exit(&to_hal->topology_tree_mutex);
533                 TNF_PROBE_1(t1394_read_error, S1394_TNF_SL_ATREQ_ERROR, "",
534                     tnf_string, msg, "Failed in s1394_setup_asynch_command()");
535                 TNF_PROBE_0_DEBUG(t1394_read_exit,
536                     S1394_TNF_SL_ATREQ_STACK, "");
537                 return (DDI_FAILURE);
538         }

540         /*
541          * If this command was sent during a bus reset,
542          * then put it onto the pending Q.
543          */
544         if (state == S1394_HAL_RESET) {
545                 /* Remove cmd from outstanding request Q */
546                 s1394_remove_q_asynch_cmd(to_hal, cmd);
547                 /* Are we on the bus reset event stack? */
548                 if (s1394_on_br_thread(to_hal) == B_TRUE) {
549                         /* Blocking commands are not allowed */
550                         if (cmd->cmd_options & CMD1394_BLOCKING) {
551                                 mutex_exit(&to_hal->topology_tree_mutex);
552                                 s_priv->cmd_in_use = B_FALSE;
553                                 cmd->cmd_result    = CMD1394_EINVALID_CONTEXT;
554                                 TNF_PROBE_1(t1394_read_error,
555                                     S1394_TNF_SL_ATREQ_ERROR, "", tnf_string,
556                                     msg, "CMD1394_BLOCKING in bus reset "
557                                     "context");
558                                 TNF_PROBE_0_DEBUG(t1394_read_exit,
559                                     S1394_TNF_SL_ATREQ_STACK, "");
560                                 return (DDI_FAILURE);
561                         }
562                 }

564                 s1394_pending_q_insert(to_hal, cmd, S1394_PENDING_Q_FRONT);
565                 mutex_exit(&to_hal->topology_tree_mutex);

567                 /* Block (if necessary) */
568                 goto block_on_asynch_cmd;
569         }
570         mutex_exit(&to_hal->topology_tree_mutex);

572         /* Send the command out */
573         ret = s1394_xfer_asynch_command(to_hal, cmd, &err);

575         if (ret != DDI_SUCCESS) {
576                 if (err == CMD1394_ESTALE_GENERATION) {
577                         /* Remove cmd from outstanding request Q */
578                         s1394_remove_q_asynch_cmd(to_hal, cmd);
579                         s1394_pending_q_insert(to_hal, cmd,
580                             S1394_PENDING_Q_FRONT);

582                         /* Block (if necessary) */
583                         goto block_on_asynch_cmd;

585                 } else {
586                         /* Remove cmd from outstanding request Q */
587                         s1394_remove_q_asynch_cmd(to_hal, cmd);
```



```
589                         s_priv->cmd_in_use = B_FALSE;

591                         /* Copy error code into result */
592                         cmd->cmd_result    = err;

594                         TNF_PROBE_1(t1394_read_error, S1394_TNF_SL_ATREQ_ERROR,
595                             "", tnf_string, msg, "Failed in "
596                             "s1394_xfer_asynch_command()");
597                         TNF_PROBE_0_DEBUG(t1394_read_exit,
598                             S1394_TNF_SL_ATREQ_STACK, "");
599                         return (DDI_FAILURE);
600                 }
601         } else {
602                 /* Block (if necessary) */
603                 goto block_on_asynch_cmd;
604         }

606 block_on_asynch_cmd:
607         s1394_block_on_asynch_cmd(cmd);

609         TNF_PROBE_0_DEBUG(t1394_read_exit,
610             S1394_TNF_SL_ATREQ_STACK, "");
611         return (DDI_SUCCESS);
612 }

614 /*
615  * Function:    t1394_write()
616  * Input(s):    t1394_hdl               The target "handle" returned by
617  *                                          t1394_attach()
618  *              cmd                     Pointer to the command to send
619  *
620  * Output(s):   DDI_SUCCESS             Target successful sent the command
621  *              DDI_FAILURE             Target failed to send command
622  *
623  * Description: t1394_write() attempts to send an asynchronous write request
624  *              onto the 1394 bus.
625  */
626 int
627 t1394_write(t1394_handle_t t1394_hdl, cmd1394_cmd_t *cmd)
628 {
629         s1394_hal_t       *to_hal;
630         s1394_target_t    *target;
631         s1394_cmd_priv_t  *s_priv;
632         s1394_hal_state_t state;
633         int               ret;
634         int               err;

636         TNF_PROBE_0_DEBUG(t1394_write_enter, S1394_TNF_SL_ATREQ_STACK, "");

638         ASSERT(t1394_hdl != NULL);
639         ASSERT(cmd != NULL);

641         /* Get the Services Layer private area */
642         s_priv = S1394_GET_CMD_PRIV(cmd);

644         /* Is this command currently in use? */
645         if (s_priv->cmd_in_use == B_TRUE) {
646                 TNF_PROBE_1(t1394_write_error, S1394_TNF_SL_ATREQ_ERROR, "",
647                     tnf_string, msg, "Attempted to resend an in-use command");
648                 TNF_PROBE_0_DEBUG(t1394_write_exit, S1394_TNF_SL_ATREQ_STACK,
649                     "");
650                 ASSERT(s_priv->cmd_in_use == B_FALSE);
651                 return (DDI_FAILURE);
652         }
```

```
654         target = (s1394_target_t *)t1394_hdl;
655
656         /* Set-up the destination of the command */
657         to_hal = target->on_hal;
658
659         /* Is this an FA request? */
660         if (s_priv->cmd_ext_type == S1394_CMD_EXT_FA) {
661                 if (S1394_IS_CMD_FCP(s_priv) &&
662                     (s1394_fcp_write_check_cmd(cmd) != DDI_SUCCESS)) {
663                         TNF_PROBE_0_DEBUG(t1394_write_exit,
664                             S1394_TNF_SL_ATREQ_STACK, "");
665                         return (DDI_FAILURE);
666                 }
667                 s1394_fa_convert_cmd(to_hal, cmd);
668         }
669
670         /* No status (default) */
671         cmd->cmd_result = CMD1394_NOSTATUS;
672
673         /* Check for proper command type */
674         if ((cmd->cmd_type != CMD1394_ASYNCH_WR_QUAD) &&
675             (cmd->cmd_type != CMD1394_ASYNCH_WR_BLOCK)) {
676                 cmd->cmd_result = CMD1394_EINVALID_COMMAND;
677                 s1394_fa_check_restore_cmd(to_hal, cmd);
678                 TNF_PROBE_1(t1394_write_error, S1394_TNF_SL_ATREQ_ERROR, "",
679                     tnf_string, msg, "Invalid command type specified");
680                 TNF_PROBE_0_DEBUG(t1394_write_exit, S1394_TNF_SL_ATREQ_STACK,
681                     "");
682                 return (DDI_FAILURE);
683         }
684
685         /* Is this a blocking command on interrupt stack? */
686         if ((cmd->cmd_options & CMD1394_BLOCKING) &&
687             (servicing_interrupt())) {
688                 cmd->cmd_result = CMD1394_EINVALID_CONTEXT;
689                 s1394_fa_check_restore_cmd(to_hal, cmd);
690                 TNF_PROBE_1(t1394_write_error, S1394_TNF_SL_ATREQ_ERROR, "",
691                     tnf_string, msg, "Tried to use CMD1394_BLOCKING in intr "
692                     "context");
693                 TNF_PROBE_0_DEBUG(t1394_write_exit, S1394_TNF_SL_ATREQ_STACK,
694                     "");
695                 return (DDI_FAILURE);
696         }
697
698         mutex_enter(&to_hal->topology_tree_mutex);
699         state = to_hal->hal_state;
700         if (state != S1394_HAL_NORMAL) {
701                 ret = s1394_HAL_asynch_error(to_hal, cmd, state);
702                 if (ret != CMD1394_CMDSUCCESS) {
703                         cmd->cmd_result = ret;
704                         mutex_exit(&to_hal->topology_tree_mutex);
705                         s1394_fa_check_restore_cmd(to_hal, cmd);
706                         return (DDI_FAILURE);
707                 }
708         }
709
710         ret = s1394_setup_asynch_command(to_hal, target, cmd,
711             S1394_CMD_WRITE, &err);
712
713         /* Command has now been put onto the queue! */
714         if (ret != DDI_SUCCESS) {
715                 /* Copy error code into result */
716                 cmd->cmd_result = err;
717                 mutex_exit(&to_hal->topology_tree_mutex);
718                 s1394_fa_check_restore_cmd(to_hal, cmd);
719                 TNF_PROBE_1(t1394_write_error, S1394_TNF_SL_ATREQ_ERROR, "",
```



```
720                     tnf_string, msg, "Failed in s1394_setup_asynch_command()");
721                 TNF_PROBE_0_DEBUG(t1394_write_exit, S1394_TNF_SL_ATREQ_STACK,
722                     "");
723                 return (DDI_FAILURE);
724         }
725
726         /*
727          * If this command was sent during a bus reset,
728          * then put it onto the pending Q.
729          */
730         if (state == S1394_HAL_RESET) {
731                 /* Remove cmd from outstanding request Q */
732                 s1394_remove_q_asynch_cmd(to_hal, cmd);
733                 /* Are we on the bus reset event stack? */
734                 if (s1394_on_br_thread(to_hal) == B_TRUE) {
735                         /* Blocking commands are not allowed */
736                         if (cmd->cmd_options & CMD1394_BLOCKING) {
737                                 mutex_exit(&to_hal->topology_tree_mutex);
738                                 s_priv->cmd_in_use = B_FALSE;
739                                 cmd->cmd_result    = CMD1394_EINVALID_CONTEXT;
740                                 s1394_fa_check_restore_cmd(to_hal, cmd);
741                                 TNF_PROBE_1(t1394_write_error,
742                                     S1394_TNF_SL_ATREQ_ERROR, "", tnf_string,
743                                     msg, "CMD1394_BLOCKING in bus reset cntxt");
744                                 TNF_PROBE_0_DEBUG(t1394_write_exit,
745                                     S1394_TNF_SL_ATREQ_STACK, "");
746                                 return (DDI_FAILURE);
747                         }
748                 }
749
750                 s1394_pending_q_insert(to_hal, cmd, S1394_PENDING_Q_FRONT);
751                 mutex_exit(&to_hal->topology_tree_mutex);
752
753                 /* Block (if necessary) */
754                 s1394_block_on_asynch_cmd(cmd);
755
756                 TNF_PROBE_0_DEBUG(t1394_write_exit, S1394_TNF_SL_ATREQ_STACK,
757                     "");
758                 return (DDI_SUCCESS);
759         }
760         mutex_exit(&to_hal->topology_tree_mutex);
761
762         /* Send the command out */
763         ret = s1394_xfer_asynch_command(to_hal, cmd, &err);
764
765         if (ret != DDI_SUCCESS) {
766                 if (err == CMD1394_ESTALE_GENERATION) {
767                         /* Remove cmd from outstanding request Q */
768                         s1394_remove_q_asynch_cmd(to_hal, cmd);
769                         s1394_pending_q_insert(to_hal, cmd,
770                             S1394_PENDING_Q_FRONT);
771
772                         /* Block (if necessary) */
773                         s1394_block_on_asynch_cmd(cmd);
774
775                         TNF_PROBE_0_DEBUG(t1394_write_exit,
776                             S1394_TNF_SL_ATREQ_STACK, "");
777                         return (DDI_SUCCESS);
778                 } else {
779                         /* Remove cmd from outstanding request Q */
780                         s1394_remove_q_asynch_cmd(to_hal, cmd);
781
782                         s_priv->cmd_in_use = B_FALSE;
783
784                         /* Copy error code into result */
785                         cmd->cmd_result = err;
```

```
787                         s1394_fa_check_restore_cmd(to_hal, cmd);
788                         TNF_PROBE_1(t1394_write_error,
789                             S1394_TNF_SL_ATREQ_ERROR, "", tnf_string, msg,
790                             "Failed in s1394_xfer_asynch_command()");
791                         TNF_PROBE_0_DEBUG(t1394_write_exit,
792                             S1394_TNF_SL_ATREQ_STACK, "");
793                         return (DDI_FAILURE);
794                 }
795         } else {
796                 /* Block (if necessary) */
797                 s1394_block_on_asynch_cmd(cmd);

799                 TNF_PROBE_0_DEBUG(t1394_write_exit, S1394_TNF_SL_ATREQ_STACK,
800                     "");
801                 return (DDI_SUCCESS);
802         }
803 }

805 /*
806  * Function:    t1394_lock()
807  * Input(s):    t1394_hdl               The target "handle" returned by
808  *                                          t1394_attach()
809  *              cmd                     Pointer to the command to send
810  *
811  * Output(s):   DDI_SUCCESS             Target successful sent the command
812  *              DDI_FAILURE             Target failed to send command
813  *
814  * Description: t1394_lock() attempts to send an asynchronous lock request
815  *              onto the 1394 bus.
816  */
817 int
818 t1394_lock(t1394_handle_t t1394_hdl, cmd1394_cmd_t *cmd)
819 {
820         s1394_hal_t         *to_hal;
821         s1394_target_t      *target;
822         s1394_cmd_priv_t    *s_priv;
823         s1394_hal_state_t   state;
824         cmd1394_lock_type_t lock_type;
825         uint_t              num_retries;
826         int                 ret;

828         TNF_PROBE_0_DEBUG(t1394_lock_enter, S1394_TNF_SL_ATREQ_STACK, "");

830         ASSERT(t1394_hdl != NULL);
831         ASSERT(cmd != NULL);

833         /* Get the Services Layer private area */
834         s_priv = S1394_GET_CMD_PRIV(cmd);

836         /* Is this command currently in use? */
837         if (s_priv->cmd_in_use == B_TRUE) {
838                 TNF_PROBE_1(t1394_lock_error, S1394_TNF_SL_ATREQ_ERROR, "",
839                     tnf_string, msg, "Attempted to resend an in-use command");
840                 TNF_PROBE_0_DEBUG(t1394_lock_exit, S1394_TNF_SL_ATREQ_STACK,
841                     "");
842                 ASSERT(s_priv->cmd_in_use == B_FALSE);
843                 return (DDI_FAILURE);
844         }

846         target = (s1394_target_t *)t1394_hdl;

848         /* Set-up the destination of the command */
849         to_hal = target->on_hal;

851         mutex_enter(&to_hal->topology_tree_mutex);
```



```
852         state = to_hal->hal_state;
853         if (state != S1394_HAL_NORMAL) {
854                 ret = s1394_HAL_asynch_error(to_hal, cmd, state);
855                 if (ret != CMD1394_CMDSUCCESS) {
856                         cmd->cmd_result = ret;
857                         mutex_exit(&to_hal->topology_tree_mutex);
858                         return (DDI_FAILURE);
859                 }
860         }
861         mutex_exit(&to_hal->topology_tree_mutex);

863         /* Check for proper command type */
864         if ((cmd->cmd_type != CMD1394_ASYNCH_LOCK_32) &&
865             (cmd->cmd_type != CMD1394_ASYNCH_LOCK_64)) {
866                 cmd->cmd_result = CMD1394_EINVALID_COMMAND;
867                 TNF_PROBE_1(t1394_lock_error, S1394_TNF_SL_ATREQ_ERROR, "",
868                     tnf_string, msg, "Invalid command type sent to "
869                     "t1394_lock()");
870                 TNF_PROBE_0_DEBUG(t1394_lock_exit, S1394_TNF_SL_ATREQ_STACK,
871                     "");
872                 return (DDI_FAILURE);
873         }

875         /* No status (default) */
876         cmd->cmd_result = CMD1394_NOSTATUS;

878         /* Is this a blocking command on interrupt stack? */
879         if ((cmd->cmd_options & CMD1394_BLOCKING) &&
880             (servicing_interrupt())) {
881                 cmd->cmd_result = CMD1394_EINVALID_CONTEXT;
882                 TNF_PROBE_1(t1394_lock_error, S1394_TNF_SL_ATREQ_ERROR, "",
883                     tnf_string, msg, "Tried to use CMD1394_BLOCKING in intr "
884                     "context");
885                 TNF_PROBE_0_DEBUG(t1394_lock_exit, S1394_TNF_SL_ATREQ_STACK,
886                     "");
887                 return (DDI_FAILURE);
888         }

890         if (cmd->cmd_type == CMD1394_ASYNCH_LOCK_32) {
891                 lock_type       = cmd->cmd_u.l32.lock_type;
892                 num_retries     = cmd->cmd_u.l32.num_retries;
893         } else {        /* (cmd->cmd_type == CMD1394_ASYNCH_LOCK_64) */
894                 lock_type       = cmd->cmd_u.l64.lock_type;
895                 num_retries     = cmd->cmd_u.l64.num_retries;
896         }

898         /* Make sure num_retries is reasonable */
899         ASSERT(num_retries <= MAX_NUMBER_OF_LOCK_RETRIES);

901         switch (lock_type) {
902         case CMD1394_LOCK_MASK_SWAP:
903         case CMD1394_LOCK_FETCH_ADD:
904         case CMD1394_LOCK_LITTLE_ADD:
905         case CMD1394_LOCK_BOUNDED_ADD:
906         case CMD1394_LOCK_WRAP_ADD:
907         case CMD1394_LOCK_COMPARE_SWAP:
908                 ret = s1394_compare_swap(to_hal, target, cmd);
909                 break;

911         case CMD1394_LOCK_BIT_AND:
912         case CMD1394_LOCK_BIT_OR:
913         case CMD1394_LOCK_BIT_XOR:
914         case CMD1394_LOCK_INCREMENT:
915         case CMD1394_LOCK_DECREMENT:
916         case CMD1394_LOCK_ADD:
917         case CMD1394_LOCK_SUBTRACT:
```

```
918         case CMD1394_LOCK_THRESH_ADD:
919         case CMD1394_LOCK_THRESH_SUBTRACT:
920         case CMD1394_LOCK_CLIP_ADD:
921         case CMD1394_LOCK_CLIP_SUBTRACT:
922                 ret = s1394_split_lock_req(to_hal, target, cmd);
923                 break;

925         default:
926                 TNF_PROBE_1(t1394_lock_error, S1394_TNF_SL_ATREQ_ERROR, "",
927                     tnf_string, msg, "Invalid lock_type in command");
928                 cmd->cmd_result = CMD1394_EINVALID_COMMAND;
929                 ret = DDI_FAILURE;
930                 break;
931         }

933         TNF_PROBE_0_DEBUG(t1394_lock_exit, S1394_TNF_SL_ATREQ_STACK, "");
934         return (ret);
935 }

937 /*
938  * Function:    t1394_alloc_addr()
939  * Input(s):    t1394_hdl               The target "handle" returned by
940  *                                          t1394_attach()
941  *              addr_allocp             The structure used to specify the type,
942  *                                          size, permissions, and callbacks
943  *                                          (if any) for the requested block
944  *                                          of 1394 address space
945  *              flags                   The flags parameter is unused (for now)
946  *
947  * Output(s):   result                  Used to pass more specific info back
948  *                                          to target
949  *
950  * Description: t1394_alloc_addr() requests that part of the 1394 Address Space
951  *              on the local node be set aside for this target driver, and
952  *              associated with this address space should be some permissions
953  *              and callbacks.  If the request is unable to be fulfilled,
954  *              t1394_alloc_addr() will return DDI_FAILURE and result will
955  *              indicate the reason.  T1394_EINVALID_PARAM indicates that the
956  *              combination of flags given is invalid, and T1394_EALLOC_ADDR
957  *              indicates that the requested type of address space is
958  *              unavailable.
959  */
960 /* ARGSUSED */
961 int
962 t1394_alloc_addr(t1394_handle_t t1394_hdl, t1394_alloc_addr_t *addr_allocp,
963     uint_t flags, int *result)
964 {
965         s1394_hal_t     *hal;
966         s1394_target_t  *target;
967         uint64_t        addr_lo;
968         uint64_t        addr_hi;
969         int             err;

971         TNF_PROBE_0_DEBUG(t1394_alloc_addr_enter, S1394_TNF_SL_ARREQ_STACK,
972             "");

974         ASSERT(t1394_hdl != NULL);
975         ASSERT(addr_allocp != NULL);

977         target = (s1394_target_t *)t1394_hdl;

979         /* Find the HAL this target resides on */
980         hal = target->on_hal;

982         /* Get the bounds of the request */
983         addr_lo = addr_allocp->aa_address;
```



```
984         addr_hi = addr_lo + addr_allocp->aa_length;

986         /* Check combination of flags */
987         if ((addr_allocp->aa_enable & T1394_ADDR_RDENBL) &&
988             (addr_allocp->aa_evts.recv_read_request == NULL) &&
989             (addr_allocp->aa_kmem_bufp == NULL)) {
990                 if ((addr_allocp->aa_type != T1394_ADDR_FIXED)  ||
991                     (addr_lo < hal->physical_addr_lo)           ||
992                     (addr_hi > hal->physical_addr_hi)) {

994                         /*
995                          * Reads are enabled, but target doesn't want to
996                          * be notified and hasn't given backing store
997                          */
998                         *result = T1394_EINVALID_PARAM;

1000                        TNF_PROBE_1(t1394_alloc_addr_error,
1001                            S1394_TNF_SL_ARREQ_ERROR, "", tnf_string, msg,
1002                            "Invalid flags "
1003                            "(RDs on, notify off, no backing store)");
1004                        TNF_PROBE_0_DEBUG(t1394_alloc_addr_exit,
1005                            S1394_TNF_SL_ARREQ_STACK, "");

1007                        /* kstats - addr alloc failures */
1008                        hal->hal_kstats->addr_alloc_fail++;
1009                        return (DDI_FAILURE);
1010                } else {
1011                        addr_allocp->aa_enable &= ~T1394_ADDR_RDENBL;
1012                }
1013        }

1015        if ((addr_allocp->aa_enable & T1394_ADDR_WRENBL) &&
1016            (addr_allocp->aa_evts.recv_write_request == NULL) &&
1017            (addr_allocp->aa_kmem_bufp == NULL)) {
1018                if ((addr_allocp->aa_type != T1394_ADDR_FIXED)  ||
1019                    (addr_lo < hal->physical_addr_lo)           ||
1020                    (addr_hi > hal->physical_addr_hi)) {

1022                        /*
1023                         * Writes are enabled, but target doesn't want to
1024                         * be notified and hasn't given backing store
1025                         */
1026                        *result = T1394_EINVALID_PARAM;

1028                        TNF_PROBE_1(t1394_alloc_addr_error,
1029                            S1394_TNF_SL_ARREQ_ERROR, "", tnf_string, msg,
1030                            "Invalid flags "
1031                            "(WRs on, notify off, no backing store)");
1032                        TNF_PROBE_0_DEBUG(t1394_alloc_addr_exit,
1033                            S1394_TNF_SL_ARREQ_STACK, "");

1035                        /* kstats - addr alloc failures */
1036                        hal->hal_kstats->addr_alloc_fail++;
1037                        return (DDI_FAILURE);
1038                } else {
1039                        addr_allocp->aa_enable &= ~T1394_ADDR_WRENBL;
1040                }
1041        }

1043        if ((addr_allocp->aa_enable & T1394_ADDR_LKENBL) &&
1044            (addr_allocp->aa_evts.recv_lock_request == NULL) &&
1045            (addr_allocp->aa_kmem_bufp == NULL)) {
1046                if ((addr_allocp->aa_type != T1394_ADDR_FIXED)  ||
1047                    (addr_lo < hal->physical_addr_lo)           ||
1048                    (addr_hi > hal->physical_addr_hi)) {
```

```
1050                        /*
1051                         * Locks are enabled, but target doesn't want to
1052                         * be notified and hasn't given backing store
1053                         */
1054                        *result = T1394_EINVALID_PARAM;

1056                        TNF_PROBE_1(t1394_alloc_addr_error,
1057                            S1394_TNF_SL_ARREQ_ERROR, "", tnf_string, msg,
1058                            "Invalid flags "
1059                            "(LKs on, notify off, no backing store)");
1060                        TNF_PROBE_0_DEBUG(t1394_alloc_addr_exit,
1061                            S1394_TNF_SL_ARREQ_STACK, "");

1063                        /* kstats - addr alloc failures */
1064                        hal->hal_kstats->addr_alloc_fail++;
1065                        return (DDI_FAILURE);
1066                } else {
1067                        addr_allocp->aa_enable &= ~T1394_ADDR_LKENBL;
1068                }
1069        }

1071        /* If not T1394_ADDR_FIXED, then allocate a block */
1072        if (addr_allocp->aa_type != T1394_ADDR_FIXED) {
1073                err = s1394_request_addr_blk((s1394_hal_t *)target->on_hal,
1074                                    addr_allocp);
1075                if (err != DDI_SUCCESS) {
1076                        *result = T1394_EALLOC_ADDR;
1077                        /* kstats - addr alloc failures */
1078                        hal->hal_kstats->addr_alloc_fail++;
1079                } else {
1080                        *result = T1394_NOERROR;
1081                }
1082                TNF_PROBE_0_DEBUG(t1394_alloc_addr_exit,
1083                    S1394_TNF_SL_ARREQ_STACK, "");
1084                return (err);
1085        } else {
1086                err = s1394_claim_addr_blk((s1394_hal_t *)target->on_hal,
1087                                    addr_allocp);
1088                if (err != DDI_SUCCESS) {
1089                        *result = T1394_EALLOC_ADDR;
1090                        /* kstats - addr alloc failures */
1091                        hal->hal_kstats->addr_alloc_fail++;
1092                } else {
1093                        *result = T1394_NOERROR;
1094                        /* If physical, update the AR request counter */
1095                        if ((addr_lo >= hal->physical_addr_lo) &&
1096                            (addr_hi <= hal->physical_addr_hi)) {
1097                                rw_enter(&hal->target_list_rwlock, RW_WRITER);
1098                                target->physical_arreq_enabled++;
1099                                rw_exit(&hal->target_list_rwlock);

1101                                s1394_physical_arreq_set_one(target);
1102                        }
1103                }
1104                TNF_PROBE_0_DEBUG(t1394_alloc_addr_exit,
1105                    S1394_TNF_SL_ARREQ_STACK, "");
1106                return (err);
1107        }
1108 }

1110 /*
1111  * Function:    t1394_free_addr()
1112  * Input(s):    t1394_hdl               The target "handle" returned by
1113  *                                          t1394_attach()
1114  *              addr_hdl                The address "handle" returned by the
1115  *                                         the t1394_alloc_addr() routine
```



```
1116  *              flags                   The flags parameter is unused (for now)
1117  *
1118  * Output(s):   DDI_SUCCESS             Target successfully freed memory
1119  *              DDI_FAILURE             Target failed to free the memory block
1120  *
1121  * Description: t1394_free_addr() attempts to free up memory that has been
1122  *              allocated by the target using t1394_alloc_addr().
1123  */
1124 /* ARGSUSED */
1125 int
1126 t1394_free_addr(t1394_handle_t t1394_hdl, t1394_addr_handle_t *addr_hdl,
1127     uint_t flags)
1128 {
1129        s1394_addr_space_blk_t  *curr_blk;
1130        s1394_hal_t             *hal;
1131        s1394_target_t          *target;

1133        TNF_PROBE_0_DEBUG(t1394_free_addr_enter, S1394_TNF_SL_ARREQ_STACK, "");

1135        ASSERT(t1394_hdl != NULL);
1136        ASSERT(addr_hdl != NULL);

1138        target = (s1394_target_t *)t1394_hdl;

1140        /* Find the HAL this target resides on */
1141        hal = target->on_hal;

1143        curr_blk = (s1394_addr_space_blk_t *)(*addr_hdl);

1145        if (s1394_free_addr_blk(hal, curr_blk) != DDI_SUCCESS) {
1146                TNF_PROBE_0_DEBUG(t1394_free_addr_exit,
1147                    S1394_TNF_SL_ARREQ_STACK, "");
1148                return (DDI_FAILURE);
1149        }

1151        /* If physical, update the AR request counter */
1152        if (curr_blk->addr_type == T1394_ADDR_FIXED) {
1153                target->physical_arreq_enabled--;
1154                s1394_physical_arreq_clear_one(target);
1155        }

1157        *addr_hdl = NULL;

1159        /* kstats - number of addr frees */
1160        hal->hal_kstats->addr_space_free++;

1162        TNF_PROBE_0_DEBUG(t1394_free_addr_exit, S1394_TNF_SL_ARREQ_STACK, "");
1163        return (DDI_SUCCESS);
1164 }

1166 /*
1167  * Function:    t1394_recv_request_done()
1168  * Input(s):    t1394_hdl               The target "handle" returned by
1169  *                                          t1394_attach()
1170  *              resp                    Pointer to the command which the
1171  *                                          target received in it's callback
1172  *              flags                   The flags parameter is unused (for now)
1173  *
1174  * Output(s):   DDI_SUCCESS             Target successfully returned command
1175  *                                          to the 1394 Software Framework,
1176  *                                          and, if necessary, sent response
1177  *              DDI_FAILURE             Target failed to return the command to
1178  *                                          the 1394 Software Framework
1179  *
1180  * Description: t1394_recv_request_done() takes the command that is given and
1181  *              determines whether that command requires a response to be
```

```
 *              sent on the 1394 bus.  If it is necessary and it's response
 *              code (cmd_result) has been set appropriately, then a response
 *              will be sent.  If no response is necessary (broadcast or
 *              posted write), then the command resources are reclaimed.
 */
/* ARGSUSED */
int
t1394_recv_request_done(t1394_handle_t t1394_hdl, cmd1394_cmd_t *resp,
    uint_t flags)
{
        s1394_hal_t      *hal;
        s1394_cmd_priv_t *s_priv;
        h1394_cmd_priv_t *h_priv;
        mblk_t           *curr_blk;
        size_t           msgb_len;
        size_t           size;
        int              ret;
        boolean_t        response = B_TRUE;
        boolean_t        posted_write = B_FALSE;
        boolean_t        write_cmd = B_FALSE;
        boolean_t        mblk_too_small;

        TNF_PROBE_0_DEBUG(t1394_recv_request_done_enter,
            S1394_TNF_SL_ARREQ_STACK, "");

        ASSERT(t1394_hdl != NULL);
        ASSERT(resp != NULL);

        /* Find the HAL this target resides on */
        hal = ((s1394_target_t *)t1394_hdl)->on_hal;

        /* Get the Services Layer private area */
        s_priv = S1394_GET_CMD_PRIV(resp);

        /* Get a pointer to the HAL private struct */
        h_priv = (h1394_cmd_priv_t *)&s_priv->hal_cmd_private;

        /* Is this an FA request? */
        if (s_priv->cmd_ext_type == S1394_CMD_EXT_FA) {
                s1394_fa_convert_cmd(hal, resp);
        }

        /* Is this a write request? */
        if ((resp->cmd_type == CMD1394_ASYNCH_WR_QUAD) ||
            (resp->cmd_type == CMD1394_ASYNCH_WR_BLOCK)) {
                write_cmd = B_TRUE;
                /* Is this a posted write request? */
                posted_write = s_priv->posted_write;
        }

        /* If broadcast or posted write cmd, don't send response */
        if ((resp->broadcast == 1) ||
            ((write_cmd == B_TRUE) && (posted_write == B_TRUE)))
                response = B_FALSE;

        if (response == B_FALSE) {
                if ((write_cmd == B_TRUE) && (posted_write == B_TRUE)) {
                        /* kstats - Posted Write error */
                        hal->hal_kstats->arreq_posted_write_error++;
                }

                /* Free the command - Pass it back to the HAL */
                HAL_CALL(hal).response_complete(hal->halinfo.hal_private, resp,
                    h_priv);
                TNF_PROBE_0_DEBUG(t1394_recv_request_done_exit,
                    S1394_TNF_SL_ARREQ_STACK, "");
```



```
                return (DDI_SUCCESS);
        }

        ASSERT(response == B_TRUE);

        /* Verify valid response code */
        switch (resp->cmd_result) {
        case IEEE1394_RESP_COMPLETE:
                /* Is the mblk_t too small? */
                if (resp->cmd_type == CMD1394_ASYNCH_RD_BLOCK) {
                        curr_blk = resp->cmd_u.b.data_block;
                        size     = resp->cmd_u.b.blk_length;
                        msgb_len = 0;
                        mblk_too_small = B_TRUE;

                        if (curr_blk == NULL) {
                                TNF_PROBE_1(t1394_recv_request_done_error,
                                    S1394_TNF_SL_ARREQ_ERROR, "", tnf_string,
                                    msg, "mblk_t is NULL in response");
                                TNF_PROBE_0_DEBUG(t1394_recv_request_done_exit,
                                    S1394_TNF_SL_ARREQ_STACK, "");
                                /*
                                 * Free the command - Pass it back
                                 * to the HAL
                                 */
                                HAL_CALL(hal).response_complete(
                                    hal->halinfo.hal_private, resp, h_priv);
                                ASSERT(curr_blk != NULL);
                                return (DDI_FAILURE);
                        }

                        while (curr_blk != NULL) {
                                msgb_len +=
                                    (curr_blk->b_wptr - curr_blk->b_rptr);

                                if (msgb_len >= size) {
                                        mblk_too_small = B_FALSE;
                                        break;
                                }
                                curr_blk = curr_blk->b_cont;
                        }

                        if (mblk_too_small == B_TRUE) {
                                TNF_PROBE_1(t1394_recv_request_done_error,
                                    S1394_TNF_SL_ARREQ_ERROR, "", tnf_string,
                                    msg, "mblk_t too small in response");
                                TNF_PROBE_0_DEBUG(t1394_recv_request_done_exit,
                                    S1394_TNF_SL_ARREQ_STACK, "");
                                /*
                                 * Free the command - Pass it back
                                 * to the HAL
                                 */
                                HAL_CALL(hal).response_complete(
                                    hal->halinfo.hal_private, resp, h_priv);
                                ASSERT(mblk_too_small != B_TRUE);
                                return (DDI_FAILURE);
                        }
                }
                /* FALLTHROUGH */
        case IEEE1394_RESP_CONFLICT_ERROR:
        case IEEE1394_RESP_DATA_ERROR:
        case IEEE1394_RESP_TYPE_ERROR:
        case IEEE1394_RESP_ADDRESS_ERROR:
                ret = s1394_send_response(hal, resp);
                TNF_PROBE_0_DEBUG(t1394_recv_request_done_exit,
                    S1394_TNF_SL_ARREQ_STACK, "");
```

```
1314                 return (ret);

1316         default:
1317                 TNF_PROBE_1(t1394_recv_request_done_error,
1318                     S1394_TNF_SL_ARREQ_ERROR, "", tnf_string, msg,
1319                     "Invalid response code");
1320                 TNF_PROBE_0_DEBUG(t1394_recv_request_done_exit,
1321                     S1394_TNF_SL_ARREQ_STACK, "");
1322                 return (DDI_FAILURE);
1323         }
1324 }


1327 /*
1328  * Function:    t1394_fcp_register_controller()
1329  * Input(s):    t1394_hdl               The target "handle" returned by
1330  *                                          t1394_attach()
1331  *              evts                    The structure in which the target
1332  *                                          specifies its callback routines
1333  *
1334  *              flags                   The flags parameter is unused (for now)
1335  *
1336  * Output(s):   DDI_SUCCESS             Successfully registered.
1337  *
1338  *              DDI_FAILURE             Not registered due to failure.
1339  *
1340  * Description: Used to register the target within the Framework as an FCP
1341  *              controller.
1342  */
1343 /* ARGSUSED */
1344 int
1345 t1394_fcp_register_controller(t1394_handle_t t1394_hdl, t1394_fcp_evts_t *evts,
1346     uint_t flags)
1347 {
1348         int             result;

1350         TNF_PROBE_0_DEBUG(t1394_fcp_register_controller_enter,
1351             S1394_TNF_SL_FCP_STACK, "");

1353         ASSERT(t1394_hdl != NULL);

1355         result = s1394_fcp_register_ctl((s1394_target_t *)t1394_hdl, evts);

1357         TNF_PROBE_0_DEBUG(t1394_fcp_register_controller_exit,
1358             S1394_TNF_SL_FCP_STACK, "");
1359         return (result);
1360 }

1362 /*
1363  * Function:    t1394_fcp_unregister_controller()
1364  * Input(s):    t1394_hdl               The target "handle" returned by
1365  *                                          t1394_attach()
1366  *
1367  * Output(s):   DDI_SUCCESS             Successfully unregistered.
1368  *
1369  *              DDI_FAILURE             Not unregistered due to failure.
1370  *
1371  * Description: Used to unregister the target within the Framework as an FCP
1372  *              controller.
1373  */
1374 int
1375 t1394_fcp_unregister_controller(t1394_handle_t t1394_hdl)
1376 {
1377         int             result;

1379         TNF_PROBE_0_DEBUG(t1394_fcp_unregister_controller_enter,
```



```
1380             S1394_TNF_SL_FCP_STACK, "");

1382         ASSERT(t1394_hdl != NULL);

1384         result = s1394_fcp_unregister_ctl((s1394_target_t *)t1394_hdl);

1386         TNF_PROBE_0_DEBUG(t1394_fcp_unregister_controller_exit,
1387             S1394_TNF_SL_FCP_STACK, "");
1388         return (result);
1389 }

1391 /*
1392  * Function:    t1394_fcp_register_target()
1393  * Input(s):    t1394_hdl               The target "handle" returned by
1394  *                                          t1394_attach()
1395  *              evts                    The structure in which the target
1396  *                                          specifies its callback routines
1397  *
1398  *              flags                   The flags parameter is unused (for now)
1399  *
1400  * Output(s):   DDI_SUCCESS             Successfully registered.
1401  *
1402  *              DDI_FAILURE             Not registered due to failure.
1403  *
1404  * Description: Used to register the target within the Framework as an FCP
1405  *              target.
1406  */
1407 /* ARGSUSED */
1408 int
1409 t1394_fcp_register_target(t1394_handle_t t1394_hdl, t1394_fcp_evts_t *evts,
1410     uint_t flags)
1411 {
1412         int             result;

1414         TNF_PROBE_0_DEBUG(t1394_fcp_register_target_enter,
1415             S1394_TNF_SL_FCP_STACK, "");

1417         ASSERT(t1394_hdl != NULL);

1419         result = s1394_fcp_register_tgt((s1394_target_t *)t1394_hdl, evts);

1421         TNF_PROBE_0_DEBUG(t1394_fcp_register_target_exit,
1422             S1394_TNF_SL_FCP_STACK, "");
1423         return (result);
1424 }

1426 /*
1427  * Function:    t1394_fcp_unregister_target()
1428  * Input(s):    t1394_hdl               The target "handle" returned by
1429  *                                          t1394_attach()
1430  *
1431  * Output(s):   DDI_SUCCESS             Successfully unregistered.
1432  *
1433  *              DDI_FAILURE             Not unregistered due to failure.
1434  *
1435  * Description: Used to unregister the target within the Framework as an FCP
1436  *              target.
1437  */
1438 int
1439 t1394_fcp_unregister_target(t1394_handle_t t1394_hdl)
1440 {
1441         int             result;

1443         TNF_PROBE_0_DEBUG(t1394_fcp_unregister_target_enter,
1444             S1394_TNF_SL_FCP_STACK, "");
```

```
1446         ASSERT(t1394_hdl != NULL);

1448         result = s1394_fcp_unregister_tgt((s1394_target_t *)t1394_hdl);

1450         TNF_PROBE_0_DEBUG(t1394_fcp_unregister_target_exit,
1451             S1394_TNF_SL_FCP_STACK, "");
1452         return (result);
1453 }

1455 /*
1456  * Function:    t1394_cmp_register()
1457  * Input(s):    t1394_hdl               The target "handle" returned by
1458  *                                          t1394_attach()
1459  *              evts                    The structure in which the target
1460  *                                          specifies its callback routines
1461  *
1462  * Output(s):   DDI_SUCCESS             Successfully registered.
1463  *
1464  *              DDI_FAILURE             Not registered due to failure.
1465  *
1466  * Description: Used to register the target within the Framework as a CMP
1467  *              device.
1468  */
1469 /* ARGSUSED */
1470 int
1471 t1394_cmp_register(t1394_handle_t t1394_hdl, t1394_cmp_evts_t *evts,
1472     uint_t flags)
1473 {
1474         int             result;

1476         TNF_PROBE_0_DEBUG(t1394_cmp_register_enter, S1394_TNF_SL_CMP_STACK, "");

1478         ASSERT(t1394_hdl != NULL);

1480         result = s1394_cmp_register((s1394_target_t *)t1394_hdl, evts);

1482         TNF_PROBE_0_DEBUG(t1394_cmp_register_exit, S1394_TNF_SL_CMP_STACK, "");
1483         return (result);
1484 }

1486 /*
1487  * Function:    t1394_cmp_unregister()
1488  * Input(s):    t1394_hdl               The target "handle" returned by
1489  *                                          t1394_attach()
1490  *              evts                    The structure in which the target
1491  *                                          specifies its callback routines
1492  *
1493  * Output(s):   DDI_SUCCESS             Successfully registered.
1494  *
1495  *              DDI_FAILURE             Not registered due to failure.
1496  *
1497  * Description: Used to unregister the target within the Framework as a CMP
1498  *              device.
1499  */
1500 int
1501 t1394_cmp_unregister(t1394_handle_t t1394_hdl)
1502 {
1503         int             result;

1505         TNF_PROBE_0_DEBUG(t1394_cmp_unregister_enter, S1394_TNF_SL_CMP_STACK,
1506             "");

1508         ASSERT(t1394_hdl != NULL);

1510         result = s1394_cmp_unregister((s1394_target_t *)t1394_hdl);
```



```
1512         TNF_PROBE_0_DEBUG(t1394_cmp_unregister_exit, S1394_TNF_SL_CMP_STACK,
1513             "");
1514         return (result);
1515 }

1517 /*
1518  * Function:    t1394_cmp_read()
1519  * Input(s):    t1394_hdl               The target "handle" returned by
1520  *                                          t1394_attach()
1521  *              reg                     Register type.
1522  *              valp                    Returned register value.
1523  *
1524  * Output(s):   DDI_SUCCESS             Successfully registered.
1525  *
1526  *              DDI_FAILURE             Not registered due to failure.
1527  *
1528  * Description: Used to read a CMP register value.
1529  */
1530 int
1531 t1394_cmp_read(t1394_handle_t t1394_hdl, t1394_cmp_reg_t reg, uint32_t *valp)
1532 {
1533         int             result;

1535         TNF_PROBE_0_DEBUG(t1394_cmp_read_enter, S1394_TNF_SL_CMP_STACK, "");

1537         ASSERT(t1394_hdl != NULL);

1539         result = s1394_cmp_read((s1394_target_t *)t1394_hdl, reg, valp);

1541         TNF_PROBE_0_DEBUG(t1394_cmp_read_exit, S1394_TNF_SL_CMP_STACK, "");
1542         return (result);
1543 }

1545 /*
1546  * Function:    t1394_cmp_cas()
1547  * Input(s):    t1394_hdl               The target "handle" returned by
1548  *                                          t1394_attach()
1549  *              reg                     Register type.
1550  *              arg_val                 Compare argument.
1551  *              new_val                 New register value.
1552  *              old_valp                Returned original register value.
1553  *
1554  * Output(s):   DDI_SUCCESS             Successfully registered.
1555  *
1556  *              DDI_FAILURE             Not registered due to failure.
1557  *
1558  * Description: Used to compare-swap a CMP register value.
1559  */
1560 int
1561 t1394_cmp_cas(t1394_handle_t t1394_hdl, t1394_cmp_reg_t reg, uint32_t arg_val,
1562     uint32_t new_val, uint32_t *old_valp)
1563 {
1564         int             result;

1566         TNF_PROBE_0_DEBUG(t1394_cmp_read_enter, S1394_TNF_SL_CMP_STACK, "");

1568         ASSERT(t1394_hdl != NULL);

1570         result = s1394_cmp_cas((s1394_target_t *)t1394_hdl, reg, arg_val,
1571                                 new_val, old_valp);

1573         TNF_PROBE_0_DEBUG(t1394_cmp_read_exit, S1394_TNF_SL_CMP_STACK, "");
1574         return (result);
1575 }

1577 /*
```

```
 * Function:    t1394_alloc_isoch_single()
 * Input(s):    t1394_hdl               The target "handle" returned by
 *                                          t1394_attach()
 *              sii                     The structure used to set up the
 *                                          overall characteristics of the
 *                                          isochronous stream
 *              flags                   The flags parameter is unused (for now)
 *
 * Output(s):   setup_args              Contains the channel number that was
 *                                          allocated
 *              t1394_single_hdl        This in the isoch "handle" used in
 *                                          t1394_free_isoch_single()
 *              result                  Used to pass more specific info back
 *                                          to target
 *
 * Description: t1394_alloc_isoch_single() is used to direct the 1394 Software
 *              Framework to allocate an isochronous channel and bandwidth
 *              from the Isochronous Resource Manager (IRM).  If a bus reset
 *              occurs, the 1394 Software Framework attempts to reallocate the
 *              same resources, calling the rsrc_fail_target() callback if
 *              it is unsuccessful.
 */
/* ARGSUSED */
int
t1394_alloc_isoch_single(t1394_handle_t t1394_hdl,
    t1394_isoch_singleinfo_t *sii, uint_t flags,
    t1394_isoch_single_out_t *output_args,
    t1394_isoch_single_handle_t *t1394_single_hdl, int *result)
{
        s1394_hal_t             *hal;
        s1394_isoch_cec_t       *cec_new;
        t1394_join_isochinfo_t  jii;
        int                     ret;
        int                     err;

        TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_enter,
            S1394_TNF_SL_ISOCH_STACK, "");

        ASSERT(t1394_hdl != NULL);
        ASSERT(t1394_single_hdl != NULL);
        ASSERT(sii != NULL);

        hal = ((s1394_target_t *)t1394_hdl)->on_hal;

        /* Check for invalid channel_mask */
        if (sii->si_channel_mask == 0) {
                TNF_PROBE_1(t1394_alloc_isoch_single_error,
                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
                    "Invalid channel mask");
                TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_exit,
                    S1394_TNF_SL_ISOCH_STACK, "");
                return (DDI_FAILURE);
        }

        /* Check for invalid bandwidth */
        if ((sii->si_bandwidth <= IEEE1394_BANDWIDTH_MIN) ||
            (sii->si_bandwidth > IEEE1394_BANDWIDTH_MAX)) {
                TNF_PROBE_1(t1394_alloc_isoch_single_error,
                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
                    "Invalid bandwidth requirements");
                TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_exit,
                    S1394_TNF_SL_ISOCH_STACK, "");
                return (DDI_FAILURE);
        }

        /* Verify that rsrc_fail_target() callback is non-NULL */
```



```
        if (sii->rsrc_fail_target == NULL) {
                TNF_PROBE_1(t1394_alloc_isoch_single_error,
                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
                    "Invalid callback specified");
                TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_exit,
                    S1394_TNF_SL_ISOCH_STACK, "");
                return (DDI_FAILURE);
        }

        /*
         * Allocate an Isoch CEC of type S1394_SINGLE
         */

        /* Allocate the Isoch CEC structure */
        cec_new = kmem_zalloc(sizeof (s1394_isoch_cec_t), KM_SLEEP);

        /* Initialize the structure type */
        cec_new->cec_type = S1394_SINGLE;

        /* Create the mutex and "in_callbacks" cv */
        mutex_init(&cec_new->isoch_cec_mutex, NULL, MUTEX_DRIVER,
            hal->halinfo.hw_interrupt);
        cv_init(&cec_new->in_callbacks_cv, NULL, CV_DRIVER,
            hal->halinfo.hw_interrupt);

        /* Initialize the Isoch CEC's member list */
        cec_new->cec_member_list_head = NULL;
        cec_new->cec_member_list_tail = NULL;

        /* Initialize the filters */
        cec_new->filter_min_speed       = sii->si_speed;
        cec_new->filter_max_speed       = sii->si_speed;
        cec_new->filter_current_speed   = cec_new->filter_max_speed;
        cec_new->filter_channel_mask    = sii->si_channel_mask;
        cec_new->bandwidth              = sii->si_bandwidth;
        cec_new->state_transitions      = ISOCH_CEC_FREE | ISOCH_CEC_JOIN |
                                             ISOCH_CEC_SETUP;

        mutex_enter(&hal->isoch_cec_list_mutex);

        /* Insert Isoch CEC into the HAL's list */
        s1394_isoch_cec_list_insert(hal, cec_new);

        mutex_exit(&hal->isoch_cec_list_mutex);

        /*
         * Join the newly created Isoch CEC
         */
        jii.req_channel_mask    = sii->si_channel_mask;
        jii.req_max_speed       = sii->si_speed;
        jii.jii_options         = T1394_TALKER;
        jii.isoch_cec_evts_arg  = sii->single_evt_arg;

        /* All events are NULL except rsrc_fail_target() */
        jii.isoch_cec_evts.setup_target     = NULL;
        jii.isoch_cec_evts.start_target     = NULL;
        jii.isoch_cec_evts.stop_target      = NULL;
        jii.isoch_cec_evts.stop_target      = NULL;
        jii.isoch_cec_evts.teardown_target  = NULL;
        jii.isoch_cec_evts.rsrc_fail_target = sii->rsrc_fail_target;

        ret = t1394_join_isoch_cec(t1394_hdl,
            (t1394_isoch_cec_handle_t)cec_new, 0, &jii);

        if (ret != DDI_SUCCESS) {
                TNF_PROBE_1(t1394_alloc_isoch_single_error,
```

```
1710                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1711                    "Unexpected error from t1394_join_isoch_cec()");

1713                ret = t1394_free_isoch_cec(t1394_hdl, flags,
1714                    (t1394_isoch_cec_handle_t *)&cec_new);
1715                if (ret != DDI_SUCCESS) {
1716                        /* Unable to free the Isoch CEC */
1717                        TNF_PROBE_1(t1394_alloc_isoch_single_error,
1718                            S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1719                            "Unexpected error from t1394_free_isoch_cec()");
1720                        ASSERT(0);
1721                }

1723                /* Handle is nulled out before returning */
1724                *t1394_single_hdl = NULL;

1726                TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_exit,
1727                    S1394_TNF_SL_ISOCH_STACK, "");
1728                return (DDI_FAILURE);
1729        }

1731        /*
1732         * Setup the isoch resources, etc.
1733         */
1734        ret = t1394_setup_isoch_cec(t1394_hdl,
1735            (t1394_isoch_cec_handle_t)cec_new, 0, &err);

1737        if (ret != DDI_SUCCESS) {
1738                TNF_PROBE_1(t1394_alloc_isoch_single_error,
1739                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1740                    "Unexpected error from t1394_setup_isoch_cec()");

1742                *result = err;

1744                /* Leave the Isoch CEC */
1745                ret = t1394_leave_isoch_cec(t1394_hdl,
1746                    (t1394_isoch_cec_handle_t)cec_new, 0);
1747                if (ret != DDI_SUCCESS) {
1748                        /* Unable to leave the Isoch CEC */
1749                        TNF_PROBE_1(t1394_alloc_isoch_single_error,
1750                            S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1751                            "Unexpected error from t1394_leave_isoch_cec()");
1752                        ASSERT(0);
1753                }

1755                /* Free up the Isoch CEC */
1756                ret = t1394_free_isoch_cec(t1394_hdl, flags,
1757                    (t1394_isoch_cec_handle_t *)&cec_new);
1758                if (ret != DDI_SUCCESS) {
1759                        /* Unable to free the Isoch CEC */
1760                        TNF_PROBE_1(t1394_alloc_isoch_single_error,
1761                            S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1762                            "Unexpected error from t1394_free_isoch_cec()");
1763                        ASSERT(0);
1764                }

1766                /* Handle is nulled out before returning */
1767                *t1394_single_hdl = NULL;

1769                TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_exit,
1770                    S1394_TNF_SL_ISOCH_STACK, "");
1771                return (DDI_FAILURE);
1772        }

1774        /* Return the setup_args - channel num and speed */
1775        mutex_enter(&cec_new->isoch_cec_mutex);
```



```
1776        output_args->channel_num  = cec_new->realloc_chnl_num;
1777        mutex_exit(&cec_new->isoch_cec_mutex);

1779        /* Update the handle */
1780        *t1394_single_hdl = (t1394_isoch_single_handle_t)cec_new;

1782        TNF_PROBE_0_DEBUG(t1394_alloc_isoch_single_exit,
1783            S1394_TNF_SL_ISOCH_STACK, "");
1784        return (DDI_SUCCESS);
1785 }

1787 /*
1788  * Function:    t1394_free_isoch_single()
1789  * Input(s):    t1394_hdl               The target "handle" returned by
1790  *                                          t1394_attach()
1791  *              t1394_single_hdl        The isoch "handle" return by
1792  *                                          t1394_alloc_isoch_single()
1793  *              flags                   The flags parameter is unused (for now)
1794  *
1795  * Output(s):   None
1796  *
1797  * Description: t1394_free_isoch_single() frees the isochronous resources
1798  *              and the handle that were allocated during the call to
1799  *              t1394_alloc_isoch_single().
1800  */
1801 /* ARGSUSED */
1802 void
1803 t1394_free_isoch_single(t1394_handle_t t1394_hdl,
1804     t1394_isoch_single_handle_t *t1394_single_hdl, uint_t flags)
1805 {
1806        s1394_isoch_cec_t *cec_curr;
1807        int               ret;

1809        TNF_PROBE_0_DEBUG(t1394_free_isoch_single_enter,
1810            S1394_TNF_SL_ISOCH_STACK, "");

1812        ASSERT(t1394_hdl != NULL);
1813        ASSERT(t1394_single_hdl != NULL);

1815        /* Convert the handle to an Isoch CEC pointer */
1816        cec_curr = (s1394_isoch_cec_t *)(*t1394_single_hdl);

1818        /*
1819         * Teardown the isoch resources, etc.
1820         */
1821        ret = t1394_teardown_isoch_cec(t1394_hdl,
1822            (t1394_isoch_cec_handle_t)cec_curr, 0);
1823        if (ret != DDI_SUCCESS) {
1824                /* Unable to teardown the Isoch CEC */
1825                TNF_PROBE_1(t1394_free_isoch_single_error,
1826                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1827                    "Unexpected error from t1394_teardown_isoch_cec()");
1828                ASSERT(0);
1829        }

1831        /*
1832         * Leave the Isoch CEC
1833         */
1834        ret = t1394_leave_isoch_cec(t1394_hdl,
1835            (t1394_isoch_cec_handle_t)cec_curr, 0);
1836        if (ret != DDI_SUCCESS) {
1837                /* Unable to leave the Isoch CEC */
1838                TNF_PROBE_1(t1394_free_isoch_single_error,
1839                    S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1840                    "Unexpected error from t1394_leave_isoch_cec()");
1841                ASSERT(0);
```

```
1842         }

1844         /*
1845          * Free the Isoch CEC
1846          */
1847         ret = t1394_free_isoch_cec(t1394_hdl, flags,
1848             (t1394_isoch_cec_handle_t *)&cec_curr);
1849         if (ret != DDI_SUCCESS) {
1850                 /* Unable to free the Isoch CEC */
1851                 TNF_PROBE_1(t1394_free_isoch_single_error,
1852                     S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1853                     "Unexpected error from t1394_free_isoch_cec()");
1854                 ASSERT(0);
1855         }

1857         /* Handle is nulled out before returning */
1858         *t1394_single_hdl = NULL;

1860         TNF_PROBE_0_DEBUG(t1394_free_isoch_single_exit,
1861             S1394_TNF_SL_ISOCH_STACK, "");
1862 }

1864 /*
1865  * Function:    t1394_alloc_isoch_cec()
1866  * Input(s):    t1394_hdl               The target "handle" returned by
1867  *                                          t1394_attach()
1868  *              props                   The structure used to set up the
1869  *                                          overall characteristics of for
1870  *                                          the Isoch CEC.
1871  *              flags                   The flags parameter is unused (for now)
1872  *
1873  * Output(s):   t1394_isoch_cec_hdl     The Isoch CEC "handle" used in all
1874  *                                          subsequent isoch_cec() calls
1875  *
1876  * Description: t1394_alloc_isoch_cec() allocates and initializes an
1877  *              isochronous channel event coordinator (Isoch CEC) for use
1878  *              in managing and coordinating activity for an isoch channel
1879  */
1880 /* ARGSUSED */
1881 int
1882 t1394_alloc_isoch_cec(t1394_handle_t t1394_hdl, t1394_isoch_cec_props_t *props,
1883     uint_t flags, t1394_isoch_cec_handle_t *t1394_isoch_cec_hdl)
1884 {
1885         s1394_hal_t       *hal;
1886         s1394_isoch_cec_t *cec_new;
1887         uint64_t          temp;

1889         TNF_PROBE_0_DEBUG(t1394_alloc_isoch_cec_enter,
1890             S1394_TNF_SL_ISOCH_STACK, "");

1892         ASSERT(t1394_hdl != NULL);
1893         ASSERT(t1394_isoch_cec_hdl != NULL);
1894         ASSERT(props != NULL);

1896         hal = ((s1394_target_t *)t1394_hdl)->on_hal;

1898         /* Check for invalid channel_mask */
1899         if (props->cec_channel_mask == 0) {
1900                 TNF_PROBE_1(t1394_alloc_isoch_cec_error,
1901                     S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1902                     "Invalid channel mask");
1903                 TNF_PROBE_0_DEBUG(t1394_alloc_isoch_cec_exit,
1904                     S1394_TNF_SL_ISOCH_STACK, "");
1905                 return (DDI_FAILURE);
1906         }
```



```
1908         /* Test conditions specific to T1394_NO_IRM_ALLOC */
1909         temp = props->cec_channel_mask;
1910         if (props->cec_options & T1394_NO_IRM_ALLOC) {
1911                 /* If T1394_NO_IRM_ALLOC, then only one bit should be set */
1912                 if (!ISP2(temp)) {
  35                 if ((temp & (temp - 1)) != 0) {
1913                         TNF_PROBE_1(t1394_alloc_isoch_cec_error,
1914                             S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1915                             "Invalid channel mask");
1916                         TNF_PROBE_0_DEBUG(t1394_alloc_isoch_cec_exit,
1917                             S1394_TNF_SL_ISOCH_STACK, "");
1918                         return (DDI_FAILURE);
1919                 }

1921                 /* If T1394_NO_IRM_ALLOC, then speeds should be equal */
1922                 if (props->cec_min_speed != props->cec_max_speed) {
1923                         TNF_PROBE_1(t1394_alloc_isoch_cec_error,
1924                             S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1925                             "Invalid speeds (min != max)");
1926                         TNF_PROBE_0_DEBUG(t1394_alloc_isoch_cec_exit,
1927                             S1394_TNF_SL_ISOCH_STACK, "");
1928                         return (DDI_FAILURE);
1929                 }
1930         }

1932         /* Check for invalid bandwidth */
1933         if ((props->cec_bandwidth <= IEEE1394_BANDWIDTH_MIN) ||
1934             (props->cec_bandwidth > IEEE1394_BANDWIDTH_MAX)) {
1935                 TNF_PROBE_1(t1394_alloc_isoch_cec_error,
1936                     S1394_TNF_SL_ISOCH_ERROR, "", tnf_string, msg,
1937                     "Invalid bandwidth requirements");
1938                 TNF_PROBE_0_DEBUG(t1394_alloc_isoch_cec_exit,
1939                     S1394_TNF_SL_ISOCH_STACK, "");
1940                 return (DDI_FAILURE);
1941         }

1943         /* Allocate the Isoch CEC structure */
1944         cec_new = kmem_zalloc(sizeof (s1394_isoch_cec_t), KM_SLEEP);

1946         /* Initialize the structure type */
1947         cec_new->cec_type = S1394_PEER_TO_PEER;

1949         /* Create the mutex and "in_callbacks" cv */
1950         mutex_init(&cec_new->isoch_cec_mutex, NULL, MUTEX_DRIVER,
1951             hal->halinfo.hw_interrupt);
1952         cv_init(&cec_new->in_callbacks_cv, NULL, CV_DRIVER,
1953             hal->halinfo.hw_interrupt);

1955         /* Initialize the Isoch CEC's member list */
1956         cec_new->cec_member_list_head   = NULL;
1957         cec_new->cec_member_list_tail   = NULL;

1959         /* Initialize the filters */
1960         cec_new->filter_min_speed       = props->cec_min_speed;
1961         cec_new->filter_max_speed       = props->cec_max_speed;
1962         cec_new->filter_current_speed   = cec_new->filter_max_speed;
1963         cec_new->filter_channel_mask    = props->cec_channel_mask;
1964         cec_new->bandwidth              = props->cec_bandwidth;
1965         cec_new->cec_options            = props->cec_options;
1966         cec_new->state_transitions      = ISOCH_CEC_FREE | ISOCH_CEC_JOIN |
1967                                           ISOCH_CEC_SETUP;

1969         mutex_enter(&hal->isoch_cec_list_mutex);

1971         /* Insert Isoch CEC into the HAL's list */
1972         s1394_isoch_cec_list_insert(hal, cec_new);
```

```
1974         mutex_exit(&hal->isoch_cec_list_mutex);

1976         /* Update the handle and return */
1977         *t1394_isoch_cec_hdl = (t1394_isoch_cec_handle_t)cec_new;

1979         TNF_PROBE_0_DEBUG(t1394_alloc_isoch_cec_exit,
1980             S1394_TNF_SL_ISOCH_STACK, "");
1981         return (DDI_SUCCESS);
1982 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   89603 Thu Oct 23 10:42:11 2014
new/usr/src/uts/common/io/arn/arn_main.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2008 Atheros Communications Inc.
 *
 * Permission to use, copy, modify, and/or distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/param.h>
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/stream.h>
#include <sys/termio.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/cmn_err.h>
#include <sys/stropts.h>
#include <sys/strsubr.h>
#include <sys/strtty.h>
#include <sys/kbio.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/consdev.h>
#include <sys/kmem.h>
#include <sys/modctl.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/pci.h>
#include <sys/errno.h>
#include <sys/mac_provider.h>
#include <sys/dlpi.h>
#include <sys/ethernet.h>
#include <sys/list.h>
#include <sys/byteorder.h>
#include <sys/strsun.h>
#include <sys/policy.h>
#include <inet/common.h>
#include <inet/nd.h>
#include <inet/mi.h>
#include <inet/wifi_ioctl.h>
#include <sys/mac_wifi.h>
#include <sys/net80211.h>
#include <sys/net80211_proto.h>
#include <sys/net80211_ht.h>
```



```
#include "arn_ath9k.h"
#include "arn_core.h"
#include "arn_reg.h"
#include "arn_hw.h"

#define ARN_MAX_RSSI    45      /* max rssi */

/*
 * Default 11n reates supported by this station.
 */
extern struct ieee80211_htrateset ieee80211_rateset_11n;

/*
 * PIO access attributes for registers
 */
static ddi_device_acc_attr_t arn_reg_accattr = {
        DDI_DEVICE_ATTR_V0,
        DDI_STRUCTURE_LE_ACC,
        DDI_STRICTORDER_ACC,
        DDI_DEFAULT_ACC
};

/*
 * DMA access attributes for descriptors: NOT to be byte swapped.
 */
static ddi_device_acc_attr_t arn_desc_accattr = {
        DDI_DEVICE_ATTR_V0,
        DDI_STRUCTURE_LE_ACC,
        DDI_STRICTORDER_ACC,
        DDI_DEFAULT_ACC
};

/*
 * Describes the chip's DMA engine
 */
static ddi_dma_attr_t arn_dma_attr = {
        DMA_ATTR_V0,    /* version number */
        0,                              /* low address */
        0xffffffffU,    /* high address */
        0x3ffffU,               /* counter register max */
        1,                              /* alignment */
        0xFFF,                  /* burst sizes */
        1,                              /* minimum transfer size */
        0x3ffffU,               /* max transfer size */
        0xffffffffU,    /* address register max */
        1,                              /* no scatter-gather */
        1,                              /* granularity of device */
        0,                              /* DMA flags */
};

static ddi_dma_attr_t arn_desc_dma_attr = {
        DMA_ATTR_V0,    /* version number */
        0,                              /* low address */
        0xffffffffU,    /* high address */
        0xffffffffU,    /* counter register max */
        0x1000,                 /* alignment */
        0xFFF,                  /* burst sizes */
        1,                              /* minimum transfer size */
        0xffffffffU,    /* max transfer size */
        0xffffffffU,    /* address register max */
        1,                              /* no scatter-gather */
        1,                              /* granularity of device */
        0,                              /* DMA flags */
};

#define ATH_DEF_CACHE_BYTES     32 /* default cache line size */
```

```
129 static kmutex_t arn_loglock;
130 static void *arn_soft_state_p = NULL;
131 static int arn_dwelltime = 200; /* scan interval */

133 static int      arn_m_stat(void *,  uint_t, uint64_t *);
134 static int      arn_m_start(void *);
135 static void     arn_m_stop(void *);
136 static int      arn_m_promisc(void *, boolean_t);
137 static int      arn_m_multicst(void *, boolean_t, const uint8_t *);
138 static int      arn_m_unicst(void *, const uint8_t *);
139 static mblk_t   *arn_m_tx(void *, mblk_t *);
140 static void     arn_m_ioctl(void *, queue_t *, mblk_t *);
141 static int      arn_m_setprop(void *, const char *, mac_prop_id_t,
142     uint_t, const void *);
143 static int      arn_m_getprop(void *, const char *, mac_prop_id_t,
144     uint_t, void *);
145 static void     arn_m_propinfo(void *, const char *, mac_prop_id_t,
146     mac_prop_info_handle_t);

148 /* MAC Callcack Functions */
149 static mac_callbacks_t arn_m_callbacks = {
150         MC_IOCTL | MC_SETPROP | MC_GETPROP | MC_PROPINFO,
151         arn_m_stat,
152         arn_m_start,
153         arn_m_stop,
154         arn_m_promisc,
155         arn_m_multicst,
156         arn_m_unicst,
157         arn_m_tx,
158         NULL,
159         arn_m_ioctl,
160         NULL,
161         NULL,
162         NULL,
163         arn_m_setprop,
164         arn_m_getprop,
165         arn_m_propinfo
166 };

168 /*
169  * ARN_DBG_HW
170  * ARN_DBG_REG_IO
171  * ARN_DBG_QUEUE
172  * ARN_DBG_EEPROM
173  * ARN_DBG_XMIT
174  * ARN_DBG_RECV
175  * ARN_DBG_CALIBRATE
176  * ARN_DBG_CHANNEL
177  * ARN_DBG_INTERRUPT
178  * ARN_DBG_REGULATORY
179  * ARN_DBG_ANI
180  * ARN_DBG_POWER_MGMT
181  * ARN_DBG_KEYCACHE
182  * ARN_DBG_BEACON
183  * ARN_DBG_RATE
184  * ARN_DBG_INIT
185  * ARN_DBG_ATTACH
186  * ARN_DBG_DEATCH
187  * ARN_DBG_AGGR
188  * ARN_DBG_RESET
189  * ARN_DBG_FATAL
190  * ARN_DBG_ANY
191  * ARN_DBG_ALL
192  */
193 uint32_t arn_dbg_mask = 0;
```



```
195 /*
196  * Exception/warning cases not leading to panic.
197  */
198 void
199 arn_problem(const int8_t *fmt, ...)
200 {
201         va_list args;

203         mutex_enter(&arn_loglock);

205         va_start(args, fmt);
206         vcmn_err(CE_WARN, fmt, args);
207         va_end(args);

209         mutex_exit(&arn_loglock);
210 }

212 /*
213  * Normal log information independent of debug.
214  */
215 void
216 arn_log(const int8_t *fmt, ...)
217 {
218         va_list args;

220         mutex_enter(&arn_loglock);

222         va_start(args, fmt);
223         vcmn_err(CE_CONT, fmt, args);
224         va_end(args);

226         mutex_exit(&arn_loglock);
227 }

229 void
230 arn_dbg(uint32_t dbg_flags, const int8_t *fmt, ...)
231 {
232         va_list args;

234         if (dbg_flags & arn_dbg_mask) {
235                 mutex_enter(&arn_loglock);
236                 va_start(args, fmt);
237                 vcmn_err(CE_CONT, fmt, args);
238                 va_end(args);
239                 mutex_exit(&arn_loglock);
240         }
241 }

243 /*
244  * Read and write, they both share the same lock. We do this to serialize
245  * reads and writes on Atheros 802.11n PCI devices only. This is required
246  * as the FIFO on these devices can only accept sanely 2 requests. After
247  * that the device goes bananas. Serializing the reads/writes prevents this
248  * from happening.
249  */
250 void
251 arn_iowrite32(struct ath_hal *ah, uint32_t reg_offset, uint32_t val)
252 {
253         struct arn_softc *sc = ah->ah_sc;
254         if (ah->ah_config.serialize_regmode == SER_REG_MODE_ON) {
255                 mutex_enter(&sc->sc_serial_rw);
256                 ddi_put32(sc->sc_io_handle,
257                     (uint32_t *)((uintptr_t)(sc->mem) + (reg_offset)), val);
258                 mutex_exit(&sc->sc_serial_rw);
259         } else {
```

```
260                 ddi_put32(sc->sc_io_handle,
261                     (uint32_t *)((uintptr_t)(sc->mem) + (reg_offset)), val);
262         }
263 }

265 unsigned int
266 arn_ioread32(struct ath_hal *ah, uint32_t reg_offset)
267 {
268         uint32_t val;
269         struct arn_softc *sc = ah->ah_sc;
270         if (ah->ah_config.serialize_regmode == SER_REG_MODE_ON) {
271                 mutex_enter(&sc->sc_serial_rw);
272                 val = ddi_get32(sc->sc_io_handle,
273                     (uint32_t *)((uintptr_t)(sc->mem) + (reg_offset)));
274                 mutex_exit(&sc->sc_serial_rw);
275         } else {
276                 val = ddi_get32(sc->sc_io_handle,
277                     (uint32_t *)((uintptr_t)(sc->mem) + (reg_offset)));
278         }

280         return (val);
281 }

283 /*
284  * Allocate an area of memory and a DMA handle for accessing it
285  */
286 static int
287 arn_alloc_dma_mem(dev_info_t *devinfo, ddi_dma_attr_t *dma_attr, size_t memsize,
288     ddi_device_acc_attr_t *attr_p, uint_t alloc_flags,
289     uint_t bind_flags, dma_area_t *dma_p)
290 {
291         int err;

293         /*
294          * Allocate handle
295          */
296         err = ddi_dma_alloc_handle(devinfo, dma_attr,
297             DDI_DMA_SLEEP, NULL, &dma_p->dma_hdl);
298         if (err != DDI_SUCCESS)
299                 return (DDI_FAILURE);

301         /*
302          * Allocate memory
303          */
304         err = ddi_dma_mem_alloc(dma_p->dma_hdl, memsize, attr_p,
305             alloc_flags, DDI_DMA_SLEEP, NULL, &dma_p->mem_va,
306             &dma_p->alength, &dma_p->acc_hdl);
307         if (err != DDI_SUCCESS)
308                 return (DDI_FAILURE);

310         /*
311          * Bind the two together
312          */
313         err = ddi_dma_addr_bind_handle(dma_p->dma_hdl, NULL,
314             dma_p->mem_va, dma_p->alength, bind_flags,
315             DDI_DMA_SLEEP, NULL, &dma_p->cookie, &dma_p->ncookies);
316         if (err != DDI_DMA_MAPPED)
317                 return (DDI_FAILURE);

319         dma_p->nslots = ~0U;
320         dma_p->size = ~0U;
321         dma_p->token = ~0U;
322         dma_p->offset = 0;
323         return (DDI_SUCCESS);
324 }
```



```
326 /*
327  * Free one allocated area of DMAable memory
328  */
329 static void
330 arn_free_dma_mem(dma_area_t *dma_p)
331 {
332         if (dma_p->dma_hdl != NULL) {
333                 (void) ddi_dma_unbind_handle(dma_p->dma_hdl);
334                 if (dma_p->acc_hdl != NULL) {
335                         ddi_dma_mem_free(&dma_p->acc_hdl);
336                         dma_p->acc_hdl = NULL;
337                 }
338                 ddi_dma_free_handle(&dma_p->dma_hdl);
339                 dma_p->ncookies = 0;
340                 dma_p->dma_hdl = NULL;
341         }
342 }

344 /*
345  * Initialize tx, rx. or beacon buffer list. Allocate DMA memory for
346  * each buffer.
347  */
348 static int
349 arn_buflist_setup(dev_info_t *devinfo,
350     struct arn_softc *sc,
351     list_t *bflist,
352     struct ath_buf **pbf,
353     struct ath_desc **pds,
354     int nbuf,
355     uint_t dmabflags,
356     uint32_t buflen)
357 {
358         int i, err;
359         struct ath_buf *bf = *pbf;
360         struct ath_desc *ds = *pds;

362         list_create(bflist, sizeof (struct ath_buf),
363             offsetof(struct ath_buf, bf_node));
364         for (i = 0; i < nbuf; i++, bf++, ds++) {
365                 bf->bf_desc = ds;
366                 bf->bf_daddr = sc->sc_desc_dma.cookie.dmac_address +
367                     ((uintptr_t)ds - (uintptr_t)sc->sc_desc);
368                 list_insert_tail(bflist, bf);

370                 /* alloc DMA memory */
371                 err = arn_alloc_dma_mem(devinfo, &arn_dma_attr,
372                     buflen, &arn_desc_accattr, DDI_DMA_STREAMING,
373                     dmabflags, &bf->bf_dma);
374                 if (err != DDI_SUCCESS)
375                         return (err);
376         }
377         *pbf = bf;
378         *pds = ds;

380         return (DDI_SUCCESS);
381 }

383 /*
384  * Destroy tx, rx or beacon buffer list. Free DMA memory.
385  */
386 static void
387 arn_buflist_cleanup(list_t *buflist)
388 {
389         struct ath_buf *bf;

391         if (!buflist)
```

```
392                 return;

394         bf = list_head(buflist);
395         while (bf != NULL) {
396                 if (bf->bf_m != NULL) {
397                         freemsg(bf->bf_m);
398                         bf->bf_m = NULL;
399                 }
400                 /* Free DMA buffer */
401                 arn_free_dma_mem(&bf->bf_dma);
402                 if (bf->bf_in != NULL) {
403                         ieee80211_free_node(bf->bf_in);
404                         bf->bf_in = NULL;
405                 }
406                 list_remove(buflist, bf);
407                 bf = list_head(buflist);
408         }
409         list_destroy(buflist);
410 }

412 static void
413 arn_desc_free(struct arn_softc *sc)
414 {
415         arn_buflist_cleanup(&sc->sc_txbuf_list);
416         arn_buflist_cleanup(&sc->sc_rxbuf_list);
417 #ifdef ARN_IBSS
418         arn_buflist_cleanup(&sc->sc_bcbuf_list);
419 #endif

421         /* Free descriptor DMA buffer */
422         arn_free_dma_mem(&sc->sc_desc_dma);

424         kmem_free((void *)sc->sc_vbufptr, sc->sc_vbuflen);
425         sc->sc_vbufptr = NULL;
426 }

428 static int
429 arn_desc_alloc(dev_info_t *devinfo, struct arn_softc *sc)
430 {
431         int err;
432         size_t size;
433         struct ath_desc *ds;
434         struct ath_buf *bf;

436 #ifdef ARN_IBSS
437         size = sizeof (struct ath_desc) * (ATH_TXBUF + ATH_RXBUF + ATH_BCBUF);
438 #else
439         size = sizeof (struct ath_desc) * (ATH_TXBUF + ATH_RXBUF);
440 #endif

442         err = arn_alloc_dma_mem(devinfo, &arn_desc_dma_attr, size,
443             &arn_desc_accattr, DDI_DMA_CONSISTENT,
444             DDI_DMA_RDWR | DDI_DMA_CONSISTENT, &sc->sc_desc_dma);

446         /* virtual address of the first descriptor */
447         sc->sc_desc = (struct ath_desc *)sc->sc_desc_dma.mem_va;

449         ds = sc->sc_desc;
450         ARN_DBG((ARN_DBG_INIT, "arn: arn_desc_alloc(): DMA map: "
451             "%p (%d) -> %p\n",
452             sc->sc_desc, sc->sc_desc_dma.alength,
453             sc->sc_desc_dma.cookie.dmac_address));

455         /* allocate data structures to describe TX/RX DMA buffers */
456 #ifdef ARN_IBSS
457         sc->sc_vbuflen = sizeof (struct ath_buf) * (ATH_TXBUF + ATH_RXBUF +
```



```
458             ATH_BCBUF);
459 #else
460         sc->sc_vbuflen = sizeof (struct ath_buf) * (ATH_TXBUF + ATH_RXBUF);
461 #endif
462         bf = (struct ath_buf *)kmem_zalloc(sc->sc_vbuflen, KM_SLEEP);
463         sc->sc_vbufptr = bf;

465         /* DMA buffer size for each TX/RX packet */
466 #ifdef ARN_TX_AGGREGRATION
467         sc->tx_dmabuf_size =
468             roundup((IEEE80211_MAX_MPDU_LEN + 3840 * 2),
469             min(sc->sc_cachelsz, (uint16_t)64));
470 #else
471         sc->tx_dmabuf_size =
472             roundup(IEEE80211_MAX_MPDU_LEN, min(sc->sc_cachelsz, (uint16_t)64));
473 #endif
474         sc->rx_dmabuf_size =
475             roundup(IEEE80211_MAX_MPDU_LEN, min(sc->sc_cachelsz, (uint16_t)64));

477         /* create RX buffer list */
478         err = arn_buflist_setup(devinfo, sc, &sc->sc_rxbuf_list, &bf, &ds,
479             ATH_RXBUF, DDI_DMA_READ | DDI_DMA_STREAMING, sc->rx_dmabuf_size);
480         if (err != DDI_SUCCESS) {
481                 arn_desc_free(sc);
482                 return (err);
483         }

485         /* create TX buffer list */
486         err = arn_buflist_setup(devinfo, sc, &sc->sc_txbuf_list, &bf, &ds,
487             ATH_TXBUF, DDI_DMA_STREAMING, sc->tx_dmabuf_size);
488         if (err != DDI_SUCCESS) {
489                 arn_desc_free(sc);
490                 return (err);
491         }

493         /* create beacon buffer list */
494 #ifdef ARN_IBSS
495         err = arn_buflist_setup(devinfo, sc, &sc->sc_bcbuf_list, &bf, &ds,
496             ATH_BCBUF, DDI_DMA_STREAMING);
497         if (err != DDI_SUCCESS) {
498                 arn_desc_free(sc);
499                 return (err);
500         }
501 #endif

503         return (DDI_SUCCESS);
504 }

506 static struct ath_rate_table *
507 /* LINTED E_STATIC_UNUSED */
508 arn_get_ratetable(struct arn_softc *sc, uint32_t mode)
509 {
510         struct ath_rate_table *rate_table = NULL;

512         switch (mode) {
513         case IEEE80211_MODE_11A:
514                 rate_table = sc->hw_rate_table[ATH9K_MODE_11A];
515                 break;
516         case IEEE80211_MODE_11B:
517                 rate_table = sc->hw_rate_table[ATH9K_MODE_11B];
518                 break;
519         case IEEE80211_MODE_11G:
520                 rate_table = sc->hw_rate_table[ATH9K_MODE_11G];
521                 break;
522 #ifdef ARB_11N
523         case IEEE80211_MODE_11NA_HT20:
```

```
524                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT20];
525                 break;
526         case IEEE80211_MODE_11NG_HT20:
527                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT20];
528                 break;
529         case IEEE80211_MODE_11NA_HT40PLUS:
530                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT40PLUS];
531                 break;
532         case IEEE80211_MODE_11NA_HT40MINUS:
533                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT40MINUS];
534                 break;
535         case IEEE80211_MODE_11NG_HT40PLUS:
536                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT40PLUS];
537                 break;
538         case IEEE80211_MODE_11NG_HT40MINUS:
539                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT40MINUS];
540                 break;
541 #endif
542         default:
543                 ARN_DBG((ARN_DBG_FATAL, "arn: arn_get_ratetable(): "
544                     "invalid mode %u\n", mode));
545                 return (NULL);
546         }

548         return (rate_table);

550 }

552 static void
553 arn_setcurmode(struct arn_softc *sc, enum wireless_mode mode)
554 {
555         struct ath_rate_table *rt;
556         int i;

558         for (i = 0; i < sizeof (sc->asc_rixmap); i++)
559                 sc->asc_rixmap[i] = 0xff;

561         rt = sc->hw_rate_table[mode];
562         ASSERT(rt != NULL);

564         for (i = 0; i < rt->rate_cnt; i++)
565                 sc->asc_rixmap[rt->info[i].dot11rate &
566                     IEEE80211_RATE_VAL] = (uint8_t)i; /* LINT */

568         sc->sc_currates = rt;
569         sc->sc_curmode = mode;

571         /*
572          * All protection frames are transmited at 2Mb/s for
573          * 11g, otherwise at 1Mb/s.
574          * XXX select protection rate index from rate table.
575          */
576         sc->sc_protrix = (mode == ATH9K_MODE_11G ? 1 : 0);
577 }

579 static enum wireless_mode
580 arn_chan2mode(struct ath9k_channel *chan)
581 {
582         if (chan->chanmode == CHANNEL_A)
583                 return (ATH9K_MODE_11A);
584         else if (chan->chanmode == CHANNEL_G)
585                 return (ATH9K_MODE_11G);
586         else if (chan->chanmode == CHANNEL_B)
587                 return (ATH9K_MODE_11B);
588         else if (chan->chanmode == CHANNEL_A_HT20)
589                 return (ATH9K_MODE_11NA_HT20);
```



```
590         else if (chan->chanmode == CHANNEL_G_HT20)
591                 return (ATH9K_MODE_11NG_HT20);
592         else if (chan->chanmode == CHANNEL_A_HT40PLUS)
593                 return (ATH9K_MODE_11NA_HT40PLUS);
594         else if (chan->chanmode == CHANNEL_A_HT40MINUS)
595                 return (ATH9K_MODE_11NA_HT40MINUS);
596         else if (chan->chanmode == CHANNEL_G_HT40PLUS)
597                 return (ATH9K_MODE_11NG_HT40PLUS);
598         else if (chan->chanmode == CHANNEL_G_HT40MINUS)
599                 return (ATH9K_MODE_11NG_HT40MINUS);

601         return (ATH9K_MODE_11B);
602 }

604 static void
605 arn_update_txpow(struct arn_softc *sc)
606 {
607         struct ath_hal  *ah = sc->sc_ah;
608         uint32_t txpow;

610         if (sc->sc_curtxpow != sc->sc_config.txpowlimit) {
611                 (void) ath9k_hw_set_txpowerlimit(ah, sc->sc_config.txpowlimit);
612                 /* read back in case value is clamped */
613                 (void) ath9k_hw_getcapability(ah, ATH9K_CAP_TXPOW, 1, &txpow);
614                 sc->sc_curtxpow = (uint32_t)txpow;
615         }
616 }

618 uint8_t
619 parse_mpdudensity(uint8_t mpdudensity)
620 {
621         /*
622          * 802.11n D2.0 defined values for "Minimum MPDU Start Spacing":
623          *   0 for no restriction
624          *   1 for 1/4 us
625          *   2 for 1/2 us
626          *   3 for 1 us
627          *   4 for 2 us
628          *   5 for 4 us
629          *   6 for 8 us
630          *   7 for 16 us
631          */
632         switch (mpdudensity) {
633         case 0:
634                 return (0);
635         case 1:
636         case 2:
637         case 3:
638                 /*
639                  * Our lower layer calculations limit our
640                  * precision to 1 microsecond
641                  */
642                 return (1);
643         case 4:
644                 return (2);
645         case 5:
646                 return (4);
647         case 6:
648                 return (8);
649         case 7:
650                 return (16);
651         default:
652                 return (0);
653         }
654 }
```

```
656 static void
657 arn_setup_rates(struct arn_softc *sc, uint32_t mode)
658 {
659         int i, maxrates;
660         struct ath_rate_table *rate_table = NULL;
661         struct ieee80211_rateset *rateset;
662         ieee80211com_t *ic = (ieee80211com_t *)sc;

664         /* rate_table = arn_get_ratetable(sc, mode); */
665         switch (mode) {
666         case IEEE80211_MODE_11A:
667                 rate_table = sc->hw_rate_table[ATH9K_MODE_11A];
668                 break;
669         case IEEE80211_MODE_11B:
670                 rate_table = sc->hw_rate_table[ATH9K_MODE_11B];
671                 break;
672         case IEEE80211_MODE_11G:
673                 rate_table = sc->hw_rate_table[ATH9K_MODE_11G];
674                 break;
675 #ifdef ARN_11N
676         case IEEE80211_MODE_11NA_HT20:
677                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT20];
678                 break;
679         case IEEE80211_MODE_11NG_HT20:
680                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT20];
681                 break;
682         case IEEE80211_MODE_11NA_HT40PLUS:
683                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT40PLUS];
684                 break;
685         case IEEE80211_MODE_11NA_HT40MINUS:
686                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT40MINUS];
687                 break;
688         case IEEE80211_MODE_11NG_HT40PLUS:
689                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT40PLUS];
690                 break;
691         case IEEE80211_MODE_11NG_HT40MINUS:
692                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT40MINUS];
693                 break;
694 #endif
695         default:
696                 ARN_DBG((ARN_DBG_RATE, "arn: arn_get_ratetable(): "
697                     "invalid mode %u\n", mode));
698                 break;
699         }
700         if (rate_table == NULL)
701                 return;
702         if (rate_table->rate_cnt > ATH_RATE_MAX) {
703                 ARN_DBG((ARN_DBG_RATE, "arn: arn_rate_setup(): "
704                     "rate table too small (%u > %u)\n",
705                     rate_table->rate_cnt, IEEE80211_RATE_MAXSIZE));
706                 maxrates = ATH_RATE_MAX;
707         } else
708                 maxrates = rate_table->rate_cnt;

710         ARN_DBG((ARN_DBG_RATE, "arn: arn_rate_setup(): "
711             "maxrates is %d\n", maxrates));

713         rateset = &ic->ic_sup_rates[mode];
714         for (i = 0; i < maxrates; i++) {
715                 rateset->ir_rates[i] = rate_table->info[i].dot11rate;
716                 ARN_DBG((ARN_DBG_RATE, "arn: arn_rate_setup(): "
717                     "%d\n", rate_table->info[i].dot11rate));
718         }
719         rateset->ir_nrates = (uint8_t)maxrates; /* ??? */
720 }
```



```
722 static int
723 arn_setup_channels(struct arn_softc *sc)
724 {
725         struct ath_hal *ah = sc->sc_ah;
726         ieee80211com_t *ic = (ieee80211com_t *)sc;
727         int nchan, i, index;
728         uint8_t regclassids[ATH_REGCLASSIDS_MAX];
729         uint32_t nregclass = 0;
730         struct ath9k_channel *c;

732         /* Fill in ah->ah_channels */
733         if (!ath9k_regd_init_channels(ah, ATH_CHAN_MAX, (uint32_t *)&nchan,
734             regclassids, ATH_REGCLASSIDS_MAX, &nregclass, CTRY_DEFAULT,
735             B_FALSE, 1)) {
736                 uint32_t rd = ah->ah_currentRD;
737                 ARN_DBG((ARN_DBG_CHANNEL, "arn: arn_setup_channels(): "
738                     "unable to collect channel list; "
739                     "regdomain likely %u country code %u\n",
740                     rd, CTRY_DEFAULT));
741                 return (EINVAL);
742         }

744         ARN_DBG((ARN_DBG_CHANNEL, "arn: arn_setup_channels(): "
745             "number of channel is %d\n", nchan));

747         for (i = 0; i < nchan; i++) {
748                 c = &ah->ah_channels[i];
749                 uint32_t flags;
750                 index = ath9k_hw_mhz2ieee(ah, c->channel, c->channelFlags);

752                 if (index > IEEE80211_CHAN_MAX) {
753                         ARN_DBG((ARN_DBG_CHANNEL,
754                             "arn: arn_setup_channels(): "
755                             "bad hal channel %d (%u/%x) ignored\n",
756                             index, c->channel, c->channelFlags));
757                         continue;
758                 }
759                 /* NB: flags are known to be compatible */
760                 if (index < 0) {
761                         /*
762                          * can't handle frequency <2400MHz (negative
763                          * channels) right now
764                          */
765                         ARN_DBG((ARN_DBG_CHANNEL,
766                             "arn: arn_setup_channels(): "
767                             "hal channel %d (%u/%x) "
768                             "cannot be handled, ignored\n",
769                             index, c->channel, c->channelFlags));
770                         continue;
771                 }

773                 /*
774                  * Calculate net80211 flags; most are compatible
775                  * but some need massaging.  Note the static turbo
776                  * conversion can be removed once net80211 is updated
777                  * to understand static vs. dynamic turbo.
778                  */

780                 flags = c->channelFlags & (CHANNEL_ALL | CHANNEL_PASSIVE);

782                 if (ic->ic_sup_channels[index].ich_freq == 0) {
783                         ic->ic_sup_channels[index].ich_freq = c->channel;
784                         ic->ic_sup_channels[index].ich_flags = flags;
785                 } else {
786                         /* channels overlap; e.g. 11g and 11b */
787                         ic->ic_sup_channels[index].ich_flags |= flags;
```

```
788                 }
789                 if ((c->channelFlags & CHANNEL_G) == CHANNEL_G) {
790                         sc->sc_have11g = 1;
791                         ic->ic_caps |= IEEE80211_C_SHPREAMBLE |
792                             IEEE80211_C_SHSLOT; /* short slot time */
793                 }
794         }

796         return (0);
797 }

799 uint32_t
800 arn_chan2flags(ieee80211com_t *isc, struct ieee80211_channel *chan)
801 {
802         uint32_t channel_mode;
803         switch (ieee80211_chan2mode(isc, chan)) {
804         case IEEE80211_MODE_11NA:
805                 if (chan->ich_flags & IEEE80211_CHAN_HT40U)
806                         channel_mode = CHANNEL_A_HT40PLUS;
807                 else if (chan->ich_flags & IEEE80211_CHAN_HT40D)
808                         channel_mode = CHANNEL_A_HT40MINUS;
809                 else
810                         channel_mode = CHANNEL_A_HT20;
811                 break;
812         case IEEE80211_MODE_11NG:
813                 if (chan->ich_flags & IEEE80211_CHAN_HT40U)
814                         channel_mode = CHANNEL_G_HT40PLUS;
815                 else if (chan->ich_flags & IEEE80211_CHAN_HT40D)
816                         channel_mode = CHANNEL_G_HT40MINUS;
817                 else
818                         channel_mode = CHANNEL_G_HT20;
819                 break;
820         case IEEE80211_MODE_TURBO_G:
821         case IEEE80211_MODE_STURBO_A:
822         case IEEE80211_MODE_TURBO_A:
823                 channel_mode = 0;
824                 break;
825         case IEEE80211_MODE_11A:
826                 channel_mode = CHANNEL_A;
827                 break;
828         case IEEE80211_MODE_11G:
829                 channel_mode = CHANNEL_B;
830                 break;
831         case IEEE80211_MODE_11B:
832                 channel_mode = CHANNEL_G;
833                 break;
834         case IEEE80211_MODE_FH:
835                 channel_mode = 0;
836                 break;
837         default:
838                 break;
839         }

841         return (channel_mode);
842 }

844 /*
845  * Update internal state after a channel change.
846  */
847 void
848 arn_chan_change(struct arn_softc *sc, struct ieee80211_channel *chan)
849 {
850         struct ieee80211com *ic = &sc->sc_isc;
851         enum ieee80211_phymode mode;
852         enum wireless_mode wlmode;
```



```
854         /*
855          * Change channels and update the h/w rate map
856          * if we're switching; e.g. 11a to 11b/g.
857          */
858         mode = ieee80211_chan2mode(ic, chan);
859         switch (mode) {
860         case IEEE80211_MODE_11A:
861                 wlmode = ATH9K_MODE_11A;
862                 break;
863         case IEEE80211_MODE_11B:
864                 wlmode = ATH9K_MODE_11B;
865                 break;
866         case IEEE80211_MODE_11G:
867                 wlmode = ATH9K_MODE_11B;
868                 break;
869         default:
870                 break;
871         }
872         if (wlmode != sc->sc_curmode)
873                 arn_setcurmode(sc, wlmode);

875 }

877 /*
878  * Set/change channels.  If the channel is really being changed, it's done
879  * by reseting the chip.  To accomplish this we must first cleanup any pending
880  * DMA, then restart stuff.
881  */
882 static int
883 arn_set_channel(struct arn_softc *sc, struct ath9k_channel *hchan)
884 {
885         struct ath_hal *ah = sc->sc_ah;
886         ieee80211com_t *ic = &sc->sc_isc;
887         boolean_t fastcc = B_TRUE;
888         boolean_t  stopped;
889         struct ieee80211_channel chan;
890         enum wireless_mode curmode;

892         if (sc->sc_flags & SC_OP_INVALID)
893                 return (EIO);

895         if (hchan->channel != sc->sc_ah->ah_curchan->channel ||
896             hchan->channelFlags != sc->sc_ah->ah_curchan->channelFlags ||
897             (sc->sc_flags & SC_OP_CHAINMASK_UPDATE) ||
898             (sc->sc_flags & SC_OP_FULL_RESET)) {
899                 int status;

901                 /*
902                  * This is only performed if the channel settings have
903                  * actually changed.
904                  *
905                  * To switch channels clear any pending DMA operations;
906                  * wait long enough for the RX fifo to drain, reset the
907                  * hardware at the new frequency, and then re-enable
908                  * the relevant bits of the h/w.
909                  */
910                 (void) ath9k_hw_set_interrupts(ah, 0);  /* disable interrupts */
911                 arn_draintxq(sc, B_FALSE);      /* clear pending tx frames */
912                 stopped = arn_stoprecv(sc);     /* turn off frame recv */

914                 /*
915                  * XXX: do not flush receive queue here. We don't want
916                  * to flush data frames already in queue because of
917                  * changing channel.
918                  */
```

```
920                 if (!stopped || (sc->sc_flags & SC_OP_FULL_RESET))
921                         fastcc = B_FALSE;

923                 ARN_DBG((ARN_DBG_CHANNEL, "arn: arn_set_channel(): "
924                     "(%u MHz) -> (%u MHz), cflags:%x, chanwidth: %d\n",
925                     sc->sc_ah->ah_curchan->channel,
926                     hchan->channel, hchan->channelFlags, sc->tx_chan_width));

928                 if (!ath9k_hw_reset(ah, hchan, sc->tx_chan_width,
929                     sc->sc_tx_chainmask, sc->sc_rx_chainmask,
930                     sc->sc_ht_extprotspacing, fastcc, &status)) {
931                         ARN_DBG((ARN_DBG_FATAL, "arn: arn_set_channel(): "
932                             "unable to reset channel %u (%uMhz) "
933                             "flags 0x%x hal status %u\n",
934                             ath9k_hw_mhz2ieee(ah, hchan->channel,
935                             hchan->channelFlags),
936                             hchan->channel, hchan->channelFlags, status));
937                         return (EIO);
938                 }

940                 sc->sc_curchan = *hchan;

942                 sc->sc_flags &= ~SC_OP_CHAINMASK_UPDATE;
943                 sc->sc_flags &= ~SC_OP_FULL_RESET;

945                 if (arn_startrecv(sc) != 0) {
946                         arn_problem("arn: arn_set_channel(): "
947                             "unable to restart recv logic\n");
948                         return (EIO);
949                 }

951                 chan.ich_freq = hchan->channel;
952                 chan.ich_flags = hchan->channelFlags;
953                 ic->ic_ibss_chan = &chan;

955                 /*
956                  * Change channels and update the h/w rate map
957                  * if we're switching; e.g. 11a to 11b/g.
958                  */
959                 curmode = arn_chan2mode(hchan);
960                 if (curmode != sc->sc_curmode)
961                         arn_setcurmode(sc, arn_chan2mode(hchan));

963                 arn_update_txpow(sc);

965                 (void) ath9k_hw_set_interrupts(ah, sc->sc_imask);
966         }

968         return (0);
969 }

971 /*
972  *  This routine performs the periodic noise floor calibration function
973  *  that is used to adjust and optimize the chip performance.  This
974  *  takes environmental changes (location, temperature) into account.
975  *  When the task is complete, it reschedules itself depending on the
976  *  appropriate interval that was calculated.
977  */
978 static void
979 arn_ani_calibrate(void *arg)

981 {
982         ieee80211com_t *ic = (ieee80211com_t *)arg;
983         struct arn_softc *sc = (struct arn_softc *)ic;
984         struct ath_hal *ah = sc->sc_ah;
985         boolean_t longcal = B_FALSE;
```



```
986         boolean_t shortcal = B_FALSE;
987         boolean_t aniflag = B_FALSE;
988         unsigned int timestamp = drv_hztousec(ddi_get_lbolt())/1000;
989         uint32_t cal_interval;

991         /*
992          * don't calibrate when we're scanning.
993          * we are most likely not on our home channel.
994          */
995         if (ic->ic_state != IEEE80211_S_RUN)
996                 goto settimer;

998         /* Long calibration runs independently of short calibration. */
999         if ((timestamp - sc->sc_ani.sc_longcal_timer) >= ATH_LONG_CALINTERVAL) {
1000                longcal = B_TRUE;
1001                ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
1002                    "%s: longcal @%lu\n", __func__, drv_hztousec));
1003                sc->sc_ani.sc_longcal_timer = timestamp;
1004        }

1006        /* Short calibration applies only while sc_caldone is FALSE */
1007        if (!sc->sc_ani.sc_caldone) {
1008                if ((timestamp - sc->sc_ani.sc_shortcal_timer) >=
1009                    ATH_SHORT_CALINTERVAL) {
1010                        shortcal = B_TRUE;
1011                        ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
1012                            "%s: shortcal @%lu\n",
1013                            __func__, drv_hztousec));
1014                        sc->sc_ani.sc_shortcal_timer = timestamp;
1015                        sc->sc_ani.sc_resetcal_timer = timestamp;
1016                }
1017        } else {
1018                if ((timestamp - sc->sc_ani.sc_resetcal_timer) >=
1019                    ATH_RESTART_CALINTERVAL) {
1020                        ath9k_hw_reset_calvalid(ah, ah->ah_curchan,
1021                                                &sc->sc_ani.sc_caldone);
1022                        if (sc->sc_ani.sc_caldone)
1023                                sc->sc_ani.sc_resetcal_timer = timestamp;
1024                }
1025        }

1027        /* Verify whether we must check ANI */
1028        if ((timestamp - sc->sc_ani.sc_checkani_timer) >=
1029            ATH_ANI_POLLINTERVAL) {
1030                aniflag = B_TRUE;
1031                sc->sc_ani.sc_checkani_timer = timestamp;
1032        }

1034        /* Skip all processing if there's nothing to do. */
1035        if (longcal || shortcal || aniflag) {
1036                /* Call ANI routine if necessary */
1037                if (aniflag)
1038                        ath9k_hw_ani_monitor(ah, &sc->sc_halstats,
1039                            ah->ah_curchan);

1041                /* Perform calibration if necessary */
1042                if (longcal || shortcal) {
1043                        boolean_t iscaldone = B_FALSE;

1045                        if (ath9k_hw_calibrate(ah, ah->ah_curchan,
1046                            sc->sc_rx_chainmask, longcal, &iscaldone)) {
1047                                if (longcal)
1048                                        sc->sc_ani.sc_noise_floor =
1049                                            ath9k_hw_getchan_noise(ah,
1050                                            ah->ah_curchan);
```

```
1052                                ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
1053                                    "%s: calibrate chan %u/%x nf: %d\n",
1054                                    __func__,
1055                                    ah->ah_curchan->channel,
1056                                    ah->ah_curchan->channelFlags,
1057                                    sc->sc_ani.sc_noise_floor));
1058                        } else {
1059                                ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
1060                                    "%s: calibrate chan %u/%x failed\n",
1061                                    __func__,
1062                                    ah->ah_curchan->channel,
1063                                    ah->ah_curchan->channelFlags));
1064                        }
1065                        sc->sc_ani.sc_caldone = iscaldone;
1066                }
1067        }

1069 settimer:
1070        /*
1071         * Set timer interval based on previous results.
1072         * The interval must be the shortest necessary to satisfy ANI,
1073         * short calibration and long calibration.
1074         */
1075        cal_interval = ATH_LONG_CALINTERVAL;
1076        if (sc->sc_ah->ah_config.enable_ani)
1077                cal_interval =
1078                    min(cal_interval, (uint32_t)ATH_ANI_POLLINTERVAL);

1080        if (!sc->sc_ani.sc_caldone)
1081                cal_interval = min(cal_interval,
1082                    (uint32_t)ATH_SHORT_CALINTERVAL);

1084        sc->sc_scan_timer = 0;
1085        sc->sc_scan_timer = timeout(arn_ani_calibrate, (void *)sc,
1086            drv_usectohz(cal_interval * 1000));
1087 }

1089 static void
1090 arn_stop_caltimer(struct arn_softc *sc)
1091 {
1092        timeout_id_t tmp_id = 0;

1094        while ((sc->sc_cal_timer != 0) && (tmp_id != sc->sc_cal_timer)) {
1095                tmp_id = sc->sc_cal_timer;
1096                (void) untimeout(tmp_id);
1097        }
1098        sc->sc_cal_timer = 0;
1099 }

1101 static uint_t
1102 arn_isr(caddr_t arg)
1103 {
1104        /* LINTED E_BAD_PTR_CAST_ALIGN */
1105        struct arn_softc *sc = (struct arn_softc *)arg;
1106        struct ath_hal *ah = sc->sc_ah;
1107        enum ath9k_int status;
1108        ieee80211com_t *ic = (ieee80211com_t *)sc;

1110        ARN_LOCK(sc);

1112        if (sc->sc_flags & SC_OP_INVALID) {
1113                /*
1114                 * The hardware is not ready/present, don't
1115                 * touch anything. Note this can happen early
1116                 * on if the IRQ is shared.
1117                 */
```



```
1118                ARN_UNLOCK(sc);
1119                return (DDI_INTR_UNCLAIMED);
1120        }
1121        if (!ath9k_hw_intrpend(ah)) {   /* shared irq, not for us */
1122                ARN_UNLOCK(sc);
1123                return (DDI_INTR_UNCLAIMED);
1124        }

1126        /*
1127         * Figure out the reason(s) for the interrupt. Note
1128         * that the hal returns a pseudo-ISR that may include
1129         * bits we haven't explicitly enabled so we mask the
1130         * value to insure we only process bits we requested.
1131         */
1132        (void) ath9k_hw_getisr(ah, &status); /* NB: clears ISR too */

1134        status &= sc->sc_imask; /* discard unasked-for bits */

1136        /*
1137         * If there are no status bits set, then this interrupt was not
1138         * for me (should have been caught above).
1139         */
1140        if (!status) {
1141                ARN_UNLOCK(sc);
1142                return (DDI_INTR_UNCLAIMED);
1143        }

1145        sc->sc_intrstatus = status;

1147        if (status & ATH9K_INT_FATAL) {
1148                /* need a chip reset */
1149                ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1150                    "ATH9K_INT_FATAL\n"));
1151                goto reset;
1152        } else if (status & ATH9K_INT_RXORN) {
1153                /* need a chip reset */
1154                ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1155                    "ATH9K_INT_RXORN\n"));
1156                goto reset;
1157        } else {
1158                if (status & ATH9K_INT_RXEOL) {
1159                        /*
1160                         * NB: the hardware should re-read the link when
1161                         * RXE bit is written, but it doesn't work
1162                         * at least on older hardware revs.
1163                         */
1164                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1165                            "ATH9K_INT_RXEOL\n"));
1166                        sc->sc_rxlink = NULL;
1167                }
1168                if (status & ATH9K_INT_TXURN) {
1169                        /* bump tx trigger level */
1170                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1171                            "ATH9K_INT_TXURN\n"));
1172                        (void) ath9k_hw_updatetxtriglevel(ah, B_TRUE);
1173                }
1174                /* XXX: optimize this */
1175                if (status & ATH9K_INT_RX) {
1176                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1177                            "ATH9K_INT_RX\n"));
1178                        sc->sc_rx_pend = 1;
1179                        ddi_trigger_softintr(sc->sc_softint_id);
1180                }
1181                if (status & ATH9K_INT_TX) {
1182                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1183                            "ATH9K_INT_TX\n"));
```

```
1184                        if (ddi_taskq_dispatch(sc->sc_tq,
1185                            arn_tx_int_proc, sc, DDI_NOSLEEP) !=
1186                            DDI_SUCCESS) {
1187                                arn_problem("arn: arn_isr(): "
1188                                    "No memory for tx taskq\n");
1189                                }
1190                        }
1191 #ifdef ARN_ATH9K_INT_MIB
1192                if (status & ATH9K_INT_MIB) {
1193                        /*
1194                         * Disable interrupts until we service the MIB
1195                         * interrupt; otherwise it will continue to
1196                         * fire.
1197                         */
1198                        (void) ath9k_hw_set_interrupts(ah, 0);
1199                        /*
1200                         * Let the hal handle the event. We assume
1201                         * it will clear whatever condition caused
1202                         * the interrupt.
1203                         */
1204                        ath9k_hw_procmibevent(ah, &sc->sc_halstats);
1205                        (void) ath9k_hw_set_interrupts(ah, sc->sc_imask);
1206                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1207                            "ATH9K_INT_MIB\n"));
1208                }
1209 #endif

1211 #ifdef ARN_ATH9K_INT_TIM_TIMER
1212                if (status & ATH9K_INT_TIM_TIMER) {
1213                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1214                            "ATH9K_INT_TIM_TIMER\n"));
1215                        if (!(ah->ah_caps.hw_caps &
1216                            ATH9K_HW_CAP_AUTOSLEEP)) {
1217                                /*
1218                                 * Clear RxAbort bit so that we can
1219                                 * receive frames
1220                                 */
1221                                ath9k_hw_setrxabort(ah, 0);
1222                                goto reset;
1223                        }
1224                }
1225 #endif

1227                if (status & ATH9K_INT_BMISS) {
1228                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1229                            "ATH9K_INT_BMISS\n"));
1230 #ifdef ARN_HW_BEACON_MISS_HANDLE
1231                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1232                            "handle beacon mmiss by H/W mechanism\n"));
1233                        if (ddi_taskq_dispatch(sc->sc_tq, arn_bmiss_proc,
1234                            sc, DDI_NOSLEEP) != DDI_SUCCESS) {
1235                                arn_problem("arn: arn_isr(): "
1236                                    "No memory available for bmiss taskq\n");
1237                        }
1238 #else
1239                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1240                            "handle beacon mmiss by S/W mechanism\n"));
1241 #endif /* ARN_HW_BEACON_MISS_HANDLE */
1242                }

1244                ARN_UNLOCK(sc);

1246 #ifdef ARN_ATH9K_INT_CST
1247                /* carrier sense timeout */
1248                if (status & ATH9K_INT_CST) {
1249                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
```



```
1250                            "ATH9K_INT_CST\n"));
1251                        return (DDI_INTR_CLAIMED);
1252                }
1253 #endif

1255                if (status & ATH9K_INT_SWBA) {
1256                        ARN_DBG((ARN_DBG_INTERRUPT, "arn: arn_isr(): "
1257                            "ATH9K_INT_SWBA\n"));
1258                        /* This will occur only in Host-AP or Ad-Hoc mode */
1259                        return (DDI_INTR_CLAIMED);
1260                }
1261        }

1263        return (DDI_INTR_CLAIMED);
1264 reset:
1265        ARN_DBG((ARN_DBG_INTERRUPT, "Rset for fatal err\n"));
1266        (void) arn_reset(ic);
1267        ARN_UNLOCK(sc);
1268        return (DDI_INTR_CLAIMED);
1269 }

1271 static int
1272 arn_get_channel(struct arn_softc *sc, struct ieee80211_channel *chan)
1273 {
1274        int i;

1276        for (i = 0; i < sc->sc_ah->ah_nchan; i++) {
1277                if (sc->sc_ah->ah_channels[i].channel == chan->ich_freq)
1278                        return (i);
1279        }

1281        return (-1);
1282 }

1284 int
1285 arn_reset(ieee80211com_t *ic)
1286 {
1287        struct arn_softc *sc = (struct arn_softc *)ic;
1288        struct ath_hal *ah = sc->sc_ah;
1289        int status;
1290        int error = 0;

1292        (void) ath9k_hw_set_interrupts(ah, 0);
1293        arn_draintxq(sc, 0);
1294        (void) arn_stoprecv(sc);

1296        if (!ath9k_hw_reset(ah, sc->sc_ah->ah_curchan, sc->tx_chan_width,
1297            sc->sc_tx_chainmask, sc->sc_rx_chainmask,
1298            sc->sc_ht_extprotspacing, B_FALSE, &status)) {
1299                ARN_DBG((ARN_DBG_RESET, "arn: arn_reset(): "
1300                    "unable to reset hardware; hal status %u\n", status));
1301                error = EIO;
1302        }

1304        if (arn_startrecv(sc) != 0)
1305                ARN_DBG((ARN_DBG_RESET, "arn: arn_reset(): "
1306                    "unable to start recv logic\n"));

1308        /*
1309         * We may be doing a reset in response to a request
1310         * that changes the channel so update any state that
1311         * might change as a result.
1312         */
1313        arn_setcurmode(sc, arn_chan2mode(sc->sc_ah->ah_curchan));

1315        arn_update_txpow(sc);
```

```
1317         if (sc->sc_flags & SC_OP_BEACONS)
1318                 arn_beacon_config(sc);  /* restart beacons */

1320         (void) ath9k_hw_set_interrupts(ah, sc->sc_imask);

1322         return (error);
1323 }

1325 int
1326 arn_get_hal_qnum(uint16_t queue, struct arn_softc *sc)
1327 {
1328         int qnum;

1330         switch (queue) {
1331         case WME_AC_VO:
1332                 qnum = sc->sc_haltype2q[ATH9K_WME_AC_VO];
1333                 break;
1334         case WME_AC_VI:
1335                 qnum = sc->sc_haltype2q[ATH9K_WME_AC_VI];
1336                 break;
1337         case WME_AC_BE:
1338                 qnum = sc->sc_haltype2q[ATH9K_WME_AC_BE];
1339                 break;
1340         case WME_AC_BK:
1341                 qnum = sc->sc_haltype2q[ATH9K_WME_AC_BK];
1342                 break;
1343         default:
1344                 qnum = sc->sc_haltype2q[ATH9K_WME_AC_BE];
1345                 break;
1346         }

1348         return (qnum);
1349 }

1351 static struct {
1352         uint32_t version;
1353         const char *name;
1354 } ath_mac_bb_names[] = {
1355         { AR_SREV_VERSION_5416_PCI,     "5416" },
1356         { AR_SREV_VERSION_5416_PCIE,    "5418" },
1357         { AR_SREV_VERSION_9100,         "9100" },
1358         { AR_SREV_VERSION_9160,         "9160" },
1359         { AR_SREV_VERSION_9280,         "9280" },
1360         { AR_SREV_VERSION_9285,         "9285" }
1361 };

1363 static struct {
1364         uint16_t version;
1365         const char *name;
1366 } ath_rf_names[] = {
1367         { 0,                            "5133" },
1368         { AR_RAD5133_SREV_MAJOR,        "5133" },
1369         { AR_RAD5122_SREV_MAJOR,        "5122" },
1370         { AR_RAD2133_SREV_MAJOR,        "2133" },
1371         { AR_RAD2122_SREV_MAJOR,        "2122" }
1372 };

1374 /*
1375  * Return the MAC/BB name. "????" is returned if the MAC/BB is unknown.
1376  */

1378 static const char *
1379 arn_mac_bb_name(uint32_t mac_bb_version)
1380 {
1381         int i;
```



```
1383         for (i = 0; i < ARRAY_SIZE(ath_mac_bb_names); i++) {
1384                 if (ath_mac_bb_names[i].version == mac_bb_version) {
1385                         return (ath_mac_bb_names[i].name);
1386                 }
1387         }

1389         return ("????");
1390 }

1392 /*
1393  * Return the RF name. "????" is returned if the RF is unknown.
1394  */

1396 static const char *
1397 arn_rf_name(uint16_t rf_version)
1398 {
1399         int i;

1401         for (i = 0; i < ARRAY_SIZE(ath_rf_names); i++) {
1402                 if (ath_rf_names[i].version == rf_version) {
1403                         return (ath_rf_names[i].name);
1404                 }
1405         }

1407         return ("????");
1408 }

1410 static void
1411 arn_next_scan(void *arg)
1412 {
1413         ieee80211com_t *ic = arg;
1414         struct arn_softc *sc = (struct arn_softc *)ic;

1416         sc->sc_scan_timer = 0;
1417         if (ic->ic_state == IEEE80211_S_SCAN) {
1418                 sc->sc_scan_timer = timeout(arn_next_scan, (void *)sc,
1419                     drv_usectohz(arn_dwelltime * 1000));
1420                 ieee80211_next_scan(ic);
1421         }
1422 }

1424 static void
1425 arn_stop_scantimer(struct arn_softc *sc)
1426 {
1427         timeout_id_t tmp_id = 0;

1429         while ((sc->sc_scan_timer != 0) && (tmp_id != sc->sc_scan_timer)) {
1430                 tmp_id = sc->sc_scan_timer;
1431                 (void) untimeout(tmp_id);
1432         }
1433         sc->sc_scan_timer = 0;
1434 }

1436 static int32_t
1437 arn_newstate(ieee80211com_t *ic, enum ieee80211_state nstate, int arg)
1438 {
1439         struct arn_softc *sc = (struct arn_softc *)ic;
1440         struct ath_hal *ah = sc->sc_ah;
1441         struct ieee80211_node *in;
1442         int32_t i, error;
1443         uint8_t *bssid;
1444         uint32_t rfilt;
1445         enum ieee80211_state ostate;
1446         struct ath9k_channel *channel;
1447         int pos;
```

```
1449         /* Should set up & init LED here */

1451         if (sc->sc_flags & SC_OP_INVALID)
1452                 return (0);

1454         ostate = ic->ic_state;
1455         ARN_DBG((ARN_DBG_INIT, "arn: arn_newstate(): "
1456             "%x -> %x!\n", ostate, nstate));

1458         ARN_LOCK(sc);

1460         if (nstate != IEEE80211_S_SCAN)
1461                 arn_stop_scantimer(sc);
1462         if (nstate != IEEE80211_S_RUN)
1463                 arn_stop_caltimer(sc);

1465         /* Should set LED here */

1467         if (nstate == IEEE80211_S_INIT) {
1468                 sc->sc_imask &= ~(ATH9K_INT_SWBA | ATH9K_INT_BMISS);
1469                 /*
1470                  * Disable interrupts.
1471                  */
1472                 (void) ath9k_hw_set_interrupts
1473                     (ah, sc->sc_imask &~ ATH9K_INT_GLOBAL);

1475 #ifdef ARN_IBSS
1476                 if (ic->ic_opmode == IEEE80211_M_IBSS) {
1477                         (void) ath9k_hw_stoptxdma(ah, sc->sc_beaconq);
1478                         arn_beacon_return(sc);
1479                 }
1480 #endif
1481                 ARN_UNLOCK(sc);
1482                 ieee80211_stop_watchdog(ic);
1483                 goto done;
1484         }
1485         in = ic->ic_bss;

1487         pos = arn_get_channel(sc, ic->ic_curchan);

1489         if (pos == -1) {
1490                 ARN_DBG((ARN_DBG_FATAL, "arn: "
1491                     "%s: Invalid channel\n", __func__));
1492                 error = EINVAL;
1493                 ARN_UNLOCK(sc);
1494                 goto bad;
1495         }

1497         if (in->in_htcap & IEEE80211_HTCAP_CHWIDTH40) {
1498                 arn_update_chainmask(sc);
1499                 sc->tx_chan_width = ATH9K_HT_MACMODE_2040;
1500         } else
1501                 sc->tx_chan_width = ATH9K_HT_MACMODE_20;

1503         sc->sc_ah->ah_channels[pos].chanmode =
1504             arn_chan2flags(ic, ic->ic_curchan);
1505         channel = &sc->sc_ah->ah_channels[pos];
1506         if (channel == NULL) {
1507                 arn_problem("arn_newstate(): channel == NULL");
1508                 ARN_UNLOCK(sc);
1509                 goto bad;
1510         }
1511         error = arn_set_channel(sc, channel);
1512         if (error != 0) {
1513                 if (nstate != IEEE80211_S_SCAN) {
```



```
1514                         ARN_UNLOCK(sc);
1515                         ieee80211_reset_chan(ic);
1516                         goto bad;
1517                 }
1518         }

1520         /*
1521          * Get the receive filter according to the
1522          * operating mode and state
1523          */
1524         rfilt = arn_calcrxfilter(sc);

1526         if (nstate == IEEE80211_S_SCAN)
1527                 bssid = ic->ic_macaddr;
1528         else
1529                 bssid = in->in_bssid;

1531         ath9k_hw_setrxfilter(ah, rfilt);

1533         if (nstate == IEEE80211_S_RUN && ic->ic_opmode != IEEE80211_M_IBSS)
1534                 ath9k_hw_write_associd(ah, bssid, in->in_associd);
1535         else
1536                 ath9k_hw_write_associd(ah, bssid, 0);

1538         /* Check for WLAN_CAPABILITY_PRIVACY ? */
1539         if (ic->ic_flags & IEEE80211_F_PRIVACY) {
1540                 for (i = 0; i < IEEE80211_WEP_NKID; i++) {
1541                         if (ath9k_hw_keyisvalid(ah, (uint16_t)i))
1542                                 (void) ath9k_hw_keysetmac(ah, (uint16_t)i,
1543                                     bssid);
1544                 }
1545         }

1547         if (nstate == IEEE80211_S_RUN) {
1548                 switch (ic->ic_opmode) {
1549 #ifdef ARN_IBSS
1550                 case IEEE80211_M_IBSS:
1551                         /*
1552                          * Allocate and setup the beacon frame.
1553                          * Stop any previous beacon DMA.
1554                          */
1555                         (void) ath9k_hw_stoptxdma(ah, sc->sc_beaconq);
1556                         arn_beacon_return(sc);
1557                         error = arn_beacon_alloc(sc, in);
1558                         if (error != 0) {
1559                                 ARN_UNLOCK(sc);
1560                                 goto bad;
1561                         }
1562                         /*
1563                          * If joining an adhoc network defer beacon timer
1564                          * configuration to the next beacon frame so we
1565                          * have a current TSF to use.  Otherwise we're
1566                          * starting an ibss/bss so there's no need to delay.
1567                          */
1568                         if (ic->ic_opmode == IEEE80211_M_IBSS &&
1569                             ic->ic_bss->in_tstamp.tsf != 0) {
1570                                 sc->sc_bsync = 1;
1571                         } else {
1572                                 arn_beacon_config(sc);
1573                         }
1574                         break;
1575 #endif /* ARN_IBSS */
1576                 case IEEE80211_M_STA:
1577                         if (ostate != IEEE80211_S_RUN) {
1578                                 /*
1579                                  * Defer beacon timer configuration to the next
```

```
1580                                 * beacon frame so we have a current TSF to use.
1581                                 * Any TSF collected when scanning is likely old
1582                                 */
1583 #ifdef ARN_IBSS
1584                                sc->sc_bsync = 1;
1585 #else
1586                                /* Configure the beacon and sleep timers. */
1587                                arn_beacon_config(sc);
1588                                /* Reset rssi stats */
1589                                sc->sc_halstats.ns_avgbrssi =
1590                                    ATH_RSSI_DUMMY_MARKER;
1591                                sc->sc_halstats.ns_avgrssi =
1592                                    ATH_RSSI_DUMMY_MARKER;
1593                                sc->sc_halstats.ns_avgtxrssi =
1594                                    ATH_RSSI_DUMMY_MARKER;
1595                                sc->sc_halstats.ns_avgtxrate =
1596                                    ATH_RATE_DUMMY_MARKER;
1597 /* end */

1599 #endif /* ARN_IBSS */
1600                        }
1601                        break;
1602                default:
1603                        break;
1604                }
1605        } else {
1606                sc->sc_imask &= ~(ATH9K_INT_SWBA | ATH9K_INT_BMISS);
1607                (void) ath9k_hw_set_interrupts(ah, sc->sc_imask);
1608        }

1610        /*
1611         * Reset the rate control state.
1612         */
1613        arn_rate_ctl_reset(sc, nstate);

1615        ARN_UNLOCK(sc);
1616 done:
1617        /*
1618         * Invoke the parent method to complete the work.
1619         */
1620        error = sc->sc_newstate(ic, nstate, arg);

1622        /*
1623         * Finally, start any timers.
1624         */
1625        if (nstate == IEEE80211_S_RUN) {
1626                ieee80211_start_watchdog(ic, 1);
1627                ASSERT(sc->sc_cal_timer == 0);
1628                sc->sc_cal_timer = timeout(arn_ani_calibrate, (void *)sc,
1629                    drv_usectohz(100 * 1000));
1630        } else if ((nstate == IEEE80211_S_SCAN) && (ostate != nstate)) {
1631                /* start ap/neighbor scan timer */
1632                /* ASSERT(sc->sc_scan_timer == 0); */
1633                if (sc->sc_scan_timer != 0) {
1634                        (void) untimeout(sc->sc_scan_timer);
1635                        sc->sc_scan_timer = 0;
1636                }
1637                sc->sc_scan_timer = timeout(arn_next_scan, (void *)sc,
1638                    drv_usectohz(arn_dwelltime * 1000));
1639        }

1641 bad:
1642        return (error);
1643 }

1645 static void
```



```
1646 arn_watchdog(void *arg)
1647 {
1648        struct arn_softc *sc = arg;
1649        ieee80211com_t *ic = &sc->sc_isc;
1650        int ntimer = 0;

1652        ARN_LOCK(sc);
1653        ic->ic_watchdog_timer = 0;
1654        if (sc->sc_flags & SC_OP_INVALID) {
1655                ARN_UNLOCK(sc);
1656                return;
1657        }

1659        if (ic->ic_state == IEEE80211_S_RUN) {
1660                /*
1661                 * Start the background rate control thread if we
1662                 * are not configured to use a fixed xmit rate.
1663                 */
1664 #ifdef ARN_LEGACY_RC
1665                if (ic->ic_fixed_rate == IEEE80211_FIXED_RATE_NONE) {
1666                        sc->sc_stats.ast_rate_calls ++;
1667                        if (ic->ic_opmode == IEEE80211_M_STA)
1668                                arn_rate_ctl(ic, ic->ic_bss);
1669                        else
1670                                ieee80211_iterate_nodes(&ic->ic_sta,
1671                                    arn_rate_ctl, sc);
1672                }
1673 #endif /* ARN_LEGACY_RC */

1675 #ifdef ARN_HW_BEACON_MISS_HANDLE
1676                /* nothing to do here */
1677 #else
1678                /* currently set 10 seconds as beacon miss threshold */
1679                if (ic->ic_beaconmiss++ > 100) {
1680                        ARN_DBG((ARN_DBG_BEACON, "arn_watchdog():"
1681                            "Beacon missed for 10 seconds, run"
1682                            "ieee80211_new_state(ic, IEEE80211_S_INIT, -1)\n"));
1683                        ARN_UNLOCK(sc);
1684                        (void) ieee80211_new_state(ic, IEEE80211_S_INIT, -1);
1685                        return;
1686                }
1687 #endif /* ARN_HW_BEACON_MISS_HANDLE */

1689                        ntimer = 1;
1690                }
1691                ARN_UNLOCK(sc);

1693                ieee80211_watchdog(ic);
1694                if (ntimer != 0)
1695                        ieee80211_start_watchdog(ic, ntimer);
1696 }

1698 /* ARGSUSED */
1699 static struct ieee80211_node *
1700 arn_node_alloc(ieee80211com_t *ic)
1701 {
1702        struct ath_node *an;
1703 #ifdef ARN_TX_AGGREGATION
1704        struct arn_softc *sc = (struct arn_softc *)ic;
1705 #endif

1707        an = kmem_zalloc(sizeof (struct ath_node), KM_SLEEP);

1709        /* legacy rate control */
1710 #ifdef ARN_LEGACY_RC
1711        arn_rate_update(sc, &an->an_node, 0);
```

```
#endif

#ifdef ARN_TX_AGGREGATION
        if (sc->sc_flags & SC_OP_TXAGGR) {
                arn_tx_node_init(sc, an);
        }
#endif /* ARN_TX_AGGREGATION */

        an->last_rssi = ATH_RSSI_DUMMY_MARKER;

        return ((an != NULL) ? &an->an_node : NULL);
}

static void
arn_node_free(struct ieee80211_node *in)
{
        ieee80211com_t *ic = in->in_ic;
        struct arn_softc *sc = (struct arn_softc *)ic;
        struct ath_buf *bf;
        struct ath_txq *txq;
        int32_t i;

#ifdef ARN_TX_AGGREGATION
        if (sc->sc_flags & SC_OP_TXAGGR)
                arn_tx_node_cleanup(sc, in);
#endif /* TX_AGGREGATION */

        for (i = 0; i < ATH9K_NUM_TX_QUEUES; i++) {
                if (ARN_TXQ_SETUP(sc, i)) {
                        txq = &sc->sc_txq[i];
                        mutex_enter(&txq->axq_lock);
                        bf = list_head(&txq->axq_list);
                        while (bf != NULL) {
                                if (bf->bf_in == in) {
                                        bf->bf_in = NULL;
                                }
                                bf = list_next(&txq->axq_list, bf);
                        }
                        mutex_exit(&txq->axq_lock);
                }
        }

        ic->ic_node_cleanup(in);

        if (in->in_wpa_ie != NULL)
                ieee80211_free(in->in_wpa_ie);

        if (in->in_wme_ie != NULL)
                ieee80211_free(in->in_wme_ie);

        if (in->in_htcap_ie != NULL)
                ieee80211_free(in->in_htcap_ie);

        kmem_free(in, sizeof (struct ath_node));
}

/*
 * Allocate tx/rx key slots for TKIP.  We allocate one slot for
 * each key. MIC is right after the decrypt/encrypt key.
 */
static uint16_t
arn_key_alloc_pair(struct arn_softc *sc, ieee80211_keyix *txkeyix,
    ieee80211_keyix *rxkeyix)
{
        uint16_t i, keyix;
```



```
        ASSERT(!sc->sc_splitmic);
        for (i = 0; i < ARRAY_SIZE(sc->sc_keymap)/4; i++) {
                uint8_t b = sc->sc_keymap[i];
                if (b == 0xff)
                        continue;
                for (keyix = i * NBBY; keyix < (i + 1) * NBBY;
                    keyix++, b >>= 1) {
                        if ((b & 1) || is_set(keyix+64, sc->sc_keymap)) {
                                /* full pair unavailable */
                                continue;
                        }
                        set_bit(keyix, sc->sc_keymap);
                        set_bit(keyix+64, sc->sc_keymap);
                        ARN_DBG((ARN_DBG_KEYCACHE,
                            "arn_key_alloc_pair(): key pair %u,%u\n",
                            keyix, keyix+64));
                        *txkeyix = *rxkeyix = keyix;
                        return (1);
                }
        }
        ARN_DBG((ARN_DBG_KEYCACHE, "arn_key_alloc_pair():"
            " out of pair space\n"));

        return (0);
}

/*
 * Allocate tx/rx key slots for TKIP.  We allocate two slots for
 * each key, one for decrypt/encrypt and the other for the MIC.
 */
static int
arn_key_alloc_2pair(struct arn_softc *sc, ieee80211_keyix *txkeyix,
    ieee80211_keyix *rxkeyix)
{
        uint16_t i, keyix;

        ASSERT(sc->sc_splitmic);
        for (i = 0; i < ARRAY_SIZE(sc->sc_keymap)/4; i++) {
                uint8_t b = sc->sc_keymap[i];
                if (b != 0xff) {
                        /*
                         * One or more slots in this byte are free.
                         */
                        keyix = i*NBBY;
                        while (b & 1) {
                again:
                                keyix++;
                                b >>= 1;
                        }
                        /* XXX IEEE80211_KEY_XMIT | IEEE80211_KEY_RECV */
                        if (is_set(keyix+32, sc->sc_keymap) ||
                            is_set(keyix+64, sc->sc_keymap) ||
                            is_set(keyix+32+64, sc->sc_keymap)) {
                                /* full pair unavailable */
                                if (keyix == (i+1)*NBBY) {
                                        /* no slots were appropriate, advance */
                                        continue;
                                }
                                goto again;
                        }
                        set_bit(keyix, sc->sc_keymap);
                        set_bit(keyix+64, sc->sc_keymap);
                        set_bit(keyix+32, sc->sc_keymap);
                        set_bit(keyix+32+64, sc->sc_keymap);
                        ARN_DBG((ARN_DBG_KEYCACHE,
                            "arn_key_alloc_2pair(): key pair %u,%u %u,%u\n",
```

```
1844                            keyix, keyix+64,
1845                            keyix+32, keyix+32+64));
1846                        *txkeyix = *rxkeyix = keyix;
1847                        return (1);
1848                }
1849        }
1850        ARN_DBG((ARN_DBG_KEYCACHE, "arn_key_alloc_2pair(): "
1851            " out of pair space\n"));

1853        return (0);
1854 }
1855 /*
1856  * Allocate a single key cache slot.
1857  */
1858 static int
1859 arn_key_alloc_single(struct arn_softc *sc, ieee80211_keyix *txkeyix,
1860     ieee80211_keyix *rxkeyix)
1861 {
1862        uint16_t i, keyix;

1864        /* try i,i+32,i+64,i+32+64 to minimize key pair conflicts */
1865        for (i = 0; i < ARRAY_SIZE(sc->sc_keymap); i++) {
1866                uint8_t b = sc->sc_keymap[i];

1868                if (b != 0xff) {
1869                        /*
1870                         * One or more slots are free.
1871                         */
1872                        keyix = i*NBBY;
1873                        while (b & 1)
1874                                keyix++, b >>= 1;
1875                        set_bit(keyix, sc->sc_keymap);
1876                        ARN_DBG((ARN_DBG_KEYCACHE, "arn_key_alloc_single(): "
1877                            "key %u\n", keyix));
1878                        *txkeyix = *rxkeyix = keyix;
1879                        return (1);
1880                }
1881        }
1882        return (0);
1883 }

1885 /*
1886  * Allocate one or more key cache slots for a unicast key.  The
1887  * key itself is needed only to identify the cipher.  For hardware
1888  * TKIP with split cipher+MIC keys we allocate two key cache slot
1889  * pairs so that we can setup separate TX and RX MIC keys.  Note
1890  * that the MIC key for a TKIP key at slot i is assumed by the
1891  * hardware to be at slot i+64.  This limits TKIP keys to the first
1892  * 64 entries.
1893  */
1894 /* ARGSUSED */
1895 int
1896 arn_key_alloc(ieee80211com_t *ic, const struct ieee80211_key *k,
1897     ieee80211_keyix *keyix, ieee80211_keyix *rxkeyix)
1898 {
1899        struct arn_softc *sc = (struct arn_softc *)ic;

1901        /*
1902         * We allocate two pair for TKIP when using the h/w to do
1903         * the MIC.  For everything else, including software crypto,
1904         * we allocate a single entry.  Note that s/w crypto requires
1905         * a pass-through slot on the 5211 and 5212.  The 5210 does
1906         * not support pass-through cache entries and we map all
1907         * those requests to slot 0.
1908         */
1909        if (k->wk_flags & IEEE80211_KEY_SWCRYPT) {
```



```
1910                return (arn_key_alloc_single(sc, keyix, rxkeyix));
1911        } else if (k->wk_cipher->ic_cipher == IEEE80211_CIPHER_TKIP &&
1912            (k->wk_flags & IEEE80211_KEY_SWMIC) == 0) {
1913                if (sc->sc_splitmic)
1914                        return (arn_key_alloc_2pair(sc, keyix, rxkeyix));
1915                else
1916                        return (arn_key_alloc_pair(sc, keyix, rxkeyix));
1917        } else {
1918                return (arn_key_alloc_single(sc, keyix, rxkeyix));
1919        }
1920 }

1922 /*
1923  * Delete an entry in the key cache allocated by ath_key_alloc.
1924  */
1925 int
1926 arn_key_delete(ieee80211com_t *ic, const struct ieee80211_key *k)
1927 {
1928        struct arn_softc *sc = (struct arn_softc *)ic;
1929        struct ath_hal *ah = sc->sc_ah;
1930        const struct ieee80211_cipher *cip = k->wk_cipher;
1931        ieee80211_keyix keyix = k->wk_keyix;

1933        ARN_DBG((ARN_DBG_KEYCACHE, "arn_key_delete():"
1934            " delete key %u ic_cipher=0x%x\n", keyix, cip->ic_cipher));

1936        (void) ath9k_hw_keyreset(ah, keyix);
1937        /*
1938         * Handle split tx/rx keying required for TKIP with h/w MIC.
1939         */
1940        if (cip->ic_cipher == IEEE80211_CIPHER_TKIP &&
1941            (k->wk_flags & IEEE80211_KEY_SWMIC) == 0 && sc->sc_splitmic)
1942                (void) ath9k_hw_keyreset(ah, keyix+32);         /* RX key */

1944        if (keyix >= IEEE80211_WEP_NKID) {
1945                /*
1946                 * Don’t touch keymap entries for global keys so
1947                 * they are never considered for dynamic allocation.
1948                 */
1949                clr_bit(keyix, sc->sc_keymap);
1950                if (cip->ic_cipher == IEEE80211_CIPHER_TKIP &&
1951                    (k->wk_flags & IEEE80211_KEY_SWMIC) == 0) {
1952                        /*
1953                         * If splitmic is true +64 is TX key MIC,
1954                         * else +64 is RX key + RX key MIC.
1955                         */
1956                        clr_bit(keyix+64, sc->sc_keymap);
1957                        if (sc->sc_splitmic) {
1958                                /* Rx key */
1959                                clr_bit(keyix+32, sc->sc_keymap);
1960                                /* RX key MIC */
1961                                clr_bit(keyix+32+64, sc->sc_keymap);
1962                        }
1963                }
1964        }
1965        return (1);
1966 }

1968 /*
1969  * Set a TKIP key into the hardware.  This handles the
1970  * potential distribution of key state to multiple key
1971  * cache slots for TKIP.
1972  */
1973 static int
1974 arn_keyset_tkip(struct arn_softc *sc, const struct ieee80211_key *k,
1975     struct ath9k_keyval *hk, const uint8_t mac[IEEE80211_ADDR_LEN])
```

```
{
        uint8_t *key_rxmic = NULL;
        uint8_t *key_txmic = NULL;
        uint8_t  *key = (uint8_t *)&(k->wk_key[0]);
        struct ath_hal *ah = sc->sc_ah;

        key_txmic = key + 16;
        key_rxmic = key + 24;

        if (mac == NULL) {
                /* Group key installation */
                (void) memcpy(hk->kv_mic,  key_rxmic, sizeof (hk->kv_mic));
                return (ath9k_hw_set_keycache_entry(ah, k->wk_keyix, hk,
                    mac, B_FALSE));
        }
        if (!sc->sc_splitmic) {
                /*
                 * data key goes at first index,
                 * the hal handles the MIC keys at index+64.
                 */
                (void) memcpy(hk->kv_mic, key_rxmic, sizeof (hk->kv_mic));
                (void) memcpy(hk->kv_txmic, key_txmic, sizeof (hk->kv_txmic));
                return (ath9k_hw_set_keycache_entry(ah, k->wk_keyix, hk,
                    mac, B_FALSE));
        }
        /*
         * TX key goes at first index, RX key at +32.
         * The hal handles the MIC keys at index+64.
         */
        (void) memcpy(hk->kv_mic, key_txmic, sizeof (hk->kv_mic));
        if (!(ath9k_hw_set_keycache_entry(ah, k->wk_keyix, hk, NULL,
            B_FALSE))) {
                /* Txmic entry failed. No need to proceed further */
                ARN_DBG((ARN_DBG_KEYCACHE,
                    "%s Setting TX MIC Key Failed\n", __func__));
                return (0);
        }

        (void) memcpy(hk->kv_mic, key_rxmic, sizeof (hk->kv_mic));

        /* XXX delete tx key on failure? */
        return (ath9k_hw_set_keycache_entry(ah, k->wk_keyix, hk, mac, B_FALSE));

}

int
arn_key_set(ieee80211com_t *ic, const struct ieee80211_key *k,
    const uint8_t mac[IEEE80211_ADDR_LEN])
{
        struct arn_softc *sc = (struct arn_softc *)ic;
        const struct ieee80211_cipher *cip = k->wk_cipher;
        struct ath9k_keyval hk;

        /* cipher table */
        static const uint8_t ciphermap[] = {
                ATH9K_CIPHER_WEP,               /* IEEE80211_CIPHER_WEP */
                ATH9K_CIPHER_TKIP,              /* IEEE80211_CIPHER_TKIP */
                ATH9K_CIPHER_AES_OCB,   /* IEEE80211_CIPHER_AES_OCB */
                ATH9K_CIPHER_AES_CCM,   /* IEEE80211_CIPHER_AES_CCM */
                ATH9K_CIPHER_CKIP,              /* IEEE80211_CIPHER_CKIP */
                ATH9K_CIPHER_CLR,               /* IEEE80211_CIPHER_NONE */
        };

        bzero(&hk, sizeof (hk));

        /*
```



```
         * Software crypto uses a "clear key" so non-crypto
         * state kept in the key cache are maintainedd so that
         * rx frames have an entry to match.
         */
        if ((k->wk_flags & IEEE80211_KEY_SWCRYPT) == 0) {
                ASSERT(cip->ic_cipher < 6);
                hk.kv_type = ciphermap[cip->ic_cipher];
                hk.kv_len = k->wk_keylen;
                bcopy(k->wk_key, hk.kv_val, k->wk_keylen);
        } else {
                hk.kv_type = ATH9K_CIPHER_CLR;
        }

        if (hk.kv_type == ATH9K_CIPHER_TKIP &&
            (k->wk_flags & IEEE80211_KEY_SWMIC) == 0) {
                return (arn_keyset_tkip(sc, k, &hk, mac));
        } else {
                return (ath9k_hw_set_keycache_entry(sc->sc_ah,
                    k->wk_keyix, &hk, mac, B_FALSE));
        }
}

/*
 * Enable/Disable short slot timing
 */
void
arn_set_shortslot(ieee80211com_t *ic, int onoff)
{
        struct ath_hal *ah = ((struct arn_softc *)ic)->sc_ah;

        if (onoff)
                (void) ath9k_hw_setslottime(ah, ATH9K_SLOT_TIME_9);
        else
                (void) ath9k_hw_setslottime(ah, ATH9K_SLOT_TIME_20);
}

static int
arn_open(struct arn_softc *sc)
{
        ieee80211com_t *ic = (ieee80211com_t *)sc;
        struct ieee80211_channel *curchan = ic->ic_curchan;
        struct ath9k_channel *init_channel;
        int error = 0, pos, status;

        ARN_LOCK_ASSERT(sc);

        pos = arn_get_channel(sc, curchan);
        if (pos == -1) {
                ARN_DBG((ARN_DBG_FATAL, "arn: "
                    "%s: Invalid channel\n", __func__));
                error = EINVAL;
                goto error;
        }

        sc->tx_chan_width = ATH9K_HT_MACMODE_20;

        if (sc->sc_curmode == ATH9K_MODE_11A) {
                sc->sc_ah->ah_channels[pos].chanmode = CHANNEL_A;
        } else {
                sc->sc_ah->ah_channels[pos].chanmode = CHANNEL_G;
        }

        init_channel = &sc->sc_ah->ah_channels[pos];

        /* Reset SERDES registers */
        ath9k_hw_configpcipowersave(sc->sc_ah, 0);
```

```
2109         /*
2110          * The basic interface to setting the hardware in a good
2111          * state is ''reset''.  On return the hardware is known to
2112          * be powered up and with interrupts disabled.  This must
2113          * be followed by initialization of the appropriate bits
2114          * and then setup of the interrupt mask.
2115          */
2116         if (!ath9k_hw_reset(sc->sc_ah, init_channel,
2117             sc->tx_chan_width, sc->sc_tx_chainmask,
2118             sc->sc_rx_chainmask, sc->sc_ht_extprotspacing,
2119             B_FALSE, &status)) {
2120                 ARN_DBG((ARN_DBG_FATAL, "arn: "
2121                     "%s: unable to reset hardware; hal status %u "
2122                     "(freq %u flags 0x%x)\n", __func__, status,
2123                     init_channel->channel, init_channel->channelFlags));
2124
2125                 error = EIO;
2126                 goto error;
2127         }
2128
2129         /*
2130          * This is needed only to setup initial state
2131          * but it's best done after a reset.
2132          */
2133         arn_update_txpow(sc);
2134
2135         /*
2136          * Setup the hardware after reset:
2137          * The receive engine is set going.
2138          * Frame transmit is handled entirely
2139          * in the frame output path; there's nothing to do
2140          * here except setup the interrupt mask.
2141          */
2142         if (arn_startrecv(sc) != 0) {
2143                 ARN_DBG((ARN_DBG_INIT, "arn: "
2144                     "%s: unable to start recv logic\n", __func__));
2145                 error = EIO;
2146                 goto error;
2147         }
2148
2149         /* Setup our intr mask. */
2150         sc->sc_imask = ATH9K_INT_RX | ATH9K_INT_TX |
2151             ATH9K_INT_RXEOL | ATH9K_INT_RXORN |
2152             ATH9K_INT_FATAL | ATH9K_INT_GLOBAL;
2153 #ifdef ARN_ATH9K_HW_CAP_GTT
2154         if (sc->sc_ah->ah_caps.hw_caps & ATH9K_HW_CAP_GTT)
2155                 sc->sc_imask |= ATH9K_INT_GTT;
2156 #endif
2157
2158 #ifdef ARN_ATH9K_HW_CAP_GTT
2159         if (sc->sc_ah->ah_caps.hw_caps & ATH9K_HW_CAP_HT)
2160                 sc->sc_imask |= ATH9K_INT_CST;
2161 #endif
2162
2163         /*
2164          * Enable MIB interrupts when there are hardware phy counters.
2165          * Note we only do this (at the moment) for station mode.
2166          */
2167 #ifdef ARN_ATH9K_INT_MIB
2168         if (ath9k_hw_phycounters(sc->sc_ah) &&
2169             ((sc->sc_ah->ah_opmode == ATH9K_M_STA) ||
2170             (sc->sc_ah->ah_opmode == ATH9K_M_IBSS)))
2171                 sc->sc_imask |= ATH9K_INT_MIB;
2172 #endif
2173         /*
```



```
2174          * Some hardware processes the TIM IE and fires an
2175          * interrupt when the TIM bit is set.  For hardware
2176          * that does, if not overridden by configuration,
2177          * enable the TIM interrupt when operating as station.
2178          */
2179 #ifdef ARN_ATH9K_INT_TIM
2180         if ((sc->sc_ah->ah_caps.hw_caps & ATH9K_HW_CAP_ENHANCEDPM) &&
2181             (sc->sc_ah->ah_opmode == ATH9K_M_STA) &&
2182             !sc->sc_config.swBeaconProcess)
2183                 sc->sc_imask |= ATH9K_INT_TIM;
2184 #endif
2185         if (arn_chan2mode(init_channel) != sc->sc_curmode)
2186                 arn_setcurmode(sc, arn_chan2mode(init_channel));
2187         ARN_DBG((ARN_DBG_INIT, "arn: "
2188             "%s: current mode after arn_setcurmode is %d\n",
2189             __func__, sc->sc_curmode));
2190
2191         sc->sc_isrunning = 1;
2192
2193         /* Disable BMISS interrupt when we're not associated */
2194         sc->sc_imask &= ~(ATH9K_INT_SWBA | ATH9K_INT_BMISS);
2195         (void) ath9k_hw_set_interrupts(sc->sc_ah, sc->sc_imask);
2196
2197         return (0);
2198
2199 error:
2200         return (error);
2201 }
2202
2203 static void
2204 arn_close(struct arn_softc *sc)
2205 {
2206         ieee80211com_t *ic = (ieee80211com_t *)sc;
2207         struct ath_hal *ah = sc->sc_ah;
2208
2209         ARN_LOCK_ASSERT(sc);
2210
2211         if (!sc->sc_isrunning)
2212                 return;
2213
2214         /*
2215          * Shutdown the hardware and driver
2216          * Note that some of this work is not possible if the
2217          * hardware is gone (invalid).
2218          */
2219         ARN_UNLOCK(sc);
2220         ieee80211_new_state(ic, IEEE80211_S_INIT, -1);
2221         ieee80211_stop_watchdog(ic);
2222         ARN_LOCK(sc);
2223
2224         /*
2225          * make sure h/w will not generate any interrupt
2226          * before setting the invalid flag.
2227          */
2228         (void) ath9k_hw_set_interrupts(ah, 0);
2229
2230         if (!(sc->sc_flags & SC_OP_INVALID)) {
2231                 arn_draintxq(sc, 0);
2232                 (void) arn_stoprecv(sc);
2233                 (void) ath9k_hw_phy_disable(ah);
2234         } else {
2235                 sc->sc_rxlink = NULL;
2236         }
2237
2238         sc->sc_isrunning = 0;
2239 }
```

```
2241 /*
2242  * MAC callback functions
2243  */
2244 static int
2245 arn_m_stat(void *arg, uint_t stat, uint64_t *val)
2246 {
2247         struct arn_softc *sc = arg;
2248         ieee80211com_t *ic = (ieee80211com_t *)sc;
2249         struct ieee80211_node *in;
2250         struct ieee80211_rateset *rs;

2252         ARN_LOCK(sc);
2253         switch (stat) {
2254         case MAC_STAT_IFSPEED:
2255                 in = ic->ic_bss;
2256                 rs = &in->in_rates;
2257                 *val = (rs->ir_rates[in->in_txrate] & IEEE80211_RATE_VAL) / 2 *
2258                     1000000ull;
2259                 break;
2260         case MAC_STAT_NOXMTBUF:
2261                 *val = sc->sc_stats.ast_tx_nobuf +
2262                     sc->sc_stats.ast_tx_nobufmgt;
2263                 break;
2264         case MAC_STAT_IERRORS:
2265                 *val = sc->sc_stats.ast_rx_tooshort;
2266                 break;
2267         case MAC_STAT_RBYTES:
2268                 *val = ic->ic_stats.is_rx_bytes;
2269                 break;
2270         case MAC_STAT_IPACKETS:
2271                 *val = ic->ic_stats.is_rx_frags;
2272                 break;
2273         case MAC_STAT_OBYTES:
2274                 *val = ic->ic_stats.is_tx_bytes;
2275                 break;
2276         case MAC_STAT_OPACKETS:
2277                 *val = ic->ic_stats.is_tx_frags;
2278                 break;
2279         case MAC_STAT_OERRORS:
2280         case WIFI_STAT_TX_FAILED:
2281                 *val = sc->sc_stats.ast_tx_fifoerr +
2282                     sc->sc_stats.ast_tx_xretries +
2283                     sc->sc_stats.ast_tx_discard;
2284                 break;
2285         case WIFI_STAT_TX_RETRANS:
2286                 *val = sc->sc_stats.ast_tx_xretries;
2287                 break;
2288         case WIFI_STAT_FCS_ERRORS:
2289                 *val = sc->sc_stats.ast_rx_crcerr;
2290                 break;
2291         case WIFI_STAT_WEP_ERRORS:
2292                 *val = sc->sc_stats.ast_rx_badcrypt;
2293                 break;
2294         case WIFI_STAT_TX_FRAGS:
2295         case WIFI_STAT_MCAST_TX:
2296         case WIFI_STAT_RTS_SUCCESS:
2297         case WIFI_STAT_RTS_FAILURE:
2298         case WIFI_STAT_ACK_FAILURE:
2299         case WIFI_STAT_RX_FRAGS:
2300         case WIFI_STAT_MCAST_RX:
2301         case WIFI_STAT_RX_DUPS:
2302                 ARN_UNLOCK(sc);
2303                 return (ieee80211_stat(ic, stat, val));
2304         default:
2305                 ARN_UNLOCK(sc);
```



```
2306                 return (ENOTSUP);
2307         }
2308         ARN_UNLOCK(sc);

2310         return (0);
2311 }

2313 int
2314 arn_m_start(void *arg)
2315 {
2316         struct arn_softc *sc = arg;
2317         int err = 0;

2319         ARN_LOCK(sc);

2321         /*
2322          * Stop anything previously setup.  This is safe
2323          * whether this is the first time through or not.
2324          */

2326         arn_close(sc);

2328         if ((err = arn_open(sc)) != 0) {
2329                 ARN_UNLOCK(sc);
2330                 return (err);
2331         }

2333         /* H/W is reday now */
2334         sc->sc_flags &= ~SC_OP_INVALID;

2336         ARN_UNLOCK(sc);

2338         return (0);
2339 }

2341 static void
2342 arn_m_stop(void *arg)
2343 {
2344         struct arn_softc *sc = arg;

2346         ARN_LOCK(sc);
2347         arn_close(sc);

2349         /* disable HAL and put h/w to sleep */
2350         (void) ath9k_hw_disable(sc->sc_ah);
2351         ath9k_hw_configpcipowersave(sc->sc_ah, 1);

2353         /* XXX: hardware will not be ready in suspend state */
2354         sc->sc_flags |= SC_OP_INVALID;
2355         ARN_UNLOCK(sc);
2356 }

2358 static int
2359 arn_m_promisc(void *arg, boolean_t on)
2360 {
2361         struct arn_softc *sc = arg;
2362         struct ath_hal *ah = sc->sc_ah;
2363         uint32_t rfilt;

2365         ARN_LOCK(sc);

2367         rfilt = ath9k_hw_getrxfilter(ah);
2368         if (on)
2369                 rfilt |= ATH9K_RX_FILTER_PROM;
2370         else
2371                 rfilt &= ~ATH9K_RX_FILTER_PROM;
```

```
2372         sc->sc_promisc = on;
2373         ath9k_hw_setrxfilter(ah, rfilt);

2375         ARN_UNLOCK(sc);

2377         return (0);
2378 }

2380 static int
2381 arn_m_multicst(void *arg, boolean_t add, const uint8_t *mca)
2382 {
2383         struct arn_softc *sc = arg;
2384         struct ath_hal *ah = sc->sc_ah;
2385         uint32_t val, index, bit;
2386         uint8_t pos;
2387         uint32_t *mfilt = sc->sc_mcast_hash;

2389         ARN_LOCK(sc);

2391         /* calculate XOR of eight 6bit values */
2392         val = ARN_LE_READ_32(mca + 0);
2393         pos = (val >> 18) ^ (val >> 12) ^ (val >> 6) ^ val;
2394         val = ARN_LE_READ_32(mca + 3);
2395         pos ^= (val >> 18) ^ (val >> 12) ^ (val >> 6) ^ val;
2396         pos &= 0x3f;
2397         index = pos / 32;
2398         bit = 1 << (pos % 32);

2400         if (add) {      /* enable multicast */
2401                 sc->sc_mcast_refs[pos]++;
2402                 mfilt[index] |= bit;
2403         } else {        /* disable multicast */
2404                 if (--sc->sc_mcast_refs[pos] == 0)
2405                         mfilt[index] &= ~bit;
2406         }
2407         ath9k_hw_setmcastfilter(ah, mfilt[0], mfilt[1]);

2409         ARN_UNLOCK(sc);
2410         return (0);
2411 }

2413 static int
2414 arn_m_unicst(void *arg, const uint8_t *macaddr)
2415 {
2416         struct arn_softc *sc = arg;
2417         struct ath_hal *ah = sc->sc_ah;
2418         ieee80211com_t *ic = (ieee80211com_t *)sc;

2420         ARN_DBG((ARN_DBG_XMIT, "ath: ath_gld_saddr(): "
2421             "%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
2422             macaddr[0], macaddr[1], macaddr[2],
2423             macaddr[3], macaddr[4], macaddr[5]));

2425         ARN_LOCK(sc);
2426         IEEE80211_ADDR_COPY(sc->sc_isc.ic_macaddr, macaddr);
2427         (void) ath9k_hw_setmac(ah, sc->sc_isc.ic_macaddr);
2428         (void) arn_reset(ic);
2429         ARN_UNLOCK(sc);
2430         return (0);
2431 }

2433 static mblk_t *
2434 arn_m_tx(void *arg, mblk_t *mp)
2435 {
2436         struct arn_softc *sc = arg;
2437         int error = 0;
```



```
2438         mblk_t *next;
2439         ieee80211com_t *ic = (ieee80211com_t *)sc;

2441         /*
2442          * No data frames go out unless we're associated; this
2443          * should not happen as the 802.11 layer does not enable
2444          * the xmit queue until we enter the RUN state.
2445          */
2446         if (ic->ic_state != IEEE80211_S_RUN) {
2447                 ARN_DBG((ARN_DBG_XMIT, "arn: arn_m_tx(): "
2448                     "discard, state %u\n", ic->ic_state));
2449                 sc->sc_stats.ast_tx_discard++;
2450                 freemsgchain(mp);
2451                 return (NULL);
2452         }

2454         while (mp != NULL) {
2455                 next = mp->b_next;
2456                 mp->b_next = NULL;
2457                 error = arn_tx(ic, mp, IEEE80211_FC0_TYPE_DATA);
2458                 if (error != 0) {
2459                         mp->b_next = next;
2460                         if (error == ENOMEM) {
2461                                 break;
2462                         } else {
2463                                 freemsgchain(mp);
2464                                 return (NULL);
2465                         }
2466                 }
2467                 mp = next;
2468         }

2470         return (mp);
2471 }

2473 static void
2474 arn_m_ioctl(void *arg, queue_t *wq, mblk_t *mp)
2475 {
2476         struct arn_softc *sc = arg;
2477         int32_t err;

2479         err = ieee80211_ioctl(&sc->sc_isc, wq, mp);

2481         ARN_LOCK(sc);
2482         if (err == ENETRESET) {
2483                 if (!(sc->sc_flags & SC_OP_INVALID)) {
2484                         ARN_UNLOCK(sc);

2486                         (void) arn_m_start(sc);

2488                         (void) ieee80211_new_state(&sc->sc_isc,
2489                             IEEE80211_S_SCAN, -1);
2490                         ARN_LOCK(sc);
2491                 }
2492         }
2493         ARN_UNLOCK(sc);
2494 }

2496 static int
2497 arn_m_setprop(void *arg, const char *pr_name, mac_prop_id_t wldp_pr_num,
2498     uint_t wldp_length, const void *wldp_buf)
2499 {
2500         struct arn_softc *sc = arg;
2501         int     err;

2503         err = ieee80211_setprop(&sc->sc_isc, pr_name, wldp_pr_num,
```

```
2504             wldp_length, wldp_buf);

2506         ARN_LOCK(sc);

2508         if (err == ENETRESET) {
2509                 if (!(sc->sc_flags & SC_OP_INVALID)) {
2510                         ARN_UNLOCK(sc);
2511                         (void) arn_m_start(sc);
2512                         (void) ieee80211_new_state(&sc->sc_isc,
2513                             IEEE80211_S_SCAN, -1);
2514                         ARN_LOCK(sc);
2515                 }
2516                 err = 0;
2517         }

2519         ARN_UNLOCK(sc);

2521         return (err);
2522 }

2524 /* ARGSUSED */
2525 static int
2526 arn_m_getprop(void *arg, const char *pr_name, mac_prop_id_t wldp_pr_num,
2527     uint_t wldp_length, void *wldp_buf)
2528 {
2529         struct arn_softc *sc = arg;
2530         int     err = 0;

2532         err = ieee80211_getprop(&sc->sc_isc, pr_name, wldp_pr_num,
2533             wldp_length, wldp_buf);

2535         return (err);
2536 }

2538 static void
2539 arn_m_propinfo(void *arg, const char *pr_name, mac_prop_id_t wldp_pr_num,
2540     mac_prop_info_handle_t prh)
2541 {
2542         struct arn_softc *sc = arg;

2544         ieee80211_propinfo(&sc->sc_isc, pr_name, wldp_pr_num, prh);
2545 }

2547 /* return bus cachesize in 4B word units */
2548 static void
2549 arn_pci_config_cachesize(struct arn_softc *sc)
2550 {
2551         uint8_t csz;

2553         /*
2554          * Cache line size is used to size and align various
2555          * structures used to communicate with the hardware.
2556          */
2557         csz = pci_config_get8(sc->sc_cfg_handle, PCI_CONF_CACHE_LINESZ);
2558         if (csz == 0) {
2559                 /*
2560                  * We must have this setup properly for rx buffer
2561                  * DMA to work so force a reasonable value here if it
2562                  * comes up zero.
2563                  */
2564                 csz = ATH_DEF_CACHE_BYTES / sizeof (uint32_t);
2565                 pci_config_put8(sc->sc_cfg_handle, PCI_CONF_CACHE_LINESZ,
2566                     csz);
2567         }
2568         sc->sc_cachelsz = csz << 2;
2569 }
```



```
2571 static int
2572 arn_pci_setup(struct arn_softc *sc)
2573 {
2574         uint16_t command;

2576         /*
2577          * Enable memory mapping and bus mastering
2578          */
2579         ASSERT(sc != NULL);
2580         command = pci_config_get16(sc->sc_cfg_handle, PCI_CONF_COMM);
2581         command |= PCI_COMM_MAE | PCI_COMM_ME;
2582         pci_config_put16(sc->sc_cfg_handle, PCI_CONF_COMM, command);
2583         command = pci_config_get16(sc->sc_cfg_handle, PCI_CONF_COMM);
2584         if ((command & PCI_COMM_MAE) == 0) {
2585                 arn_problem("arn: arn_pci_setup(): "
2586                     "failed to enable memory mapping\n");
2587                 return (EIO);
2588         }
2589         if ((command & PCI_COMM_ME) == 0) {
2590                 arn_problem("arn: arn_pci_setup(): "
2591                     "failed to enable bus mastering\n");
2592                 return (EIO);
2593         }
2594         ARN_DBG((ARN_DBG_INIT, "arn: arn_pci_setup(): "
2595             "set command reg to 0x%x \n", command));

2597         return (0);
2598 }

2600 static void
2601 arn_get_hw_encap(struct arn_softc *sc)
2602 {
2603         ieee80211com_t *ic;
2604         struct ath_hal *ah;

2606         ic = (ieee80211com_t *)sc;
2607         ah = sc->sc_ah;

2609         if (ath9k_hw_getcapability(ah, ATH9K_CAP_CIPHER,
2610             ATH9K_CIPHER_AES_CCM, NULL))
2611                 ic->ic_caps |= IEEE80211_C_AES_CCM;
2612         if (ath9k_hw_getcapability(ah, ATH9K_CAP_CIPHER,
2613             ATH9K_CIPHER_AES_OCB, NULL))
2614                 ic->ic_caps |= IEEE80211_C_AES;
2615         if (ath9k_hw_getcapability(ah, ATH9K_CAP_CIPHER,
2616             ATH9K_CIPHER_TKIP, NULL))
2617                 ic->ic_caps |= IEEE80211_C_TKIP;
2618         if (ath9k_hw_getcapability(ah, ATH9K_CAP_CIPHER,
2619             ATH9K_CIPHER_WEP, NULL))
2620                 ic->ic_caps |= IEEE80211_C_WEP;
2621         if (ath9k_hw_getcapability(ah, ATH9K_CAP_CIPHER,
2622             ATH9K_CIPHER_MIC, NULL))
2623                 ic->ic_caps |= IEEE80211_C_TKIPMIC;
2624 }

2626 static void
2627 arn_setup_ht_cap(struct arn_softc *sc)
2628 {
2629 #define ATH9K_HT_CAP_MAXRXAMPDU_65536 0x3       /* 2 ^ 16 */
2630 #define ATH9K_HT_CAP_MPDUDENSITY_8 0x6          /* 8 usec */

2632         /* LINTED E_FUNC_SET_NOT_USED */
2633         uint8_t tx_streams;
2634         uint8_t rx_streams;
```

```
2636         arn_ht_conf *ht_info = &sc->sc_ht_conf;

2638         ht_info->ht_supported = B_TRUE;

2640         /* Todo: IEEE80211_HTCAP_SMPS */
2641         ht_info->cap = IEEE80211_HTCAP_CHWIDTH40|
2642             IEEE80211_HTCAP_SHORTGI40 |
2643             IEEE80211_HTCAP_DSSSCCK40;

2645         ht_info->ampdu_factor = ATH9K_HT_CAP_MAXRXAMPDU_65536;
2646         ht_info->ampdu_density = ATH9K_HT_CAP_MPDUDENSITY_8;

2648         /* set up supported mcs set */
2649         (void) memset(&ht_info->rx_mcs_mask, 0, sizeof (ht_info->rx_mcs_mask));
2650         tx_streams = ISP2(sc->sc_ah->ah_caps.tx_chainmask) ? 1 : 2;
2651         rx_streams = ISP2(sc->sc_ah->ah_caps.rx_chainmask) ? 1 : 2;
  22         tx_streams =
  23             !(sc->sc_ah->ah_caps.tx_chainmask &
  24             (sc->sc_ah->ah_caps.tx_chainmask - 1)) ? 1 : 2;
  25         rx_streams =
  26             !(sc->sc_ah->ah_caps.rx_chainmask &
  27             (sc->sc_ah->ah_caps.rx_chainmask - 1)) ? 1 : 2;

2653         ht_info->rx_mcs_mask[0] = 0xff;
2654         if (rx_streams >= 2)
2655                 ht_info->rx_mcs_mask[1] = 0xff;
2656 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   97030 Thu Oct 23 10:42:12 2014
new/usr/src/uts/common/io/comstar/lu/stmf_sbd/sbd.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/scsi/scsi.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/disp.h>
#include <sys/byteorder.h>
#include <sys/pathname.h>
#include <sys/atomic.h>
#include <sys/nvpair.h>
#include <sys/fs/zfs.h>
#include <sys/sdt.h>
#include <sys/dkio.h>
#include <sys/zfs_ioctl.h>

#include <sys/stmf.h>
#include <sys/lpif.h>
#include <sys/stmf_ioctl.h>
#include <sys/stmf_sbd_ioctl.h>

#include "stmf_sbd.h"
#include "sbd_impl.h"

#define SBD_IS_ZVOL(zvol)       (strncmp("/dev/zvol", zvol, 9))

extern sbd_status_t sbd_pgr_meta_init(sbd_lu_t *sl);
extern sbd_status_t sbd_pgr_meta_load(sbd_lu_t *sl);
extern void sbd_pgr_reset(sbd_lu_t *sl);

static int sbd_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
```



```
    void **result);
static int sbd_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int sbd_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static int sbd_open(dev_t *devp, int flag, int otype, cred_t *credp);
static int sbd_close(dev_t dev, int flag, int otype, cred_t *credp);
static int stmf_sbd_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
    cred_t *credp, int *rval);
void sbd_lp_cb(stmf_lu_provider_t *lp, int cmd, void *arg, uint32_t flags);
stmf_status_t sbd_proxy_reg_lu(uint8_t *luid, void *proxy_reg_arg,
    uint32_t proxy_reg_arg_len);
stmf_status_t sbd_proxy_dereg_lu(uint8_t *luid, void *proxy_reg_arg,
    uint32_t proxy_reg_arg_len);
stmf_status_t sbd_proxy_msg(uint8_t *luid, void *proxy_arg,
    uint32_t proxy_arg_len, uint32_t type);
int sbd_create_register_lu(sbd_create_and_reg_lu_t *slu, int struct_sz,
    uint32_t *err_ret);
int sbd_create_standby_lu(sbd_create_standby_lu_t *slu, uint32_t *err_ret);
int sbd_set_lu_standby(sbd_set_lu_standby_t *stlu, uint32_t *err_ret);
int sbd_import_lu(sbd_import_lu_t *ilu, int struct_sz, uint32_t *err_ret,
    int no_register, sbd_lu_t **slr);
int sbd_import_active_lu(sbd_import_lu_t *ilu, sbd_lu_t *sl, uint32_t *err_ret);
int sbd_delete_lu(sbd_delete_lu_t *dlu, int struct_sz, uint32_t *err_ret);
int sbd_modify_lu(sbd_modify_lu_t *mlu, int struct_sz, uint32_t *err_ret);
int sbd_set_global_props(sbd_global_props_t *mlu, int struct_sz,
    uint32_t *err_ret);
int sbd_get_global_props(sbd_global_props_t *oslp, uint32_t oslp_sz,
    uint32_t *err_ret);
int sbd_get_lu_props(sbd_lu_props_t *islp, uint32_t islp_sz,
    sbd_lu_props_t *oslp, uint32_t oslp_sz, uint32_t *err_ret);
static char *sbd_get_zvol_name(sbd_lu_t *);
static int sbd_get_unmap_props(sbd_unmap_props_t *sup, sbd_unmap_props_t *osup,
    uint32_t *err_ret);
sbd_status_t sbd_create_zfs_meta_object(sbd_lu_t *sl);
sbd_status_t sbd_open_zfs_meta(sbd_lu_t *sl);
sbd_status_t sbd_read_zfs_meta(sbd_lu_t *sl, uint8_t *buf, uint64_t sz,
    uint64_t off);
sbd_status_t sbd_write_zfs_meta(sbd_lu_t *sl, uint8_t *buf, uint64_t sz,
    uint64_t off);
sbd_status_t sbd_update_zfs_prop(sbd_lu_t *sl);
int sbd_is_zvol(char *path);
int sbd_zvolget(char *zvol_name, char **comstarprop);
int sbd_zvolset(char *zvol_name, char *comstarprop);
char sbd_ctoi(char c);
void sbd_close_lu(sbd_lu_t *sl);

static ldi_ident_t      sbd_zfs_ident;
static stmf_lu_provider_t *sbd_lp;
static sbd_lu_t         *sbd_lu_list = NULL;
static kmutex_t         sbd_lock;
static dev_info_t       *sbd_dip;
static uint32_t         sbd_lu_count = 0;

/* Global property settings for the logical unit */
char sbd_vendor_id[]    = "SUN     ";
char sbd_product_id[]   = "COMSTAR         ";
char sbd_revision[]     = "1.0 ";
char *sbd_mgmt_url = NULL;
uint16_t sbd_mgmt_url_alloc_size = 0;
krwlock_t sbd_global_prop_lock;

static char sbd_name[] = "sbd";

static struct cb_ops sbd_cb_ops = {
        sbd_open,                       /* open */
        sbd_close,                      /* close */
        nodev,                          /* strategy */
```

```
128         nodev,                          /* print */
129         nodev,                          /* dump */
130         nodev,                          /* read */
131         nodev,                          /* write */
132         stmf_sbd_ioctl,                 /* ioctl */
133         nodev,                          /* devmap */
134         nodev,                          /* mmap */
135         nodev,                          /* segmap */
136         nochpoll,                       /* chpoll */
137         ddi_prop_op,                    /* cb_prop_op */
138         0,                              /* streamtab */
139         D_NEW | D_MP,                   /* cb_flag */
140         CB_REV,                         /* rev */
141         nodev,                          /* aread */
142         nodev                           /* awrite */
143 };

145 static struct dev_ops sbd_ops = {
146         DEVO_REV,
147         0,
148         sbd_getinfo,
149         nulldev,                /* identify */
150         nulldev,                /* probe */
151         sbd_attach,
152         sbd_detach,
153         nodev,                  /* reset */
154         &sbd_cb_ops,
155         NULL,                   /* bus_ops */
156         NULL                    /* power */
157 };

159 #define SBD_NAME        "COMSTAR SBD"

161 static struct modldrv modldrv = {
162         &mod_driverops,
163         SBD_NAME,
164         &sbd_ops
165 };

167 static struct modlinkage modlinkage = {
168         MODREV_1,
169         &modldrv,
170         NULL
171 };

173 int
174 _init(void)
175 {
176         int ret;

178         ret = mod_install(&modlinkage);
179         if (ret)
180                 return (ret);
181         sbd_lp = (stmf_lu_provider_t *)stmf_alloc(STMF_STRUCT_LU_PROVIDER,
182             0, 0);
183         sbd_lp->lp_lpif_rev = LPIF_REV_2;
184         sbd_lp->lp_instance = 0;
185         sbd_lp->lp_name = sbd_name;
186         sbd_lp->lp_cb = sbd_lp_cb;
187         sbd_lp->lp_alua_support = 1;
188         sbd_lp->lp_proxy_msg = sbd_proxy_msg;
189         sbd_zfs_ident = ldi_ident_from_anon();

191         if (stmf_register_lu_provider(sbd_lp) != STMF_SUCCESS) {
192                 (void) mod_remove(&modlinkage);
193                 stmf_free(sbd_lp);
```



```
194                 return (EINVAL);
195         }
196         mutex_init(&sbd_lock, NULL, MUTEX_DRIVER, NULL);
197         rw_init(&sbd_global_prop_lock, NULL, RW_DRIVER, NULL);
198         return (0);
199 }

201 int
202 _fini(void)
203 {
204         int ret;

206         /*
207          * If we have registered lus, then make sure they are all offline
208          * if so then deregister them. This should drop the sbd_lu_count
209          * to zero.
210          */
211         if (sbd_lu_count) {
212                 sbd_lu_t *slu;

214                 /* See if all of them are offline */
215                 mutex_enter(&sbd_lock);
216                 for (slu = sbd_lu_list; slu != NULL; slu = slu->sl_next) {
217                         if ((slu->sl_state != STMF_STATE_OFFLINE) ||
218                             slu->sl_state_not_acked) {
219                                 mutex_exit(&sbd_lock);
220                                 return (EBUSY);
221                         }
222                 }
223                 mutex_exit(&sbd_lock);

225 #if 0
226                 /* ok start deregistering them */
227                 while (sbd_lu_list) {
228                         sbd_store_t *sst = sbd_lu_list->sl_sst;
229                         if (sst->sst_deregister_lu(sst) != STMF_SUCCESS)
230                                 return (EBUSY);
231                 }
232 #endif
233                 return (EBUSY);
234         }
235         if (stmf_deregister_lu_provider(sbd_lp) != STMF_SUCCESS)
236                 return (EBUSY);
237         ret = mod_remove(&modlinkage);
238         if (ret != 0) {
239                 (void) stmf_register_lu_provider(sbd_lp);
240                 return (ret);
241         }
242         stmf_free(sbd_lp);
243         mutex_destroy(&sbd_lock);
244         rw_destroy(&sbd_global_prop_lock);
245         ldi_ident_release(sbd_zfs_ident);
246         return (0);
247 }

249 int
250 _info(struct modinfo *modinfop)
251 {
252         return (mod_info(&modlinkage, modinfop));
253 }

255 /* ARGSUSED */
256 static int
257 sbd_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void **result)
258 {
259         switch (cmd) {
```

```
260         case DDI_INFO_DEVT2DEVINFO:
261                 *result = sbd_dip;
262                 break;
263         case DDI_INFO_DEVT2INSTANCE:
264                 *result = (void *)(uintptr_t)ddi_get_instance(sbd_dip);
265                 break;
266         default:
267                 return (DDI_FAILURE);
268         }
270         return (DDI_SUCCESS);
271 }
273 static int
274 sbd_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
275 {
276         switch (cmd) {
277         case DDI_ATTACH:
278                 sbd_dip = dip;
280                 if (ddi_create_minor_node(dip, "admin", S_IFCHR, 0,
281                     DDI_NT_STMF_LP, 0) != DDI_SUCCESS) {
282                         break;
283                 }
284                 ddi_report_dev(dip);
285                 return (DDI_SUCCESS);
286         }
288         return (DDI_FAILURE);
289 }
291 static int
292 sbd_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
293 {
294         switch (cmd) {
295         case DDI_DETACH:
296                 ddi_remove_minor_node(dip, 0);
297                 return (DDI_SUCCESS);
298         }
300         return (DDI_FAILURE);
301 }
303 /* ARGSUSED */
304 static int
305 sbd_open(dev_t *devp, int flag, int otype, cred_t *credp)
306 {
307         if (otype != OTYP_CHR)
308                 return (EINVAL);
309         return (0);
310 }
312 /* ARGSUSED */
313 static int
314 sbd_close(dev_t dev, int flag, int otype, cred_t *credp)
315 {
316         return (0);
317 }
319 /* ARGSUSED */
320 static int
321 stmf_sbd_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
322         cred_t *credp, int *rval)
323 {
324         stmf_iocdata_t          *iocd;
325         void                    *ibuf   = NULL;
```



```
326         void                    *obuf   = NULL;
327         sbd_lu_t                *nsl;
328         int                     i;
329         int                     ret;
331         if (drv_priv(credp) != 0) {
332                 return (EPERM);
333         }
335         ret = stmf_copyin_iocdata(data, mode, &iocd, &ibuf, &obuf);
336         if (ret)
337                 return (ret);
338         iocd->stmf_error = 0;
340         switch (cmd) {
341         case SBD_IOCTL_CREATE_AND_REGISTER_LU:
342                 if (iocd->stmf_ibuf_size <
343                     (sizeof (sbd_create_and_reg_lu_t) - 8)) {
344                         ret = EFAULT;
345                         break;
346                 }
347                 if ((iocd->stmf_obuf_size == 0) ||
348                     (iocd->stmf_obuf_size > iocd->stmf_ibuf_size)) {
349                         ret = EINVAL;
350                         break;
351                 }
352                 ret = sbd_create_register_lu((sbd_create_and_reg_lu_t *)
353                     ibuf, iocd->stmf_ibuf_size, &iocd->stmf_error);
354                 bcopy(ibuf, obuf, iocd->stmf_obuf_size);
355                 break;
356         case SBD_IOCTL_SET_LU_STANDBY:
357                 if (iocd->stmf_ibuf_size < sizeof (sbd_set_lu_standby_t)) {
358                         ret = EFAULT;
359                         break;
360                 }
361                 if (iocd->stmf_obuf_size) {
362                         ret = EINVAL;
363                         break;
364                 }
365                 ret = sbd_set_lu_standby((sbd_set_lu_standby_t *)ibuf,
366                     &iocd->stmf_error);
367                 break;
368         case SBD_IOCTL_IMPORT_LU:
369                 if (iocd->stmf_ibuf_size <
370                     (sizeof (sbd_import_lu_t) - 8)) {
371                         ret = EFAULT;
372                         break;
373                 }
374                 if ((iocd->stmf_obuf_size == 0) ||
375                     (iocd->stmf_obuf_size > iocd->stmf_ibuf_size)) {
376                         ret = EINVAL;
377                         break;
378                 }
379                 ret = sbd_import_lu((sbd_import_lu_t *)ibuf,
380                     iocd->stmf_ibuf_size, &iocd->stmf_error, 0, NULL);
381                 bcopy(ibuf, obuf, iocd->stmf_obuf_size);
382                 break;
383         case SBD_IOCTL_DELETE_LU:
384                 if (iocd->stmf_ibuf_size < (sizeof (sbd_delete_lu_t) - 8)) {
385                         ret = EFAULT;
386                         break;
387                 }
388                 if (iocd->stmf_obuf_size) {
389                         ret = EINVAL;
390                         break;
391                 }
```

```
392                 ret = sbd_delete_lu((sbd_delete_lu_t *)ibuf,
393                     iocd->stmf_ibuf_size, &iocd->stmf_error);
394                 break;
395         case SBD_IOCTL_MODIFY_LU:
396                 if (iocd->stmf_ibuf_size < (sizeof (sbd_modify_lu_t) - 8)) {
397                         ret = EFAULT;
398                         break;
399                 }
400                 if (iocd->stmf_obuf_size) {
401                         ret = EINVAL;
402                         break;
403                 }
404                 ret = sbd_modify_lu((sbd_modify_lu_t *)ibuf,
405                     iocd->stmf_ibuf_size, &iocd->stmf_error);
406                 break;
407         case SBD_IOCTL_SET_GLOBAL_LU:
408                 if (iocd->stmf_ibuf_size < (sizeof (sbd_global_props_t) - 8)) {
409                         ret = EFAULT;
410                         break;
411                 }
412                 if (iocd->stmf_obuf_size) {
413                         ret = EINVAL;
414                         break;
415                 }
416                 ret = sbd_set_global_props((sbd_global_props_t *)ibuf,
417                     iocd->stmf_ibuf_size, &iocd->stmf_error);
418                 break;
419         case SBD_IOCTL_GET_GLOBAL_LU:
420                 if (iocd->stmf_ibuf_size) {
421                         ret = EINVAL;
422                         break;
423                 }
424                 if (iocd->stmf_obuf_size < sizeof (sbd_global_props_t)) {
425                         ret = EINVAL;
426                         break;
427                 }
428                 ret = sbd_get_global_props((sbd_global_props_t *)obuf,
429                     iocd->stmf_obuf_size, &iocd->stmf_error);
430                 break;
431         case SBD_IOCTL_GET_LU_PROPS:
432                 if (iocd->stmf_ibuf_size < (sizeof (sbd_lu_props_t) - 8)) {
433                         ret = EFAULT;
434                         break;
435                 }
436                 if (iocd->stmf_obuf_size < sizeof (sbd_lu_props_t)) {
437                         ret = EINVAL;
438                         break;
439                 }
440                 ret = sbd_get_lu_props((sbd_lu_props_t *)ibuf,
441                     iocd->stmf_ibuf_size, (sbd_lu_props_t *)obuf,
442                     iocd->stmf_obuf_size, &iocd->stmf_error);
443                 break;
444         case SBD_IOCTL_GET_LU_LIST:
445                 mutex_enter(&sbd_lock);
446                 iocd->stmf_obuf_max_nentries = sbd_lu_count;
447                 iocd->stmf_obuf_nentries = min((iocd->stmf_obuf_size >> 4),
448                     sbd_lu_count);
449                 for (nsl = sbd_lu_list, i = 0; nsl &&
450                     (i < iocd->stmf_obuf_nentries); i++, nsl = nsl->sl_next) {
451                         bcopy(nsl->sl_device_id + 4,
452                             &(((uint8_t *)obuf)[i << 4]), 16);
453                 }
454                 mutex_exit(&sbd_lock);
455                 ret = 0;
456                 iocd->stmf_error = 0;
457                 break;
```



```
458         case SBD_IOCTL_GET_UNMAP_PROPS:
459                 if (iocd->stmf_ibuf_size < sizeof (sbd_unmap_props_t)) {
460                         ret = EFAULT;
461                         break;
462                 }
463                 if (iocd->stmf_obuf_size < sizeof (sbd_unmap_props_t)) {
464                         ret = EINVAL;
465                         break;
466                 }
467                 ret = sbd_get_unmap_props((sbd_unmap_props_t *)ibuf,
468                     (sbd_unmap_props_t *)obuf, &iocd->stmf_error);
469                 break;
470         default:
471                 ret = ENOTTY;
472         }

474         if (ret == 0) {
475                 ret = stmf_copyout_iocdata(data, mode, iocd, obuf);
476         } else if (iocd->stmf_error) {
477                 (void) stmf_copyout_iocdata(data, mode, iocd, obuf);
478         }
479         if (obuf) {
480                 kmem_free(obuf, iocd->stmf_obuf_size);
481                 obuf = NULL;
482         }
483         if (ibuf) {
484                 kmem_free(ibuf, iocd->stmf_ibuf_size);
485                 ibuf = NULL;
486         }
487         kmem_free(iocd, sizeof (stmf_iocdata_t));
488         return (ret);
489 }

491 /* ARGSUSED */
492 void
493 sbd_lp_cb(stmf_lu_provider_t *lp, int cmd, void *arg, uint32_t flags)
494 {
495         nvpair_t        *np;
496         char            *s;
497         sbd_import_lu_t *ilu;
498         uint32_t        ilu_sz;
499         uint32_t        struct_sz;
500         uint32_t        err_ret;
501         int             iret;

503         if ((cmd != STMF_PROVIDER_DATA_UPDATED) || (arg == NULL)) {
504                 return;
505         }

507         if ((flags & (STMF_PCB_STMF_ONLINING | STMF_PCB_PREG_COMPLETE)) == 0) {
508                 return;
509         }

511         np = NULL;
512         ilu_sz = 1024;
513         ilu = (sbd_import_lu_t *)kmem_zalloc(ilu_sz, KM_SLEEP);
514         while ((np = nvlist_next_nvpair((nvlist_t *)arg, np)) != NULL) {
515                 if (nvpair_type(np) != DATA_TYPE_STRING) {
516                         continue;
517                 }
518                 if (nvpair_value_string(np, &s) != 0) {
519                         continue;
520                 }
521                 struct_sz = max(8, strlen(s) + 1);
522                 struct_sz += sizeof (sbd_import_lu_t) - 8;
523                 if (struct_sz > ilu_sz) {
```

```
524                         kmem_free(ilu, ilu_sz);
525                         ilu_sz = struct_sz + 32;
526                         ilu = (sbd_import_lu_t *)kmem_zalloc(ilu_sz, KM_SLEEP);
527                 }
528                 ilu->ilu_struct_size = struct_sz;
529                 (void) strcpy(ilu->ilu_meta_fname, s);
530                 iret = sbd_import_lu(ilu, struct_sz, &err_ret, 0, NULL);
531                 if (iret) {
532                         stmf_trace(0, "sbd_lp_cb: import_lu failed, ret = %d, "
533                             "err_ret = %d", iret, err_ret);
534                 } else {
535                         stmf_trace(0, "Imported the LU %s", nvpair_name(np));
536                 }
537         }

539         if (ilu) {
540                 kmem_free(ilu, ilu_sz);
541                 ilu = NULL;
542         }
543 }

545 sbd_status_t
546 sbd_link_lu(sbd_lu_t *sl)
547 {
548         sbd_lu_t *nsl;

550         mutex_enter(&sbd_lu_lock);
551         mutex_enter(&sl->sl_lock);
552         ASSERT(sl->sl_trans_op != SL_OP_NONE);

554         if (sl->sl_flags & SL_LINKED) {
555                 mutex_exit(&sbd_lu_lock);
556                 mutex_exit(&sl->sl_lock);
557                 return (SBD_ALREADY);
558         }
559         for (nsl = sbd_lu_list; nsl; nsl = nsl->sl_next) {
560                 if (strcmp(nsl->sl_name, sl->sl_name) == 0)
561                         break;
562         }
563         if (nsl) {
564                 mutex_exit(&sbd_lu_lock);
565                 mutex_exit(&sl->sl_lock);
566                 return (SBD_ALREADY);
567         }
568         sl->sl_next = sbd_lu_list;
569         sbd_lu_list = sl;
570         sl->sl_flags |= SL_LINKED;
571         mutex_exit(&sbd_lu_lock);
572         mutex_exit(&sl->sl_lock);
573         return (SBD_SUCCESS);
574 }

576 void
577 sbd_unlink_lu(sbd_lu_t *sl)
578 {
579         sbd_lu_t **ppnsl;

581         mutex_enter(&sbd_lu_lock);
582         mutex_enter(&sl->sl_lock);
583         ASSERT(sl->sl_trans_op != SL_OP_NONE);

585         ASSERT(sl->sl_flags & SL_LINKED);
586         for (ppnsl = &sbd_lu_list; *ppnsl; ppnsl = &((*ppnsl)->sl_next)) {
587                 if (*ppnsl == sl)
588                         break;
589         }
```



```
590         ASSERT(*ppnsl);
591         *ppnsl = (*ppnsl)->sl_next;
592         sl->sl_flags &= ~SL_LINKED;
593         mutex_exit(&sbd_lu_lock);
594         mutex_exit(&sl->sl_lock);
595 }

597 sbd_status_t
598 sbd_find_and_lock_lu(uint8_t *guid, uint8_t *meta_name, uint8_t op,
599     sbd_lu_t **ppsl)
600 {
601         sbd_lu_t *sl;
602         int found = 0;
603         sbd_status_t sret;

605         mutex_enter(&sbd_lu_lock);
606         for (sl = sbd_lu_list; sl; sl = sl->sl_next) {
607                 if (guid) {
608                         found = bcmp(sl->sl_device_id + 4, guid, 16) == 0;
609                 } else {
610                         found = strcmp(sl->sl_name, (char *)meta_name) == 0;
611                 }
612                 if (found)
613                         break;
614         }
615         if (!found) {
616                 mutex_exit(&sbd_lu_lock);
617                 return (SBD_NOT_FOUND);
618         }
619         mutex_enter(&sl->sl_lock);
620         if (sl->sl_trans_op == SL_OP_NONE) {
621                 sl->sl_trans_op = op;
622                 *ppsl = sl;
623                 sret = SBD_SUCCESS;
624         } else {
625                 sret = SBD_BUSY;
626         }
627         mutex_exit(&sl->sl_lock);
628         mutex_exit(&sbd_lu_lock);
629         return (sret);
630 }

632 sbd_status_t
633 sbd_read_meta(sbd_lu_t *sl, uint64_t offset, uint64_t size, uint8_t *buf)
634 {
635         uint64_t        meta_align;
636         uint64_t        starting_off;
637         uint64_t        data_off;
638         uint64_t        ending_off;
639         uint64_t        io_size;
640         uint8_t         *io_buf;
641         vnode_t         *vp;
642         sbd_status_t    ret;
643         ssize_t         resid;
644         int             vret;

646         ASSERT(sl->sl_flags & SL_META_OPENED);
647         if (sl->sl_flags & SL_SHARED_META) {
648                 meta_align = (((uint64_t)1) << sl->sl_data_blocksize_shift) - 1;
649                 vp = sl->sl_data_vp;
650                 ASSERT(vp);
651         } else {
652                 meta_align = (((uint64_t)1) << sl->sl_meta_blocksize_shift) - 1;
653                 if ((sl->sl_flags & SL_ZFS_META) == 0) {
654                         vp = sl->sl_meta_vp;
655                         ASSERT(vp);
```

```
656                 }
657         }
658         starting_off = offset & ~(meta_align);
659         data_off = offset & meta_align;
660         ending_off = (offset + size + meta_align) & (~meta_align);
661         if (ending_off > sl->sl_meta_size_used) {
662                 bzero(buf, size);
663                 if (starting_off >= sl->sl_meta_size_used) {
664                         return (SBD_SUCCESS);
665                 }
666                 ending_off = (sl->sl_meta_size_used + meta_align) &
667                     (~meta_align);
668                 if (size > (ending_off - (starting_off + data_off))) {
669                         size = ending_off - (starting_off + data_off);
670                 }
671         }
672         io_size = ending_off - starting_off;
673         io_buf = (uint8_t *)kmem_zalloc(io_size, KM_SLEEP);
674         ASSERT((starting_off + io_size) <= sl->sl_total_meta_size);

676         /*
677          * Don't proceed if the device has been closed
678          * This can occur on an access state change to standby or
679          * a delete. The writer lock is acquired before closing the
680          * lu. If importing, reading the metadata is valid, hence
681          * the check on SL_OP_IMPORT_LU.
682          */
683         rw_enter(&sl->sl_access_state_lock, RW_READER);
684         if ((sl->sl_flags & SL_MEDIA_LOADED) == 0 &&
685             sl->sl_trans_op != SL_OP_IMPORT_LU) {
686                 rw_exit(&sl->sl_access_state_lock);
687                 ret = SBD_FILEIO_FAILURE;
688                 goto sbd_read_meta_failure;
689         }
690         if (sl->sl_flags & SL_ZFS_META) {
691                 if ((ret = sbd_read_zfs_meta(sl, io_buf, io_size,
692                     starting_off)) != SBD_SUCCESS) {
693                         rw_exit(&sl->sl_access_state_lock);
694                         goto sbd_read_meta_failure;
695                 }
696         } else {
697                 vret = vn_rdwr(UIO_READ, vp, (caddr_t)io_buf, (ssize_t)io_size,
698                     (offset_t)starting_off, UIO_SYSSPACE, FRSYNC,
699                     RLIM64_INFINITY, CRED(), &resid);

701                 if (vret || resid) {
702                         ret = SBD_FILEIO_FAILURE | vret;
703                         rw_exit(&sl->sl_access_state_lock);
704                         goto sbd_read_meta_failure;
705                 }
706         }
707         rw_exit(&sl->sl_access_state_lock);

709         bcopy(io_buf + data_off, buf, size);
710         ret = SBD_SUCCESS;

712 sbd_read_meta_failure:
713         kmem_free(io_buf, io_size);
714         return (ret);
715 }

717 sbd_status_t
718 sbd_write_meta(sbd_lu_t *sl, uint64_t offset, uint64_t size, uint8_t *buf)
719 {
720         uint64_t        meta_align;
721         uint64_t        starting_off;
```



```
722         uint64_t        data_off;
723         uint64_t        ending_off;
724         uint64_t        io_size;
725         uint8_t         *io_buf;
726         vnode_t         *vp;
727         sbd_status_t    ret;
728         ssize_t         resid;
729         int             vret;

731         ASSERT(sl->sl_flags & SL_META_OPENED);
732         if (sl->sl_flags & SL_SHARED_META) {
733                 meta_align = (((uint64_t)1) << sl->sl_data_blocksize_shift) - 1;
734                 vp = sl->sl_data_vp;
735                 ASSERT(vp);
736         } else {
737                 meta_align = (((uint64_t)1) << sl->sl_meta_blocksize_shift) - 1;
738                 if ((sl->sl_flags & SL_ZFS_META) == 0) {
739                         vp = sl->sl_meta_vp;
740                         ASSERT(vp);
741                 }
742         }
743         starting_off = offset & ~(meta_align);
744         data_off = offset & meta_align;
745         ending_off = (offset + size + meta_align) & (~meta_align);
746         io_size = ending_off - starting_off;
747         io_buf = (uint8_t *)kmem_zalloc(io_size, KM_SLEEP);
748         ret = sbd_read_meta(sl, starting_off, io_size, io_buf);
749         if (ret != SBD_SUCCESS) {
750                 goto sbd_write_meta_failure;
751         }
752         bcopy(buf, io_buf + data_off, size);
753         /*
754          * Don't proceed if the device has been closed
755          * This can occur on an access state change to standby or
756          * a delete. The writer lock is acquired before closing the
757          * lu. If importing, reading the metadata is valid, hence
758          * the check on SL_OP_IMPORT_LU.
759          */
760         rw_enter(&sl->sl_access_state_lock, RW_READER);
761         if ((sl->sl_flags & SL_MEDIA_LOADED) == 0 &&
762             sl->sl_trans_op != SL_OP_IMPORT_LU) {
763                 rw_exit(&sl->sl_access_state_lock);
764                 ret = SBD_FILEIO_FAILURE;
765                 goto sbd_write_meta_failure;
766         }
767         if (sl->sl_flags & SL_ZFS_META) {
768                 if ((ret = sbd_write_zfs_meta(sl, io_buf, io_size,
769                     starting_off)) != SBD_SUCCESS) {
770                         rw_exit(&sl->sl_access_state_lock);
771                         goto sbd_write_meta_failure;
772                 }
773         } else {
774                 vret = vn_rdwr(UIO_WRITE, vp, (caddr_t)io_buf, (ssize_t)io_size,
775                     (offset_t)starting_off, UIO_SYSSPACE, FDSYNC,
776                     RLIM64_INFINITY, CRED(), &resid);

778                 if (vret || resid) {
779                         ret = SBD_FILEIO_FAILURE | vret;
780                         rw_exit(&sl->sl_access_state_lock);
781                         goto sbd_write_meta_failure;
782                 }
783         }
784         rw_exit(&sl->sl_access_state_lock);

786         ret = SBD_SUCCESS;
```

```
788 sbd_write_meta_failure:
789         kmem_free(io_buf, io_size);
790         return (ret);
791 }
792
793 uint8_t
794 sbd_calc_sum(uint8_t *buf, int size)
795 {
796         uint8_t s = 0;
797
798         while (size > 0)
799                 s += buf[--size];
800
801         return (s);
802 }
803
804 uint8_t
805 sbd_calc_section_sum(sm_section_hdr_t *sm, uint32_t sz)
806 {
807         uint8_t s, o;
808
809         o = sm->sms_chksum;
810         sm->sms_chksum = 0;
811         s = sbd_calc_sum((uint8_t *)sm, sz);
812         sm->sms_chksum = o;
813
814         return (s);
815 }
816
817 uint32_t
818 sbd_strlen(char *str, uint32_t maxlen)
819 {
820         uint32_t i;
821
822         for (i = 0; i < maxlen; i++) {
823                 if (str[i] == 0)
824                         return (i);
825         }
826         return (i);
827 }
828
829 void
830 sbd_swap_meta_start(sbd_meta_start_t *sm)
831 {
832         if (sm->sm_magic == SBD_MAGIC)
833                 return;
834         sm->sm_magic            = BSWAP_64(sm->sm_magic);
835         sm->sm_meta_size        = BSWAP_64(sm->sm_meta_size);
836         sm->sm_meta_size_used   = BSWAP_64(sm->sm_meta_size_used);
837         sm->sm_ver_major        = BSWAP_16(sm->sm_ver_major);
838         sm->sm_ver_minor        = BSWAP_16(sm->sm_ver_minor);
839         sm->sm_ver_subminor     = BSWAP_16(sm->sm_ver_subminor);
840 }
841
842 void
843 sbd_swap_section_hdr(sm_section_hdr_t *sm)
844 {
845         if (sm->sms_data_order == SMS_DATA_ORDER)
846                 return;
847         sm->sms_offset          = BSWAP_64(sm->sms_offset);
848         sm->sms_size            = BSWAP_32(sm->sms_size);
849         sm->sms_id              = BSWAP_16(sm->sms_id);
850         sm->sms_chksum          += SMS_DATA_ORDER - sm->sms_data_order;
851         sm->sms_data_order      = SMS_DATA_ORDER;
852 }
```



```
853
854 void
855 sbd_swap_lu_info_1_0(sbd_lu_info_1_0_t *sli)
856 {
857         sbd_swap_section_hdr(&sli->sli_sms_header);
858         if (sli->sli_data_order == SMS_DATA_ORDER)
859                 return;
860         sli->sli_sms_header.sms_chksum  += SMS_DATA_ORDER - sli->sli_data_order;
861         sli->sli_data_order             = SMS_DATA_ORDER;
862         sli->sli_total_store_size       = BSWAP_64(sli->sli_total_store_size);
863         sli->sli_total_meta_size        = BSWAP_64(sli->sli_total_meta_size);
864         sli->sli_lu_data_offset         = BSWAP_64(sli->sli_lu_data_offset);
865         sli->sli_lu_data_size           = BSWAP_64(sli->sli_lu_data_size);
866         sli->sli_flags                  = BSWAP_32(sli->sli_flags);
867         sli->sli_blocksize              = BSWAP_16(sli->sli_blocksize);
868 }
869
870 void
871 sbd_swap_lu_info_1_1(sbd_lu_info_1_1_t *sli)
872 {
873         sbd_swap_section_hdr(&sli->sli_sms_header);
874         if (sli->sli_data_order == SMS_DATA_ORDER)
875                 return;
876         sli->sli_sms_header.sms_chksum  += SMS_DATA_ORDER - sli->sli_data_order;
877         sli->sli_data_order             = SMS_DATA_ORDER;
878         sli->sli_flags                  = BSWAP_32(sli->sli_flags);
879         sli->sli_lu_size                = BSWAP_64(sli->sli_lu_size);
880         sli->sli_meta_fname_offset      = BSWAP_64(sli->sli_meta_fname_offset);
881         sli->sli_data_fname_offset      = BSWAP_64(sli->sli_data_fname_offset);
882         sli->sli_serial_offset          = BSWAP_64(sli->sli_serial_offset);
883         sli->sli_alias_offset           = BSWAP_64(sli->sli_alias_offset);
884         sli->sli_mgmt_url_offset        = BSWAP_64(sli->sli_mgmt_url_offset);
885 }
886
887 sbd_status_t
888 sbd_load_section_hdr(sbd_lu_t *sl, sm_section_hdr_t *sms)
889 {
890         sm_section_hdr_t        h;
891         uint64_t                st;
892         sbd_status_t            ret;
893
894         for (st = sl->sl_meta_offset + sizeof (sbd_meta_start_t);
895             st < sl->sl_meta_size_used; st += h.sms_size) {
896                 if ((ret = sbd_read_meta(sl, st, sizeof (sm_section_hdr_t),
897                     (uint8_t *)&h)) != SBD_SUCCESS) {
898                         return (ret);
899                 }
900                 if (h.sms_data_order != SMS_DATA_ORDER) {
901                         sbd_swap_section_hdr(&h);
902                 }
903                 if ((h.sms_data_order != SMS_DATA_ORDER) ||
904                     (h.sms_offset != st) || (h.sms_size < sizeof (h)) ||
905                     ((st + h.sms_size) > sl->sl_meta_size_used)) {
906                         return (SBD_META_CORRUPTED);
907                 }
908                 if (h.sms_id == sms->sms_id) {
909                         bcopy(&h, sms, sizeof (h));
910                         return (SBD_SUCCESS);
911                 }
912         }
913
914         return (SBD_NOT_FOUND);
915 }
916
917 sbd_status_t
918 sbd_load_meta_start(sbd_lu_t *sl)
919 {
```

```
920         sbd_meta_start_t *sm;
921         sbd_status_t ret;

923         /* Fake meta params initially */
924         sl->sl_total_meta_size = (uint64_t)-1;
925         sl->sl_meta_size_used = sl->sl_meta_offset + sizeof (sbd_meta_start_t);

927         sm = kmem_zalloc(sizeof (*sm), KM_SLEEP);
928         ret = sbd_read_meta(sl, sl->sl_meta_offset, sizeof (*sm),
929             (uint8_t *)sm);
930         if (ret != SBD_SUCCESS) {
931                 goto load_meta_start_failed;
932         }

934         if (sm->sm_magic != SBD_MAGIC) {
935                 sbd_swap_meta_start(sm);
936         }

938         if ((sm->sm_magic != SBD_MAGIC) || (sbd_calc_sum((uint8_t *)sm,
939             sizeof (*sm) - 1) != sm->sm_chksum)) {
940                 ret = SBD_META_CORRUPTED;
941                 goto load_meta_start_failed;
942         }

944         if (sm->sm_ver_major != SBD_VER_MAJOR) {
945                 ret = SBD_NOT_SUPPORTED;
946                 goto load_meta_start_failed;
947         }

949         sl->sl_total_meta_size = sm->sm_meta_size;
950         sl->sl_meta_size_used = sm->sm_meta_size_used;
951         ret = SBD_SUCCESS;

953 load_meta_start_failed:
954         kmem_free(sm, sizeof (*sm));
955         return (ret);
956 }

958 sbd_status_t
959 sbd_write_meta_start(sbd_lu_t *sl, uint64_t meta_size, uint64_t meta_size_used)
960 {
961         sbd_meta_start_t *sm;
962         sbd_status_t ret;

964         sm = (sbd_meta_start_t *)kmem_zalloc(sizeof (sbd_meta_start_t),
965             KM_SLEEP);

967         sm->sm_magic = SBD_MAGIC;
968         sm->sm_meta_size = meta_size;
969         sm->sm_meta_size_used = meta_size_used;
970         sm->sm_ver_major = SBD_VER_MAJOR;
971         sm->sm_ver_minor = SBD_VER_MINOR;
972         sm->sm_ver_subminor = SBD_VER_SUBMINOR;
973         sm->sm_chksum = sbd_calc_sum((uint8_t *)sm, sizeof (*sm) - 1);

975         ret = sbd_write_meta(sl, sl->sl_meta_offset, sizeof (*sm),
976             (uint8_t *)sm);
977         kmem_free(sm, sizeof (*sm));

979         return (ret);
980 }

982 sbd_status_t
983 sbd_read_meta_section(sbd_lu_t *sl, sm_section_hdr_t **ppsms, uint16_t sms_id)
984 {
985         sbd_status_t ret;
```



```
986         sm_section_hdr_t sms;
987         int alloced = 0;

989         mutex_enter(&sl->sl_metadata_lock);
990         if (((*ppsms) == NULL) || ((*ppsms)->sms_offset == 0)) {
991                 bzero(&sms, sizeof (sm_section_hdr_t));
992                 sms.sms_id = sms_id;
993                 if ((ret = sbd_load_section_hdr(sl, &sms)) != SBD_SUCCESS) {
994                         mutex_exit(&sl->sl_metadata_lock);
995                         return (ret);
996                 } else {
997                         if ((*ppsms) == NULL) {
998                                 *ppsms = (sm_section_hdr_t *)kmem_zalloc(
999                                     sms.sms_size, KM_SLEEP);
1000                                alloced = 1;
1001                        }
1002                        bcopy(&sms, *ppsms, sizeof (sm_section_hdr_t));
1003                }
1004        }

1006        ret = sbd_read_meta(sl, (*ppsms)->sms_offset, (*ppsms)->sms_size,
1007            (uint8_t *)(*ppsms));
1008        if (ret == SBD_SUCCESS) {
1009                uint8_t s;
1010                if ((*ppsms)->sms_data_order != SMS_DATA_ORDER)
1011                        sbd_swap_section_hdr(*ppsms);
1012                if ((*ppsms)->sms_id != SMS_ID_UNUSED) {
1013                        s = sbd_calc_section_sum(*ppsms, (*ppsms)->sms_size);
1014                        if (s != (*ppsms)->sms_chksum)
1015                                ret = SBD_META_CORRUPTED;
1016                }
1017        }
1018        mutex_exit(&sl->sl_metadata_lock);

1020        if ((ret != SBD_SUCCESS) && alloced)
1021                kmem_free(*ppsms, sms.sms_size);
1022        return (ret);
1023 }

1025 sbd_status_t
1026 sbd_load_section_hdr_unbuffered(sbd_lu_t *sl, sm_section_hdr_t *sms)
1027 {
1028         sbd_status_t    ret;

1030         /*
1031          * Bypass buffering and re-read the meta data from permanent storage.
1032          */
1033         if (sl->sl_flags & SL_ZFS_META) {
1034                 if ((ret = sbd_open_zfs_meta(sl)) != SBD_SUCCESS) {
1035                         return (ret);
1036                 }
1037         }
1038         /* Re-get the meta sizes into sl */
1039         if ((ret = sbd_load_meta_start(sl)) != SBD_SUCCESS) {
1040                 return (ret);
1041         }
1042         return (sbd_load_section_hdr(sl, sms));
1043 }

1045 sbd_status_t
1046 sbd_write_meta_section(sbd_lu_t *sl, sm_section_hdr_t *sms)
1047 {
1048         sm_section_hdr_t t;
1049         uint64_t off, s;
1050         uint64_t unused_start;
1051         sbd_status_t ret;
```

```
1052         sbd_status_t write_meta_ret = SBD_SUCCESS;
1053         uint8_t *cb;
1054         int meta_size_changed = 0;
1055         sm_section_hdr_t sms_before_unused = {0};

1057         mutex_enter(&sl->sl_metadata_lock);
1058 write_meta_section_again:
1059         if (sms->sms_offset) {
1060                 /*
1061                  * If the section already exists and the size is the
1062                  * same as this new data then overwrite in place. If
1063                  * the sizes are different then mark the existing as
1064                  * unused and look for free space.
1065                  */
1066                 ret = sbd_read_meta(sl, sms->sms_offset, sizeof (t),
1067                     (uint8_t *)&t);
1068                 if (ret != SBD_SUCCESS) {
1069                         mutex_exit(&sl->sl_metadata_lock);
1070                         return (ret);
1071                 }
1072                 if (t.sms_data_order != SMS_DATA_ORDER) {
1073                         sbd_swap_section_hdr(&t);
1074                 }
1075                 if (t.sms_id != sms->sms_id) {
1076                         mutex_exit(&sl->sl_metadata_lock);
1077                         return (SBD_INVALID_ARG);
1078                 }
1079                 if (t.sms_size == sms->sms_size) {
1080                         ret = sbd_write_meta(sl, sms->sms_offset,
1081                             sms->sms_size, (uint8_t *)sms);
1082                         mutex_exit(&sl->sl_metadata_lock);
1083                         return (ret);
1084                 }
1085                 sms_before_unused = t;

1087                 t.sms_id = SMS_ID_UNUSED;
1088                 /*
1089                  * For unused sections we only use chksum of the header. for
1090                  * all other sections, the chksum is for the entire section.
1091                  */
1092                 t.sms_chksum = sbd_calc_section_sum(&t, sizeof (t));
1093                 ret = sbd_write_meta(sl, t.sms_offset, sizeof (t),
1094                     (uint8_t *)&t);
1095                 if (ret != SBD_SUCCESS) {
1096                         mutex_exit(&sl->sl_metadata_lock);
1097                         return (ret);
1098                 }
1099                 sms->sms_offset = 0;
1100         } else {
1101                 /* Section location is unknown, search for it. */
1102                 t.sms_id = sms->sms_id;
1103                 t.sms_data_order = SMS_DATA_ORDER;
1104                 ret = sbd_load_section_hdr(sl, &t);
1105                 if (ret == SBD_SUCCESS) {
1106                         sms->sms_offset = t.sms_offset;
1107                         sms->sms_chksum =
1108                             sbd_calc_section_sum(sms, sms->sms_size);
1109                         goto write_meta_section_again;
1110                 } else if (ret != SBD_NOT_FOUND) {
1111                         mutex_exit(&sl->sl_metadata_lock);
1112                         return (ret);
1113                 }
1114         }

1116         /*
1117          * At this point we know that section does not already exist.
```



```
1118          * Find space large enough to hold the section or grow meta if
1119          * possible.
1120          */
1121         unused_start = 0;
1122         s = 0;  /* size of space found */

1124         /*
1125          * Search all sections for unused space of sufficient size.
1126          * The first one found is taken. Contiguous unused sections
1127          * will be combined.
1128          */
1129         for (off = sl->sl_meta_offset + sizeof (sbd_meta_start_t);
1130             off < sl->sl_meta_size_used; off += t.sms_size) {
1131                 ret = sbd_read_meta(sl, off, sizeof (t), (uint8_t *)&t);
1132                 if (ret != SBD_SUCCESS) {
1133                         mutex_exit(&sl->sl_metadata_lock);
1134                         return (ret);
1135                 }
1136                 if (t.sms_data_order != SMS_DATA_ORDER)
1137                         sbd_swap_section_hdr(&t);
1138                 if (t.sms_size == 0) {
1139                         mutex_exit(&sl->sl_metadata_lock);
1140                         return (SBD_META_CORRUPTED);
1141                 }
1142                 if (t.sms_id == SMS_ID_UNUSED) {
1143                         if (unused_start == 0)
1144                                 unused_start = off;
1145                         /*
1146                          * Calculate size of the unused space, break out
1147                          * if it satisfies the requirement.
1148                          */
1149                         s = t.sms_size - unused_start + off;
1150                         if ((s == sms->sms_size) || (s >= (sms->sms_size +
1151                             sizeof (t)))) {
1152                                 break;
1153                         } else {
1154                                 s = 0;
1155                         }
1156                 } else {
1157                         unused_start = 0;
1158                 }
1159         }

1161         off = (unused_start == 0) ? sl->sl_meta_size_used : unused_start;
1162         /*
1163          * If none found, how much room is at the end?
1164          * See if the data can be expanded.
1165          */
1166         if (s == 0) {
1167                 s = sl->sl_total_meta_size - off;
1168                 if (s >= sms->sms_size || !(sl->sl_flags & SL_SHARED_META)) {
1169                         s = sms->sms_size;
1170                         meta_size_changed = 1;
1171                 } else {
1172                         s = 0;
1173                 }
1174         }

1176         if (s == 0) {
1177                 mutex_exit(&sl->sl_metadata_lock);
1178                 return (SBD_ALLOC_FAILURE);
1179         }

1181         sms->sms_offset = off;
1182         sms->sms_chksum = sbd_calc_section_sum(sms, sms->sms_size);
1183         /*
```

```
1184          * Since we may have to write more than one section (current +
1185          * any unused), use a combined buffer.
1186          */
1187         cb = kmem_zalloc(s, KM_SLEEP);
1188         bcopy(sms, cb, sms->sms_size);
1189         if (s > sms->sms_size) {
1190                 t.sms_offset = off + sms->sms_size;
1191                 t.sms_size = s - sms->sms_size;
1192                 t.sms_id = SMS_ID_UNUSED;
1193                 t.sms_data_order = SMS_DATA_ORDER;
1194                 t.sms_chksum = sbd_calc_section_sum(&t, sizeof (t));
1195                 bcopy(&t, cb + sms->sms_size, sizeof (t));
1196         }
1197         /*
1198          * Two write events & statuses take place. Failure writing the
1199          * meta section takes precedence, can possibly be rolled back,
1200          * & gets reported. Else return status from writing the meta start.
1201          */
1202         ret = SBD_SUCCESS; /* Set a default, it's not always loaded below. */
1203         if (meta_size_changed) {
1204                 uint64_t old_meta_size;
1205                 uint64_t old_sz_used = sl->sl_meta_size_used; /* save a copy */
1206                 old_meta_size = sl->sl_total_meta_size; /* save a copy */

1208                 write_meta_ret = sbd_write_meta(sl, off, s, cb);
1209                 if (write_meta_ret == SBD_SUCCESS) {
1210                         sl->sl_meta_size_used = off + s;
1211                         if (sl->sl_total_meta_size < sl->sl_meta_size_used) {
1212                                 uint64_t meta_align =
1213                                     (((uint64_t)1) <<
1214                                     sl->sl_meta_blocksize_shift) - 1;
1215                                 sl->sl_total_meta_size =
1216                                     (sl->sl_meta_size_used + meta_align) &
1217                                     (~meta_align);
1218                         }
1219                         ret = sbd_write_meta_start(sl, sl->sl_total_meta_size,
1220                             sl->sl_meta_size_used);
1221                         if (ret != SBD_SUCCESS) {
1222                                 sl->sl_meta_size_used = old_sz_used;
1223                                 sl->sl_total_meta_size = old_meta_size;
1224                         }
1225                 } else {
1226                         sl->sl_meta_size_used = old_sz_used;
1227                         sl->sl_total_meta_size = old_meta_size;
1228                 }
1229         } else {
1230                 write_meta_ret = sbd_write_meta(sl, off, s, cb);
1231         }
1232         if ((write_meta_ret != SBD_SUCCESS) &&
1233             (sms_before_unused.sms_offset != 0)) {
1234                 sm_section_hdr_t new_sms;
1235                 sm_section_hdr_t *unused_sms;
1236                 /*
1237                  * On failure writing the meta section attempt to undo
1238                  * the change to unused.
1239                  * Re-read the meta data from permanent storage.
1240                  * The section id can't exist for undo to be possible.
1241                  * Read what should be the entire old section data and
1242                  * insure the old data's still present by validating
1243                  * against it's old checksum.
1244                  */
1245                 new_sms.sms_id = sms->sms_id;
1246                 new_sms.sms_data_order = SMS_DATA_ORDER;
1247                 if (sbd_load_section_hdr_unbuffered(sl, &new_sms) !=
1248                     SBD_NOT_FOUND) {
1249                         goto done;
```



```
1250                 }
1251                 unused_sms = kmem_zalloc(sms_before_unused.sms_size, KM_SLEEP);
1252                 if (sbd_read_meta(sl, sms_before_unused.sms_offset,
1253                     sms_before_unused.sms_size,
1254                     (uint8_t *)unused_sms) != SBD_SUCCESS) {
1255                         goto done;
1256                 }
1257                 if (unused_sms->sms_data_order != SMS_DATA_ORDER) {
1258                         sbd_swap_section_hdr(unused_sms);
1259                 }
1260                 if (unused_sms->sms_id != SMS_ID_UNUSED) {
1261                         goto done;
1262                 }
1263                 if (unused_sms->sms_offset != sms_before_unused.sms_offset) {
1264                         goto done;
1265                 }
1266                 if (unused_sms->sms_size != sms_before_unused.sms_size) {
1267                         goto done;
1268                 }
1269                 unused_sms->sms_id = sms_before_unused.sms_id;
1270                 if (sbd_calc_section_sum(unused_sms,
1271                     sizeof (sm_section_hdr_t)) !=
1272                     sbd_calc_section_sum(&sms_before_unused,
1273                     sizeof (sm_section_hdr_t))) {
1274                         goto done;
1275                 }
1276                 unused_sms->sms_chksum =
1277                     sbd_calc_section_sum(unused_sms, unused_sms->sms_size);
1278                 if (unused_sms->sms_chksum != sms_before_unused.sms_chksum) {
1279                         goto done;
1280                 }
1281                 (void) sbd_write_meta(sl, unused_sms->sms_offset,
1282                     sizeof (sm_section_hdr_t), (uint8_t *)unused_sms);
1283         }
1284 done:
1285         mutex_exit(&sl->sl_metadata_lock);
1286         kmem_free(cb, s);
1287         if (write_meta_ret != SBD_SUCCESS) {
1288                 return (write_meta_ret);
1289         }
1290         return (ret);
1291 }

1293 sbd_status_t
1294 sbd_write_lu_info(sbd_lu_t *sl)
1295 {
1296         sbd_lu_info_1_1_t *sli;
1297         int s;
1298         uint8_t *p;
1299         char *zvol_name = NULL;
1300         sbd_status_t ret;

1302         mutex_enter(&sl->sl_lock);

1304         s = sl->sl_serial_no_size;
1305         if ((sl->sl_flags & (SL_SHARED_META | SL_ZFS_META)) == 0) {
1306                 if (sl->sl_data_filename) {
1307                         s += strlen(sl->sl_data_filename) + 1;
1308                 }
1309         }
1310         if (sl->sl_flags & SL_ZFS_META) {
1311                 zvol_name = sbd_get_zvol_name(sl);
1312                 s += strlen(zvol_name) + 1;
1313         }
1314         if (sl->sl_alias) {
1315                 s += strlen(sl->sl_alias) + 1;
```

```
1316         }
1317         if (sl->sl_mgmt_url) {
1318                 s += strlen(sl->sl_mgmt_url) + 1;
1319         }
1320         sli = (sbd_lu_info_1_1_t *)kmem_zalloc(sizeof (*sli) + s, KM_SLEEP);
1321         p = sli->sli_buf;
1322         if ((sl->sl_flags & (SL_SHARED_META | SL_ZFS_META)) == 0) {
1323                 sli->sli_flags |= SLI_SEPARATE_META;
1324                 (void) strcpy((char *)p, sl->sl_data_filename);
1325                 sli->sli_data_fname_offset =
1326                     (uintptr_t)p - (uintptr_t)sli->sli_buf;
1327                 sli->sli_flags |= SLI_DATA_FNAME_VALID;
1328                 p += strlen(sl->sl_data_filename) + 1;
1329         }
1330         if (sl->sl_flags & SL_ZFS_META) {
1331                 (void) strcpy((char *)p, zvol_name);
1332                 sli->sli_meta_fname_offset =
1333                     (uintptr_t)p - (uintptr_t)sli->sli_buf;
1334                 sli->sli_flags |= SLI_META_FNAME_VALID | SLI_ZFS_META;
1335                 p += strlen(zvol_name) + 1;
1336                 kmem_free(zvol_name, strlen(zvol_name) + 1);
1337                 zvol_name = NULL;
1338         }
1339         if (sl->sl_alias) {
1340                 (void) strcpy((char *)p, sl->sl_alias);
1341                 sli->sli_alias_offset =
1342                     (uintptr_t)p - (uintptr_t)sli->sli_buf;
1343                 sli->sli_flags |= SLI_ALIAS_VALID;
1344                 p += strlen(sl->sl_alias) + 1;
1345         }
1346         if (sl->sl_mgmt_url) {
1347                 (void) strcpy((char *)p, sl->sl_mgmt_url);
1348                 sli->sli_mgmt_url_offset =
1349                     (uintptr_t)p - (uintptr_t)sli->sli_buf;
1350                 sli->sli_flags |= SLI_MGMT_URL_VALID;
1351                 p += strlen(sl->sl_mgmt_url) + 1;
1352         }
1353         if (sl->sl_flags & SL_WRITE_PROTECTED) {
1354                 sli->sli_flags |= SLI_WRITE_PROTECTED;
1355         }
1356         if (sl->sl_flags & SL_SAVED_WRITE_CACHE_DISABLE) {
1357                 sli->sli_flags |= SLI_WRITEBACK_CACHE_DISABLE;
1358         }
1359         if (sl->sl_flags & SL_VID_VALID) {
1360                 bcopy(sl->sl_vendor_id, sli->sli_vid, 8);
1361                 sli->sli_flags |= SLI_VID_VALID;
1362         }
1363         if (sl->sl_flags & SL_PID_VALID) {
1364                 bcopy(sl->sl_product_id, sli->sli_pid, 16);
1365                 sli->sli_flags |= SLI_PID_VALID;
1366         }
1367         if (sl->sl_flags & SL_REV_VALID) {
1368                 bcopy(sl->sl_revision, sli->sli_rev, 4);
1369                 sli->sli_flags |= SLI_REV_VALID;
1370         }
1371         if (sl->sl_serial_no_size) {
1372                 bcopy(sl->sl_serial_no, p, sl->sl_serial_no_size);
1373                 sli->sli_serial_size = sl->sl_serial_no_size;
1374                 sli->sli_serial_offset =
1375                     (uintptr_t)p - (uintptr_t)sli->sli_buf;
1376                 sli->sli_flags |= SLI_SERIAL_VALID;
1377                 p += sli->sli_serial_size;
1378         }
1379         sli->sli_lu_size = sl->sl_lu_size;
1380         sli->sli_data_blocksize_shift = sl->sl_data_blocksize_shift;
1381         sli->sli_data_order = SMS_DATA_ORDER;
```



```
1382         bcopy(sl->sl_device_id, sli->sli_device_id, 20);

1384         sli->sli_sms_header.sms_size = sizeof (*sli) + s;
1385         sli->sli_sms_header.sms_id = SMS_ID_LU_INFO_1_1;
1386         sli->sli_sms_header.sms_data_order = SMS_DATA_ORDER;

1388         mutex_exit(&sl->sl_lock);
1389         ret = sbd_write_meta_section(sl, (sm_section_hdr_t *)sli);
1390         kmem_free(sli, sizeof (*sli) + s);
1391         return (ret);
1392 }

1394 /*
1395  * Will scribble SL_UNMAP_ENABLED into sl_flags if we succeed.
1396  */
1397 static void
1398 do_unmap_setup(sbd_lu_t *sl)
1399 {
1400         ASSERT((sl->sl_flags & SL_UNMAP_ENABLED) == 0);

1402         if ((sl->sl_flags & SL_ZFS_META) == 0)
1403                 return; /* No UNMAP for you. */

1405         sl->sl_flags |= SL_UNMAP_ENABLED;
1406 }

1408 int
1409 sbd_populate_and_register_lu(sbd_lu_t *sl, uint32_t *err_ret)
1410 {
1411         stmf_lu_t *lu = sl->sl_lu;
1412         stmf_status_t ret;

1414         do_unmap_setup(sl);

1416         lu->lu_id = (scsi_devid_desc_t *)sl->sl_device_id;
1417         if (sl->sl_alias) {
1418                 lu->lu_alias = sl->sl_alias;
1419         } else {
1420                 lu->lu_alias = sl->sl_name;
1421         }
1422         if (sl->sl_access_state == SBD_LU_STANDBY) {
1423                 /* call set access state */
1424                 ret = stmf_set_lu_access(lu, STMF_LU_STANDBY);
1425                 if (ret != STMF_SUCCESS) {
1426                         *err_ret = SBD_RET_ACCESS_STATE_FAILED;
1427                         return (EIO);
1428                 }
1429         }
1430         /* set proxy_reg_cb_arg to meta filename */
1431         if (sl->sl_meta_filename) {
1432                 lu->lu_proxy_reg_arg = sl->sl_meta_filename;
1433                 lu->lu_proxy_reg_arg_len = strlen(sl->sl_meta_filename) + 1;
1434         } else {
1435                 lu->lu_proxy_reg_arg = sl->sl_data_filename;
1436                 lu->lu_proxy_reg_arg_len = strlen(sl->sl_data_filename) + 1;
1437         }
1438         lu->lu_lp = sbd_lp;
1439         lu->lu_task_alloc = sbd_task_alloc;
1440         lu->lu_new_task = sbd_new_task;
1441         lu->lu_dbuf_xfer_done = sbd_dbuf_xfer_done;
1442         lu->lu_send_status_done = sbd_send_status_done;
1443         lu->lu_task_free = sbd_task_free;
1444         lu->lu_abort = sbd_abort;
1445         lu->lu_dbuf_free = sbd_dbuf_free;
1446         lu->lu_ctl = sbd_ctl;
1447         lu->lu_info = sbd_info;
```

```
1448         sl->sl_state = STMF_STATE_OFFLINE;

1450         if ((ret = stmf_register_lu(lu)) != STMF_SUCCESS) {
1451                 stmf_trace(0, "Failed to register with framework, ret=%llx",
1452                     ret);
1453                 if (ret == STMF_ALREADY) {
1454                         *err_ret = SBD_RET_GUID_ALREADY_REGISTERED;
1455                 }
1456                 return (EIO);
1457         }

1459         *err_ret = 0;
1460         return (0);
1461 }

1463 int
1464 sbd_open_data_file(sbd_lu_t *sl, uint32_t *err_ret, int lu_size_valid,
1465     int vp_valid, int keep_open)
1466 {
1467         int ret;
1468         int flag;
1469         ulong_t nbits;
1470         uint64_t supported_size;
1471         vattr_t vattr;
1472         enum vtype vt;
1473         struct dk_cinfo dki;
1474         int unused;

1476         mutex_enter(&sl->sl_lock);
1477         if (vp_valid) {
1478                 goto odf_over_open;
1479         }
1480         if (sl->sl_data_filename[0] != '/') {
1481                 *err_ret = SBD_RET_DATA_PATH_NOT_ABSOLUTE;
1482                 mutex_exit(&sl->sl_lock);
1483                 return (EINVAL);
1484         }
1485         if ((ret = lookupname(sl->sl_data_filename, UIO_SYSSPACE, FOLLOW,
1486             NULLVPP, &sl->sl_data_vp)) != 0) {
1487                 *err_ret = SBD_RET_DATA_FILE_LOOKUP_FAILED;
1488                 mutex_exit(&sl->sl_lock);
1489                 return (ret);
1490         }
1491         sl->sl_data_vtype = vt = sl->sl_data_vp->v_type;
1492         VN_RELE(sl->sl_data_vp);
1493         if ((vt != VREG) && (vt != VCHR) && (vt != VBLK)) {
1494                 *err_ret = SBD_RET_WRONG_DATA_FILE_TYPE;
1495                 mutex_exit(&sl->sl_lock);
1496                 return (EINVAL);
1497         }
1498         if (sl->sl_flags & SL_WRITE_PROTECTED) {
1499                 flag = FREAD | FOFFMAX;
1500         } else {
1501                 flag = FREAD | FWRITE | FOFFMAX | FEXCL;
1502         }
1503         if ((ret = vn_open(sl->sl_data_filename, UIO_SYSSPACE, flag, 0,
1504             &sl->sl_data_vp, 0, 0)) != 0) {
1505                 *err_ret = SBD_RET_DATA_FILE_OPEN_FAILED;
1506                 mutex_exit(&sl->sl_lock);
1507                 return (ret);
1508         }
1509 odf_over_open:
1510         vattr.va_mask = AT_SIZE;
1511         if ((ret = VOP_GETATTR(sl->sl_data_vp, &vattr, 0, CRED(), NULL)) != 0) {
1512                 *err_ret = SBD_RET_DATA_FILE_GETATTR_FAILED;
1513                 goto odf_close_data_and_exit;
```



```
1514         }
1515         if ((vt != VREG) && (vattr.va_size == 0)) {
1516                 /*
1517                  * Its a zero byte block or char device. This cannot be
1518                  * a raw disk.
1519                  */
1520                 *err_ret = SBD_RET_WRONG_DATA_FILE_TYPE;
1521                 ret = EINVAL;
1522                 goto odf_close_data_and_exit;
1523         }
1524         /* sl_data_readable size includes any metadata. */
1525         sl->sl_data_readable_size = vattr.va_size;

1527         if (VOP_PATHCONF(sl->sl_data_vp, _PC_FILESIZEBITS, &nbits,
1528             CRED(), NULL) != 0) {
1529                 nbits = 0;
1530         }
1531         /* nbits cannot be greater than 64 */
1532         sl->sl_data_fs_nbits = (uint8_t)nbits;
1533         if (lu_size_valid) {
1534                 sl->sl_total_data_size = sl->sl_lu_size;
1535                 if (sl->sl_flags & SL_SHARED_META) {
1536                         sl->sl_total_data_size += SHARED_META_DATA_SIZE;
1537                 }
1538                 if ((nbits > 0) && (nbits < 64)) {
1539                         /*
1540                          * The expression below is correct only if nbits is
1541                          * positive and less than 64.
1542                          */
1543                         supported_size = (((uint64_t)1) << nbits) - 1;
1544                         if (sl->sl_total_data_size > supported_size) {
1545                                 *err_ret = SBD_RET_SIZE_NOT_SUPPORTED_BY_FS;
1546                                 ret = EINVAL;
1547                                 goto odf_close_data_and_exit;
1548                         }
1549                 }
1550         } else {
1551                 sl->sl_total_data_size = vattr.va_size;
1552                 if (sl->sl_flags & SL_SHARED_META) {
1553                         if (vattr.va_size > SHARED_META_DATA_SIZE) {
1554                                 sl->sl_lu_size = vattr.va_size -
1555                                     SHARED_META_DATA_SIZE;
1556                         } else {
1557                                 *err_ret = SBD_RET_FILE_SIZE_ERROR;
1558                                 ret = EINVAL;
1559                                 goto odf_close_data_and_exit;
1560                         }
1561                 } else {
1562                         sl->sl_lu_size = vattr.va_size;
1563                 }
1564         }
1565         if (sl->sl_lu_size < SBD_MIN_LU_SIZE) {
1566                 *err_ret = SBD_RET_FILE_SIZE_ERROR;
1567                 ret = EINVAL;
1568                 goto odf_close_data_and_exit;
1569         }
1570         if (sl->sl_lu_size &
1571             ((((uint64_t)1) << sl->sl_data_blocksize_shift) - 1)) {
1572                 *err_ret = SBD_RET_FILE_ALIGN_ERROR;
1573                 ret = EINVAL;
1574                 goto odf_close_data_and_exit;
1575         }
1576         /*
1577          * Get the minor device for direct zvol access
1578          */
1579         if (sl->sl_flags & SL_ZFS_META) {
```

```
1580                 if ((ret = VOP_IOCTL(sl->sl_data_vp, DKIOCINFO, (intptr_t)&dki,
1581                     FKIOCTL, kcred, &unused, NULL)) != 0) {
1582                         cmn_err(CE_WARN, "ioctl(DKIOCINFO) failed %d", ret);
1583                         /* zvol reserves 0, so this would fail later */
1584                         sl->sl_zvol_minor = 0;
1585                 } else {
1586                         sl->sl_zvol_minor = dki.dki_unit;
1587                         if (sbd_zvol_get_volume_params(sl) == 0)
1588                                 sl->sl_flags |= SL_CALL_ZVOL;
1589                 }
1590         }
1591         sl->sl_flags |= SL_MEDIA_LOADED;
1592         mutex_exit(&sl->sl_lock);
1593         return (0);

1595 odf_close_data_and_exit:
1596         if (!keep_open) {
1597                 (void) VOP_CLOSE(sl->sl_data_vp, flag, 1, 0, CRED(), NULL);
1598                 VN_RELE(sl->sl_data_vp);
1599         }
1600         mutex_exit(&sl->sl_lock);
1601         return (ret);
1602 }

1604 void
1605 sbd_close_lu(sbd_lu_t *sl)
1606 {
1607         int flag;

1609         if (((sl->sl_flags & SL_SHARED_META) == 0) &&
1610             (sl->sl_flags & SL_META_OPENED)) {
1611                 if (sl->sl_flags & SL_ZFS_META) {
1612                         rw_destroy(&sl->sl_zfs_meta_lock);
1613                         if (sl->sl_zfs_meta) {
1614                                 kmem_free(sl->sl_zfs_meta, ZAP_MAXVALUELEN / 2);
1615                                 sl->sl_zfs_meta = NULL;
1616                         }
1617                 } else {
1618                         flag = FREAD | FWRITE | FOFFMAX | FEXCL;
1619                         (void) VOP_CLOSE(sl->sl_meta_vp, flag, 1, 0,
1620                             CRED(), NULL);
1621                         VN_RELE(sl->sl_meta_vp);
1622                 }
1623                 sl->sl_flags &= ~SL_META_OPENED;
1624         }
1625         if (sl->sl_flags & SL_MEDIA_LOADED) {
1626                 if (sl->sl_flags & SL_WRITE_PROTECTED) {
1627                         flag = FREAD | FOFFMAX;
1628                 } else {
1629                         flag = FREAD | FWRITE | FOFFMAX | FEXCL;
1630                 }
1631                 (void) VOP_CLOSE(sl->sl_data_vp, flag, 1, 0, CRED(), NULL);
1632                 VN_RELE(sl->sl_data_vp);
1633                 sl->sl_flags &= ~SL_MEDIA_LOADED;
1634                 if (sl->sl_flags & SL_SHARED_META) {
1635                         sl->sl_flags &= ~SL_META_OPENED;
1636                 }
1637         }
1638 }

1640 int
1641 sbd_set_lu_standby(sbd_set_lu_standby_t *stlu, uint32_t *err_ret)
1642 {
1643         sbd_lu_t *sl;
1644         sbd_status_t sret;
1645         stmf_status_t stret;
```



```
1646         uint8_t old_access_state;

1648         sret = sbd_find_and_lock_lu(stlu->stlu_guid, NULL,
1649             SL_OP_MODIFY_LU, &sl);
1650         if (sret != SBD_SUCCESS) {
1651                 if (sret == SBD_BUSY) {
1652                         *err_ret = SBD_RET_LU_BUSY;
1653                         return (EBUSY);
1654                 } else if (sret == SBD_NOT_FOUND) {
1655                         *err_ret = SBD_RET_NOT_FOUND;
1656                         return (ENOENT);
1657                 }
1658                 *err_ret = SBD_RET_ACCESS_STATE_FAILED;
1659                 return (EIO);
1660         }

1662         old_access_state = sl->sl_access_state;
1663         sl->sl_access_state = SBD_LU_TRANSITION_TO_STANDBY;
1664         stret = stmf_set_lu_access((stmf_lu_t *)sl->sl_lu, STMF_LU_STANDBY);
1665         if (stret != STMF_SUCCESS) {
1666                 sl->sl_trans_op = SL_OP_NONE;
1667                 *err_ret = SBD_RET_ACCESS_STATE_FAILED;
1668                 sl->sl_access_state = old_access_state;
1669                 return (EIO);
1670         }

1672         /*
1673          * acquire the writer lock here to ensure we're not pulling
1674          * the rug from the vn_rdwr to the backing store
1675          */
1676         rw_enter(&sl->sl_access_state_lock, RW_WRITER);
1677         sbd_close_lu(sl);
1678         rw_exit(&sl->sl_access_state_lock);

1680         sl->sl_trans_op = SL_OP_NONE;
1681         return (0);
1682 }

1684 int
1685 sbd_close_delete_lu(sbd_lu_t *sl, int ret)
1686 {

1688         /*
1689          * acquire the writer lock here to ensure we're not pulling
1690          * the rug from the vn_rdwr to the backing store
1691          */
1692         rw_enter(&sl->sl_access_state_lock, RW_WRITER);
1693         sbd_close_lu(sl);
1694         rw_exit(&sl->sl_access_state_lock);

1696         if (sl->sl_flags & SL_LINKED)
1697                 sbd_unlink_lu(sl);
1698         mutex_destroy(&sl->sl_metadata_lock);
1699         mutex_destroy(&sl->sl_lock);
1700         rw_destroy(&sl->sl_pgr->pgr_lock);
1701         rw_destroy(&sl->sl_access_state_lock);
1702         if (sl->sl_serial_no_alloc_size) {
1703                 kmem_free(sl->sl_serial_no, sl->sl_serial_no_alloc_size);
1704         }
1705         if (sl->sl_data_fname_alloc_size) {
1706                 kmem_free(sl->sl_data_filename, sl->sl_data_fname_alloc_size);
1707         }
1708         if (sl->sl_alias_alloc_size) {
1709                 kmem_free(sl->sl_alias, sl->sl_alias_alloc_size);
1710         }
1711         if (sl->sl_mgmt_url_alloc_size) {
```

```
1712                kmem_free(sl->sl_mgmt_url, sl->sl_mgmt_url_alloc_size);
1713        }
1714        stmf_free(sl->sl_lu);
1715        return (ret);
1716 }

1718 int
1719 sbd_create_register_lu(sbd_create_and_reg_lu_t *slu, int struct_sz,
1720     uint32_t *err_ret)
1721 {
1722        char *namebuf;
1723        sbd_lu_t *sl;
1724        stmf_lu_t *lu;
1725        char *p;
1726        int sz;
1727        int alloc_sz;
1728        int ret = EIO;
1729        int flag;
1730        int wcd = 0;
1731        uint32_t hid = 0;
1732        enum vtype vt;

1734        sz = struct_sz - sizeof (sbd_create_and_reg_lu_t) + 8 + 1;

1736        *err_ret = 0;

1738        /* Lets validate various offsets */
1739        if (((slu->slu_meta_fname_valid) &&
1740            (slu->slu_meta_fname_off >= sz)) ||
1741            (slu->slu_data_fname_off >= sz) ||
1742            ((slu->slu_alias_valid) &&
1743            (slu->slu_alias_off >= sz)) ||
1744            ((slu->slu_mgmt_url_valid) &&
1745            (slu->slu_mgmt_url_off >= sz)) ||
1746            ((slu->slu_serial_valid) &&
1747            ((slu->slu_serial_off + slu->slu_serial_size) >= sz))) {
1748                return (EINVAL);
1749        }

1751        namebuf = kmem_zalloc(sz, KM_SLEEP);
1752        bcopy(slu->slu_buf, namebuf, sz - 1);
1753        namebuf[sz - 1] = 0;

1755        alloc_sz = sizeof (sbd_lu_t) + sizeof (sbd_pgr_t);
1756        if (slu->slu_meta_fname_valid) {
1757                alloc_sz += strlen(namebuf + slu->slu_meta_fname_off) + 1;
1758        }
1759        alloc_sz += strlen(namebuf + slu->slu_data_fname_off) + 1;
1760        if (slu->slu_alias_valid) {
1761                alloc_sz += strlen(namebuf + slu->slu_alias_off) + 1;
1762        }
1763        if (slu->slu_mgmt_url_valid) {
1764                alloc_sz += strlen(namebuf + slu->slu_mgmt_url_off) + 1;
1765        }
1766        if (slu->slu_serial_valid) {
1767                alloc_sz += slu->slu_serial_size;
1768        }

1770        lu = (stmf_lu_t *)stmf_alloc(STMF_STRUCT_STMF_LU, alloc_sz, 0);
1771        if (lu == NULL) {
1772                kmem_free(namebuf, sz);
1773                return (ENOMEM);
1774        }
1775        sl = (sbd_lu_t *)lu->lu_provider_private;
1776        bzero(sl, alloc_sz);
1777        sl->sl_lu = lu;
```



```
1778        sl->sl_alloc_size = alloc_sz;
1779        sl->sl_pgr = (sbd_pgr_t *)(sl + 1);
1780        rw_init(&sl->sl_pgr->pgr_lock, NULL, RW_DRIVER, NULL);
1781        mutex_init(&sl->sl_lock, NULL, MUTEX_DRIVER, NULL);
1782        mutex_init(&sl->sl_metadata_lock, NULL, MUTEX_DRIVER, NULL);
1783        rw_init(&sl->sl_access_state_lock, NULL, RW_DRIVER, NULL);
1784        p = ((char *)sl) + sizeof (sbd_lu_t) + sizeof (sbd_pgr_t);
1785        sl->sl_data_filename = p;
1786        (void) strcpy(sl->sl_data_filename, namebuf + slu->slu_data_fname_off);
1787        p += strlen(sl->sl_data_filename) + 1;
1788        sl->sl_meta_offset = SBD_META_OFFSET;
1789        sl->sl_access_state = SBD_LU_ACTIVE;
1790        if (slu->slu_meta_fname_valid) {
1791                sl->sl_alias = sl->sl_name = sl->sl_meta_filename = p;
1792                (void) strcpy(sl->sl_meta_filename, namebuf +
1793                    slu->slu_meta_fname_off);
1794                p += strlen(sl->sl_meta_filename) + 1;
1795        } else {
1796                sl->sl_alias = sl->sl_name = sl->sl_data_filename;
1797                if (sbd_is_zvol(sl->sl_data_filename)) {
1798                        sl->sl_flags |= SL_ZFS_META;
1799                        sl->sl_meta_offset = 0;
1800                } else {
1801                        sl->sl_flags |= SL_SHARED_META;
1802                        sl->sl_data_offset = SHARED_META_DATA_SIZE;
1803                        sl->sl_total_meta_size = SHARED_META_DATA_SIZE;
1804                        sl->sl_meta_size_used = 0;
1805                }
1806        }
1807        if (slu->slu_alias_valid) {
1808                sl->sl_alias = p;
1809                (void) strcpy(p, namebuf + slu->slu_alias_off);
1810                p += strlen(sl->sl_alias) + 1;
1811        }
1812        if (slu->slu_mgmt_url_valid) {
1813                sl->sl_mgmt_url = p;
1814                (void) strcpy(p, namebuf + slu->slu_mgmt_url_off);
1815                p += strlen(sl->sl_mgmt_url) + 1;
1816        }
1817        if (slu->slu_serial_valid) {
1818                sl->sl_serial_no = (uint8_t *)p;
1819                bcopy(namebuf + slu->slu_serial_off, sl->sl_serial_no,
1820                    slu->slu_serial_size);
1821                sl->sl_serial_no_size = slu->slu_serial_size;
1822                p += slu->slu_serial_size;
1823        }
1824        kmem_free(namebuf, sz);
1825        if (slu->slu_vid_valid) {
1826                bcopy(slu->slu_vid, sl->sl_vendor_id, 8);
1827                sl->sl_flags |= SL_VID_VALID;
1828        }
1829        if (slu->slu_pid_valid) {
1830                bcopy(slu->slu_pid, sl->sl_product_id, 16);
1831                sl->sl_flags |= SL_PID_VALID;
1832        }
1833        if (slu->slu_rev_valid) {
1834                bcopy(slu->slu_rev, sl->sl_revision, 4);
1835                sl->sl_flags |= SL_REV_VALID;
1836        }
1837        if (slu->slu_write_protected) {
1838                sl->sl_flags |= SL_WRITE_PROTECTED;
1839        }
1840        if (slu->slu_blksize_valid) {
1841                if (!ISP2(slu->slu_blksize) ||
  28                *if ((slu->slu_blksize & (slu->slu_blksize - 1)) ||*
1842                    (slu->slu_blksize > (32 * 1024)) ||
```

```
1843                    (slu->slu_blksize == 0)) {
1844                        *err_ret = SBD_RET_INVALID_BLKSIZE;
1845                        ret = EINVAL;
1846                        goto scm_err_out;
1847                }
1848                while ((1 << sl->sl_data_blocksize_shift) != slu->slu_blksize) {
1849                        sl->sl_data_blocksize_shift++;
1850                }
1851        } else {
1852                sl->sl_data_blocksize_shift = 9;        /* 512 by default */
1853                slu->slu_blksize = 512;
1854        }

1856        /* Now lets start creating meta */
1857        sl->sl_trans_op = SL_OP_CREATE_REGISTER_LU;
1858        if (sbd_link_lu(sl) != SBD_SUCCESS) {
1859                *err_ret = SBD_RET_FILE_ALREADY_REGISTERED;
1860                ret = EALREADY;
1861                goto scm_err_out;
1862        }

1864        /* 1st focus on the data store */
1865        if (slu->slu_lu_size_valid) {
1866                sl->sl_lu_size = slu->slu_lu_size;
1867        }
1868        ret = sbd_open_data_file(sl, err_ret, slu->slu_lu_size_valid, 0, 0);
1869        slu->slu_ret_filesize_nbits = sl->sl_data_fs_nbits;
1870        slu->slu_lu_size = sl->sl_lu_size;
1871        if (ret) {
1872                goto scm_err_out;
1873        }

1875        /*
1876         * Check if we were explicitly asked to disable/enable write
1877         * cache on the device, otherwise get current device setting.
1878         */
1879        if (slu->slu_writeback_cache_disable_valid) {
1880                if (slu->slu_writeback_cache_disable) {
1881                        /*
1882                         * Set write cache disable on the device. If it fails,
1883                         * we'll support it using sync/flush.
1884                         */
1885                        (void) sbd_wcd_set(1, sl);
1886                        wcd = 1;
1887                } else {
1888                        /*
1889                         * Set write cache enable on the device. If it fails,
1890                         * return an error.
1891                         */
1892                        if (sbd_wcd_set(0, sl) != SBD_SUCCESS) {
1893                                *err_ret = SBD_RET_WRITE_CACHE_SET_FAILED;
1894                                ret = EFAULT;
1895                                goto scm_err_out;
1896                        }
1897                }
1898        } else {
1899                sbd_wcd_get(&wcd, sl);
1900        }

1902        if (wcd) {
1903                sl->sl_flags |= SL_WRITEBACK_CACHE_DISABLE |
1904                    SL_SAVED_WRITE_CACHE_DISABLE;
1905        }

1907        if (sl->sl_flags & SL_SHARED_META) {
1908                goto over_meta_open;
```



```
1909        }
1910        if (sl->sl_flags & SL_ZFS_META) {
1911                if (sbd_create_zfs_meta_object(sl) != SBD_SUCCESS) {
1912                        *err_ret = SBD_RET_ZFS_META_CREATE_FAILED;
1913                        ret = ENOMEM;
1914                        goto scm_err_out;
1915                }
1916                sl->sl_meta_blocksize_shift = 0;
1917                goto over_meta_create;
1918        }
1919        if ((ret = lookupname(sl->sl_meta_filename, UIO_SYSSPACE, FOLLOW,
1920            NULLVPP, &sl->sl_meta_vp)) != 0) {
1921                *err_ret = SBD_RET_META_FILE_LOOKUP_FAILED;
1922                goto scm_err_out;
1923        }
1924        sl->sl_meta_vtype = vt = sl->sl_meta_vp->v_type;
1925        VN_RELE(sl->sl_meta_vp);
1926        if ((vt != VREG) && (vt != VCHR) && (vt != VBLK)) {
1927                *err_ret = SBD_RET_WRONG_META_FILE_TYPE;
1928                ret = EINVAL;
1929                goto scm_err_out;
1930        }
1931        if (vt == VREG) {
1932                sl->sl_meta_blocksize_shift = 0;
1933        } else {
1934                sl->sl_meta_blocksize_shift = 9;
1935        }
1936        flag = FREAD | FWRITE | FOFFMAX | FEXCL;
1937        if ((ret = vn_open(sl->sl_meta_filename, UIO_SYSSPACE, flag, 0,
1938            &sl->sl_meta_vp, 0, 0)) != 0) {
1939                *err_ret = SBD_RET_META_FILE_OPEN_FAILED;
1940                goto scm_err_out;
1941        }
1942 over_meta_create:
1943        sl->sl_total_meta_size = sl->sl_meta_offset + sizeof (sbd_meta_start_t);
1944        sl->sl_total_meta_size +=
1945            (((uint64_t)1) << sl->sl_meta_blocksize_shift) - 1;
1946        sl->sl_total_meta_size &=
1947            ~((((uint64_t)1) << sl->sl_meta_blocksize_shift) - 1);
1948        sl->sl_meta_size_used = 0;
1949 over_meta_open:
1950        sl->sl_flags |= SL_META_OPENED;

1952        sl->sl_device_id[3] = 16;
1953        if (slu->slu_guid_valid) {
1954                sl->sl_device_id[0] = 0xf1;
1955                sl->sl_device_id[1] = 3;
1956                sl->sl_device_id[2] = 0;
1957                bcopy(slu->slu_guid, sl->sl_device_id + 4, 16);
1958        } else {
1959                if (slu->slu_host_id_valid)
1960                        hid = slu->slu_host_id;
1961                if (!slu->slu_company_id_valid)
1962                        slu->slu_company_id = COMPANY_ID_SUN;
1963                if (stmf_scsilib_uniq_lu_id2(slu->slu_company_id, hid,
1964                    (scsi_devid_desc_t *)&sl->sl_device_id[0]) !=
1965                    STMF_SUCCESS) {
1966                        *err_ret = SBD_RET_META_CREATION_FAILED;
1967                        ret = EIO;
1968                        goto scm_err_out;
1969                }
1970                bcopy(sl->sl_device_id + 4, slu->slu_guid, 16);
1971        }

1973        /* Lets create the meta now */
1974        mutex_enter(&sl->sl_metadata_lock);
```

```
1975         if (sbd_write_meta_start(sl, sl->sl_total_meta_size,
1976             sizeof (sbd_meta_start_t)) != SBD_SUCCESS) {
1977                 mutex_exit(&sl->sl_metadata_lock);
1978                 *err_ret = SBD_RET_META_CREATION_FAILED;
1979                 ret = EIO;
1980                 goto scm_err_out;
1981         }
1982         mutex_exit(&sl->sl_metadata_lock);
1983         sl->sl_meta_size_used = sl->sl_meta_offset + sizeof (sbd_meta_start_t);

1985         if (sbd_write_lu_info(sl) != SBD_SUCCESS) {
1986                 *err_ret = SBD_RET_META_CREATION_FAILED;
1987                 ret = EIO;
1988                 goto scm_err_out;
1989         }

1991         if (sbd_pgr_meta_init(sl) != SBD_SUCCESS) {
1992                 *err_ret = SBD_RET_META_CREATION_FAILED;
1993                 ret = EIO;
1994                 goto scm_err_out;
1995         }

1997         /*
1998          * Update the zvol separately as this need only be called upon
1999          * completion of the metadata initialization.
2000          */
2001         if (sl->sl_flags & SL_ZFS_META) {
2002                 if (sbd_update_zfs_prop(sl) != SBD_SUCCESS) {
2003                         *err_ret = SBD_RET_META_CREATION_FAILED;
2004                         ret = EIO;
2005                         goto scm_err_out;
2006                 }
2007         }

2009         ret = sbd_populate_and_register_lu(sl, err_ret);
2010         if (ret) {
2011                 goto scm_err_out;
2012         }

2014         sl->sl_trans_op = SL_OP_NONE;
2015         atomic_inc_32(&sbd_lu_count);
2016         return (0);

2018 scm_err_out:
2019         return (sbd_close_delete_lu(sl, ret));
2020 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   82549 Thu Oct 23 10:42:12 2014
new/usr/src/uts/common/io/comstar/port/fct/discovery.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/scsi/scsi.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/disp.h>
#include <sys/byteorder.h>
#include <sys/varargs.h>
#include <sys/atomic.h>
#include <sys/sdt.h>

#include <sys/stmf.h>
#include <sys/stmf_ioctl.h>
#include <sys/portif.h>
#include <sys/fct.h>
#include <sys/fctio.h>

#include "fct_impl.h"
#include "discovery.h"

disc_action_t fct_handle_local_port_event(fct_i_local_port_t *iport);
disc_action_t fct_walk_discovery_queue(fct_i_local_port_t *iport);
disc_action_t fct_process_els(fct_i_local_port_t *iport,
    fct_i_remote_port_t *irp);
fct_status_t fct_send_accrjt(fct_cmd_t *cmd, uint8_t accrjt,
    uint8_t reason, uint8_t expl);
disc_action_t fct_link_init_complete(fct_i_local_port_t *iport);
fct_status_t fct_complete_previous_li_cmd(fct_i_local_port_t *iport);
fct_status_t fct_sol_plogi(fct_i_local_port_t *iport, uint32_t id,
    fct_cmd_t **ret_ppcmd, int implicit);
fct_status_t fct_sol_ct(fct_i_local_port_t *iport, uint32_t id,
    fct_cmd_t **ret_ppcmd, uint16_t opcode);
fct_status_t fct_ns_scr(fct_i_local_port_t *iport, uint32_t id,
```



```
    fct_cmd_t **ret_ppcmd);
static disc_action_t fct_check_cmdlist(fct_i_local_port_t *iport);
static disc_action_t fct_check_solcmd_queue(fct_i_local_port_t *iport);
static void fct_rscn_verify(fct_i_local_port_t *iport,
    uint8_t *rscn_req_payload, uint32_t rscn_req_size);
void fct_gid_cb(fct_i_cmd_t *icmd);

char *fct_els_names[] = { 0, "LS_RJT", "ACC", "PLOGI", "FLOGI", "LOGO",
                                "ABTX", "RCS", "RES", "RSS", "RSI", "ESTS",
                                "ESTC", "ADVC", "RTV", "RLS",
        /* 0x10 */              "ECHO", "TEST", "RRQ", "REC", "SRR", 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x20 */              "PRLI", "PRLO", "SCN", "TPLS",
                                "TPRLO", 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x30 */              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x40 */              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x50 */              "PDISC", "FDISC", "ADISC", "RNC", "FARP",
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 0x60 */              "FAN", "RSCN", "SCR", 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0,
        /* 0x70 */              "LINIT", "LPC", "LSTS", 0, 0, 0, 0, 0,
                                "RNID", "RLIR", "LIRR", 0, 0, 0, 0, 0
                };

extern uint32_t fct_rscn_options;

/*
 * NOTE: if anybody drops the iport_worker_lock then they should not return
 * DISC_ACTION_NO_WORK. Which also means, dont drop the lock if you have
 * nothing to do. Or else return DISC_ACTION_RESCAN or DISC_ACTION_DELAY_RESCAN.
 * But you cannot be infinitly returning those so have some logic to
 * determine that there is nothing to do without dropping the lock.
 */
void
fct_port_worker(void *arg)
{
        fct_local_port_t        *port = (fct_local_port_t *)arg;
        fct_i_local_port_t      *iport = (fct_i_local_port_t *)
            port->port_fct_private;
        disc_action_t           suggested_action;
        clock_t                 dl, short_delay, long_delay;
        int64_t                 tmp_delay;

        iport->iport_cmdcheck_clock = ddi_get_lbolt() +
            drv_usectohz(FCT_CMDLIST_CHECK_SECONDS * 1000000);
        short_delay = drv_usectohz(10000);
        long_delay = drv_usectohz(1000000);

        stmf_trace(iport->iport_alias, "iport is %p", iport);
        /* Discovery loop */
        mutex_enter(&iport->iport_worker_lock);
        atomic_or_32(&iport->iport_flags, IPORT_WORKER_RUNNING);
        while ((iport->iport_flags & IPORT_TERMINATE_WORKER) == 0) {
                suggested_action = DISC_ACTION_NO_WORK;
                /*
                 * Local port events are of the highest prioriy
                 */
                if (iport->iport_event_head) {
                        suggested_action |= fct_handle_local_port_event(iport);
                }

                /*
                 * We could post solicited ELSes to discovery queue.
                 * solicited CT will be processed inside fct_check_solcmd_queue
                 */
                if (iport->iport_solcmd_queue) {
```

```
128                     suggested_action |= fct_check_solcmd_queue(iport);
129             }
131             /*
132              * All solicited and unsolicited ELS will be handled here
133              */
134             if (iport->iport_rpwe_head) {
135                     suggested_action |= fct_walk_discovery_queue(iport);
136             }
138             /*
139              * We only process it when there's no outstanding link init CMD
140              */
141             if ((iport->iport_link_state == PORT_STATE_LINK_INIT_START) &&
142                 !(iport->iport_li_state & (LI_STATE_FLAG_CMD_WAITING |
143                 LI_STATE_FLAG_NO_LI_YET))) {
144                     suggested_action |= fct_process_link_init(iport);
145             }
147             /*
148              * We process cmd aborting in the end
149              */
150             if (iport->iport_abort_queue) {
151                     suggested_action |= fct_cmd_terminator(iport);
152             }
154             /*
155              * Check cmd max/free
156              */
157             if (iport->iport_cmdcheck_clock <= ddi_get_lbolt()) {
158                     suggested_action |= fct_check_cmdlist(iport);
159                     iport->iport_cmdcheck_clock = ddi_get_lbolt() +
160                         drv_usectohz(FCT_CMDLIST_CHECK_SECONDS * 1000000);
161                     iport->iport_max_active_ncmds = 0;
162             }
164             if (iport->iport_offline_prstate != FCT_OPR_DONE) {
165                     suggested_action |= fct_handle_port_offline(iport);
166             }
168             if (suggested_action & DISC_ACTION_RESCAN) {
169                     continue;
170             } else if (suggested_action & DISC_ACTION_DELAY_RESCAN) {
171                     /*
172                      * This is not very optimum as whoever returned
173                      * DISC_ACTION_DELAY_RESCAN must have dropped the lock
174                      * and more things might have queued up. But since
175                      * we are only doing small delays, it only delays
176                      * things by a few ms, which is okey.
177                      */
178                     if (suggested_action & DISC_ACTION_USE_SHORT_DELAY) {
179                             dl = short_delay;
180                     } else {
181                             dl = long_delay;
182                     }
183                     atomic_or_32(&iport->iport_flags,
184                         IPORT_WORKER_DOING_TIMEDWAIT);
185                     (void) cv_reltimedwait(&iport->iport_worker_cv,
186                         &iport->iport_worker_lock, dl, TR_CLOCK_TICK);
187                     atomic_and_32(&iport->iport_flags,
188                         ~IPORT_WORKER_DOING_TIMEDWAIT);
189             } else {
190                     atomic_or_32(&iport->iport_flags,
191                         IPORT_WORKER_DOING_WAIT);
192                     tmp_delay = (int64_t)(iport->iport_cmdcheck_clock -
193                         ddi_get_lbolt());
```



```
194                     if (tmp_delay < 0) {
195                             tmp_delay = (int64_t)short_delay;
196                     }
197                     (void) cv_reltimedwait(&iport->iport_worker_cv,
198                         &iport->iport_worker_lock, (clock_t)tmp_delay,
199                         TR_CLOCK_TICK);
200                     atomic_and_32(&iport->iport_flags,
201                         ~IPORT_WORKER_DOING_WAIT);
202             }
203     }
205     atomic_and_32(&iport->iport_flags, ~IPORT_WORKER_RUNNING);
206     mutex_exit(&iport->iport_worker_lock);
207 }
209 static char *topologies[] = { "Unknown", "Direct Pt-to-Pt", "Private Loop",
210                               "Unknown", "Unknown", "Fabric Pt-to-Pt",
211                               "Public Loop" };
213 void
214 fct_li_to_txt(fct_link_info_t *li, char *topology, char *speed)
215 {
216     uint8_t s = li->port_speed;
218     if (li->port_topology > PORT_TOPOLOGY_PUBLIC_LOOP) {
219             (void) sprintf(topology, "Invalid %02x", li->port_topology);
220     } else {
221             (void) strcpy(topology, topologies[li->port_topology]);
222     }
224     if ((s == 0) || ((s & 0xf00) != 0) || !ISP2(s)) {
 25     if ((s == 0) || ((s & 0xf00) != 0) || ((s & (s - 1)) != 0)) {
225             speed[0] = '?';
226     } else if (s == PORT_SPEED_10G) {
227             speed[0] = '1';
228             speed[1] = '0';
229             speed[2] = 'G';
230             speed[3] = 0;
231     } else {
232             speed[0] = '0' + li->port_speed;
233             speed[1] = 'G';
234             speed[2] = 0;
235     }
236 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   33072 Thu Oct 23 10:42:12 2014
new/usr/src/uts/common/io/drm/drmP.h
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * drmP.h -- Private header for Direct Rendering Manager -*- linux-c -*-
 * Created: Mon Jan  4 10:05:05 1999 by faith@precisioninsight.com
 */
/*
 * Copyright 1999 Precision Insight, Inc., Cedar Park, Texas.
 * Copyright 2000 VA Linux Systems, Inc., Sunnyvale, California.
 * Copyright (c) 2009, Intel Corporation.
 * All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice (including the next
 * paragraph) shall be included in all copies or substantial portions of the
 * Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
 * VA LINUX SYSTEMS AND/OR ITS SUPPLIERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
 * OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
 * ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * Authors:
 *    Rickard E. (Rik) Faith <faith@valinux.com>
 *    Gareth Hughes <gareth@valinux.com>
 *
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _DRMP_H
#define _DRMP_H

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <sys/cmn_err.h>
#include <sys/varargs.h>
#include <sys/pci.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>
#include <sys/pmem.h>
#include <sys/agpgart.h>
#include <sys/time.h>
#include "drm_atomic.h"
#include "drm.h"
```



```
#include "queue.h"
#include "drm_linux_list.h"

#ifndef __inline__
#define __inline__      inline
#endif

#if !defined(__FUNCTION__)
#if defined(C99)
#define __FUNCTION__ __func__
#else
#define __FUNCTION__    " "
#endif
#endif

/* DRM space units */
#define DRM_PAGE_SHIFT                  PAGESHIFT
#define DRM_PAGE_SIZE                   (1 << DRM_PAGE_SHIFT)
#define DRM_PAGE_OFFSET                 (DRM_PAGE_SIZE - 1)
#define DRM_PAGE_MASK                   ~(DRM_PAGE_SIZE - 1)
#define DRM_MB2PAGES(x)                 ((x) << 8)
#define DRM_PAGES2BYTES(x)              ((x) << DRM_PAGE_SHIFT)
#define DRM_BYTES2PAGES(x)              ((x) >> DRM_PAGE_SHIFT)
#define DRM_PAGES2KB(x)                 ((x) << 2)
#define DRM_ALIGNED(offset)             (((offset) & DRM_PAGE_OFFSET) == 0)

#define PAGE_SHIFT                      DRM_PAGE_SHIFT
#define PAGE_SIZE                       DRM_PAGE_SIZE

#define DRM_MAX_INSTANCES       8
#define DRM_DEVNODE             "drm"
#define DRM_UNOPENED            0
#define DRM_OPENED              1

#define DRM_HASH_SIZE           16 /* Size of key hash table */
#define DRM_KERNEL_CONTEXT      0  /* Change drm_resctx if changed */
#define DRM_RESERVED_CONTEXTS   1  /* Change drm_resctx if changed */

#define DRM_MEM_DMA        0
#define DRM_MEM_SAREA      1
#define DRM_MEM_DRIVER     2
#define DRM_MEM_MAGIC      3
#define DRM_MEM_IOCTLS     4
#define DRM_MEM_MAPS       5
#define DRM_MEM_BUFS       6
#define DRM_MEM_SEGS       7
#define DRM_MEM_PAGES      8
#define DRM_MEM_FILES     9
#define DRM_MEM_QUEUES    10
#define DRM_MEM_CMDS      11
#define DRM_MEM_MAPPINGS  12
#define DRM_MEM_BUFLISTS  13
#define DRM_MEM_DRMLISTS  14
#define DRM_MEM_TOTALDRM  15
#define DRM_MEM_BOUNDDRM  16
#define DRM_MEM_CTXBITMAP 17
#define DRM_MEM_STUB      18
#define DRM_MEM_SGLISTS   19
#define DRM_MEM_AGPLISTS  20
#define DRM_MEM_CTXLIST   21
#define DRM_MEM_MM              22
#define DRM_MEM_HASHTAB         23
#define DRM_MEM_OBJECTS         24

#define DRM_MAX_CTXBITMAP (PAGE_SIZE * 8)
#define DRM_MAP_HASH_OFFSET 0x10000000
```

```
#define DRM_MAP_HASH_ORDER 12
#define DRM_OBJECT_HASH_ORDER 12
#define DRM_FILE_PAGE_OFFSET_START ((0xFFFFFFFFUL >> PAGE_SHIFT) + 1)
#define DRM_FILE_PAGE_OFFSET_SIZE ((0xFFFFFFFFUL >> PAGE_SHIFT) * 16)
#define DRM_MM_INIT_MAX_PAGES 256


/* Internal types and structures */
#define DRM_ARRAY_SIZE(x) (sizeof (x) / sizeof (x[0]))
#define DRM_MIN(a, b) ((a) < (b) ? (a) : (b))
#define DRM_MAX(a, b) ((a) > (b) ? (a) : (b))

#define DRM_IF_VERSION(maj, min) (maj << 16 | min)

#define __OS_HAS_AGP    1

#define DRM_DEV_MOD     (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP)
#define DRM_DEV_UID     0
#define DRM_DEV_GID     0

#define DRM_CURRENTPID          ddi_get_pid()
#define DRM_SPINLOCK(l)         mutex_enter(l)
#define DRM_SPINUNLOCK(u)       mutex_exit(u)
#define DRM_SPINLOCK_ASSERT(l)
#define DRM_LOCK()      mutex_enter(&dev->dev_lock)
#define DRM_UNLOCK()    mutex_exit(&dev->dev_lock)
#define DRM_LOCK_OWNED()        ASSERT(mutex_owned(&dev->dev_lock))
#define spin_lock_irqsave(l, flag)              mutex_enter(l)
#define spin_unlock_irqrestore(u, flag) mutex_exit(u)
#define spin_lock(l)    mutex_enter(l)
#define spin_unlock(u)  mutex_exit(u)


#define DRM_UDELAY(sec)  delay(drv_usectohz(sec *1000))
#define DRM_MEMORYBARRIER()

typedef struct drm_file         drm_file_t;
typedef struct drm_device       drm_device_t;
typedef struct drm_driver_info drm_driver_t;

#define DRM_DEVICE      drm_device_t *dev = dev1
#define DRM_IOCTL_ARGS  \
        drm_device_t *dev1, intptr_t data, drm_file_t *fpriv, int mode

#define DRM_COPYFROM_WITH_RETURN(dest, src, size)       \
        if (ddi_copyin((src), (dest), (size), 0)) {     \
                DRM_ERROR("%s: copy from user failed", __func__);       \
                return (EFAULT);        \
        }

#define DRM_COPYTO_WITH_RETURN(dest, src, size) \
        if (ddi_copyout((src), (dest), (size), 0)) {    \
                DRM_ERROR("%s: copy to user failed", __func__); \
                return (EFAULT);        \
        }

#define DRM_COPY_FROM_USER(dest, src, size) \
        ddi_copyin((src), (dest), (size), 0) /* flag for src */

#define DRM_COPY_TO_USER(dest, src, size) \
        ddi_copyout((src), (dest), (size), 0) /* flags for dest */

#define DRM_COPY_FROM_USER_UNCHECKED(arg1, arg2, arg3)  \
        ddi_copyin((arg2), (arg1), (arg3), 0)

#define DRM_COPY_TO_USER_UNCHECKED(arg1, arg2, arg3)        \
```



```
        ddi_copyout((arg2), arg1, arg3, 0)

#define DRM_READ8(map, offset) \
        *(volatile uint8_t *)((uintptr_t)((map)->dev_addr) + (offset))
#define DRM_READ16(map, offset) \
        *(volatile uint16_t *)((uintptr_t)((map)->dev_addr) + (offset))
#define DRM_READ32(map, offset) \
        *(volatile uint32_t *)((uintptr_t)((map)->dev_addr) + (offset))
#define DRM_WRITE8(map, offset, val) \
        *(volatile uint8_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)
#define DRM_WRITE16(map, offset, val) \
        *(volatile uint16_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)
#define DRM_WRITE32(map, offset, val) \
        *(volatile uint32_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)

typedef struct drm_wait_queue {
        kcondvar_t      cv;
        kmutex_t        lock;
}wait_queue_head_t;

#define DRM_INIT_WAITQUEUE(q, pri)      \
{ \
        mutex_init(&(q)->lock, NULL, MUTEX_DRIVER, pri); \
        cv_init(&(q)->cv, NULL, CV_DRIVER, NULL);       \
}

#define DRM_FINI_WAITQUEUE(q)   \
{ \
        mutex_destroy(&(q)->lock);      \
        cv_destroy(&(q)->cv);   \
}

#define DRM_WAKEUP(q)   \
{ \
        mutex_enter(&(q)->lock); \
        cv_broadcast(&(q)->cv); \
        mutex_exit(&(q)->lock); \
}

#define jiffies ddi_get_lbolt()

#define DRM_WAIT_ON(ret, q, timeout, condition)                         \
        mutex_enter(&(q)->lock);                                        \
        while (!(condition)) {                                          \
                ret = cv_reltimedwait_sig(&(q)->cv, &(q)->lock, timeout,\
                    TR_CLOCK_TICK);                                     \
                if (ret == -1) {                                        \
                        ret = EBUSY;                                    \
                        break;                                          \
                } else if (ret == 0) {                                  \
                        ret = EINTR;                                    \
                        break;                                          \
                } else {                                                \
                        ret = 0;                                        \
                }                                                       \
        }                                                               \
        mutex_exit(&(q)->lock);

#define DRM_WAIT(ret, q, condition)  \
mutex_enter(&(q)->lock);        \
if (!(condition)) {     \
        ret = cv_timedwait_sig(&(q)->cv, &(q)->lock, jiffies + 30 * DRM_HZ); \
        if (ret == -1) {                                        \
                /* gfx maybe hang */    \
                if (!(condition))       \
                        ret = -2;       \
```

```
260         } else {        \
261                 ret = 0;        \
262         }       \
263 } \
264 mutex_exit(&(q)->lock);


267 #define DRM_GETSAREA()                                          \
268 {                                                               \
269         drm_local_map_t *map;                                   \
270         DRM_SPINLOCK_ASSERT(&dev->dev_lock);                    \
271         TAILQ_FOREACH(map, &dev->maplist, link) {               \
272                 if (map->type == _DRM_SHM &&                    \
273                     map->flags & _DRM_CONTAINS_LOCK) {          \
274                         dev_priv->sarea = map;                  \
275                         break;                                  \
276                 }                                               \
277         }                                                       \
278 }

280 #define LOCK_TEST_WITH_RETURN(dev, fpriv)                               \
281         if (!_DRM_LOCK_IS_HELD(dev->lock.hw_lock->lock) ||              \
282             dev->lock.filp != fpriv) {                                  \
283                 DRM_DEBUG("%s called without lock held", __func__);     \
284                 return (EINVAL);        \
285         }

287 #define DRM_IRQ_ARGS    caddr_t arg
288 #define IRQ_HANDLED             DDI_INTR_CLAIMED
289 #define IRQ_NONE                DDI_INTR_UNCLAIMED

291 enum {
292         DRM_IS_NOT_AGP,
293         DRM_IS_AGP,
294         DRM_MIGHT_BE_AGP
295 };

297 /* Capabilities taken from src/sys/dev/pci/pcireg.h. */
298 #ifndef PCIY_AGP
299 #define PCIY_AGP                0x02
300 #endif

302 #ifndef PCIY_EXPRESS
303 #define PCIY_EXPRESS            0x10
304 #endif

306 #define PAGE_ALIGN(addr)        (((addr) + DRM_PAGE_SIZE - 1) & DRM_PAGE_MASK)
307 #define DRM_SUSER(p)            (crgetsgid(p) == 0 || crgetsuid(p) == 0)

309 #define DRM_GEM_OBJIDR_HASHNODE 1024
310 #define idr_list_for_each(entry, head) \
311         for (int key = 0; key < DRM_GEM_OBJIDR_HASHNODE; key++) \
312                 list_for_each(entry, &(head)->next[key])

314 /*
315  * wait for 400 milliseconds
316  */
317 #define DRM_HZ                  drv_usectohz(400000)

319 typedef unsigned long dma_addr_t;
320 typedef uint64_t        u64;
321 typedef uint32_t        u32;
322 typedef uint16_t        u16;
323 typedef uint8_t         u8;
324 typedef uint_t          irqreturn_t;
```



```
326 #define DRM_SUPPORT     1
327 #define DRM_UNSUPPORT   0

329 #define __OS_HAS_AGP    1

 45 #define __offsetof(type, field) ((size_t)(&((type *)0)->field))
 46 #define offsetof(type, field)   __offsetof(type, field)

331 typedef struct drm_pci_id_list
332 {
333         int vendor;
334         int device;
335         long driver_private;
336         char *name;
337 } drm_pci_id_list_t;
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   18236 Thu Oct 23 10:42:12 2014
new/usr/src/uts/common/io/ib/adapters/hermon/hermon_cfg.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * hermon_cfg.c
 *    Hermon Configuration Profile Routines
 *
 *    Implements the routines necessary for initializing and (later) tearing
 *    down the list of Hermon configuration information.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/hermon/hermon.h>

/*
 * Below are the elements that make up the Hermon configuration profile.
 * For advanced users who wish to alter these values, this can be done via
 * the /etc/system file. By default, values are assigned to the number of
 * supported resources, either from the HCA's reported capacities or by
 * a by-design limit in the driver.
 */

/* Number of supported QPs, CQs and SRQs */
uint32_t hermon_log_num_qp              = HERMON_NUM_QP_SHIFT;
uint32_t hermon_log_num_cq              = HERMON_NUM_CQ_SHIFT;
uint32_t hermon_log_num_srq             = HERMON_NUM_SRQ_SHIFT;

/* Number of supported SGL per WQE for SQ/RQ, and for SRQ */
/* XXX use the same for all queues if limitation in srq.h is resolved */
uint32_t hermon_wqe_max_sgl             = HERMON_NUM_SGL_PER_WQE;
uint32_t hermon_srq_max_sgl             = HERMON_SRQ_MAX_SGL;
```



```
/* Maximum "responder resources" (in) and "initiator depth" (out) per QP */
uint32_t hermon_log_num_rdb_per_qp      = HERMON_LOG_NUM_RDB_PER_QP;

/*
 * Number of multicast groups (MCGs), number of QP per MCG, and the number
 * of entries (from the total number) in the multicast group "hash table"
 */
uint32_t hermon_log_num_mcg             = HERMON_NUM_MCG_SHIFT;
uint32_t hermon_log_num_qp_per_mcg      = HERMON_NUM_QP_PER_MCG;
uint32_t hermon_log_num_mcg_hash        = HERMON_NUM_MCG_HASH_SHIFT;

/* Number of UD AVs */
uint32_t hermon_log_num_ah              = HERMON_NUM_AH_SHIFT;

/* Number of EQs and their default size */
uint32_t hermon_log_num_eq              = HERMON_NUM_EQ_SHIFT;
uint32_t hermon_log_eq_sz               = HERMON_DEFAULT_EQ_SZ_SHIFT;

/*
 * Number of supported MPTs, MTTs and also the maximum MPT size.
 */
uint32_t hermon_log_num_mtt             = HERMON_NUM_MTT_SHIFT;
uint32_t hermon_log_num_dmpt            = HERMON_NUM_DMPT_SHIFT;
uint32_t hermon_log_max_mrw_sz          = HERMON_MAX_MEM_MPT_SHIFT;

/*
 * Number of supported UAR (User Access Regions) for this HCA.
 * We could in the future read in uar_sz from devlim, and thus
 * derive the number of UAR. Since this is derived from PAGESIZE,
 * however, this means that x86 systems would have twice as many
 * UARs as SPARC systems. Therefore for consistency's sake, we will
 * just use 1024 pages, which is the maximum on SPARC systems.
 */
uint32_t hermon_log_num_uar             = HERMON_NUM_UAR_SHIFT;

/*
 * Number of remaps allowed for FMR before a sync is required.  This value
 * determines how many times we can fmr_deregister() before the underlying fmr
 * framework places the region to wait for an MTT_SYNC operation, cleaning up
 * the old mappings.
 */
uint32_t hermon_fmr_num_remaps          = HERMON_FMR_MAX_REMAPS;

/*
 * Number of supported Hermon mailboxes ("In" and "Out") and their maximum
 * sizes, respectively
 */
uint32_t hermon_log_num_inmbox          = HERMON_NUM_MAILBOXES_SHIFT;
uint32_t hermon_log_num_outmbox         = HERMON_NUM_MAILBOXES_SHIFT;
uint32_t hermon_log_inmbox_size         = HERMON_MBOX_SIZE_SHIFT;
uint32_t hermon_log_outmbox_size        = HERMON_MBOX_SIZE_SHIFT;
uint32_t hermon_log_num_intr_inmbox     = HERMON_NUM_INTR_MAILBOXES_SHIFT;
uint32_t hermon_log_num_intr_outmbox    = HERMON_NUM_INTR_MAILBOXES_SHIFT;

/* Number of supported Protection Domains (PD) */
uint32_t hermon_log_num_pd              = HERMON_NUM_PD_SHIFT;

/*
 * Number of total supported PKeys per PKey table (i.e.
 * per port).  Also the number of SGID per GID table.
 */
uint32_t hermon_log_max_pkeytbl         = HERMON_NUM_PKEYTBL_SHIFT;
uint32_t hermon_log_max_gidtbl          = HERMON_NUM_GIDTBL_SHIFT;

/* Maximum supported MTU and portwidth */
```

```
128 uint32_t hermon_max_mtu                 = HERMON_MAX_MTU;
129 uint32_t hermon_max_port_width          = HERMON_MAX_PORT_WIDTH;

131 /* Number of supported Virtual Lanes (VL) */
132 uint32_t hermon_max_vlcap               = HERMON_MAX_VLCAP;

134 /*
135  * Whether or not to use the built-in (i.e. in firmware) agents for QP0 and
136  * QP1, respectively.
137  */
138 uint32_t hermon_qp0_agents_in_fw        = 0;
139 uint32_t hermon_qp1_agents_in_fw        = 0;

141 /*
142  * Whether DMA mappings should bypass the PCI IOMMU or not.
143  * hermon_iommu_bypass is a global setting for all memory addresses.
144  */
145 uint32_t hermon_iommu_bypass            = 1;

147 /*
148  * Whether *DATA* buffers should be bound w/ Relaxed Ordering (RO) turned on
149  * via the SW workaround (HCAs don't support RO in HW).  Defaulted on,
150  * though care must be taken w/ some Userland clients that *MAY* have
151  * peeked in the data to understand when data xfer was done - MPI does
152  * as an efficiency
153  */

155 uint32_t hermon_kernel_data_ro          = HERMON_RO_ENABLED;    /* default */
156 uint32_t hermon_user_data_ro            = HERMON_RO_ENABLED;    /* default */

158 /*
159  * Whether Hermon should use MSI (Message Signaled Interrupts), if available.
160  * Note: 0 indicates 'legacy interrupt', 1 indicates MSI (if available)
161  */
162 uint32_t hermon_use_msi_if_avail        = 1;

164 /*
165  * This is a patchable variable that determines the time we will wait after
166  * initiating SW reset before we do our first read from Hermon config space.
167  * If this value is set too small (less than the default 100ms), it is
168  * possible for Hermon hardware to be unready to respond to the config cycle
169  * reads.  This could cause master abort on the PCI bridge.  Note: If
170  * "hermon_sw_reset_delay" is set to zero, then no software reset of the Hermon
171  * device will be attempted.
172  */
173 uint32_t hermon_sw_reset_delay          = HERMON_SW_RESET_DELAY;

175 /*
176  * These are patchable variables for hermon command polling. The poll_delay is
177  * the number of usec to wait in-between calls to poll the 'go' bit.  The
178  * poll_max is the total number of usec to loop in waiting for the 'go' bit to
179  * clear.
180  */
181 uint32_t hermon_cmd_poll_delay          = HERMON_CMD_POLL_DELAY;
182 uint32_t hermon_cmd_poll_max            = HERMON_CMD_POLL_MAX;

184 /*
185  * This is a patchable variable that determines the frequency with which
186  * the AckReq bit will be set in outgoing RC packets.  The AckReq bit will be
187  * set in at least every 2^hermon_qp_ackreq_freq packets (but at least once
188  * per message, i.e. in the last packet).  Tuning this value can increase
189  * IB fabric utilization by cutting down on the number of unnecessary ACKs.
190  */
191 uint32_t hermon_qp_ackreq_freq          = HERMON_QP_ACKREQ_FREQ;

193 static void hermon_cfg_wqe_sizes(hermon_state_t *state,
```



```
194     hermon_cfg_profile_t *cp);
195 #ifdef __sparc
196 static void hermon_check_iommu_bypass(hermon_state_t *state,
197     hermon_cfg_profile_t *cp);
198 #endif

200 /*
201  * hermon_cfg_profile_init_phase1()
202  *    Context: Only called from attach() path context
203  */
204 int
205 hermon_cfg_profile_init_phase1(hermon_state_t *state)
206 {
207         hermon_cfg_profile_t    *cp;

209         /*
210          * Allocate space for the configuration profile structure
211          */
212         cp = (hermon_cfg_profile_t *)kmem_zalloc(sizeof (hermon_cfg_profile_t),
213             KM_SLEEP);

215         /*
216          * Common to all profiles.
217          */
218         cp->cp_qp0_agents_in_fw         = hermon_qp0_agents_in_fw;
219         cp->cp_qp1_agents_in_fw         = hermon_qp1_agents_in_fw;
220         cp->cp_sw_reset_delay           = hermon_sw_reset_delay;
221         cp->cp_cmd_poll_delay           = hermon_cmd_poll_delay;
222         cp->cp_cmd_poll_max             = hermon_cmd_poll_max;
223         cp->cp_ackreq_freq              = hermon_qp_ackreq_freq;
224         cp->cp_fmr_max_remaps           = hermon_fmr_num_remaps;

226         /*
227          * Although most of the configuration is enabled in "phase2" of the
228          * cfg_profile_init, we have to setup the OUT mailboxes soon, since
229          * they are used immediately after this "phase1" completes, to run the
230          * firmware and get the device limits, which we'll need for 'phase2'.
231          * That's done in rsrc_init_phase1, called shortly after we do this
232          * and the sw reset - see hermon.c
233          */
234         if (state->hs_cfg_profile_setting == HERMON_CFG_MEMFREE) {
235                 cp->cp_log_num_outmbox          = hermon_log_num_outmbox;
236                 cp->cp_log_outmbox_size         = hermon_log_outmbox_size;
237                 cp->cp_log_num_inmbox           = hermon_log_num_inmbox;
238                 cp->cp_log_inmbox_size          = hermon_log_inmbox_size;
239                 cp->cp_log_num_intr_inmbox      = hermon_log_num_intr_inmbox;
240                 cp->cp_log_num_intr_outmbox     = hermon_log_num_intr_outmbox;

242         } else {
243                 return (DDI_FAILURE);
244         }

246         /*
247          * Set IOMMU bypass or not.  Ensure consistency of flags with
248          * architecture type.
249          */
250 #ifdef __sparc
251         if (hermon_iommu_bypass == 1) {
252                 hermon_check_iommu_bypass(state, cp);
253         } else {
254                 cp->cp_iommu_bypass = HERMON_BINDMEM_NORMAL;
255         }
256 #else
257         cp->cp_iommu_bypass = HERMON_BINDMEM_NORMAL;
258 #endif
```

```
260         /* Attach the configuration profile to Hermon softstate */
261         state->hs_cfg_profile = cp;
262
263         return (DDI_SUCCESS);
264 }
265
266 /*
267  * hermon_cfg_profile_init_phase2()
268  *    Context: Only called from attach() path context
269  */
270 int
271 hermon_cfg_profile_init_phase2(hermon_state_t *state)
272 {
273         hermon_cfg_profile_t    *cp;
274         hermon_hw_querydevlim_t *devlim;
275         hermon_hw_query_port_t  *port;
276         uint32_t                num, size;
277         int                     i;
278
279         /* Read in the device limits */
280         devlim = &state->hs_devlim;
281         /* and the port information */
282         port = &state->hs_queryport;
283
284         /* Read the configuration profile */
285         cp = state->hs_cfg_profile;
286
287         /*
288          * We configure all Hermon HCAs with the same profile, which
289          * is based upon the default value assignments above. If we want to
290          * add additional profiles in the future, they can be added here.
291          * Note the reference to "Memfree" is a holdover from Arbel/Sinai
292          */
293         if (state->hs_cfg_profile_setting != HERMON_CFG_MEMFREE) {
294                 return (DDI_FAILURE);
295         }
296
297         /*
298          * Note for most configuration parameters, we use the lesser of our
299          * desired configuration value or the device-defined maximum value.
300          */
301         cp->cp_log_num_mtt      = min(hermon_log_num_mtt, devlim->log_max_mtt);
302         cp->cp_log_num_dmpt = min(hermon_log_num_dmpt, devlim->log_max_dmpt);
303         cp->cp_log_num_cmpt     = HERMON_LOG_CMPT_PER_TYPE + 2; /* times 4, */
304                                                                 /* per PRM */
305         cp->cp_log_max_mrw_sz   = min(hermon_log_max_mrw_sz,
306             devlim->log_max_mrw_sz);
307         cp->cp_log_num_pd       = min(hermon_log_num_pd, devlim->log_max_pd);
308         cp->cp_log_num_qp       = min(hermon_log_num_qp, devlim->log_max_qp);
309         cp->cp_log_num_cq       = min(hermon_log_num_cq, devlim->log_max_cq);
310         cp->cp_log_num_srq      = min(hermon_log_num_srq, devlim->log_max_srq);
311         cp->cp_log_num_eq       = min(hermon_log_num_eq, devlim->log_max_eq);
312         cp->cp_log_eq_sz        = min(hermon_log_eq_sz, devlim->log_max_eq_sz);
313         cp->cp_log_num_rdb      = cp->cp_log_num_qp +
314             min(hermon_log_num_rdb_per_qp, devlim->log_max_ra_req_qp);
315         cp->cp_hca_max_rdma_in_qp = cp->cp_hca_max_rdma_out_qp =
316             1 << min(hermon_log_num_rdb_per_qp, devlim->log_max_ra_req_qp);
317         cp->cp_num_qp_per_mcg   = max(hermon_num_qp_per_mcg,
318             HERMON_NUM_QP_PER_MCG_MIN);
319         cp->cp_num_qp_per_mcg   = min(cp->cp_num_qp_per_mcg,
320             (1 << devlim->log_max_qp_mcg) - 8);
321         cp->cp_num_qp_per_mcg   = (1 << highbit(cp->cp_num_qp_per_mcg + 7)) - 8;
322         cp->cp_log_num_mcg      = min(hermon_log_num_mcg, devlim->log_max_mcg);
323         cp->cp_log_num_mcg_hash = hermon_log_num_mcg_hash;
324
325         /* until srq_resize is debugged, disable it */
```



```
326         cp->cp_srq_resize_enabled = 0;
327
328         /* cp->cp_log_num_uar   = hermon_log_num_uar; */
329         /*
330          * now, we HAVE to calculate the number of UAR pages, so that we can
331          * get the blueflame stuff correct as well
332          */
333
334         size = devlim->log_max_uar_sz;
335         /* 1MB (2^^20) times size (2^^size) / sparc_pg (2^^13) */
336         num = (20 + size) - 13;         /* XXX - consider using PAGESHIFT */
337         if (devlim->blu_flm)
338                 num -= 1;       /* if blueflame, only half the size for UARs */
339         cp->cp_log_num_uar      = min(hermon_log_num_uar, num);
340
341
342         /* while we're at it, calculate the index of the kernel uar page */
343         /* either the reserved uar's or 128, whichever is smaller */
344         state->hs_kernel_uar_index = (devlim->num_rsvd_uar > 128) ?
345             devlim->num_rsvd_uar : 128;
346
347         cp->cp_log_max_pkeytbl  = port->log_max_pkey;
348
349         cp->cp_log_max_qp_sz    = devlim->log_max_qp_sz;
350         cp->cp_log_max_cq_sz    = devlim->log_max_cq_sz;
351         cp->cp_log_max_srq_sz   = devlim->log_max_srq_sz;
352         cp->cp_log_max_gidtbl   = port->log_max_gid;
353         cp->cp_max_mtu          = port->ib_mtu; /* XXX now from query_port */
354         cp->cp_max_port_width   = port->ib_port_wid;  /* now from query_port */
355         cp->cp_max_vlcap        = port->max_vl;
356         cp->cp_log_num_ah       = hermon_log_num_ah;
357
358         /* Paranoia, ensure no arrays indexed by port_num are out of bounds */
359         cp->cp_num_ports        = devlim->num_ports;
360         if (cp->cp_num_ports > HERMON_MAX_PORTS) {
361                 cmn_err(CE_CONT, "device has more ports (%d) than are "
362                     "supported; Using %d ports\n",
363                     cp->cp_num_ports, HERMON_MAX_PORTS);
364                 cp->cp_num_ports = HERMON_MAX_PORTS;
365         };
366
367         /* allocate variable sized arrays */
368         for (i = 0; i < HERMON_MAX_PORTS; i++) {
369                 state->hs_pkey[i] = kmem_zalloc((1 << cp->cp_log_max_pkeytbl) *
370                     sizeof (ib_pkey_t), KM_SLEEP);
371                 state->hs_guid[i] = kmem_zalloc((1 << cp->cp_log_max_gidtbl) *
372                     sizeof (ib_guid_t), KM_SLEEP);
373         }
374
375         /* Determine WQE sizes from requested max SGLs */
376         hermon_cfg_wqe_sizes(state, cp);
377
378         /* Set whether to use MSIs or not */
379         cp->cp_use_msi_if_avail = hermon_use_msi_if_avail;
380
381 #if !defined(_ELF64)
382         /*
383          * Need to reduce the hermon kernel virtual memory footprint
384          * on 32-bit kernels.
385          */
386         cp->cp_log_num_mtt      -= 6;
387         cp->cp_log_num_dmpt     -= 6;
388         cp->cp_log_num_pd       -= 6;
389         cp->cp_log_num_qp       -= 6;
390         cp->cp_log_num_cq       -= 6;
391         cp->cp_log_num_srq      -= 6;
```

```
392         cp->cp_log_num_rdb      = cp->cp_log_num_qp +
393             min(hermon_log_num_rdb_per_qp, devlim->log_max_ra_req_qp);
394         cp->cp_hca_max_rdma_in_qp = cp->cp_hca_max_rdma_out_qp =
395             1 << min(hermon_log_num_rdb_per_qp, devlim->log_max_ra_req_qp);
396 #endif

398         return (DDI_SUCCESS);
399 }


402 /*
403  * hermon_cfg_profile_fini()
404  *    Context: Only called from attach() and/or detach() path contexts
405  */
406 void
407 hermon_cfg_profile_fini(hermon_state_t *state)
408 {
409         /*
410          * Free up the space for configuration profile
411          */
412         kmem_free(state->hs_cfg_profile, sizeof (hermon_cfg_profile_t));
413 }


416 /*
417  * hermon_cfg_wqe_sizes()
418  *    Context: Only called from attach() path context
419  */
420 static void
421 hermon_cfg_wqe_sizes(hermon_state_t *state, hermon_cfg_profile_t *cp)
422 {
423         uint_t  max_size, log2;
424         uint_t  max_sgl, real_max_sgl;

426         /*
427          * Get the requested maximum number SGL per WQE from the Hermon
428          * patchable variable
429          */
430         max_sgl = hermon_wqe_max_sgl;

432         /*
433          * Use requested maximum number of SGL to calculate the max descriptor
434          * size (while guaranteeing that the descriptor size is a power-of-2
435          * cachelines).  We have to use the calculation for QP1 MLX transport
436          * because the possibility that we might need to inline a GRH, along
437          * with all the other headers and alignment restrictions, sets the
438          * maximum for the number of SGLs that we can advertise support for.
439          */
440         max_size = (HERMON_QP_WQE_MLX_QP1_HDRS + (max_sgl << 4));
441         log2 = highbit(max_size);
442         if (ISP2(max_size)) {
 34         if ((max_size & (max_size - 1)) == 0) {
443                 log2 = log2 - 1;
444         }
445         max_size = (1 << log2);

447         max_size = min(max_size, state->hs_devlim.max_desc_sz_sq);

449         /*
450          * Then use the calculated max descriptor size to determine the "real"
451          * maximum SGL (the number beyond which we would roll over to the next
452          * power-of-2).
453          */
454         real_max_sgl = (max_size - HERMON_QP_WQE_MLX_QP1_HDRS) >> 4;

456         /* Then save away this configuration information */
```



```
457         cp->cp_wqe_max_sgl      = max_sgl;
458         cp->cp_wqe_real_max_sgl = real_max_sgl;

460         /* SRQ SGL gets set to it's own patchable variable value */
461         cp->cp_srq_max_sgl              = hermon_srq_max_sgl;
462 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   91142 Thu Oct 23 10:42:12 2014
new/usr/src/uts/common/io/ib/adapters/hermon/hermon_qp.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * hermon_qp.c
 *    Hermon Queue Pair Processing Routines
 *
 *    Implements all the routines necessary for allocating, freeing, and
 *    querying the Hermon queue pairs.
 */

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>
#include <sys/sysmacros.h>

#include <sys/ib/adapters/hermon/hermon.h>
#include <sys/ib/ib_pkt_hdrs.h>

static int hermon_qp_create_qpn(hermon_state_t *state, hermon_qphdl_t qp,
    hermon_rsrc_t *qpc);
static int hermon_qpn_avl_compare(const void *q, const void *e);
static int hermon_special_qp_rsrc_alloc(hermon_state_t *state,
    ibt_sqp_type_t type, uint_t port, hermon_rsrc_t **qp_rsrc);
static int hermon_special_qp_rsrc_free(hermon_state_t *state,
    ibt_sqp_type_t type, uint_t port);
static void hermon_qp_sgl_to_logwqesz(hermon_state_t *state, uint_t num_sgl,
    uint_t real_max_sgl, hermon_qp_wq_type_t wq_type,
    uint_t *logwqesz, uint_t *max_sgl);

/*
 * hermon_qp_alloc()
 *    Context: Can be called only from user or kernel context.
 */
int
hermon_qp_alloc(hermon_state_t *state, hermon_qp_info_t *qpinfo,
```



```
    uint_t sleepflag)
{
        hermon_rsrc_t                   *qpc, *rsrc;
        hermon_rsrc_type_t              rsrc_type;
        hermon_umap_db_entry_t          *umapdb;
        hermon_qphdl_t                  qp;
        ibt_qp_alloc_attr_t             *attr_p;
        ibt_qp_alloc_flags_t            alloc_flags;
        ibt_qp_type_t                   type;
        hermon_qp_wq_type_t             swq_type;
        ibtl_qp_hdl_t                   ibt_qphdl;
        ibt_chan_sizes_t                *queuesz_p;
        ib_qpn_t                        *qpn;
        hermon_qphdl_t                  *qphdl;
        ibt_mr_attr_t                   mr_attr;
        hermon_mr_options_t             mr_op;
        hermon_srqhdl_t                 srq;
        hermon_pdhdl_t                  pd;
        hermon_cqhdl_t                  sq_cq, rq_cq;
        hermon_mrhdl_t                  mr;
        uint64_t                        value, qp_desc_off;
        uint64_t                        *thewqe, thewqesz;
        uint32_t                        *sq_buf, *rq_buf;
        uint32_t                        log_qp_sq_size, log_qp_rq_size;
        uint32_t                        sq_size, rq_size;
        uint32_t                        sq_depth, rq_depth;
        uint32_t                        sq_wqe_size, rq_wqe_size, wqesz_shift;
        uint32_t                        max_sgl, max_recv_sgl, uarpg;
        uint_t                          qp_is_umap;
        uint_t                          qp_srq_en, i, j;
        int                             status, flag;

        _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*attr_p, *queuesz_p))

        /*
         * Extract the necessary info from the hermon_qp_info_t structure
         */
        attr_p    = qpinfo->qpi_attrp;
        type      = qpinfo->qpi_type;
        ibt_qphdl = qpinfo->qpi_ibt_qphdl;
        queuesz_p = qpinfo->qpi_queueszp;
        qpn       = qpinfo->qpi_qpn;
        qphdl     = &qpinfo->qpi_qphdl;
        alloc_flags = attr_p->qp_alloc_flags;

        /*
         * Verify correctness of alloc_flags.
         *
         * 1. FEXCH and RSS are only allocated via qp_range.
         */
        if (alloc_flags & (IBT_QP_USES_FEXCH | IBT_QP_USES_RSS)) {
                return (IBT_INVALID_PARAM);
        }
        rsrc_type = HERMON_QPC;
        qp_is_umap = 0;

        /* 2. Make sure only one of these flags is set. */
        switch (alloc_flags &
            (IBT_QP_USER_MAP | IBT_QP_USES_RFCI | IBT_QP_USES_FCMD)) {
        case IBT_QP_USER_MAP:
                qp_is_umap = 1;
                break;
        case IBT_QP_USES_RFCI:
                if (type != IBT_UD_RQP)
                        return (IBT_INVALID_PARAM);
```

```
128                 switch (attr_p->qp_fc.fc_hca_port) {
129                 case 1:
130                         rsrc_type = HERMON_QPC_RFCI_PORT1;
131                         break;
132                 case 2:
133                         rsrc_type = HERMON_QPC_RFCI_PORT2;
134                         break;
135                 default:
136                         return (IBT_INVALID_PARAM);
137                 }
138                 break;
139         case IBT_QP_USES_FCMD:
140                 if (type != IBT_UD_RQP)
141                         return (IBT_INVALID_PARAM);
142                 break;
143         case 0:
144                 break;
145         default:
146                 return (IBT_INVALID_PARAM);     /* conflicting flags set */
147         }
148
149         /*
150          * Determine whether QP is being allocated for userland access or
151          * whether it is being allocated for kernel access.  If the QP is
152          * being allocated for userland access, then lookup the UAR
153          * page number for the current process.  Note:  If this is not found
154          * (e.g. if the process has not previously open()'d the Hermon driver),
155          * then an error is returned.
156          */
157         if (qp_is_umap) {
158                 status = hermon_umap_db_find(state->hs_instance, ddi_get_pid(),
159                     MLNX_UMAP_UARPG_RSRC, &value, 0, NULL);
160                 if (status != DDI_SUCCESS) {
161                         return (IBT_INVALID_PARAM);
162                 }
163                 uarpg = ((hermon_rsrc_t *)(uintptr_t)value)->hr_indx;
164         } else {
165                 uarpg = state->hs_kernel_uar_index;
166         }
167
168         /*
169          * Determine whether QP is being associated with an SRQ
170          */
171         qp_srq_en = (alloc_flags & IBT_QP_USES_SRQ) ? 1 : 0;
172         if (qp_srq_en) {
173                 /*
174                  * Check for valid SRQ handle pointers
175                  */
176                 if (attr_p->qp_ibc_srq_hdl == NULL) {
177                         status = IBT_SRQ_HDL_INVALID;
178                         goto qpalloc_fail;
179                 }
180                 srq = (hermon_srqhdl_t)attr_p->qp_ibc_srq_hdl;
181         }
182
183         /*
184          * Check for valid QP service type (only UD/RC/UC supported)
185          */
186         if (((type != IBT_UD_RQP) && (type != IBT_RC_RQP) &&
187             (type != IBT_UC_RQP))) {
188                 status = IBT_QP_SRV_TYPE_INVALID;
189                 goto qpalloc_fail;
190         }


193         /*
```



```
194          * Check for valid PD handle pointer
195          */
196         if (attr_p->qp_pd_hdl == NULL) {
197                 status = IBT_PD_HDL_INVALID;
198                 goto qpalloc_fail;
199         }
200         pd = (hermon_pdhdl_t)attr_p->qp_pd_hdl;
201
202         /*
203          * If on an SRQ, check to make sure the PD is the same
204          */
205         if (qp_srq_en && (pd->pd_pdnum != srq->srq_pdhdl->pd_pdnum)) {
206                 status = IBT_PD_HDL_INVALID;
207                 goto qpalloc_fail;
208         }
209
210         /* Increment the reference count on the protection domain (PD) */
211         hermon_pd_refcnt_inc(pd);
212
213         /*
214          * Check for valid CQ handle pointers
215          *
216          * FCMD QPs do not require a receive cq handle.
217          */
218         if (attr_p->qp_ibc_scq_hdl == NULL) {
219                 status = IBT_CQ_HDL_INVALID;
220                 goto qpalloc_fail1;
221         }
222         sq_cq = (hermon_cqhdl_t)attr_p->qp_ibc_scq_hdl;
223         if ((attr_p->qp_ibc_rcq_hdl == NULL)) {
224                 if ((alloc_flags & IBT_QP_USES_FCMD) == 0) {
225                         status = IBT_CQ_HDL_INVALID;
226                         goto qpalloc_fail1;
227                 }
228                 rq_cq = sq_cq;  /* just use the send cq */
229         } else
230                 rq_cq = (hermon_cqhdl_t)attr_p->qp_ibc_rcq_hdl;
231
232         /*
233          * Increment the reference count on the CQs.  One or both of these
234          * could return error if we determine that the given CQ is already
235          * being used with a special (SMI/GSI) QP.
236          */
237         status = hermon_cq_refcnt_inc(sq_cq, HERMON_CQ_IS_NORMAL);
238         if (status != DDI_SUCCESS) {
239                 status = IBT_CQ_HDL_INVALID;
240                 goto qpalloc_fail1;
241         }
242         status = hermon_cq_refcnt_inc(rq_cq, HERMON_CQ_IS_NORMAL);
243         if (status != DDI_SUCCESS) {
244                 status = IBT_CQ_HDL_INVALID;
245                 goto qpalloc_fail2;
246         }
247
248         /*
249          * Allocate an QP context entry.  This will be filled in with all
250          * the necessary parameters to define the Queue Pair.  Unlike
251          * other Hermon hardware resources, ownership is not immediately
252          * given to hardware in the final step here.  Instead, we must
253          * wait until the QP is later transitioned to the "Init" state before
254          * passing the QP to hardware.  If we fail here, we must undo all
255          * the reference count (CQ and PD).
256          */
257         status = hermon_rsrc_alloc(state, rsrc_type, 1, sleepflag, &qpc);
258         if (status != DDI_SUCCESS) {
259                 status = IBT_INSUFF_RESOURCE;
```

```
260                 goto qpalloc_fail3;
261         }

263         /*
264          * Allocate the software structure for tracking the queue pair
265          * (i.e. the Hermon Queue Pair handle).  If we fail here, we must
266          * undo the reference counts and the previous resource allocation.
267          */
268         status = hermon_rsrc_alloc(state, HERMON_QPHDL, 1, sleepflag, &rsrc);
269         if (status != DDI_SUCCESS) {
270                 status = IBT_INSUFF_RESOURCE;
271                 goto qpalloc_fail4;
272         }
273         qp = (hermon_qphdl_t)rsrc->hr_addr;
274         bzero(qp, sizeof (struct hermon_sw_qp_s));
275         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*qp))

277         qp->qp_alloc_flags = alloc_flags;

279         /*
280          * Calculate the QP number from QPC index.  This routine handles
281          * all of the operations necessary to keep track of used, unused,
282          * and released QP numbers.
283          */
284         if (type == IBT_UD_RQP) {
285                 qp->qp_qpnum = qpc->hr_indx;
286                 qp->qp_ring = qp->qp_qpnum << 8;
287                 qp->qp_qpn_hdl = NULL;
288         } else {
289                 status = hermon_qp_create_qpn(state, qp, qpc);
290                 if (status != DDI_SUCCESS) {
291                         status = IBT_INSUFF_RESOURCE;
292                         goto qpalloc_fail5;
293                 }
294         }

296         /*
297          * If this will be a user-mappable QP, then allocate an entry for
298          * the "userland resources database".  This will later be added to
299          * the database (after all further QP operations are successful).
300          * If we fail here, we must undo the reference counts and the
301          * previous resource allocation.
302          */
303         if (qp_is_umap) {
304                 umapdb = hermon_umap_db_alloc(state->hs_instance, qp->qp_qpnum,
305                     MLNX_UMAP_QPMEM_RSRC, (uint64_t)(uintptr_t)rsrc);
306                 if (umapdb == NULL) {
307                         status = IBT_INSUFF_RESOURCE;
308                         goto qpalloc_fail6;
309                 }
310         }

312         /*
313          * Allocate the doorbell record.  Hermon just needs one for the RQ,
314          * if the QP is not associated with an SRQ, and use uarpg (above) as
315          * the uar index
316          */

318         if (!qp_srq_en) {
319                 status = hermon_dbr_alloc(state, uarpg, &qp->qp_rq_dbr_acchdl,
320                     &qp->qp_rq_vdbr, &qp->qp_rq_pdbr, &qp->qp_rdbr_mapoffset);
321                 if (status != DDI_SUCCESS) {
322                         status = IBT_INSUFF_RESOURCE;
323                         goto qpalloc_fail6;
324                 }
325         }
```



```
327         qp->qp_uses_lso = (attr_p->qp_flags & IBT_USES_LSO);

329         /*
330          * We verify that the requested number of SGL is valid (i.e.
331          * consistent with the device limits and/or software-configured
332          * limits).  If not, then obviously the same cleanup needs to be done.
333          */
334         if (type == IBT_UD_RQP) {
335                 max_sgl = state->hs_ibtfinfo.hca_attr->hca_ud_send_sgl_sz;
336                 swq_type = HERMON_QP_WQ_TYPE_SENDQ_UD;
337         } else {
338                 max_sgl = state->hs_ibtfinfo.hca_attr->hca_conn_send_sgl_sz;
339                 swq_type = HERMON_QP_WQ_TYPE_SENDQ_CONN;
340         }
341         max_recv_sgl = state->hs_ibtfinfo.hca_attr->hca_recv_sgl_sz;
342         if ((attr_p->qp_sizes.cs_sq_sgl > max_sgl) ||
343             (!qp_srq_en && (attr_p->qp_sizes.cs_rq_sgl > max_recv_sgl))) {
344                 status = IBT_HCA_SGL_EXCEEDED;
345                 goto qpalloc_fail7;
346         }

348         /*
349          * Determine this QP's WQE stride (for both the Send and Recv WQEs).
350          * This will depend on the requested number of SGLs.  Note: this
351          * has the side-effect of also calculating the real number of SGLs
352          * (for the calculated WQE size).
353          *
354          * For QP's on an SRQ, we set these to 0.
355          */
356         if (qp_srq_en) {
357                 qp->qp_rq_log_wqesz = 0;
358                 qp->qp_rq_sgl = 0;
359         } else {
360                 hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_rq_sgl,
361                     max_recv_sgl, HERMON_QP_WQ_TYPE_RECVQ,
362                     &qp->qp_rq_log_wqesz, &qp->qp_rq_sgl);
363         }
364         hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
365             max_sgl, swq_type, &qp->qp_sq_log_wqesz, &qp->qp_sq_sgl);

367         sq_wqe_size = 1 << qp->qp_sq_log_wqesz;

369         /* NOTE: currently policy in driver, later maybe IBTF interface */
370         qp->qp_no_prefetch = 0;

372         /*
373          * for prefetching, we need to add the number of wqes in
374          * the 2k area plus one to the number requested, but
375          * ONLY for send queue.  If no_prefetch == 1 (prefetch off)
376          * it's exactly TWO wqes for the headroom
377          */
378         if (qp->qp_no_prefetch)
379                 qp->qp_sq_headroom = 2 * sq_wqe_size;
380         else
381                 qp->qp_sq_headroom = sq_wqe_size + HERMON_QP_OH_SIZE;
382         /*
383          * hdrm wqes must be integral since both sq_wqe_size &
384          * HERMON_QP_OH_SIZE are power of 2
385          */
386         qp->qp_sq_hdrmwqes = (qp->qp_sq_headroom / sq_wqe_size);


389         /*
390          * Calculate the appropriate size for the work queues.
391          * For send queue, add in the headroom wqes to the calculation.
```

```
392          * Note:  All Hermon QP work queues must be a power-of-2 in size.  Also
393          * they may not be any smaller than HERMON_QP_MIN_SIZE.  This step is
394          * to round the requested size up to the next highest power-of-2
395          */
396         /* first, adjust to a minimum and tell the caller the change */
397         attr_p->qp_sizes.cs_sq = max(attr_p->qp_sizes.cs_sq,
398             HERMON_QP_MIN_SIZE);
399         attr_p->qp_sizes.cs_rq = max(attr_p->qp_sizes.cs_rq,
400             HERMON_QP_MIN_SIZE);
401         /*
402          * now, calculate the alloc size, taking into account
403          * the headroom for the sq
404          */
405         log_qp_sq_size = highbit(attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes);
406         /* if the total is a power of two, reduce it */
407         if (ISP2(attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes))  {
407         if (((attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes)  &
408             (attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes - 1)) == 0)    {
408                 log_qp_sq_size = log_qp_sq_size - 1;
409         }

411         log_qp_rq_size = highbit(attr_p->qp_sizes.cs_rq);
412         if (ISP2(attr_p->qp_sizes.cs_rq)) {
413         if ((attr_p->qp_sizes.cs_rq & (attr_p->qp_sizes.cs_rq - 1)) == 0) {
413                 log_qp_rq_size = log_qp_rq_size - 1;
414         }

416         /*
417          * Next we verify that the rounded-up size is valid (i.e. consistent
418          * with the device limits and/or software-configured limits).  If not,
419          * then obviously we have a lot of cleanup to do before returning.
420          *
421          * NOTE: the first condition deals with the (test) case of cs_sq
422          * being just less than 2^32.  In this case, the headroom addition
423          * to the requested cs_sq will pass the test when it should not.
424          * This test no longer lets that case slip through the check.
425          */
426         if ((attr_p->qp_sizes.cs_sq >
427             (1 << state->hs_cfg_profile->cp_log_max_qp_sz)) ||
428             (log_qp_sq_size > state->hs_cfg_profile->cp_log_max_qp_sz) ||
429             (!qp_srq_en && (log_qp_rq_size >
430             state->hs_cfg_profile->cp_log_max_qp_sz))) {
431                 status = IBT_HCA_WR_EXCEEDED;
432                 goto qpalloc_fail7;
433         }

435         /*
436          * Allocate the memory for QP work queues. Since Hermon work queues
437          * are not allowed to cross a 32-bit (4GB) boundary, the alignment of
438          * the work queue memory is very important.  We used to allocate
439          * work queues (the combined receive and send queues) so that they
440          * would be aligned on their combined size.  That alignment guaranteed
441          * that they would never cross the 4GB boundary (Hermon work queues
442          * are on the order of MBs at maximum).  Now we are able to relax
443          * this alignment constraint by ensuring that the IB address assigned
444          * to the queue memory (as a result of the hermon_mr_register() call)
445          * is offset from zero.
446          * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
447          * guarantee the alignment, but when attempting to use IOMMU bypass
448          * mode we found that we were not allowed to specify any alignment
449          * that was more restrictive than the system page size.
450          * So we avoided this constraint by passing two alignment values,
451          * one for the memory allocation itself and the other for the DMA
452          * handle (for later bind).  This used to cause more memory than
453          * necessary to be allocated (in order to guarantee the more
454          * restrictive alignment contraint).  But by guaranteeing the
```



```
455          * zero-based IB virtual address for the queue, we are able to
456          * conserve this memory.
457          */
458         sq_wqe_size = 1 << qp->qp_sq_log_wqesz;
459         sq_depth    = 1 << log_qp_sq_size;
460         sq_size     = sq_depth * sq_wqe_size;

462         /* QP on SRQ sets these to 0 */
463         if (qp_srq_en) {
464                 rq_wqe_size = 0;
465                 rq_size     = 0;
466         } else {
467                 rq_wqe_size = 1 << qp->qp_rq_log_wqesz;
468                 rq_depth    = 1 << log_qp_rq_size;
469                 rq_size     = rq_depth * rq_wqe_size;
470         }

472         qp->qp_wqinfo.qa_size = sq_size + rq_size;

474         qp->qp_wqinfo.qa_alloc_align = PAGESIZE;
475         qp->qp_wqinfo.qa_bind_align  = PAGESIZE;

477         if (qp_is_umap) {
478                 qp->qp_wqinfo.qa_location = HERMON_QUEUE_LOCATION_USERLAND;
479         } else {
480                 qp->qp_wqinfo.qa_location = HERMON_QUEUE_LOCATION_NORMAL;
481         }
482         status = hermon_queue_alloc(state, &qp->qp_wqinfo, sleepflag);
483         if (status != DDI_SUCCESS) {
484                 status = IBT_INSUFF_RESOURCE;
485                 goto qpalloc_fail7;
486         }

488         /*
489          * Sort WQs in memory according to stride (*q_wqe_size), largest first
490          * If they are equal, still put the SQ first
491          */
492         qp->qp_sq_baseaddr = 0;
493         qp->qp_rq_baseaddr = 0;
494         if ((sq_wqe_size > rq_wqe_size) || (sq_wqe_size == rq_wqe_size)) {
495                 sq_buf = qp->qp_wqinfo.qa_buf_aligned;

497                 /* if this QP is on an SRQ, set the rq_buf to NULL */
498                 if (qp_srq_en) {
499                         rq_buf = NULL;
500                 } else {
501                         rq_buf = (uint32_t *)((uintptr_t)sq_buf + sq_size);
502                         qp->qp_rq_baseaddr = sq_size;
503                 }
504         } else {
505                 rq_buf = qp->qp_wqinfo.qa_buf_aligned;
506                 sq_buf = (uint32_t *)((uintptr_t)rq_buf + rq_size);
507                 qp->qp_sq_baseaddr = rq_size;
508         }

510         if (qp_is_umap == 0) {
511                 qp->qp_sq_wqhdr = hermon_wrid_wqhdr_create(sq_depth);
512                 if (qp->qp_sq_wqhdr == NULL) {
513                         status = IBT_INSUFF_RESOURCE;
514                         goto qpalloc_fail8;
515                 }
516                 if (qp_srq_en) {
517                         qp->qp_rq_wqavl.wqa_wq = srq->srq_wq_wqhdr;
518                         qp->qp_rq_wqavl.wqa_srq_en = 1;
519                         qp->qp_rq_wqavl.wqa_srq = srq;
520                 } else {
```

```
521                         qp->qp_rq_wqhdr = hermon_wrid_wqhdr_create(rq_depth);
522                         if (qp->qp_rq_wqhdr == NULL) {
523                                 status = IBT_INSUFF_RESOURCE;
524                                 goto qpalloc_fail8;
525                         }
526                         qp->qp_rq_wqavl.wqa_wq = qp->qp_rq_wqhdr;
527                 }
528                 qp->qp_sq_wqavl.wqa_qpn = qp->qp_qpnum;
529                 qp->qp_sq_wqavl.wqa_type = HERMON_WR_SEND;
530                 qp->qp_sq_wqavl.wqa_wq = qp->qp_sq_wqhdr;
531                 qp->qp_rq_wqavl.wqa_qpn = qp->qp_qpnum;
532                 qp->qp_rq_wqavl.wqa_type = HERMON_WR_RECV;
533         }

535         /*
536          * Register the memory for the QP work queues.  The memory for the
537          * QP must be registered in the Hermon cMPT tables.  This gives us the
538          * LKey to specify in the QP context later.  Note: The memory for
539          * Hermon work queues (both Send and Recv) must be contiguous and
540          * registered as a single memory region.  Note: If the QP memory is
541          * user-mappable, force DDI_DMA_CONSISTENT mapping. Also, in order to
542          * meet the alignment restriction, we pass the "mro_bind_override_addr"
543          * flag in the call to hermon_mr_register(). This guarantees that the
544          * resulting IB vaddr will be zero-based (modulo the offset into the
545          * first page). If we fail here, we still have the bunch of resource
546          * and reference count cleanup to do.
547          */
548         flag = (sleepflag == HERMON_SLEEP) ? IBT_MR_SLEEP :
549             IBT_MR_NOSLEEP;
550         mr_attr.mr_vaddr    = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned;
551         mr_attr.mr_len      = qp->qp_wqinfo.qa_size;
552         mr_attr.mr_as       = NULL;
553         mr_attr.mr_flags    = flag;
554         if (qp_is_umap) {
555                 mr_op.mro_bind_type = state->hs_cfg_profile->cp_iommu_bypass;
556         } else {
557                 /* HERMON_QUEUE_LOCATION_NORMAL */
558                 mr_op.mro_bind_type =
559                     state->hs_cfg_profile->cp_iommu_bypass;
560         }
561         mr_op.mro_bind_dmahdl = qp->qp_wqinfo.qa_dmahdl;
562         mr_op.mro_bind_override_addr = 1;
563         status = hermon_mr_register(state, pd, &mr_attr, &mr,
564             &mr_op, HERMON_QP_CMPT);
565         if (status != DDI_SUCCESS) {
566                 status = IBT_INSUFF_RESOURCE;
567                 goto qpalloc_fail9;
568         }

570         /*
571          * Calculate the offset between the kernel virtual address space
572          * and the IB virtual address space.  This will be used when
573          * posting work requests to properly initialize each WQE.
574          */
575         qp_desc_off = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned -
576             (uint64_t)mr->mr_bindinfo.bi_addr;

578         /*
579          * Fill in all the return arguments (if necessary).  This includes
580          * real work queue sizes (in wqes), real SGLs, and QP number
581          */
582         if (queuesz_p != NULL) {
583                 queuesz_p->cs_sq        =
584                     (1 << log_qp_sq_size) - qp->qp_sq_hdrmwqes;
585                 queuesz_p->cs_sq_sgl    = qp->qp_sq_sgl;
```



```
587                 /* if this QP is on an SRQ, set these to 0 */
588                 if (qp_srq_en) {
589                         queuesz_p->cs_rq        = 0;
590                         queuesz_p->cs_rq_sgl    = 0;
591                 } else {
592                         queuesz_p->cs_rq        = (1 << log_qp_rq_size);
593                         queuesz_p->cs_rq_sgl    = qp->qp_rq_sgl;
594                 }
595         }
596         if (qpn != NULL) {
597                 *qpn = (ib_qpn_t)qp->qp_qpnum;
598         }

600         /*
601          * Fill in the rest of the Hermon Queue Pair handle.
602          */
603         qp->qp_qpcrsrcp         = qpc;
604         qp->qp_rsrcp            = rsrc;
605         qp->qp_state            = HERMON_QP_RESET;
606         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
607         qp->qp_pdhdl            = pd;
608         qp->qp_mrhdl            = mr;
609         qp->qp_sq_sigtype       = (attr_p->qp_flags & IBT_WR_SIGNALED) ?
610             HERMON_QP_SQ_WR_SIGNALED : HERMON_QP_SQ_ALL_SIGNALED;
611         qp->qp_is_special       = 0;
612         qp->qp_uarpg            = uarpg;
613         qp->qp_umap_dhp         = (devmap_cookie_t)NULL;
614         qp->qp_sq_cqhdl         = sq_cq;
615         qp->qp_sq_bufsz         = (1 << log_qp_sq_size);
616         qp->qp_sq_logqsz        = log_qp_sq_size;
617         qp->qp_sq_buf           = sq_buf;
618         qp->qp_desc_off         = qp_desc_off;
619         qp->qp_rq_cqhdl         = rq_cq;
620         qp->qp_rq_buf           = rq_buf;
621         qp->qp_rlky             = (attr_p->qp_flags & IBT_FAST_REG_RES_LKEY) !=
622             0;

624         /* if this QP is on an SRQ, set rq_bufsz to 0 */
625         if (qp_srq_en) {
626                 qp->qp_rq_bufsz         = 0;
627                 qp->qp_rq_logqsz        = 0;
628         } else {
629                 qp->qp_rq_bufsz         = (1 << log_qp_rq_size);
630                 qp->qp_rq_logqsz        = log_qp_rq_size;
631         }

633         qp->qp_forward_sqd_event  = 0;
634         qp->qp_sqd_still_draining = 0;
635         qp->qp_hdlrarg          = (void *)ibt_qphdl;
636         qp->qp_mcg_refcnt       = 0;

638         /*
639          * If this QP is to be associated with an SRQ, set the SRQ handle
640          */
641         if (qp_srq_en) {
642                 qp->qp_srqhdl = srq;
643                 hermon_srq_refcnt_inc(qp->qp_srqhdl);
644         } else {
645                 qp->qp_srqhdl = NULL;
646         }

648         /* Determine the QP service type */
649         qp->qp_type = type;
650         if (type == IBT_RC_RQP) {
651                 qp->qp_serv_type = HERMON_QP_RC;
652         } else if (type == IBT_UD_RQP) {
```

```
653                 if (alloc_flags & IBT_QP_USES_RFCI)
654                         qp->qp_serv_type = HERMON_QP_RFCI;
655                 else if (alloc_flags & IBT_QP_USES_FCMD)
656                         qp->qp_serv_type = HERMON_QP_FCMND;
657                 else
658                         qp->qp_serv_type = HERMON_QP_UD;
659         } else {
660                 qp->qp_serv_type = HERMON_QP_UC;
661         }

663         /*
664          * Initialize the RQ WQEs - unlike Arbel, no Rcv init is needed
665          */

667         /*
668          * Initialize the SQ WQEs - all that needs to be done is every 64 bytes
669          * set the quadword to all F's - high-order bit is owner (init to one)
670          * and the rest for the headroom definition of prefetching
671          *
672          */
673         wqesz_shift = qp->qp_sq_log_wqesz;
674         thewqesz    = 1 << wqesz_shift;
675         thewqe = (uint64_t *)(void *)(qp->qp_sq_buf);
676         if (qp_is_umap == 0) {
677                 for (i = 0; i < sq_depth; i++) {
678                         /*
679                          * for each stride, go through and every 64 bytes
680                          * write the init value - having set the address
681                          * once, just keep incrementing it
682                          */
683                         for (j = 0; j < thewqesz; j += 64, thewqe += 8) {
684                                 *(uint32_t *)thewqe = 0xFFFFFFFF;
685                         }
686                 }
687         }

689         /* Zero out the QP context */
690         bzero(&qp->qpc, sizeof (hermon_hw_qpc_t));

692         /*
693          * Put QP handle in Hermon QPNum-to-QPHdl list.  Then fill in the
694          * "qphdl" and return success
695          */
696         hermon_icm_set_num_to_hdl(state, HERMON_QPC, qpc->hr_indx, qp);

698         /*
699          * If this is a user-mappable QP, then we need to insert the previously
700          * allocated entry into the "userland resources database".  This will
701          * allow for later lookup during devmap() (i.e. mmap()) calls.
702          */
703         if (qp_is_umap) {
704                 hermon_umap_db_add(umapdb);
705         }
706         mutex_init(&qp->qp_sq_lock, NULL, MUTEX_DRIVER,
707             DDI_INTR_PRI(state->hs_intrmsi_pri));

709         *qphdl = qp;

711         return (DDI_SUCCESS);

713 /*
714  * The following is cleanup for all possible failure cases in this routine
715  */
716 qpalloc_fail9:
717         hermon_queue_free(&qp->qp_wqinfo);
718 qpalloc_fail8:
```



```
719         if (qp->qp_sq_wqhdr)
720                 hermon_wrid_wqhdr_destroy(qp->qp_sq_wqhdr);
721         if (qp->qp_rq_wqhdr)
722                 hermon_wrid_wqhdr_destroy(qp->qp_rq_wqhdr);
723 qpalloc_fail7:
724         if (qp_is_umap) {
725                 hermon_umap_db_free(umapdb);
726         }
727         if (!qp_srq_en) {
728                 hermon_dbr_free(state, uarpg, qp->qp_rq_vdbr);
729         }

731 qpalloc_fail6:
732         /*
733          * Releasing the QPN will also free up the QPC context.  Update
734          * the QPC context pointer to indicate this.
735          */
736         if (qp->qp_qpn_hdl) {
737                 hermon_qp_release_qpn(state, qp->qp_qpn_hdl,
738                     HERMON_QPN_RELEASE);
739         } else {
740                 hermon_rsrc_free(state, &qpc);
741         }
742         qpc = NULL;
743 qpalloc_fail5:
744         hermon_rsrc_free(state, &rsrc);
745 qpalloc_fail4:
746         if (qpc) {
747                 hermon_rsrc_free(state, &qpc);
748         }
749 qpalloc_fail3:
750         hermon_cq_refcnt_dec(rq_cq);
751 qpalloc_fail2:
752         hermon_cq_refcnt_dec(sq_cq);
753 qpalloc_fail1:
754         hermon_pd_refcnt_dec(pd);
755 qpalloc_fail:
756         return (status);
757 }



761 /*
762  * hermon_special_qp_alloc()
763  *    Context: Can be called only from user or kernel context.
764  */
765 int
766 hermon_special_qp_alloc(hermon_state_t *state, hermon_qp_info_t *qpinfo,
767     uint_t sleepflag)
768 {
769         hermon_rsrc_t           *qpc, *rsrc;
770         hermon_qphdl_t          qp;
771         ibt_qp_alloc_attr_t     *attr_p;
772         ibt_sqp_type_t          type;
773         uint8_t                 port;
774         ibtl_qp_hdl_t           ibt_qphdl;
775         ibt_chan_sizes_t        *queuesz_p;
776         hermon_qphdl_t          *qphdl;
777         ibt_mr_attr_t           mr_attr;
778         hermon_mr_options_t     mr_op;
779         hermon_pdhdl_t          pd;
780         hermon_cqhdl_t          sq_cq, rq_cq;
781         hermon_mrhdl_t          mr;
782         uint64_t                qp_desc_off;
783         uint64_t                *thewqe, thewqesz;
784         uint32_t                *sq_buf, *rq_buf;
```

```
785         uint32_t                log_qp_sq_size, log_qp_rq_size;
786         uint32_t                sq_size, rq_size, max_sgl;
787         uint32_t                uarpg;
788         uint32_t                sq_depth;
789         uint32_t                sq_wqe_size, rq_wqe_size, wqesz_shift;
790         int                     status, flag, i, j;

792         /*
793          * Extract the necessary info from the hermon_qp_info_t structure
794          */
795         attr_p    = qpinfo->qpi_attrp;
796         type      = qpinfo->qpi_type;
797         port      = qpinfo->qpi_port;
798         ibt_qphdl = qpinfo->qpi_ibt_qphdl;
799         queuesz_p = qpinfo->qpi_queueszp;
800         qphdl     = &qpinfo->qpi_qphdl;

802         /*
803          * Check for valid special QP type (only SMI & GSI supported)
804          */
805         if ((type != IBT_SMI_SQP) && (type != IBT_GSI_SQP)) {
806                 status = IBT_QP_SPECIAL_TYPE_INVALID;
807                 goto spec_qpalloc_fail;
808         }

810         /*
811          * Check for valid port number
812          */
813         if (!hermon_portnum_is_valid(state, port)) {
814                 status = IBT_HCA_PORT_INVALID;
815                 goto spec_qpalloc_fail;
816         }
817         port = port - 1;

819         /*
820          * Check for valid PD handle pointer
821          */
822         if (attr_p->qp_pd_hdl == NULL) {
823                 status = IBT_PD_HDL_INVALID;
824                 goto spec_qpalloc_fail;
825         }
826         pd = (hermon_pdhdl_t)attr_p->qp_pd_hdl;

828         /* Increment the reference count on the PD */
829         hermon_pd_refcnt_inc(pd);

831         /*
832          * Check for valid CQ handle pointers
833          */
834         if ((attr_p->qp_ibc_scq_hdl == NULL) ||
835             (attr_p->qp_ibc_rcq_hdl == NULL)) {
836                 status = IBT_CQ_HDL_INVALID;
837                 goto spec_qpalloc_fail1;
838         }
839         sq_cq = (hermon_cqhdl_t)attr_p->qp_ibc_scq_hdl;
840         rq_cq = (hermon_cqhdl_t)attr_p->qp_ibc_rcq_hdl;

842         /*
843          * Increment the reference count on the CQs.  One or both of these
844          * could return error if we determine that the given CQ is already
845          * being used with a non-special QP (i.e. a normal QP).
846          */
847         status = hermon_cq_refcnt_inc(sq_cq, HERMON_CQ_IS_SPECIAL);
848         if (status != DDI_SUCCESS) {
849                 status = IBT_CQ_HDL_INVALID;
850                 goto spec_qpalloc_fail1;
```



```
851         }
852         status = hermon_cq_refcnt_inc(rq_cq, HERMON_CQ_IS_SPECIAL);
853         if (status != DDI_SUCCESS) {
854                 status = IBT_CQ_HDL_INVALID;
855                 goto spec_qpalloc_fail2;
856         }

858         /*
859          * Allocate the special QP resources.  Essentially, this allocation
860          * amounts to checking if the request special QP has already been
861          * allocated.  If successful, the QP context return is an actual
862          * QP context that has been "aliased" to act as a special QP of the
863          * appropriate type (and for the appropriate port).  Just as in
864          * hermon_qp_alloc() above, ownership for this QP context is not
865          * immediately given to hardware in the final step here.  Instead, we
866          * wait until the QP is later transitioned to the "Init" state before
867          * passing the QP to hardware.  If we fail here, we must undo all
868          * the reference count (CQ and PD).
869          */
870         status = hermon_special_qp_rsrc_alloc(state, type, port, &qpc);
871         if (status != DDI_SUCCESS) {
872                 goto spec_qpalloc_fail3;
873         }

875         /*
876          * Allocate the software structure for tracking the special queue
877          * pair (i.e. the Hermon Queue Pair handle).  If we fail here, we
878          * must undo the reference counts and the previous resource allocation.
879          */
880         status = hermon_rsrc_alloc(state, HERMON_QPHDL, 1, sleepflag, &rsrc);
881         if (status != DDI_SUCCESS) {
882                 status = IBT_INSUFF_RESOURCE;
883                 goto spec_qpalloc_fail4;
884         }
885         qp = (hermon_qphdl_t)rsrc->hr_addr;

887         bzero(qp, sizeof (struct hermon_sw_qp_s));

889         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*qp))
890         qp->qp_alloc_flags = attr_p->qp_alloc_flags;

892         /*
893          * Actual QP number is a combination of the index of the QPC and
894          * the port number.  This is because the special QP contexts must
895          * be allocated two-at-a-time.
896          */
897         qp->qp_qpnum = qpc->hr_indx + port;
898         qp->qp_ring = qp->qp_qpnum << 8;

900         uarpg = state->hs_kernel_uar_index; /* must be for spec qp */
901         /*
902          * Allocate the doorbell record.  Hermon uses only one for the RQ so
903          * alloc a qp doorbell, using uarpg (above) as the uar index
904          */

906         status = hermon_dbr_alloc(state, uarpg, &qp->qp_rq_dbr_acchdl,
907             &qp->qp_rq_vdbr, &qp->qp_rq_pdbr, &qp->qp_rdbr_mapoffset);
908         if (status != DDI_SUCCESS) {
909                 status = IBT_INSUFF_RESOURCE;
910                 goto spec_qpalloc_fail5;
911         }
912         /*
913          * Calculate the appropriate size for the work queues.
914          * Note:  All Hermon QP work queues must be a power-of-2 in size.  Also
915          * they may not be any smaller than HERMON_QP_MIN_SIZE.  This step is
916          * to round the requested size up to the next highest power-of-2
```

```
917          */
918         attr_p->qp_sizes.cs_sq =
919             max(attr_p->qp_sizes.cs_sq, HERMON_QP_MIN_SIZE);
920         attr_p->qp_sizes.cs_rq =
921             max(attr_p->qp_sizes.cs_rq, HERMON_QP_MIN_SIZE);
922         log_qp_sq_size = highbit(attr_p->qp_sizes.cs_sq);
923         if (ISP2(attr_p->qp_sizes.cs_sq)) {
924         if ((attr_p->qp_sizes.cs_sq & (attr_p->qp_sizes.cs_sq - 1)) == 0) {
924                 log_qp_sq_size = log_qp_sq_size - 1;
925         }
926         log_qp_rq_size = highbit(attr_p->qp_sizes.cs_rq);
927         if (ISP2(attr_p->qp_sizes.cs_rq)) {
928         if ((attr_p->qp_sizes.cs_rq & (attr_p->qp_sizes.cs_rq - 1)) == 0) {
928                 log_qp_rq_size = log_qp_rq_size - 1;
929         }

931         /*
932          * Next we verify that the rounded-up size is valid (i.e. consistent
933          * with the device limits and/or software-configured limits).  If not,
934          * then obviously we have a bit of cleanup to do before returning.
935          */
936         if ((log_qp_sq_size > state->hs_cfg_profile->cp_log_max_qp_sz) ||
937             (log_qp_rq_size > state->hs_cfg_profile->cp_log_max_qp_sz)) {
938                 status = IBT_HCA_WR_EXCEEDED;
939                 goto spec_qpalloc_fail5a;
940         }

942         /*
943          * Next we verify that the requested number of SGL is valid (i.e.
944          * consistent with the device limits and/or software-configured
945          * limits).  If not, then obviously the same cleanup needs to be done.
946          */
947         max_sgl = state->hs_cfg_profile->cp_wqe_real_max_sgl;
948         if ((attr_p->qp_sizes.cs_sq_sgl > max_sgl) ||
949             (attr_p->qp_sizes.cs_rq_sgl > max_sgl)) {
950                 status = IBT_HCA_SGL_EXCEEDED;
951                 goto spec_qpalloc_fail5a;
952         }

954         /*
955          * Determine this QP's WQE stride (for both the Send and Recv WQEs).
956          * This will depend on the requested number of SGLs.  Note: this
957          * has the side-effect of also calculating the real number of SGLs
958          * (for the calculated WQE size).
959          */
960         hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_rq_sgl,
961             max_sgl, HERMON_QP_WQ_TYPE_RECVQ,
962             &qp->qp_rq_log_wqesz, &qp->qp_rq_sgl);
963         if (type == IBT_SMI_SQP) {
964                 hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
965                     max_sgl, HERMON_QP_WQ_TYPE_SENDMLX_QP0,
966                     &qp->qp_sq_log_wqesz, &qp->qp_sq_sgl);
967         } else {
968                 hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
969                     max_sgl, HERMON_QP_WQ_TYPE_SENDMLX_QP1,
970                     &qp->qp_sq_log_wqesz, &qp->qp_sq_sgl);
971         }

973         /*
974          * Allocate the memory for QP work queues. Since Hermon work queues
975          * are not allowed to cross a 32-bit (4GB) boundary, the alignment of
976          * the work queue memory is very important.  We used to allocate
977          * work queues (the combined receive and send queues) so that they
978          * would be aligned on their combined size.  That alignment guaranteed
979          * that they would never cross the 4GB boundary (Hermon work queues
980          * are on the order of MBs at maximum).  Now we are able to relax
```



```
981          * this alignment constraint by ensuring that the IB address assigned
982          * to the queue memory (as a result of the hermon_mr_register() call)
983          * is offset from zero.
984          * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
985          * guarantee the alignment, but when attempting to use IOMMU bypass
986          * mode we found that we were not allowed to specify any alignment
987          * that was more restrictive than the system page size.
988          * So we avoided this constraint by passing two alignment values,
989          * one for the memory allocation itself and the other for the DMA
990          * handle (for later bind).  This used to cause more memory than
991          * necessary to be allocated (in order to guarantee the more
992          * restrictive alignment contraint).  But by guaranteeing the
993          * zero-based IB virtual address for the queue, we are able to
994          * conserve this memory.
995          */
996         sq_wqe_size = 1 << qp->qp_sq_log_wqesz;
997         sq_depth    = 1 << log_qp_sq_size;
998         sq_size     = (1 << log_qp_sq_size) * sq_wqe_size;

1000        rq_wqe_size = 1 << qp->qp_rq_log_wqesz;
1001        rq_size     = (1 << log_qp_rq_size) * rq_wqe_size;

1003        qp->qp_wqinfo.qa_size     = sq_size + rq_size;

1005        qp->qp_wqinfo.qa_alloc_align = PAGESIZE;
1006        qp->qp_wqinfo.qa_bind_align  = PAGESIZE;
1007        qp->qp_wqinfo.qa_location = HERMON_QUEUE_LOCATION_NORMAL;

1009        status = hermon_queue_alloc(state, &qp->qp_wqinfo, sleepflag);
1010        if (status != NULL) {
1011                status = IBT_INSUFF_RESOURCE;
1012                goto spec_qpalloc_fail5a;
1013        }

1015        /*
1016         * Sort WQs in memory according to depth, stride (*q_wqe_size),
1017         * biggest first. If equal, the Send Queue still goes first
1018         */
1019        qp->qp_sq_baseaddr = 0;
1020        qp->qp_rq_baseaddr = 0;
1021        if ((sq_wqe_size > rq_wqe_size) || (sq_wqe_size == rq_wqe_size)) {
1022                sq_buf = qp->qp_wqinfo.qa_buf_aligned;
1023                rq_buf = (uint32_t *)((uintptr_t)sq_buf + sq_size);
1024                qp->qp_rq_baseaddr = sq_size;
1025        } else {
1026                rq_buf = qp->qp_wqinfo.qa_buf_aligned;
1027                sq_buf = (uint32_t *)((uintptr_t)rq_buf + rq_size);
1028                qp->qp_sq_baseaddr = rq_size;
1029        }

1031        qp->qp_sq_wqhdr = hermon_wrid_wqhdr_create(sq_depth);
1032        if (qp->qp_sq_wqhdr == NULL) {
1033                status = IBT_INSUFF_RESOURCE;
1034                goto spec_qpalloc_fail6;
1035        }
1036        qp->qp_rq_wqhdr = hermon_wrid_wqhdr_create(1 << log_qp_rq_size);
1037        if (qp->qp_rq_wqhdr == NULL) {
1038                status = IBT_INSUFF_RESOURCE;
1039                goto spec_qpalloc_fail6;
1040        }
1041        qp->qp_sq_wqavl.wqa_qpn = qp->qp_qpnum;
1042        qp->qp_sq_wqavl.wqa_type = HERMON_WR_SEND;
1043        qp->qp_sq_wqavl.wqa_wq = qp->qp_sq_wqhdr;
1044        qp->qp_rq_wqavl.wqa_qpn = qp->qp_qpnum;
1045        qp->qp_rq_wqavl.wqa_type = HERMON_WR_RECV;
1046        qp->qp_rq_wqavl.wqa_wq = qp->qp_rq_wqhdr;
```

```
1048         /*
1049          * Register the memory for the special QP work queues.  The memory for
1050          * the special QP must be registered in the Hermon cMPT tables.  This
1051          * gives us the LKey to specify in the QP context later.  Note: The
1052          * memory for Hermon work queues (both Send and Recv) must be contiguous
1053          * and registered as a single memory region. Also, in order to meet the
1054          * alignment restriction, we pass the "mro_bind_override_addr" flag in
1055          * the call to hermon_mr_register(). This guarantees that the resulting
1056          * IB vaddr will be zero-based (modulo the offset into the first page).
1057          * If we fail here, we have a bunch of resource and reference count
1058          * cleanup to do.
1059          */
1060         flag = (sleepflag == HERMON_SLEEP) ? IBT_MR_SLEEP :
1061             IBT_MR_NOSLEEP;
1062         mr_attr.mr_vaddr    = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned;
1063         mr_attr.mr_len      = qp->qp_wqinfo.qa_size;
1064         mr_attr.mr_as       = NULL;
1065         mr_attr.mr_flags    = flag;

1067         mr_op.mro_bind_type = state->hs_cfg_profile->cp_iommu_bypass;
1068         mr_op.mro_bind_dmahdl = qp->qp_wqinfo.qa_dmahdl;
1069         mr_op.mro_bind_override_addr = 1;

1071         status = hermon_mr_register(state, pd, &mr_attr, &mr, &mr_op,
1072             HERMON_QP_CMPT);
1073         if (status != DDI_SUCCESS) {
1074                 status = IBT_INSUFF_RESOURCE;
1075                 goto spec_qpalloc_fail6;
1076         }

1078         /*
1079          * Calculate the offset between the kernel virtual address space
1080          * and the IB virtual address space.  This will be used when
1081          * posting work requests to properly initialize each WQE.
1082          */
1083         qp_desc_off = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned -
1084             (uint64_t)mr->mr_bindinfo.bi_addr;

1086         /* set the prefetch - initially, not prefetching */
1087         qp->qp_no_prefetch = 1;

1089         if (qp->qp_no_prefetch)
1090                 qp->qp_sq_headroom = 2 * sq_wqe_size;
1091         else
1092                 qp->qp_sq_headroom = sq_wqe_size + HERMON_QP_OH_SIZE;
1093         /*
1094          * hdrm wqes must be integral since both sq_wqe_size &
1095          * HERMON_QP_OH_SIZE are power of 2
1096          */
1097         qp->qp_sq_hdrmwqes = (qp->qp_sq_headroom / sq_wqe_size);
1098         /*
1099          * Fill in all the return arguments (if necessary).  This includes
1100          * real work queue sizes, real SGLs, and QP number (which will be
1101          * either zero or one, depending on the special QP type)
1102          */
1103         if (queuesz_p != NULL) {
1104                 queuesz_p->cs_sq        =
1105                     (1 << log_qp_sq_size) - qp->qp_sq_hdrmwqes;
1106                 queuesz_p->cs_sq_sgl    = qp->qp_sq_sgl;
1107                 queuesz_p->cs_rq        = (1 << log_qp_rq_size);
1108                 queuesz_p->cs_rq_sgl    = qp->qp_rq_sgl;
1109         }

1111         /*
1112          * Fill in the rest of the Hermon Queue Pair handle.  We can update
```



```
1113          * the following fields for use in further operations on the QP.
1114          */
1115         qp->qp_qpcrsrcp         = qpc;
1116         qp->qp_rsrcp            = rsrc;
1117         qp->qp_state            = HERMON_QP_RESET;
1118         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
1119         qp->qp_pdhdl            = pd;
1120         qp->qp_mrhdl            = mr;
1121         qp->qp_sq_sigtype       = (attr_p->qp_flags & IBT_WR_SIGNALED) ?
1122             HERMON_QP_SQ_WR_SIGNALED : HERMON_QP_SQ_ALL_SIGNALED;
1123         qp->qp_is_special       = (type == IBT_SMI_SQP) ?
1124             HERMON_QP_SMI : HERMON_QP_GSI;
1125         qp->qp_uarpg            = uarpg;
1126         qp->qp_umap_dhp         = (devmap_cookie_t)NULL;
1127         qp->qp_sq_cqhdl         = sq_cq;
1128         qp->qp_sq_bufsz         = (1 << log_qp_sq_size);
1129         qp->qp_sq_buf           = sq_buf;
1130         qp->qp_sq_logqsz        = log_qp_sq_size;
1131         qp->qp_desc_off         = qp_desc_off;
1132         qp->qp_rq_cqhdl         = rq_cq;
1133         qp->qp_rq_bufsz         = (1 << log_qp_rq_size);
1134         qp->qp_rq_buf           = rq_buf;
1135         qp->qp_rq_logqsz        = log_qp_rq_size;
1136         qp->qp_portnum          = port;
1137         qp->qp_pkeyindx         = 0;
1138         qp->qp_forward_sqd_event  = 0;
1139         qp->qp_sqd_still_draining = 0;
1140         qp->qp_hdlrarg          = (void *)ibt_qphdl;
1141         qp->qp_mcg_refcnt       = 0;
1142         qp->qp_srqhdl           = NULL;

1144         /* All special QPs are UD QP service type */
1145         qp->qp_type = IBT_UD_RQP;
1146         qp->qp_serv_type = HERMON_QP_UD;

1148         /*
1149          * Initialize the RQ WQEs - unlike Arbel, no Rcv init is needed
1150          */

1152         /*
1153          * Initialize the SQ WQEs - all that needs to be done is every 64 bytes
1154          * set the quadword to all F's - high-order bit is owner (init to one)
1155          * and the rest for the headroom definition of prefetching
1156          *
1157          */

1159         wqesz_shift = qp->qp_sq_log_wqesz;
1160         thewqesz    = 1 << wqesz_shift;
1161         thewqe = (uint64_t *)(void *)(qp->qp_sq_buf);
1162         for (i = 0; i < sq_depth; i++) {
1163                 /*
1164                  * for each stride, go through and every 64 bytes write the
1165                  * init value - having set the address once, just keep
1166                  * incrementing it
1167                  */
1168                 for (j = 0; j < thewqesz; j += 64, thewqe += 8) {
1169                         *(uint32_t *)thewqe = 0xFFFFFFFF;
1170                 }
1171         }


1174         /* Zero out the QP context */
1175         bzero(&qp->qpc, sizeof (hermon_hw_qpc_t));

1177         /*
1178          * Put QP handle in Hermon QPNum-to-QPHdl list.  Then fill in the
```

```
1179          * "qphdl" and return success
1180          */
1181         hermon_icm_set_num_to_hdl(state, HERMON_QPC, qpc->hr_indx + port, qp);

1183         mutex_init(&qp->qp_sq_lock, NULL, MUTEX_DRIVER,
1184             DDI_INTR_PRI(state->hs_intrmsi_pri));

1186         *qphdl = qp;

1188         return (DDI_SUCCESS);

1190 /*
1191  * The following is cleanup for all possible failure cases in this routine
1192  */
1193 spec_qpalloc_fail6:
1194         hermon_queue_free(&qp->qp_wqinfo);
1195         if (qp->qp_sq_wqhdr)
1196                 hermon_wrid_wqhdr_destroy(qp->qp_sq_wqhdr);
1197         if (qp->qp_rq_wqhdr)
1198                 hermon_wrid_wqhdr_destroy(qp->qp_rq_wqhdr);
1199 spec_qpalloc_fail5a:
1200         hermon_dbr_free(state, uarpg, qp->qp_rq_vdbr);
1201 spec_qpalloc_fail5:
1202         hermon_rsrc_free(state, &rsrc);
1203 spec_qpalloc_fail4:
1204         if (hermon_special_qp_rsrc_free(state, type, port) != DDI_SUCCESS) {
1205                 HERMON_WARNING(state, "failed to free special QP rsrc");
1206         }
1207 spec_qpalloc_fail3:
1208         hermon_cq_refcnt_dec(rq_cq);
1209 spec_qpalloc_fail2:
1210         hermon_cq_refcnt_dec(sq_cq);
1211 spec_qpalloc_fail1:
1212         hermon_pd_refcnt_dec(pd);
1213 spec_qpalloc_fail:
1214         return (status);
1215 }


1218 /*
1219  * hermon_qp_alloc_range()
1220  *    Context: Can be called only from user or kernel context.
1221  */
1222 int
1223 hermon_qp_alloc_range(hermon_state_t *state, uint_t log2,
1224     hermon_qp_info_t *qpinfo, ibtl_qp_hdl_t *ibt_qphdl,
1225     ibc_cq_hdl_t *send_cq, ibc_cq_hdl_t *recv_cq,
1226     hermon_qphdl_t *qphdl, uint_t sleepflag)
1227 {
1228         hermon_rsrc_t                   *qpc, *rsrc;
1229         hermon_rsrc_type_t              rsrc_type;
1230         hermon_qphdl_t                  qp;
1231         hermon_qp_range_t               *qp_range_p;
1232         ibt_qp_alloc_attr_t             *attr_p;
1233         ibt_qp_type_t                   type;
1234         hermon_qp_wq_type_t             swq_type;
1235         ibt_chan_sizes_t                *queuesz_p;
1236         ibt_mr_attr_t                   mr_attr;
1237         hermon_mr_options_t             mr_op;
1238         hermon_srqhdl_t                 srq;
1239         hermon_pdhdl_t                  pd;
1240         hermon_cqhdl_t                  sq_cq, rq_cq;
1241         hermon_mrhdl_t                  mr;
1242         uint64_t                        qp_desc_off;
1243         uint64_t                        *thewqe, thewqesz;
1244         uint32_t                        *sq_buf, *rq_buf;
```



```
1245         uint32_t                        log_qp_sq_size, log_qp_rq_size;
1246         uint32_t                        sq_size, rq_size;
1247         uint32_t                        sq_depth, rq_depth;
1248         uint32_t                        sq_wqe_size, rq_wqe_size, wqesz_shift;
1249         uint32_t                        max_sgl, max_recv_sgl, uarpg;
1250         uint_t                          qp_srq_en, i, j;
1251         int                             ii;     /* loop counter for range */
1252         int                             status, flag;
1253         uint_t                          serv_type;

1255         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*attr_p, *queuesz_p))

1257         /*
1258          * Extract the necessary info from the hermon_qp_info_t structure
1259          */
1260         attr_p    = qpinfo->qpi_attrp;
1261         type      = qpinfo->qpi_type;
1262         queuesz_p = qpinfo->qpi_queueszp;

1264         if (attr_p->qp_alloc_flags & IBT_QP_USES_RSS) {
1265                 if (log2 > state->hs_ibtfinfo.hca_attr->hca_rss_max_log2_table)
1266                         return (IBT_INSUFF_RESOURCE);
1267                 rsrc_type = HERMON_QPC;
1268                 serv_type = HERMON_QP_UD;
1269         } else if (attr_p->qp_alloc_flags & IBT_QP_USES_FEXCH) {
1270                 if (log2 > state->hs_ibtfinfo.hca_attr->hca_fexch_max_log2_qp)
1271                         return (IBT_INSUFF_RESOURCE);
1272                 switch (attr_p->qp_fc.fc_hca_port) {
1273                 case 1:
1274                         rsrc_type = HERMON_QPC_FEXCH_PORT1;
1275                         break;
1276                 case 2:
1277                         rsrc_type = HERMON_QPC_FEXCH_PORT2;
1278                         break;
1279                 default:
1280                         return (IBT_INVALID_PARAM);
1281                 }
1282                 serv_type = HERMON_QP_FEXCH;
1283         } else
1284                 return (IBT_INVALID_PARAM);

1286         /*
1287          * Determine whether QP is being allocated for userland access or
1288          * whether it is being allocated for kernel access.  If the QP is
1289          * being allocated for userland access, fail (too complex for now).
1290          */
1291         if (attr_p->qp_alloc_flags & IBT_QP_USER_MAP) {
1292                 return (IBT_NOT_SUPPORTED);
1293         } else {
1294                 uarpg = state->hs_kernel_uar_index;
1295         }

1297         /*
1298          * Determine whether QP is being associated with an SRQ
1299          */
1300         qp_srq_en = (attr_p->qp_alloc_flags & IBT_QP_USES_SRQ) ? 1 : 0;
1301         if (qp_srq_en) {
1302                 /*
1303                  * Check for valid SRQ handle pointers
1304                  */
1305                 if (attr_p->qp_ibc_srq_hdl == NULL) {
1306                         return (IBT_SRQ_HDL_INVALID);
1307                 }
1308                 srq = (hermon_srqhdl_t)attr_p->qp_ibc_srq_hdl;
1309         }
```

```
1311         /*
1312          * Check for valid QP service type (only UD supported)
1313          */
1314         if (type != IBT_UD_RQP) {
1315                 return (IBT_QP_SRV_TYPE_INVALID);
1316         }

1318         /*
1319          * Check for valid PD handle pointer
1320          */
1321         if (attr_p->qp_pd_hdl == NULL) {
1322                 return (IBT_PD_HDL_INVALID);
1323         }
1324         pd = (hermon_pdhdl_t)attr_p->qp_pd_hdl;

1326         /*
1327          * If on an SRQ, check to make sure the PD is the same
1328          */
1329         if (qp_srq_en && (pd->pd_pdnum != srq->srq_pdhdl->pd_pdnum)) {
1330                 return (IBT_PD_HDL_INVALID);
1331         }

1333         /* set loop variable here, for freeing resources on error */
1334         ii = 0;

1336         /*
1337          * Allocate 2^log2 contiguous/aligned QP context entries.  This will
1338          * be filled in with all the necessary parameters to define the
1339          * Queue Pairs.  Unlike other Hermon hardware resources, ownership
1340          * is not immediately given to hardware in the final step here.
1341          * Instead, we must wait until the QP is later transitioned to the
1342          * "Init" state before passing the QP to hardware.  If we fail here,
1343          * we must undo all the reference count (CQ and PD).
1344          */
1345         status = hermon_rsrc_alloc(state, rsrc_type, 1 << log2, sleepflag,
1346             &qpc);
1347         if (status != DDI_SUCCESS) {
1348                 return (IBT_INSUFF_RESOURCE);
1349         }

1351         if (attr_p->qp_alloc_flags & IBT_QP_USES_FEXCH)
1352                 /*
1353                  * Need to init the MKEYs for the FEXCH QPs.
1354                  *
1355                  * For FEXCH QP subranges, we return the QPN base as
1356                  * "relative" to the full FEXCH QP range for the port.
1357                  */
1358                 *(qpinfo->qpi_qpn) = hermon_fcoib_fexch_relative_qpn(state,
1359                     attr_p->qp_fc.fc_hca_port, qpc->hr_indx);
1360         else
1361                 *(qpinfo->qpi_qpn) = (ib_qpn_t)qpc->hr_indx;

1363         qp_range_p = kmem_alloc(sizeof (*qp_range_p),
1364             (sleepflag == HERMON_SLEEP) ? KM_SLEEP : KM_NOSLEEP);
1365         if (qp_range_p == NULL) {
1366                 status = IBT_INSUFF_RESOURCE;
1367                 goto qpalloc_fail0;
1368         }
1369         mutex_init(&qp_range_p->hqpr_lock, NULL, MUTEX_DRIVER,
1370             DDI_INTR_PRI(state->hs_intrmsi_pri));
1371         mutex_enter(&qp_range_p->hqpr_lock);
1372         qp_range_p->hqpr_refcnt = 1 << log2;
1373         qp_range_p->hqpr_qpcrsrc = qpc;
1374         mutex_exit(&qp_range_p->hqpr_lock);

1376 for_each_qp:
```



```
1378         /* Increment the reference count on the protection domain (PD) */
1379         hermon_pd_refcnt_inc(pd);

1381         rq_cq = (hermon_cqhdl_t)recv_cq[ii];
1382         sq_cq = (hermon_cqhdl_t)send_cq[ii];
1383         if (sq_cq == NULL) {
1384                 if (attr_p->qp_alloc_flags & IBT_QP_USES_FEXCH) {
1385                         /* if no send completions, just use rq_cq */
1386                         sq_cq = rq_cq;
1387                 } else {
1388                         status = IBT_CQ_HDL_INVALID;
1389                         goto qpalloc_fail1;
1390                 }
1391         }

1393         /*
1394          * Increment the reference count on the CQs.  One or both of these
1395          * could return error if we determine that the given CQ is already
1396          * being used with a special (SMI/GSI) QP.
1397          */
1398         status = hermon_cq_refcnt_inc(sq_cq, HERMON_CQ_IS_NORMAL);
1399         if (status != DDI_SUCCESS) {
1400                 status = IBT_CQ_HDL_INVALID;
1401                 goto qpalloc_fail1;
1402         }
1403         status = hermon_cq_refcnt_inc(rq_cq, HERMON_CQ_IS_NORMAL);
1404         if (status != DDI_SUCCESS) {
1405                 status = IBT_CQ_HDL_INVALID;
1406                 goto qpalloc_fail2;
1407         }

1409         /*
1410          * Allocate the software structure for tracking the queue pair
1411          * (i.e. the Hermon Queue Pair handle).  If we fail here, we must
1412          * undo the reference counts and the previous resource allocation.
1413          */
1414         status = hermon_rsrc_alloc(state, HERMON_QPHDL, 1, sleepflag, &rsrc);
1415         if (status != DDI_SUCCESS) {
1416                 status = IBT_INSUFF_RESOURCE;
1417                 goto qpalloc_fail4;
1418         }
1419         qp = (hermon_qphdl_t)rsrc->hr_addr;
1420         bzero(qp, sizeof (struct hermon_sw_qp_s));
1421         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*qp))
1422         qp->qp_alloc_flags = attr_p->qp_alloc_flags;

1424         /*
1425          * Calculate the QP number from QPC index.  This routine handles
1426          * all of the operations necessary to keep track of used, unused,
1427          * and released QP numbers.
1428          */
1429         qp->qp_qpnum = qpc->hr_indx + ii;
1430         qp->qp_ring = qp->qp_qpnum << 8;
1431         qp->qp_qpn_hdl = NULL;

1433         /*
1434          * Allocate the doorbell record.  Hermon just needs one for the RQ,
1435          * if the QP is not associated with an SRQ, and use uarpg (above) as
1436          * the uar index
1437          */

1439         if (!qp_srq_en) {
1440                 status = hermon_dbr_alloc(state, uarpg, &qp->qp_rq_dbr_acchdl,
1441                     &qp->qp_rq_vdbr, &qp->qp_rq_pdbr, &qp->qp_rdbr_mapoffset);
1442                 if (status != DDI_SUCCESS) {
```

```
1443                        status = IBT_INSUFF_RESOURCE;
1444                        goto qpalloc_fail6;
1445                }
1446        }

1448        qp->qp_uses_lso = (attr_p->qp_flags & IBT_USES_LSO);

1450        /*
1451         * We verify that the requested number of SGL is valid (i.e.
1452         * consistent with the device limits and/or software-configured
1453         * limits).  If not, then obviously the same cleanup needs to be done.
1454         */
1455        max_sgl = state->hs_ibtfinfo.hca_attr->hca_ud_send_sgl_sz;
1456        swq_type = HERMON_QP_WQ_TYPE_SENDQ_UD;
1457        max_recv_sgl = state->hs_ibtfinfo.hca_attr->hca_recv_sgl_sz;
1458        if ((attr_p->qp_sizes.cs_sq_sgl > max_sgl) ||
1459            (!qp_srq_en && (attr_p->qp_sizes.cs_rq_sgl > max_recv_sgl))) {
1460                status = IBT_HCA_SGL_EXCEEDED;
1461                goto qpalloc_fail7;
1462        }

1464        /*
1465         * Determine this QP's WQE stride (for both the Send and Recv WQEs).
1466         * This will depend on the requested number of SGLs.  Note: this
1467         * has the side-effect of also calculating the real number of SGLs
1468         * (for the calculated WQE size).
1469         *
1470         * For QP's on an SRQ, we set these to 0.
1471         */
1472        if (qp_srq_en) {
1473                qp->qp_rq_log_wqesz = 0;
1474                qp->qp_rq_sgl = 0;
1475        } else {
1476                hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_rq_sgl,
1477                    max_recv_sgl, HERMON_QP_WQ_TYPE_RECVQ,
1478                    &qp->qp_rq_log_wqesz, &qp->qp_rq_sgl);
1479        }
1480        hermon_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
1481            max_sgl, swq_type, &qp->qp_sq_log_wqesz, &qp->qp_sq_sgl);

1483        sq_wqe_size = 1 << qp->qp_sq_log_wqesz;

1485        /* NOTE: currently policy in driver, later maybe IBTF interface */
1486        qp->qp_no_prefetch = 0;

1488        /*
1489         * for prefetching, we need to add the number of wqes in
1490         * the 2k area plus one to the number requested, but
1491         * ONLY for send queue.  If no_prefetch == 1 (prefetch off)
1492         * it's exactly TWO wqes for the headroom
1493         */
1494        if (qp->qp_no_prefetch)
1495                qp->qp_sq_headroom = 2 * sq_wqe_size;
1496        else
1497                qp->qp_sq_headroom = sq_wqe_size + HERMON_QP_OH_SIZE;
1498        /*
1499         * hdrm wqes must be integral since both sq_wqe_size &
1500         * HERMON_QP_OH_SIZE are power of 2
1501         */
1502        qp->qp_sq_hdrmwqes = (qp->qp_sq_headroom / sq_wqe_size);


1505        /*
1506         * Calculate the appropriate size for the work queues.
1507         * For send queue, add in the headroom wqes to the calculation.
1508         * Note:  All Hermon QP work queues must be a power-of-2 in size.  Also
```



```
1509         * they may not be any smaller than HERMON_QP_MIN_SIZE.  This step is
1510         * to round the requested size up to the next highest power-of-2
1511         */
1512        /* first, adjust to a minimum and tell the caller the change */
1513        attr_p->qp_sizes.cs_sq = max(attr_p->qp_sizes.cs_sq,
1514            HERMON_QP_MIN_SIZE);
1515        attr_p->qp_sizes.cs_rq = max(attr_p->qp_sizes.cs_rq,
1516            HERMON_QP_MIN_SIZE);
1517        /*
1518         * now, calculate the alloc size, taking into account
1519         * the headroom for the sq
1520         */
1521        log_qp_sq_size = highbit(attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes);
1522        /* if the total is a power of two, reduce it */
1523        if (ISP2(attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes))  {
1524        if (((attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes)  &
1525            (attr_p->qp_sizes.cs_sq + qp->qp_sq_hdrmwqes - 1)) == 0)    {
1524                log_qp_sq_size = log_qp_sq_size - 1;
1525        }

1527        log_qp_rq_size = highbit(attr_p->qp_sizes.cs_rq);
1528        if (ISP2(attr_p->qp_sizes.cs_rq)) {
1530        if ((attr_p->qp_sizes.cs_rq & (attr_p->qp_sizes.cs_rq - 1)) == 0) {
1529                log_qp_rq_size = log_qp_rq_size - 1;
1530        }

1532        /*
1533         * Next we verify that the rounded-up size is valid (i.e. consistent
1534         * with the device limits and/or software-configured limits).  If not,
1535         * then obviously we have a lot of cleanup to do before returning.
1536         *
1537         * NOTE: the first condition deals with the (test) case of cs_sq
1538         * being just less than 2^32.  In this case, the headroom addition
1539         * to the requested cs_sq will pass the test when it should not.
1540         * This test no longer lets that case slip through the check.
1541         */
1542        if ((attr_p->qp_sizes.cs_sq >
1543            (1 << state->hs_cfg_profile->cp_log_max_qp_sz)) ||
1544            (log_qp_sq_size > state->hs_cfg_profile->cp_log_max_qp_sz) ||
1545            (!qp_srq_en && (log_qp_rq_size >
1546            state->hs_cfg_profile->cp_log_max_qp_sz))) {
1547                status = IBT_HCA_WR_EXCEEDED;
1548                goto qpalloc_fail7;
1549        }

1551        /*
1552         * Allocate the memory for QP work queues. Since Hermon work queues
1553         * are not allowed to cross a 32-bit (4GB) boundary, the alignment of
1554         * the work queue memory is very important.  We used to allocate
1555         * work queues (the combined receive and send queues) so that they
1556         * would be aligned on their combined size.  That alignment guaranteed
1557         * that they would never cross the 4GB boundary (Hermon work queues
1558         * are on the order of MBs at maximum).  Now we are able to relax
1559         * this alignment constraint by ensuring that the IB address assigned
1560         * to the queue memory (as a result of the hermon_mr_register() call)
1561         * is offset from zero.
1562         * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
1563         * guarantee the alignment, but when attempting to use IOMMU bypass
1564         * mode we found that we were not allowed to specify any alignment
1565         * that was more restrictive than the system page size.
1566         * So we avoided this constraint by passing two alignment values,
1567         * one for the memory allocation itself and the other for the DMA
1568         * handle (for later bind).  This used to cause more memory than
1569         * necessary to be allocated (in order to guarantee the more
1570         * restrictive alignment contraint).  But by guaranteeing the
1571         * zero-based IB virtual address for the queue, we are able to
```

```
1572          * conserve this memory.
1573          */
1574         sq_wqe_size = 1 << qp->qp_sq_log_wqesz;
1575         sq_depth    = 1 << log_qp_sq_size;
1576         sq_size     = sq_depth * sq_wqe_size;

1578         /* QP on SRQ sets these to 0 */
1579         if (qp_srq_en) {
1580                 rq_wqe_size = 0;
1581                 rq_size     = 0;
1582         } else {
1583                 rq_wqe_size = 1 << qp->qp_rq_log_wqesz;
1584                 rq_depth    = 1 << log_qp_rq_size;
1585                 rq_size     = rq_depth * rq_wqe_size;
1586         }

1588         qp->qp_wqinfo.qa_size = sq_size + rq_size;
1589         qp->qp_wqinfo.qa_alloc_align = PAGESIZE;
1590         qp->qp_wqinfo.qa_bind_align  = PAGESIZE;
1591         qp->qp_wqinfo.qa_location = HERMON_QUEUE_LOCATION_NORMAL;
1592         status = hermon_queue_alloc(state, &qp->qp_wqinfo, sleepflag);
1593         if (status != DDI_SUCCESS) {
1594                 status = IBT_INSUFF_RESOURCE;
1595                 goto qpalloc_fail7;
1596         }

1598         /*
1599          * Sort WQs in memory according to stride (*q_wqe_size), largest first
1600          * If they are equal, still put the SQ first
1601          */
1602         qp->qp_sq_baseaddr = 0;
1603         qp->qp_rq_baseaddr = 0;
1604         if ((sq_wqe_size > rq_wqe_size) || (sq_wqe_size == rq_wqe_size)) {
1605                 sq_buf = qp->qp_wqinfo.qa_buf_aligned;

1607                 /* if this QP is on an SRQ, set the rq_buf to NULL */
1608                 if (qp_srq_en) {
1609                         rq_buf = NULL;
1610                 } else {
1611                         rq_buf = (uint32_t *)((uintptr_t)sq_buf + sq_size);
1612                         qp->qp_rq_baseaddr = sq_size;
1613                 }
1614         } else {
1615                 rq_buf = qp->qp_wqinfo.qa_buf_aligned;
1616                 sq_buf = (uint32_t *)((uintptr_t)rq_buf + rq_size);
1617                 qp->qp_sq_baseaddr = rq_size;
1618         }

1620         qp->qp_sq_wqhdr = hermon_wrid_wqhdr_create(sq_depth);
1621         if (qp->qp_sq_wqhdr == NULL) {
1622                 status = IBT_INSUFF_RESOURCE;
1623                 goto qpalloc_fail8;
1624         }
1625         if (qp_srq_en) {
1626                 qp->qp_rq_wqavl.wqa_wq = srq->srq_wq_wqhdr;
1627                 qp->qp_rq_wqavl.wqa_srq_en = 1;
1628                 qp->qp_rq_wqavl.wqa_srq = srq;
1629         } else {
1630                 qp->qp_rq_wqhdr = hermon_wrid_wqhdr_create(rq_depth);
1631                 if (qp->qp_rq_wqhdr == NULL) {
1632                         status = IBT_INSUFF_RESOURCE;
1633                         goto qpalloc_fail8;
1634                 }
1635                 qp->qp_rq_wqavl.wqa_wq = qp->qp_rq_wqhdr;
1636         }
1637         qp->qp_sq_wqavl.wqa_qpn = qp->qp_qpnum;
```



```
1638         qp->qp_sq_wqavl.wqa_type = HERMON_WR_SEND;
1639         qp->qp_sq_wqavl.wqa_wq = qp->qp_sq_wqhdr;
1640         qp->qp_rq_wqavl.wqa_qpn = qp->qp_qpnum;
1641         qp->qp_rq_wqavl.wqa_type = HERMON_WR_RECV;

1643         /*
1644          * Register the memory for the QP work queues.  The memory for the
1645          * QP must be registered in the Hermon cMPT tables.  This gives us the
1646          * LKey to specify in the QP context later.  Note: The memory for
1647          * Hermon work queues (both Send and Recv) must be contiguous and
1648          * registered as a single memory region.  Note: If the QP memory is
1649          * user-mappable, force DDI_DMA_CONSISTENT mapping. Also, in order to
1650          * meet the alignment restriction, we pass the "mro_bind_override_addr"
1651          * flag in the call to hermon_mr_register(). This guarantees that the
1652          * resulting IB vaddr will be zero-based (modulo the offset into the
1653          * first page). If we fail here, we still have the bunch of resource
1654          * and reference count cleanup to do.
1655          */
1656         flag = (sleepflag == HERMON_SLEEP) ? IBT_MR_SLEEP :
1657             IBT_MR_NOSLEEP;
1658         mr_attr.mr_vaddr    = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned;
1659         mr_attr.mr_len      = qp->qp_wqinfo.qa_size;
1660         mr_attr.mr_as       = NULL;
1661         mr_attr.mr_flags    = flag;
1662         /* HERMON_QUEUE_LOCATION_NORMAL */
1663         mr_op.mro_bind_type =
1664             state->hs_cfg_profile->cp_iommu_bypass;
1665         mr_op.mro_bind_dmahdl = qp->qp_wqinfo.qa_dmahdl;
1666         mr_op.mro_bind_override_addr = 1;
1667         status = hermon_mr_register(state, pd, &mr_attr, &mr,
1668             &mr_op, HERMON_QP_CMPT);
1669         if (status != DDI_SUCCESS) {
1670                 status = IBT_INSUFF_RESOURCE;
1671                 goto qpalloc_fail9;
1672         }

1674         /*
1675          * Calculate the offset between the kernel virtual address space
1676          * and the IB virtual address space.  This will be used when
1677          * posting work requests to properly initialize each WQE.
1678          */
1679         qp_desc_off = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned -
1680             (uint64_t)mr->mr_bindinfo.bi_addr;

1682         /*
1683          * Fill in all the return arguments (if necessary).  This includes
1684          * real work queue sizes (in wqes), real SGLs, and QP number
1685          */
1686         if (queuesz_p != NULL) {
1687                 queuesz_p->cs_sq        =
1688                     (1 << log_qp_sq_size) - qp->qp_sq_hdrmwqes;
1689                 queuesz_p->cs_sq_sgl    = qp->qp_sq_sgl;

1691                 /* if this QP is on an SRQ, set these to 0 */
1692                 if (qp_srq_en) {
1693                         queuesz_p->cs_rq        = 0;
1694                         queuesz_p->cs_rq_sgl    = 0;
1695                 } else {
1696                         queuesz_p->cs_rq        = (1 << log_qp_rq_size);
1697                         queuesz_p->cs_rq_sgl    = qp->qp_rq_sgl;
1698                 }
1699         }

1701         /*
1702          * Fill in the rest of the Hermon Queue Pair handle.
1703          */
```

```
1704         qp->qp_qpcrsrcp         = NULL;
1705         qp->qp_rsrcp            = rsrc;
1706         qp->qp_state            = HERMON_QP_RESET;
1707         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
1708         qp->qp_pdhdl            = pd;
1709         qp->qp_mrhdl            = mr;
1710         qp->qp_sq_sigtype       = (attr_p->qp_flags & IBT_WR_SIGNALED) ?
1711             HERMON_QP_SQ_WR_SIGNALED : HERMON_QP_SQ_ALL_SIGNALED;
1712         qp->qp_is_special       = 0;
1713         qp->qp_uarpg            = uarpg;
1714         qp->qp_umap_dhp         = (devmap_cookie_t)NULL;
1715         qp->qp_sq_cqhdl         = sq_cq;
1716         qp->qp_sq_bufsz         = (1 << log_qp_sq_size);
1717         qp->qp_sq_logqsz        = log_qp_sq_size;
1718         qp->qp_sq_buf           = sq_buf;
1719         qp->qp_desc_off         = qp_desc_off;
1720         qp->qp_rq_cqhdl         = rq_cq;
1721         qp->qp_rq_buf           = rq_buf;
1722         qp->qp_rlky             = (attr_p->qp_flags & IBT_FAST_REG_RES_LKEY) !=
1723             0;

1725         /* if this QP is on an SRQ, set rq_bufsz to 0 */
1726         if (qp_srq_en) {
1727                 qp->qp_rq_bufsz         = 0;
1728                 qp->qp_rq_logqsz        = 0;
1729         } else {
1730                 qp->qp_rq_bufsz         = (1 << log_qp_rq_size);
1731                 qp->qp_rq_logqsz        = log_qp_rq_size;
1732         }

1734         qp->qp_forward_sqd_event  = 0;
1735         qp->qp_sqd_still_draining = 0;
1736         qp->qp_hdlrarg          = (void *)ibt_qphdl[ii];
1737         qp->qp_mcg_refcnt       = 0;

1739         /*
1740          * If this QP is to be associated with an SRQ, set the SRQ handle
1741          */
1742         if (qp_srq_en) {
1743                 qp->qp_srqhdl = srq;
1744                 hermon_srq_refcnt_inc(qp->qp_srqhdl);
1745         } else {
1746                 qp->qp_srqhdl = NULL;
1747         }

1749         qp->qp_type = IBT_UD_RQP;
1750         qp->qp_serv_type = serv_type;

1752         /*
1753          * Initialize the RQ WQEs - unlike Arbel, no Rcv init is needed
1754          */

1756         /*
1757          * Initialize the SQ WQEs - all that needs to be done is every 64 bytes
1758          * set the quadword to all F's - high-order bit is owner (init to one)
1759          * and the rest for the headroom definition of prefetching.
1760          */
1761         if ((attr_p->qp_alloc_flags & IBT_QP_USES_FEXCH) == 0) {
1762                 wqesz_shift = qp->qp_sq_log_wqesz;
1763                 thewqesz    = 1 << wqesz_shift;
1764                 thewqe = (uint64_t *)(void *)(qp->qp_sq_buf);
1765                 for (i = 0; i < sq_depth; i++) {
1766                         /*
1767                          * for each stride, go through and every 64 bytes
1768                          * write the init value - having set the address
1769                          * once, just keep incrementing it
```



```
1770                          */
1771                         for (j = 0; j < thewqesz; j += 64, thewqe += 8) {
1772                                 *(uint32_t *)thewqe = 0xFFFFFFFF;
1773                         }
1774                 }
1775         }

1777         /* Zero out the QP context */
1778         bzero(&qp->qpc, sizeof (hermon_hw_qpc_t));

1780         /*
1781          * Put QP handle in Hermon QPNum-to-QPHdl list.  Then fill in the
1782          * "qphdl" and return success
1783          */
1784         hermon_icm_set_num_to_hdl(state, HERMON_QPC, qpc->hr_indx + ii, qp);

1786         mutex_init(&qp->qp_sq_lock, NULL, MUTEX_DRIVER,
1787             DDI_INTR_PRI(state->hs_intrmsi_pri));

1789         qp->qp_rangep = qp_range_p;

1791         qphdl[ii] = qp;

1793         if (++ii < (1 << log2))
1794                 goto for_each_qp;

1796         return (DDI_SUCCESS);

1798 /*
1799  * The following is cleanup for all possible failure cases in this routine
1800  */
1801 qpalloc_fail9:
1802         hermon_queue_free(&qp->qp_wqinfo);
1803 qpalloc_fail8:
1804         if (qp->qp_sq_wqhdr)
1805                 hermon_wrid_wqhdr_destroy(qp->qp_sq_wqhdr);
1806         if (qp->qp_rq_wqhdr)
1807                 hermon_wrid_wqhdr_destroy(qp->qp_rq_wqhdr);
1808 qpalloc_fail7:
1809         if (!qp_srq_en) {
1810                 hermon_dbr_free(state, uarpg, qp->qp_rq_vdbr);
1811         }

1813 qpalloc_fail6:
1814         hermon_rsrc_free(state, &rsrc);
1815 qpalloc_fail4:
1816         hermon_cq_refcnt_dec(rq_cq);
1817 qpalloc_fail2:
1818         hermon_cq_refcnt_dec(sq_cq);
1819 qpalloc_fail1:
1820         hermon_pd_refcnt_dec(pd);
1821 qpalloc_fail0:
1822         if (ii == 0) {
1823                 if (qp_range_p)
1824                         kmem_free(qp_range_p, sizeof (*qp_range_p));
1825                 hermon_rsrc_free(state, &qpc);
1826         } else {
1827                 /* qp_range_p and qpc rsrc will be freed in hermon_qp_free */

1829                 mutex_enter(&qp->qp_rangep->hqpr_lock);
1830                 qp_range_p->hqpr_refcnt = ii;
1831                 mutex_exit(&qp->qp_rangep->hqpr_lock);
1832                 while (--ii >= 0) {
1833                         ibc_qpn_hdl_t qpn_hdl;
1834                         int free_status;
```

```
1836                        free_status = hermon_qp_free(state, &qphdl[ii],
1837                            IBC_FREE_QP_AND_QPN, &qpn_hdl, sleepflag);
1838                        if (free_status != DDI_SUCCESS)
1839                                cmn_err(CE_CONT, "!qp_range: status 0x%x: "
1840                                    "error status %x during free",
1841                                    status, free_status);
1842                }
1843        }

1845        return (status);
1846 }
_____unchanged_portion_omitted_


2812 /*
2813  * hermon_qp_sgl_to_logwqesz()
2814  *    Context: Can be called from interrupt or base context.
2815  */
2816 static void
2817 hermon_qp_sgl_to_logwqesz(hermon_state_t *state, uint_t num_sgl,
2818     uint_t real_max_sgl, hermon_qp_wq_type_t wq_type,
2819     uint_t *logwqesz, uint_t *max_sgl)
2820 {
2821        uint_t  max_size, log2, actual_sgl;

2823        switch (wq_type) {
2824        case HERMON_QP_WQ_TYPE_SENDQ_UD:
2825                /*
2826                 * Use requested maximum SGL to calculate max descriptor size
2827                 * (while guaranteeing that the descriptor size is a
2828                 * power-of-2 cachelines).
2829                 */
2830                max_size = (HERMON_QP_WQE_MLX_SND_HDRS + (num_sgl << 4));
2831                log2 = highbit(max_size);
2832                if (ISP2(max_size)) {
2834                if ((max_size & (max_size - 1)) == 0) {
2833                        log2 = log2 - 1;
2834                }

2836                /* Make sure descriptor is at least the minimum size */
2837                log2 = max(log2, HERMON_QP_WQE_LOG_MINIMUM);

2839                /* Calculate actual number of SGL (given WQE size) */
2840                actual_sgl = ((1 << log2) -
2841                    sizeof (hermon_hw_snd_wqe_ctrl_t)) >> 4;
2842                break;

2844        case HERMON_QP_WQ_TYPE_SENDQ_CONN:
2845                /*
2846                 * Use requested maximum SGL to calculate max descriptor size
2847                 * (while guaranteeing that the descriptor size is a
2848                 * power-of-2 cachelines).
2849                 */
2850                max_size = (HERMON_QP_WQE_MLX_SND_HDRS + (num_sgl << 4));
2851                log2 = highbit(max_size);
2852                if (ISP2(max_size)) {
2854                if ((max_size & (max_size - 1)) == 0) {
2853                        log2 = log2 - 1;
2854                }

2856                /* Make sure descriptor is at least the minimum size */
2857                log2 = max(log2, HERMON_QP_WQE_LOG_MINIMUM);

2859                /* Calculate actual number of SGL (given WQE size) */
2860                actual_sgl = ((1 << log2) - HERMON_QP_WQE_MLX_SND_HDRS) >> 4;
2861                break;
```



```
2863        case HERMON_QP_WQ_TYPE_RECVQ:
2864                /*
2865                 * Same as above (except for Recv WQEs)
2866                 */
2867                max_size = (HERMON_QP_WQE_MLX_RCV_HDRS + (num_sgl << 4));
2868                log2 = highbit(max_size);
2869                if (ISP2(max_size)) {
2871                if ((max_size & (max_size - 1)) == 0) {
2870                        log2 = log2 - 1;
2871                }

2873                /* Make sure descriptor is at least the minimum size */
2874                log2 = max(log2, HERMON_QP_WQE_LOG_MINIMUM);

2876                /* Calculate actual number of SGL (given WQE size) */
2877                actual_sgl = ((1 << log2) - HERMON_QP_WQE_MLX_RCV_HDRS) >> 4;
2878                break;

2880        case HERMON_QP_WQ_TYPE_SENDMLX_QP0:
2881                /*
2882                 * Same as above (except for MLX transport WQEs).  For these
2883                 * WQEs we have to account for the space consumed by the
2884                 * "inline" packet headers.  (This is smaller than for QP1
2885                 * below because QP0 is not allowed to send packets with a GRH.
2886                 */
2887                max_size = (HERMON_QP_WQE_MLX_QP0_HDRS + (num_sgl << 4));
2888                log2 = highbit(max_size);
2889                if (ISP2(max_size)) {
2891                if ((max_size & (max_size - 1)) == 0) {
2890                        log2 = log2 - 1;
2891                }

2893                /* Make sure descriptor is at least the minimum size */
2894                log2 = max(log2, HERMON_QP_WQE_LOG_MINIMUM);

2896                /* Calculate actual number of SGL (given WQE size) */
2897                actual_sgl = ((1 << log2) - HERMON_QP_WQE_MLX_QP0_HDRS) >> 4;
2898                break;

2900        case HERMON_QP_WQ_TYPE_SENDMLX_QP1:
2901                /*
2902                 * Same as above.  For these WQEs we again have to account for
2903                 * the space consumed by the "inline" packet headers.  (This
2904                 * is larger than for QP0 above because we have to account for
2905                 * the possibility of a GRH in each packet - and this
2906                 * introduces an alignment issue that causes us to consume
2907                 * an additional 8 bytes).
2908                 */
2909                max_size = (HERMON_QP_WQE_MLX_QP1_HDRS + (num_sgl << 4));
2910                log2 = highbit(max_size);
2911                if (ISP2(max_size)) {
2913                if ((max_size & (max_size - 1)) == 0) {
2912                        log2 = log2 - 1;
2913                }

2915                /* Make sure descriptor is at least the minimum size */
2916                log2 = max(log2, HERMON_QP_WQE_LOG_MINIMUM);

2918                /* Calculate actual number of SGL (given WQE size) */
2919                actual_sgl = ((1 << log2) - HERMON_QP_WQE_MLX_QP1_HDRS) >> 4;
2920                break;

2922        default:
2923                HERMON_WARNING(state, "unexpected work queue type");
2924                break;
```

```
2925         }

2927         /* Fill in the return values */
2928         *logwqesz = log2;
2929         *max_sgl  = min(real_max_sgl, actual_sgl);
2930 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   99416 Thu Oct 23 10:42:13 2014
new/usr/src/uts/common/io/ib/adapters/hermon/hermon_qpmod.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * hermon_qpmod.c
 *    Hermon Queue Pair Modify Routines
 *
 *    This contains all the routines necessary to implement the
 *    ModifyQP() verb.  This includes all the code for legal
 *    transitions to and from Reset, Init, RTR, RTS, SQD, SQErr,
 *    and Error.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/hermon/hermon.h>
#include <sys/ib/ib_pkt_hdrs.h>

static int hermon_qp_reset2init(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_qp_info_t *info_p);
static int hermon_qp_init2init(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int hermon_qp_init2rtr(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int hermon_qp_rtr2rts(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int hermon_qp_rts2rts(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
#ifdef HERMON_NOTNOW
static int hermon_qp_rts2sqd(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags);
#endif
```



```
static int hermon_qp_sqd2rts(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int hermon_qp_sqd2sqd(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int hermon_qp_sqerr2rts(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int hermon_qp_to_error(hermon_state_t *state, hermon_qphdl_t qp);
static int hermon_qp_reset2err(hermon_state_t *state, hermon_qphdl_t qp);

static uint_t hermon_check_rdma_enable_flags(ibt_cep_modify_flags_t flags,
    ibt_qp_info_t *info_p, hermon_hw_qpc_t *qpc);
static int hermon_qp_validate_resp_rsrc(hermon_state_t *state,
    ibt_qp_rc_attr_t *rc, uint_t *rra_max);
static int hermon_qp_validate_init_depth(hermon_state_t *state,
    ibt_qp_rc_attr_t *rc, uint_t *sra_max);
static int hermon_qp_validate_mtu(hermon_state_t *state, uint_t mtu);

/*
 * hermon_qp_modify()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
int
hermon_qp_modify(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p,
    ibt_queue_sizes_t *actual_sz)
{
        ibt_cep_state_t         cur_state, mod_state;
        ibt_cep_modify_flags_t  okflags;
        int                     status;

        /*
         * TODO add support for SUSPEND and RESUME
         */

        /*
         * Lock the QP so that we can modify it atomically.  After grabbing
         * the lock, get the current QP state.  We will use this current QP
         * state to determine the legal transitions (and the checks that need
         * to be performed.)
         * Below is a case for every possible QP state.  In each case, we
         * check that no flags are set which are not valid for the possible
         * transitions from that state.  If these tests pass and the
         * state transition we are attempting is legal, then we call one
         * of the helper functions.  Each of these functions does some
         * additional setup before posting the firmware command for the
         * appropriate state transition.
         */
        mutex_enter(&qp->qp_lock);

        /*
         * Verify that the transport type matches between the serv_type and the
         * qp_trans.  A caller to IBT must specify the qp_trans field as
         * IBT_UD_SRV, IBT_RC_SRV, or IBT_UC_SRV, depending on the QP.  We
         * check here that the correct value was specified, based on our
         * understanding of the QP serv type.
         *
         * Because callers specify part of a 'union' based on what QP type they
         * think they're working with, this ensures that we do not pickup bogus
         * data if the caller thought they were working with a different QP
         * type.
         */
        if (!(HERMON_QP_TYPE_VALID(info_p->qp_trans, qp->qp_serv_type))) {
                mutex_exit(&qp->qp_lock);
                return (IBT_QP_SRV_TYPE_INVALID);
        }
```

```
129         /*
130          * If this is a transition to RTS (which is valid from RTR, RTS,
131          * SQError, and SQ Drain) then we should honor the "current QP state"
132          * specified by the consumer.  This means converting the IBTF QP state
133          * in "info_p->qp_current_state" to an Hermon QP state.  Otherwise, we
134          * assume that we already know the current state (i.e. whatever it was
135          * last modified to or queried as - in "qp->qp_state").
136          */
137         mod_state = info_p->qp_state;
138 
139         if (flags & IBT_CEP_SET_RTR_RTS) {
140                 cur_state = HERMON_QP_RTR;              /* Ready to Receive */
141 
142         } else if ((flags & IBT_CEP_SET_STATE) &&
143             (mod_state == IBT_STATE_RTS)) {
144 
145                 /* Convert the current IBTF QP state to an Hermon QP state */
146                 switch (info_p->qp_current_state) {
147                 case IBT_STATE_RTR:
148                         cur_state = HERMON_QP_RTR;      /* Ready to Receive */
149                         break;
150                 case IBT_STATE_RTS:
151                         cur_state = HERMON_QP_RTS;      /* Ready to Send */
152                         break;
153                 case IBT_STATE_SQE:
154                         cur_state = HERMON_QP_SQERR;    /* Send Queue Error */
155                         break;
156                 case IBT_STATE_SQD:
157                         cur_state = HERMON_QP_SQD;      /* SQ Drained */
158                         break;
159                 default:
160                         mutex_exit(&qp->qp_lock);
161                         return (IBT_QP_STATE_INVALID);
162                 }
163         } else {
164                 cur_state = qp->qp_state;
165         }
166 
167         switch (cur_state) {
168         case HERMON_QP_RESET:
169                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RESET_INIT |
170                     IBT_CEP_SET_RDMA_R | IBT_CEP_SET_RDMA_W |
171                     IBT_CEP_SET_ATOMIC | IBT_CEP_SET_PKEY_IX |
172                     IBT_CEP_SET_PORT | IBT_CEP_SET_QKEY);
173 
174                 /*
175                  * Check for attempts to modify invalid attributes from the
176                  * "Reset" state
177                  */
178                 if (flags & ~okflags) {
179                         mutex_exit(&qp->qp_lock);
180                         status = IBT_QP_ATTR_RO;
181                         goto qpmod_fail;
182                 }
183 
184                 /*
185                  * Verify state transition is to either "Init", back to
186                  * "Reset", or to "Error".
187                  */
188                 if ((flags & IBT_CEP_SET_RESET_INIT) &&
189                     (flags & IBT_CEP_SET_STATE) &&
190                     (mod_state != IBT_STATE_INIT)) {
191                         /* Invalid transition - ambiguous flags */
192                         mutex_exit(&qp->qp_lock);
193                         status = IBT_QP_STATE_INVALID;
```



```
194                         goto qpmod_fail;
195 
196                 } else if ((flags & IBT_CEP_SET_RESET_INIT) ||
197                     ((flags & IBT_CEP_SET_STATE) &&
198                     (mod_state == IBT_STATE_INIT))) {
199                         /*
200                          * Attempt to transition from "Reset" to "Init"
201                          */
202                         status = hermon_qp_reset2init(state, qp, info_p);
203                         if (status != DDI_SUCCESS) {
204                                 mutex_exit(&qp->qp_lock);
205                                 goto qpmod_fail;
206                         }
207                         qp->qp_state = HERMON_QP_INIT;
208                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_INIT);
209 
210                 } else if ((flags & IBT_CEP_SET_STATE) &&
211                     (mod_state == IBT_STATE_RESET)) {
212                         /*
213                          * Attempt to transition from "Reset" back to "Reset"
214                          *    Nothing to do here really... just drop the lock
215                          *    and return success.  The qp->qp_state should
216                          *    already be set to HERMON_QP_RESET.
217                          *
218                          * Note: We return here because we do not want to fall
219                          *    through to the hermon_wrid_from_reset_handling()
220                          *    routine below (since we are not really moving
221                          *    _out_ of the "Reset" state.
222                          */
223                         mutex_exit(&qp->qp_lock);
224                         return (DDI_SUCCESS);
225 
226                 } else if ((flags & IBT_CEP_SET_STATE) &&
227                     (mod_state == IBT_STATE_ERROR)) {
228                         /*
229                          * Attempt to transition from "Reset" to "Error"
230                          */
231                         status = hermon_qp_reset2err(state, qp);
232                         if (status != DDI_SUCCESS) {
233                                 mutex_exit(&qp->qp_lock);
234                                 goto qpmod_fail;
235                         }
236                         qp->qp_state = HERMON_QP_ERR;
237                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_ERR);
238 
239                 } else {
240                         /* Invalid transition - return error */
241                         mutex_exit(&qp->qp_lock);
242                         status = IBT_QP_STATE_INVALID;
243                         goto qpmod_fail;
244                 }
245 
246                 /*
247                  * Do any additional handling necessary here for the transition
248                  * from the "Reset" state (e.g. re-initialize the workQ WRID
249                  * lists).  Note: If hermon_wrid_from_reset_handling() fails,
250                  * then we attempt to transition the QP back to the "Reset"
251                  * state.  If that fails, then it is an indication of a serious
252                  * problem (either HW or SW).  So we print out a warning
253                  * message and return failure.
254                  */
255                 status = hermon_wrid_from_reset_handling(state, qp);
256                 if (status != DDI_SUCCESS) {
257                         if (hermon_qp_to_reset(state, qp) != DDI_SUCCESS) {
258                                 HERMON_WARNING(state, "failed to reset QP");
259                         }
```

```
260                         qp->qp_state = HERMON_QP_RESET;
261                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);

263                         mutex_exit(&qp->qp_lock);
264                         goto qpmod_fail;
265                 }
266                 break;

268         case HERMON_QP_INIT:
269                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_INIT_RTR |
270                     IBT_CEP_SET_ADDS_VECT | IBT_CEP_SET_RDMARA_IN |
271                     IBT_CEP_SET_MIN_RNR_NAK | IBT_CEP_SET_ALT_PATH |
272                     IBT_CEP_SET_RDMA_R | IBT_CEP_SET_RDMA_W |
273                     IBT_CEP_SET_ATOMIC | IBT_CEP_SET_PKEY_IX |
274                     IBT_CEP_SET_QKEY | IBT_CEP_SET_PORT);

276                 /*
277                  * Check for attempts to modify invalid attributes from the
278                  * "Init" state
279                  */
280                 if (flags & ~okflags) {
281                         mutex_exit(&qp->qp_lock);
282                         status = IBT_QP_ATTR_RO;
283                         goto qpmod_fail;
284                 }

286                 /*
287                  * Verify state transition is to either "RTR", back to "Init",
288                  * to "Reset", or to "Error"
289                  */
290                 if ((flags & IBT_CEP_SET_INIT_RTR) &&
291                     (flags & IBT_CEP_SET_STATE) &&
292                     (mod_state != IBT_STATE_RTR)) {
293                         /* Invalid transition - ambiguous flags */
294                         mutex_exit(&qp->qp_lock);
295                         status = IBT_QP_STATE_INVALID;
296                         goto qpmod_fail;

298                 } else if ((flags & IBT_CEP_SET_INIT_RTR) ||
299                     ((flags & IBT_CEP_SET_STATE) &&
300                     (mod_state == IBT_STATE_RTR))) {
301                         /*
302                          * Attempt to transition from "Init" to "RTR"
303                          */
304                         status = hermon_qp_init2rtr(state, qp, flags, info_p);
305                         if (status != DDI_SUCCESS) {
306                                 mutex_exit(&qp->qp_lock);
307                                 goto qpmod_fail;
308                         }
309                         qp->qp_state = HERMON_QP_RTR;
310                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RTR);

312                 } else if ((flags & IBT_CEP_SET_STATE) &&
313                     (mod_state == IBT_STATE_INIT)) {
314                         /*
315                          * Attempt to transition from "Init" to "Init"
316                          */
317                         status = hermon_qp_init2init(state, qp, flags, info_p);
318                         if (status != DDI_SUCCESS) {
319                                 mutex_exit(&qp->qp_lock);
320                                 goto qpmod_fail;
321                         }
322                         qp->qp_state = HERMON_QP_INIT;
323                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_INIT);

325                 } else if ((flags & IBT_CEP_SET_STATE) &&
```



```
326                     (mod_state == IBT_STATE_RESET)) {
327                         /*
328                          * Attempt to transition from "Init" to "Reset"
329                          */
330                         status = hermon_qp_to_reset(state, qp);
331                         if (status != DDI_SUCCESS) {
332                                 mutex_exit(&qp->qp_lock);
333                                 goto qpmod_fail;
334                         }
335                         qp->qp_state = HERMON_QP_RESET;
336                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);

338                         /*
339                          * Do any additional handling necessary for the
340                          * transition _to_ the "Reset" state (e.g. update the
341                          * workQ WRID lists)
342                          */
343                         status = hermon_wrid_to_reset_handling(state, qp);
344                         if (status != IBT_SUCCESS) {
345                                 mutex_exit(&qp->qp_lock);
346                                 goto qpmod_fail;
347                         }

349                 } else if ((flags & IBT_CEP_SET_STATE) &&
350                     (mod_state == IBT_STATE_ERROR)) {
351                         /*
352                          * Attempt to transition from "Init" to "Error"
353                          */
354                         status = hermon_qp_to_error(state, qp);
355                         if (status != DDI_SUCCESS) {
356                                 mutex_exit(&qp->qp_lock);
357                                 goto qpmod_fail;
358                         }
359                         qp->qp_state = HERMON_QP_ERR;
360                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_ERR);

362                 } else {
363                         /* Invalid transition - return error */
364                         mutex_exit(&qp->qp_lock);
365                         status = IBT_QP_STATE_INVALID;
366                         goto qpmod_fail;
367                 }
368                 break;

370         case HERMON_QP_RTR:
371                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RTR_RTS |
372                     IBT_CEP_SET_TIMEOUT | IBT_CEP_SET_RETRY |
373                     IBT_CEP_SET_RNR_NAK_RETRY | IBT_CEP_SET_RDMARA_OUT |
374                     IBT_CEP_SET_RDMA_R | IBT_CEP_SET_RDMA_W |
375                     IBT_CEP_SET_ATOMIC | IBT_CEP_SET_QKEY |
376                     IBT_CEP_SET_ALT_PATH | IBT_CEP_SET_MIG |
377                     IBT_CEP_SET_MIN_RNR_NAK);

379                 /*
380                  * Check for attempts to modify invalid attributes from the
381                  * "RTR" state
382                  */
383                 if (flags & ~okflags) {
384                         mutex_exit(&qp->qp_lock);
385                         status = IBT_QP_ATTR_RO;
386                         goto qpmod_fail;
387                 }

389                 /*
390                  * Verify state transition is to either "RTS", "Reset",
391                  * or "Error"
```

```
392                  */
393                 if ((flags & IBT_CEP_SET_RTR_RTS) &&
394                     (flags & IBT_CEP_SET_STATE) &&
395                     (mod_state != IBT_STATE_RTS)) {
396                         /* Invalid transition - ambiguous flags */
397                         mutex_exit(&qp->qp_lock);
398                         status = IBT_QP_STATE_INVALID;
399                         goto qpmod_fail;
400
401                 } else if ((flags & IBT_CEP_SET_RTR_RTS) ||
402                     ((flags & IBT_CEP_SET_STATE) &&
403                     (mod_state == IBT_STATE_RTS))) {
404                         /*
405                          * Attempt to transition from "RTR" to "RTS"
406                          */
407                         status = hermon_qp_rtr2rts(state, qp, flags, info_p);
408                         if (status != DDI_SUCCESS) {
409                                 mutex_exit(&qp->qp_lock);
410                                 goto qpmod_fail;
411                         }
412                         qp->qp_state = HERMON_QP_RTS;
413                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RTS);
414
415                 } else if ((flags & IBT_CEP_SET_STATE) &&
416                     (mod_state == IBT_STATE_RESET)) {
417                         /*
418                          * Attempt to transition from "RTR" to "Reset"
419                          */
420                         status = hermon_qp_to_reset(state, qp);
421                         if (status != DDI_SUCCESS) {
422                                 mutex_exit(&qp->qp_lock);
423                                 goto qpmod_fail;
424                         }
425                         qp->qp_state = HERMON_QP_RESET;
426                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
427
428                         /*
429                          * Do any additional handling necessary for the
430                          * transition _to_ the "Reset" state (e.g. update the
431                          * workQ WRID lists)
432                          */
433                         status = hermon_wrid_to_reset_handling(state, qp);
434                         if (status != IBT_SUCCESS) {
435                                 mutex_exit(&qp->qp_lock);
436                                 goto qpmod_fail;
437                         }
438
439                 } else if ((flags & IBT_CEP_SET_STATE) &&
440                     (mod_state == IBT_STATE_ERROR)) {
441                         /*
442                          * Attempt to transition from "RTR" to "Error"
443                          */
444                         status = hermon_qp_to_error(state, qp);
445                         if (status != DDI_SUCCESS) {
446                                 mutex_exit(&qp->qp_lock);
447                                 goto qpmod_fail;
448                         }
449                         qp->qp_state = HERMON_QP_ERR;
450                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_ERR);
451
452                 } else {
453                         /* Invalid transition - return error */
454                         mutex_exit(&qp->qp_lock);
455                         status = IBT_QP_STATE_INVALID;
456                         goto qpmod_fail;
457                 }
```



```
458                 break;
459
460         case HERMON_QP_RTS:
461                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RDMA_R |
462                     IBT_CEP_SET_RDMA_W | IBT_CEP_SET_ATOMIC |
463                     IBT_CEP_SET_QKEY | IBT_CEP_SET_ALT_PATH |
464                     IBT_CEP_SET_MIG | IBT_CEP_SET_MIN_RNR_NAK |
465                     IBT_CEP_SET_SQD_EVENT);
466
467                 /*
468                  * Check for attempts to modify invalid attributes from the
469                  * "RTS" state
470                  */
471                 if (flags & ~okflags) {
472                         mutex_exit(&qp->qp_lock);
473                         status = IBT_QP_ATTR_RO;
474                         goto qpmod_fail;
475                 }
476
477                 /*
478                  * Verify state transition is to either "RTS", "SQD", "Reset",
479                  * or "Error"
480                  */
481                 if ((flags & IBT_CEP_SET_STATE) &&
482                     (mod_state == IBT_STATE_RTS)) {
483                         /*
484                          * Attempt to transition from "RTS" to "RTS"
485                          */
486                         status = hermon_qp_rts2rts(state, qp, flags, info_p);
487                         if (status != DDI_SUCCESS) {
488                                 mutex_exit(&qp->qp_lock);
489                                 goto qpmod_fail;
490                         }
491                         qp->qp_state = HERMON_QP_RTS;
492                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RTS);
493
494                 } else if ((flags & IBT_CEP_SET_STATE) &&
495                     (mod_state == IBT_STATE_SQD)) {
496 #ifdef HERMON_NOTNOW
497                         /*
498                          * Attempt to transition from "RTS" to "SQD"
499                          */
500                         status = hermon_qp_rts2sqd(state, qp, flags);
501                         if (status != DDI_SUCCESS) {
502                                 mutex_exit(&qp->qp_lock);
503                                 goto qpmod_fail;
504                         }
505                         qp->qp_state = HERMON_QP_SQD;
506                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_SQD);
507 #else
508                         /* hack because of the lack of fw support for SQD */
509                         mutex_exit(&qp->qp_lock);
510                         status = IBT_QP_STATE_INVALID;
511                         goto qpmod_fail;
512 #endif
513
514                 } else if ((flags & IBT_CEP_SET_STATE) &&
515                     (mod_state == IBT_STATE_RESET)) {
516                         /*
517                          * Attempt to transition from "RTS" to "Reset"
518                          */
519                         status = hermon_qp_to_reset(state, qp);
520                         if (status != DDI_SUCCESS) {
521                                 mutex_exit(&qp->qp_lock);
522                                 goto qpmod_fail;
523                         }
```

```
524                         qp->qp_state = HERMON_QP_RESET;
525                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
527                         /*
528                          * Do any additional handling necessary for the
529                          * transition _to_ the "Reset" state (e.g. update the
530                          * workQ WRID lists)
531                          */
532                         status = hermon_wrid_to_reset_handling(state, qp);
533                         if (status != IBT_SUCCESS) {
534                                 mutex_exit(&qp->qp_lock);
535                                 goto qpmod_fail;
536                         }
538                 } else if ((flags & IBT_CEP_SET_STATE) &&
539                     (mod_state == IBT_STATE_ERROR)) {
540                         /*
541                          * Attempt to transition from "RTS" to "Error"
542                          */
543                         status = hermon_qp_to_error(state, qp);
544                         if (status != DDI_SUCCESS) {
545                                 mutex_exit(&qp->qp_lock);
546                                 goto qpmod_fail;
547                         }
548                         qp->qp_state = HERMON_QP_ERR;
549                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_ERR);
551                 } else {
552                         /* Invalid transition - return error */
553                         mutex_exit(&qp->qp_lock);
554                         status = IBT_QP_STATE_INVALID;
555                         goto qpmod_fail;
556                 }
557                 break;
559         case HERMON_QP_SQERR:
560                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RDMA_R |
561                     IBT_CEP_SET_RDMA_W | IBT_CEP_SET_ATOMIC |
562                     IBT_CEP_SET_QKEY | IBT_CEP_SET_MIN_RNR_NAK);
564                 /*
565                  * Check for attempts to modify invalid attributes from the
566                  * "SQErr" state
567                  */
568                 if (flags & ~okflags) {
569                         mutex_exit(&qp->qp_lock);
570                         status = IBT_QP_ATTR_RO;
571                         goto qpmod_fail;
572                 }
574                 /*
575                  * Verify state transition is to either "RTS", "Reset", or
576                  * "Error"
577                  */
578                 if ((flags & IBT_CEP_SET_STATE) &&
579                     (mod_state == IBT_STATE_RTS)) {
580                         /*
581                          * Attempt to transition from "SQErr" to "RTS"
582                          */
583                         status = hermon_qp_sqerr2rts(state, qp, flags, info_p);
584                         if (status != DDI_SUCCESS) {
585                                 mutex_exit(&qp->qp_lock);
586                                 goto qpmod_fail;
587                         }
588                         qp->qp_state = HERMON_QP_RTS;
589                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RTS);
```



```
591                 } else if ((flags & IBT_CEP_SET_STATE) &&
592                     (mod_state == IBT_STATE_RESET)) {
593                         /*
594                          * Attempt to transition from "SQErr" to "Reset"
595                          */
596                         status = hermon_qp_to_reset(state, qp);
597                         if (status != DDI_SUCCESS) {
598                                 mutex_exit(&qp->qp_lock);
599                                 goto qpmod_fail;
600                         }
601                         qp->qp_state = HERMON_QP_RESET;
602                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
604                         /*
605                          * Do any additional handling necessary for the
606                          * transition _to_ the "Reset" state (e.g. update the
607                          * workQ WRID lists)
608                          */
609                         status = hermon_wrid_to_reset_handling(state, qp);
610                         if (status != IBT_SUCCESS) {
611                                 mutex_exit(&qp->qp_lock);
612                                 goto qpmod_fail;
613                         }
615                 } else if ((flags & IBT_CEP_SET_STATE) &&
616                     (mod_state == IBT_STATE_ERROR)) {
617                         /*
618                          * Attempt to transition from "SQErr" to "Error"
619                          */
620                         status = hermon_qp_to_error(state, qp);
621                         if (status != DDI_SUCCESS) {
622                                 mutex_exit(&qp->qp_lock);
623                                 goto qpmod_fail;
624                         }
625                         qp->qp_state = HERMON_QP_ERR;
626                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_ERR);
628                 } else {
629                         /* Invalid transition - return error */
630                         mutex_exit(&qp->qp_lock);
631                         status = IBT_QP_STATE_INVALID;
632                         goto qpmod_fail;
633                 }
634                 break;
636         case HERMON_QP_SQD:
637                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_ADDS_VECT |
638                     IBT_CEP_SET_ALT_PATH | IBT_CEP_SET_MIG |
639                     IBT_CEP_SET_RDMARA_OUT | IBT_CEP_SET_RDMARA_IN |
640                     IBT_CEP_SET_QKEY | IBT_CEP_SET_PKEY_IX |
641                     IBT_CEP_SET_TIMEOUT | IBT_CEP_SET_RETRY |
642                     IBT_CEP_SET_RNR_NAK_RETRY | IBT_CEP_SET_PORT |
643                     IBT_CEP_SET_MIN_RNR_NAK | IBT_CEP_SET_RDMA_R |
644                     IBT_CEP_SET_RDMA_W | IBT_CEP_SET_ATOMIC);
646                 /*
647                  * Check for attempts to modify invalid attributes from the
648                  * "SQD" state
649                  */
650                 if (flags & ~okflags) {
651                         mutex_exit(&qp->qp_lock);
652                         status = IBT_QP_ATTR_RO;
653                         goto qpmod_fail;
654                 }
```

```
656                 /*
657                  * Verify state transition is to either "SQD", "RTS", "Reset",
658                  * or "Error"
659                  */
660
661                 if ((flags & IBT_CEP_SET_STATE) &&
662                     (mod_state == IBT_STATE_SQD)) {
663                         /*
664                          * Attempt to transition from "SQD" to "SQD"
665                          */
666                         status = hermon_qp_sqd2sqd(state, qp, flags, info_p);
667                         if (status != DDI_SUCCESS) {
668                                 mutex_exit(&qp->qp_lock);
669                                 goto qpmod_fail;
670                         }
671                         qp->qp_state = HERMON_QP_SQD;
672                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_SQD);
673
674                 } else if ((flags & IBT_CEP_SET_STATE) &&
675                     (mod_state == IBT_STATE_RTS)) {
676                         /*
677                          * If still draining SQ, then fail transition attempt
678                          * to RTS, even though this is now done is two steps
679                          * (see below) if the consumer has tried this before
680                          * it's drained, let him fail and wait appropriately
681                          */
682                         if (qp->qp_sqd_still_draining) {
683                                 mutex_exit(&qp->qp_lock);
684                                 goto qpmod_fail;
685                         }
686                         /*
687                          * IBA 1.2 has changed - most/all the things that were
688                          * done in SQD2RTS can be done in SQD2SQD.  So make this
689                          * a 2-step process.  First, set any attributes requsted
690                          * w/ SQD2SQD, but no real transition.
691                          *
692                          * First, Attempt to transition from "SQD" to "SQD"
693                          */
694                         status = hermon_qp_sqd2sqd(state, qp, flags, info_p);
695                         if (status != DDI_SUCCESS) {
696                                 mutex_exit(&qp->qp_lock);
697                                 goto qpmod_fail;
698                         }
699                         qp->qp_state = HERMON_QP_SQD;
700                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_SQD);
701
702                         /*
703                          * The, attempt to transition from "SQD" to "RTS", but
704                          * request only the state transition, no attributes
705                          */
706
707                         status = hermon_qp_sqd2rts(state, qp,
708                             IBT_CEP_SET_STATE, info_p);
709                         if (status != DDI_SUCCESS) {
710                                 mutex_exit(&qp->qp_lock);
711                                 goto qpmod_fail;
712                         }
713                         qp->qp_state = HERMON_QP_RTS;
714                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RTS);
715
716                 } else if ((flags & IBT_CEP_SET_STATE) &&
717                     (mod_state == IBT_STATE_RESET)) {
718                         /*
719                          * Attempt to transition from "SQD" to "Reset"
720                          */
721                         status = hermon_qp_to_reset(state, qp);
```



```
722                         if (status != DDI_SUCCESS) {
723                                 mutex_exit(&qp->qp_lock);
724                                 goto qpmod_fail;
725                         }
726                         qp->qp_state = HERMON_QP_RESET;
727                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
728
729                         /*
730                          * Do any additional handling necessary for the
731                          * transition _to_ the "Reset" state (e.g. update the
732                          * workQ WRID lists)
733                          */
734                         status = hermon_wrid_to_reset_handling(state, qp);
735                         if (status != IBT_SUCCESS) {
736                                 mutex_exit(&qp->qp_lock);
737                                 goto qpmod_fail;
738                         }
739
740                 } else if ((flags & IBT_CEP_SET_STATE) &&
741                     (mod_state == IBT_STATE_ERROR)) {
742                         /*
743                          * Attempt to transition from "SQD" to "Error"
744                          */
745                         status = hermon_qp_to_error(state, qp);
746                         if (status != DDI_SUCCESS) {
747                                 mutex_exit(&qp->qp_lock);
748                                 goto qpmod_fail;
749                         }
750                         qp->qp_state = HERMON_QP_ERR;
751                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_ERR);
752
753                 } else {
754                         /* Invalid transition - return error */
755                         mutex_exit(&qp->qp_lock);
756                         status = IBT_QP_STATE_INVALID;
757                         goto qpmod_fail;
758                 }
759                 break;
760
761         case HERMON_QP_ERR:
762                 /*
763                  * Verify state transition is to either "Reset" or back to
764                  * "Error"
765                  */
766                 if ((flags & IBT_CEP_SET_STATE) &&
767                     (mod_state == IBT_STATE_RESET)) {
768                         /*
769                          * Attempt to transition from "Error" to "Reset"
770                          */
771                         status = hermon_qp_to_reset(state, qp);
772                         if (status != DDI_SUCCESS) {
773                                 mutex_exit(&qp->qp_lock);
774                                 goto qpmod_fail;
775                         }
776                         qp->qp_state = HERMON_QP_RESET;
777                         HERMON_SET_QP_POST_SEND_STATE(qp, HERMON_QP_RESET);
778
779                         /*
780                          * Do any additional handling necessary for the
781                          * transition _to_ the "Reset" state (e.g. update the
782                          * workQ WRID lists)
783                          */
784                         status = hermon_wrid_to_reset_handling(state, qp);
785                         if (status != IBT_SUCCESS) {
786                                 mutex_exit(&qp->qp_lock);
787                                 goto qpmod_fail;
```

```
788                         }
789
790                 } else if ((flags & IBT_CEP_SET_STATE) &&
791                     (mod_state == IBT_STATE_ERROR)) {
792                         /*
793                          * Attempt to transition from "Error" back to "Error"
794                          *    Nothing to do here really... just drop the lock
795                          *    and return success.  The qp->qp_state should
796                          *    already be set to HERMON_QP_ERR.
797                          *
798                          */
799                         mutex_exit(&qp->qp_lock);
800                         return (DDI_SUCCESS);
801
802                 } else {
803                         /* Invalid transition - return error */
804                         mutex_exit(&qp->qp_lock);
805                         status = IBT_QP_STATE_INVALID;
806                         goto qpmod_fail;
807                 }
808                 break;
809
810         default:
811                 /*
812                  * Invalid QP state.  If we got here then it's a warning of
813                  * a probably serious problem.  So print a message and return
814                  * failure
815                  */
816                 mutex_exit(&qp->qp_lock);
817                 HERMON_WARNING(state, "unknown QP state in modify");
818                 status = IBT_QP_STATE_INVALID;
819                 goto qpmod_fail;
820         }
821
822         mutex_exit(&qp->qp_lock);
823         return (DDI_SUCCESS);
824
825 qpmod_fail:
826         return (status);
827 }
828
829
830 /*
831  * hermon_qp_reset2init()
832  *    Context: Can be called from interrupt or base context.
833  */
834 static int
835 hermon_qp_reset2init(hermon_state_t *state, hermon_qphdl_t qp,
836     ibt_qp_info_t *info_p)
837 {
838         hermon_hw_qpc_t         *qpc;
839         ibt_qp_rc_attr_t        *rc;
840         ibt_qp_ud_attr_t        *ud;
841         ibt_qp_uc_attr_t        *uc;
842         uint_t                  portnum, pkeyindx;
843         int                     status;
844         uint32_t                cqnmask;
845         int                     qp_srq_en;
846
847         ASSERT(MUTEX_HELD(&qp->qp_lock));
848
849         /*
850          * Grab the temporary QPC entry from QP software state
851          */
852         qpc = &qp->qpc;
```



```
853
854         /*
855          * Fill in the common fields in the QPC
856          */
857
858         if (qp->qp_is_special) {
859                 qpc->serv_type  = HERMON_QP_MLX;
860         } else {
861                 qpc->serv_type  = qp->qp_serv_type;
862         }
863         qpc->pm_state           = HERMON_QP_PMSTATE_MIGRATED;
864
865         qpc->pd                 = qp->qp_pdhdl->pd_pdnum;
866
867         qpc->log_sq_stride      = qp->qp_sq_log_wqesz - 4;
868         qpc->log_rq_stride      = qp->qp_rq_log_wqesz - 4;
869         qpc->sq_no_prefetch     = qp->qp_no_prefetch;
870         qpc->log_sq_size        = highbit(qp->qp_sq_bufsz) - 1;
871         qpc->log_rq_size        = highbit(qp->qp_rq_bufsz) - 1;
872
873         qpc->usr_page           = qp->qp_uarpg;
874
875         cqnmask = (1 << state->hs_cfg_profile->cp_log_num_cq) - 1;
876         qpc->cqn_snd            =
877             (qp->qp_sq_cqhdl == NULL) ? 0 : qp->qp_sq_cqhdl->cq_cqnum & cqnmask;
878         qpc->page_offs          = qp->qp_wqinfo.qa_pgoffs >> 6;
879         qpc->cqn_rcv            =
880             (qp->qp_rq_cqhdl == NULL) ? 0 : qp->qp_rq_cqhdl->cq_cqnum & cqnmask;
881
882         /* dbr is now an address, not an index */
883         qpc->dbr_addrh          = ((uint64_t)qp->qp_rq_pdbr >> 32);
884         qpc->dbr_addrl          = ((uint64_t)qp->qp_rq_pdbr & 0xFFFFFFFC) >> 2;
885         qpc->sq_wqe_counter     = 0;
886         qpc->rq_wqe_counter     = 0;
887         /*
888          * HERMON:
889          * qpc->wqe_baseaddr is replaced by LKey from the cMPT, and
890          * page_offset, mtt_base_addr_h/l, and log2_page_size will
891          * be used to map the WQE buffer
892          * NOTE that the cMPT is created implicitly when the QP is
893          * transitioned from reset to init
894          */
895         qpc->log2_pgsz          = qp->qp_mrhdl->mr_log2_pgsz;
896         qpc->mtt_base_addrl     = (qp->qp_mrhdl->mr_mttaddr) >> 3;
897         qpc->mtt_base_addrh     = (uint32_t)((qp->qp_mrhdl->mr_mttaddr >> 32) &
898             0xFF);
899         qp_srq_en               = (qp->qp_alloc_flags & IBT_QP_USES_SRQ) != 0;
900         qpc->srq_en             = qp_srq_en;
901
902         if (qp_srq_en) {
903                 qpc->srq_number = qp->qp_srqhdl->srq_srqnum;
904         } else {
905                 qpc->srq_number = 0;
906         }
907
908         /*
909          * Fast Registration Work Requests and Reserved Lkey are enabled
910          * with the single IBT bit stored in qp_rlky.
911          */
912         qpc->fre                = qp->qp_rlky;
913         qpc->rlky               = qp->qp_rlky;
914
915         /* 1.2 verbs extensions disabled for now */
916         qpc->header_sep         = 0; /* disable header separation for now */
917         qpc->rss                = qp->qp_alloc_flags & IBT_QP_USES_RSS ? 1 : 0;
918         qpc->inline_scatter     = 0; /* disable inline scatter for now */
```

```
920         /*
921          * Now fill in the QPC fields which are specific to transport type
922          */
923         if (qp->qp_type == IBT_UD_RQP) {
924                 int my_fc_id_idx, exch_base;

926                 ud = &info_p->qp_transport.ud;

928                 /* Set the QKey */
929                 qpc->qkey = ud->ud_qkey;

931                 /*
932                  * Set MTU and message max. Hermon checks the QPC
933                  * MTU settings rather than just the port MTU,
934                  * so set it to maximum size.
935                  */
936                 qpc->mtu = HERMON_MAX_MTU;
937                 if (qp->qp_uses_lso)
938                         qpc->msg_max = state->hs_devlim.log_max_gso_sz;
939                 else if (qp->qp_is_special)
940                         qpc->msg_max = HERMON_MAX_MTU + 6;
941                 else
942                         qpc->msg_max = HERMON_QP_LOG_MAX_MSGSZ;

944                 /* Check for valid port number and fill it in */
945                 portnum = ud->ud_port;
946                 if (hermon_portnum_is_valid(state, portnum)) {
947                         qp->qp_portnum = portnum - 1;
948                         qpc->pri_addr_path.sched_q =
949                             HERMON_QP_SCHEDQ_GET(portnum - 1,
950                             0, qp->qp_is_special);
951                 } else {
952                         return (IBT_HCA_PORT_INVALID);
953                 }


956                 /* Check for valid PKey index and fill it in */
957                 pkeyindx = ud->ud_pkey_ix;
958                 if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
959                         qpc->pri_addr_path.pkey_indx = pkeyindx;
960                         qp->qp_pkeyindx = pkeyindx;
961                 } else {
962                         return (IBT_PKEY_IX_ILLEGAL);
963                 }

965                 /* fill in the RSS fields */
966                 if (qpc->rss) {
967                         struct hermon_hw_rss_s *rssp;
968                         ibt_rss_flags_t flags = ud->ud_rss.rss_flags;

970                         rssp = (struct hermon_hw_rss_s *)&qpc->pri_addr_path;
971                         rssp->log2_tbl_sz = ud->ud_rss.rss_log2_table;
972                         rssp->base_qpn = ud->ud_rss.rss_base_qpn;
973                         rssp->default_qpn = ud->ud_rss.rss_def_qpn;
974                         if (flags & IBT_RSS_ALG_XOR)
975                                 rssp->hash_fn = 0;      /* XOR Hash Function */
976                         else if (flags & IBT_RSS_ALG_TPL)
977                                 rssp->hash_fn = 1;      /* Toeplitz Hash Fn */
978                         else
979                                 return (IBT_INVALID_PARAM);
980                         rssp->ipv4 = (flags & IBT_RSS_HASH_IPV4) != 0;
981                         rssp->tcp_ipv4 = (flags & IBT_RSS_HASH_TCP_IPV4) != 0;
982                         rssp->ipv6 = (flags & IBT_RSS_HASH_IPV6) != 0;
983                         rssp->tcp_ipv4 = (flags & IBT_RSS_HASH_TCP_IPV6) != 0;
984                         bcopy(ud->ud_rss.rss_toe_key, rssp->rss_key, 40);
985                 } else if (qp->qp_serv_type == HERMON_QP_RFCI) {
```



```
986                         status = hermon_fcoib_set_id(state, portnum,
987                             qp->qp_qpnum, ud->ud_fc.fc_src_id);
988                         if (status != DDI_SUCCESS)
989                                 return (status);
990                         qp->qp_fc_attr = ud->ud_fc;
991                 } else if (qp->qp_serv_type == HERMON_QP_FEXCH) {
992                         my_fc_id_idx = hermon_fcoib_get_id_idx(state,
993                             portnum, &ud->ud_fc);
994                         if (my_fc_id_idx == -1)
995                                 return (IBT_INVALID_PARAM);
996                         qpc->my_fc_id_idx = my_fc_id_idx;

998                         status = hermon_fcoib_fexch_mkey_init(state,
999                             qp->qp_pdhdl, ud->ud_fc.fc_hca_port,
1000                            qp->qp_qpnum, HERMON_CMD_NOSLEEP_SPIN);
1001                        if (status != DDI_SUCCESS)
1002                                return (status);
1003                        qp->qp_fc_attr = ud->ud_fc;
1004                } else if (qp->qp_serv_type == HERMON_QP_FCMND) {
1005                        my_fc_id_idx = hermon_fcoib_get_id_idx(state,
1006                            portnum, &ud->ud_fc);
1007                        if (my_fc_id_idx == -1)
1008                                return (IBT_INVALID_PARAM);
1009                        qpc->my_fc_id_idx = my_fc_id_idx;
1010                        exch_base = hermon_fcoib_check_exch_base_off(state,
1011                            portnum, &ud->ud_fc);
1012                        if (exch_base == -1)
1013                                return (IBT_INVALID_PARAM);
1014                        qpc->exch_base = exch_base;
1015                        qpc->exch_size = ud->ud_fc.fc_exch_log2_sz;
1016                        qpc->qp_fc_attr = ud->ud_fc;
1017                }

1019        } else if (qp->qp_serv_type == HERMON_QP_RC) {
1020                rc = &info_p->qp_transport.rc;

1022                /* Set the RDMA (recv) enable/disable flags */
1023                qpc->rre = (info_p->qp_flags & IBT_CEP_RDMA_RD) ? 1 : 0;
1024                qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1025                qpc->rae = (info_p->qp_flags & IBT_CEP_ATOMIC)  ? 1 : 0;

1027                /* Check for valid port number and fill it in */
1028                portnum = rc->rc_path.cep_hca_port_num;
1029                if (hermon_portnum_is_valid(state, portnum)) {
1030                        qp->qp_portnum = portnum - 1;
1031                        qpc->pri_addr_path.sched_q =
1032                            HERMON_QP_SCHEDQ_GET(portnum - 1,
1033                            0, qp->qp_is_special);
1034                } else {
1035                        return (IBT_HCA_PORT_INVALID);
1036                }

1038                /* Check for valid PKey index and fill it in */
1039                pkeyindx = rc->rc_path.cep_pkey_ix;
1040                if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1041                        qpc->pri_addr_path.pkey_indx = pkeyindx;
1042                } else {
1043                        return (IBT_PKEY_IX_ILLEGAL);
1044                }

1046        } else if (qp->qp_serv_type == HERMON_QP_UC) {
1047                uc = &info_p->qp_transport.uc;

1049                /*
1050                 * Set the RDMA (recv) enable/disable flags.  Note: RDMA Read
1051                 * and Atomic are ignored by default.
```

```
1052                 */
1053                qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;

1055                /* Check for valid port number and fill it in */
1056                portnum = uc->uc_path.cep_hca_port_num;
1057                if (hermon_portnum_is_valid(state, portnum)) {
1058                        qp->qp_portnum = portnum - 1;
1059                        qpc->pri_addr_path.sched_q =
1060                            HERMON_QP_SCHEDQ_GET(portnum - 1,
1061                            0, qp->qp_is_special);
1062                } else {
1063                        return (IBT_HCA_PORT_INVALID);
1064                }

1066                /* Check for valid PKey index and fill it in */
1067                pkeyindx = uc->uc_path.cep_pkey_ix;
1068                if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1069                        qpc->pri_addr_path.pkey_indx = pkeyindx;
1070                } else {
1071                        return (IBT_PKEY_IX_ILLEGAL);
1072                }

1074        } else {
1075                /*
1076                 * Invalid QP transport type. If we got here then it's a
1077                 * warning of a probably serious problem.  So print a message
1078                 * and return failure
1079                 */
1080                HERMON_WARNING(state, "unknown QP transport type in rst2init");
1081                return (ibc_get_ci_failure(0));
1082        }

1084        /*
1085         * Post the RST2INIT_QP command to the Hermon firmware
1086         *
1087         * We do a HERMON_NOSLEEP here because we are still holding the
1088         * "qp_lock".  If we got raised to interrupt level by priority
1089         * inversion, we do not want to block in this routine waiting for
1090         * success.
1091         */
1092        status = hermon_cmn_qp_cmd_post(state, RST2INIT_QP, qpc, qp->qp_qpnum,
1093            0, HERMON_CMD_NOSLEEP_SPIN);
1094        if (status != HERMON_CMD_SUCCESS) {
1095                cmn_err(CE_NOTE, "hermon%d: RST2INIT_QP command failed: %08x\n",
1096                    state->hs_instance, status);
1097                if (status == HERMON_CMD_INVALID_STATUS) {
1098                        hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
1099                }
1100                return (ibc_get_ci_failure(0));
1101        }

1103        return (DDI_SUCCESS);
1104 }


1107 /*
1108  * hermon_qp_init2init()
1109  *    Context: Can be called from interrupt or base context.
1110  */
1111 static int
1112 hermon_qp_init2init(hermon_state_t *state, hermon_qphdl_t qp,
1113     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
1114 {
1115        hermon_hw_qpc_t         *qpc;
1116        ibt_qp_rc_attr_t        *rc;
1117        ibt_qp_ud_attr_t        *ud;
```



```
1118        ibt_qp_uc_attr_t        *uc;
1119        uint_t                  portnum, pkeyindx;
1120        uint32_t                opmask = 0;
1121        int                     status;

1123        ASSERT(MUTEX_HELD(&qp->qp_lock));

1125        /*
1126         * Grab the temporary QPC entry from QP software state
1127         */
1128        qpc = &qp->qpc;

1130        /*
1131         * Since there are no common fields to be filled in for this command,
1132         * we begin with the QPC fields which are specific to transport type.
1133         */
1134        if (qp->qp_type == IBT_UD_RQP) {
1135                ud = &info_p->qp_transport.ud;

1137                /*
1138                 * If we are attempting to modify the port for this QP, then
1139                 * check for valid port number and fill it in.  Also set the
1140                 * appropriate flag in the "opmask" parameter.
1141                 */
1142        /*
1143         * set port is not supported in init2init - however, in init2rtr it will
1144         * take the entire qpc, including the embedded sched_q in the path
1145         * structure - so, we can just skip setting the opmask for it explicitly
1146         * and allow it to be set later on
1147         */
1148                if (flags & IBT_CEP_SET_PORT) {
1149                        portnum = ud->ud_port;
1150                        if (hermon_portnum_is_valid(state, portnum)) {
1151                                qp->qp_portnum = portnum - 1; /* save it away */
1152                                qpc->pri_addr_path.sched_q =
1153                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
1154                                    0, qp->qp_is_special);
1155                        } else {
1156                                return (IBT_HCA_PORT_INVALID);
1157                        }
1158                }

1160                /*
1161                 * If we are attempting to modify the PKey index for this QP,
1162                 * then check for valid PKey index and fill it in.  Also set
1163                 * the appropriate flag in the "opmask" parameter.
1164                 */
1165                if (flags & IBT_CEP_SET_PKEY_IX) {
1166                        pkeyindx = ud->ud_pkey_ix;
1167                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1168                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1169                                opmask |= HERMON_CMD_OP_PKEYINDX;
1170                                qp->qp_pkeyindx = pkeyindx;
1171                        } else {
1172                                return (IBT_PKEY_IX_ILLEGAL);
1173                        }
1174                }

1176                /*
1177                 * If we are attempting to modify the QKey for this QP, then
1178                 * fill it in and set the appropriate flag in the "opmask"
1179                 * parameter.
1180                 */
1181                if (flags & IBT_CEP_SET_QKEY) {
1182                        qpc->qkey = ud->ud_qkey;
1183                        opmask |= HERMON_CMD_OP_QKEY;
```

```
1184                }

1186        } else if (qp->qp_serv_type == HERMON_QP_RC) {
1187                rc = &info_p->qp_transport.rc;

1189                /*
1190                 * If we are attempting to modify the port for this QP, then
1191                 * check for valid port number and fill it in.  Also set the
1192                 * appropriate flag in the "opmask" parameter.
1193                 */
1194                if (flags & IBT_CEP_SET_PORT) {
1195                        portnum = rc->rc_path.cep_hca_port_num;
1196                        if (hermon_portnum_is_valid(state, portnum)) {
1197                                qp->qp_portnum = portnum - 1;
1198                                qpc->pri_addr_path.sched_q =
1199                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
1200                                    0, qp->qp_is_special);
1201                        } else {
1202                                return (IBT_HCA_PORT_INVALID);
1203                        }

1205                }

1207                /*
1208                 * If we are attempting to modify the PKey index for this QP,
1209                 * then check for valid PKey index and fill it in.  Also set
1210                 * the appropriate flag in the "opmask" parameter.
1211                 */
1212                if (flags & IBT_CEP_SET_PKEY_IX) {
1213                        pkeyindx = rc->rc_path.cep_pkey_ix;
1214                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1215                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1216                                opmask |= HERMON_CMD_OP_PKEYINDX;
1217                        } else {
1218                                return (IBT_PKEY_IX_ILLEGAL);
1219                        }
1220                }

1222                /*
1223                 * Check if any of the flags indicate a change in the RDMA
1224                 * (recv) enable/disable flags and set the appropriate flag in
1225                 * the "opmask" parameter
1226                 */
1227                opmask |= hermon_check_rdma_enable_flags(flags, info_p, qpc);

1229        } else if (qp->qp_serv_type == HERMON_QP_UC) {
1230                uc = &info_p->qp_transport.uc;

1232                /*
1233                 * If we are attempting to modify the port for this QP, then
1234                 * check for valid port number and fill it in.  Also set the
1235                 * appropriate flag in the "opmask" parameter.
1236                 */
1237                if (flags & IBT_CEP_SET_PORT) {
1238                        portnum = uc->uc_path.cep_hca_port_num;
1239                        if (hermon_portnum_is_valid(state, portnum)) {
1240                                qp->qp_portnum = portnum - 1;
1241                                qpc->pri_addr_path.sched_q =
1242                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
1243                                    0, qp->qp_is_special);
1244                        } else {
1245                                return (IBT_HCA_PORT_INVALID);
1246                        }
1247                /* port# cannot be set in this transition - defer to init2rtr */
1248                }
```



```
1250                /*
1251                 * If we are attempting to modify the PKey index for this QP,
1252                 * then check for valid PKey index and fill it in.  Also set
1253                 * the appropriate flag in the "opmask" parameter.
1254                 */
1255                if (flags & IBT_CEP_SET_PKEY_IX) {
1256                        pkeyindx = uc->uc_path.cep_pkey_ix;
1257                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1258                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1259                                opmask |= HERMON_CMD_OP_PKEYINDX;
1260                        } else {
1261                                return (IBT_PKEY_IX_ILLEGAL);
1262                        }
1263                }

1265                /*
1266                 * Check if any of the flags indicate a change in the RDMA
1267                 * Write (recv) enable/disable and set the appropriate flag
1268                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
1269                 * not valid for UC transport.
1270                 */
1271                if (flags & IBT_CEP_SET_RDMA_W) {
1272                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1273                        opmask |= HERMON_CMD_OP_RWE;
1274                }
1275        } else {
1276                /*
1277                 * Invalid QP transport type. If we got here then it's a
1278                 * warning of a probably serious problem.  So print a message
1279                 * and return failure
1280                 */
1281                HERMON_WARNING(state, "unknown QP transport type in init2init");
1282                return (ibc_get_ci_failure(0));
1283        }

1285        /*
1286         * Post the INIT2INIT_QP command to the Hermon firmware
1287         *
1288         * We do a HERMON_NOSLEEP here because we are still holding the
1289         * "qp_lock".  If we got raised to interrupt level by priority
1290         * inversion, we do not want to block in this routine waiting for
1291         * success.
1292         */
1293        status = hermon_cmn_qp_cmd_post(state, INIT2INIT_QP, qpc, qp->qp_qpnum,
1294            opmask, HERMON_CMD_NOSLEEP_SPIN);
1295        if (status != HERMON_CMD_SUCCESS) {
1296                if (status != HERMON_CMD_BAD_QP_STATE) {
1297                        cmn_err(CE_NOTE, "hermon%d: INIT2INIT_QP command "
1298                            "failed: %08x\n", state->hs_instance, status);
1299                        if (status == HERMON_CMD_INVALID_STATUS) {
1300                                hermon_fm_ereport(state, HCA_SYS_ERR,
1301                                    HCA_ERR_SRV_LOST);
1302                        }
1303                        return (ibc_get_ci_failure(0));
1304                } else {
1305                        return (IBT_QP_STATE_INVALID);
1306                }
1307        }

1309        return (DDI_SUCCESS);
1310 }


1313 /*
1314  * hermon_qp_init2rtr()
1315  *    Context: Can be called from interrupt or base context.
```

```
 */
static int
hermon_qp_init2rtr(hermon_state_t *state, hermon_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
{
        hermon_hw_qpc_t         *qpc;
        ibt_qp_rc_attr_t        *rc;
        ibt_qp_ud_attr_t        *ud;
        ibt_qp_uc_attr_t        *uc;
        hermon_hw_addr_path_t   *qpc_path;
        ibt_adds_vect_t         *adds_vect;
        uint_t                  portnum, pkeyindx, rra_max;
        uint_t                  mtu;
        uint32_t                opmask = 0;
        int                     status;

        ASSERT(MUTEX_HELD(&qp->qp_lock));

        /*
         * Grab the temporary QPC entry from QP software state
         */
        qpc = &qp->qpc;

        /*
         * Since there are few common fields to be filled in for this command,
         * we just do the QPC fields that are specific to transport type.
         */
        if (qp->qp_type == IBT_UD_RQP) {
                ud = &info_p->qp_transport.ud;

                /*
                 * If this UD QP is also a "special QP" (QP0 or QP1), then
                 * the MTU is 256 bytes.  However, Hermon checks the QPC
                 * MTU settings rather than just the port MTU, so we will
                 * set it to maximum size for all UD.
                 */
                qpc->mtu = HERMON_MAX_MTU;
                if (qp->qp_uses_lso)
                        qpc->msg_max = state->hs_devlim.log_max_gso_sz;
                else
                        qpc->msg_max = HERMON_QP_LOG_MAX_MSGSZ;

                /*
                 * Save away the MTU value.  This is used in future sqd2sqd
                 * transitions, as the MTU must remain the same in future
                 * changes.
                 */
                qp->qp_save_mtu = qpc->mtu;

                /*
                 * If we are attempting to modify the PKey index for this QP,
                 * then check for valid PKey index and fill it in.  Also set
                 * the appropriate flag in the "opmask" parameter.
                 */
                if (flags & IBT_CEP_SET_PKEY_IX) {
                        pkeyindx = ud->ud_pkey_ix;
                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
                                qpc->pri_addr_path.pkey_indx = pkeyindx;
                                opmask |= HERMON_CMD_OP_PKEYINDX;
                                qp->qp_pkeyindx = pkeyindx;
                        } else {
                                return (IBT_PKEY_IX_ILLEGAL);
                        }
                }

                /*
```



```
                 * If we are attempting to modify the QKey for this QP, then
                 * fill it in and set the appropriate flag in the "opmask"
                 * parameter.
                 */
                if (flags & IBT_CEP_SET_QKEY) {
                        qpc->qkey = ud->ud_qkey;
                        opmask |= HERMON_CMD_OP_QKEY;
                }

        } else if (qp->qp_serv_type == HERMON_QP_RC) {
                rc = &info_p->qp_transport.rc;
                qpc_path = &qpc->pri_addr_path;
                adds_vect = &rc->rc_path.cep_adds_vect;

                /*
                 * Set the common primary address path fields
                 */
                status = hermon_set_addr_path(state, adds_vect, qpc_path,
                    HERMON_ADDRPATH_QP);
                if (status != DDI_SUCCESS) {
                        return (status);
                }
                /* set the primary port number/sched_q */
                portnum = qp->qp_portnum + 1;
                if (hermon_portnum_is_valid(state, portnum)) {
                        qpc->pri_addr_path.sched_q  =
                            HERMON_QP_SCHEDQ_GET(qp->qp_portnum,
                            adds_vect->av_srvl, qp->qp_is_special);
                } else {
                        return (IBT_HCA_PORT_INVALID);
                }

                /*
                 * The following values are apparently "required" here (as
                 * they are part of the IBA-defined "Remote Node Address
                 * Vector").  However, they are also going to be "required"
                 * later - at RTR2RTS_QP time.  Not sure why.  But we set
                 * them here anyway.
                 */
                qpc->rnr_retry          = rc->rc_rnr_retry_cnt;
                qpc->retry_cnt          = rc->rc_retry_cnt;
                qpc_path->ack_timeout   = rc->rc_path.cep_timeout;

                /*
                 * Setup the destination QP, recv PSN, MTU, max msg size,etc.
                 * Note max message size is defined to be the maximum IB
                 * allowed message size (which is 2^31 bytes).  Also max
                 * MTU is defined by HCA port properties.
                 */
                qpc->rem_qpn      = rc->rc_dst_qpn;
                qpc->next_rcv_psn = rc->rc_rq_psn;
                qpc->msg_max      = HERMON_QP_LOG_MAX_MSGSZ;
                qpc->ric          = 0;
                mtu               = rc->rc_path_mtu;

                if (hermon_qp_validate_mtu(state, mtu) != DDI_SUCCESS) {
                        return (IBT_HCA_PORT_MTU_EXCEEDED);
                }
                qpc->mtu = mtu;

                /*
                 * Save away the MTU value.  This is used in future sqd2sqd
                 * transitions, as the MTU must remain the same in future
                 * changes.
                 */
                qp->qp_save_mtu = qpc->mtu;
```

```
1449                 /*
1450                  * Though it is a "required" parameter, "min_rnr_nak" is
1451                  * optionally specifiable in Hermon.  So we force the
1452                  * optional flag here.
1453                  */
1454                 qpc->min_rnr_nak = rc->rc_min_rnr_nak;
1455                 opmask |= HERMON_CMD_OP_MINRNRNAK;

1457                 /*
1458                  * Check that the number of specified "incoming RDMA resources"
1459                  * is valid.  And if it is, then setup the "rra_max
1460                  */
1461                 if (hermon_qp_validate_resp_rsrc(state, rc, &rra_max) !=
1462                     DDI_SUCCESS) {
1463                         return (IBT_INVALID_PARAM);
1464                 }
1465                 qpc->rra_max = rra_max;

1467                 /* don't need to set up ra_buff_indx, implicit for hermon */

1469                 /*
1470                  * If we are attempting to modify the PKey index for this QP,
1471                  * then check for valid PKey index and fill it in.  Also set
1472                  * the appropriate flag in the "opmask" parameter.
1473                  */
1474                 if (flags & IBT_CEP_SET_PKEY_IX) {
1475                         pkeyindx = rc->rc_path.cep_pkey_ix;
1476                         if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1477                                 qpc->pri_addr_path.pkey_indx = pkeyindx;
1478                                 opmask |= HERMON_CMD_OP_PKEYINDX;
1479                         } else {
1480                                 return (IBT_PKEY_IX_ILLEGAL);
1481                         }
1482                 }

1484                 /*
1485                  * Check if any of the flags indicate a change in the RDMA
1486                  * (recv) enable/disable flags and set the appropriate flag in
1487                  * the "opmask" parameter
1488                  */
1489                 opmask |= hermon_check_rdma_enable_flags(flags, info_p, qpc);

1491                 /*
1492                  * Check for optional alternate path and fill in the
1493                  * appropriate QPC fields if one is specified
1494                  */
1495                 if (flags & IBT_CEP_SET_ALT_PATH) {
1496                         qpc_path = &qpc->alt_addr_path;
1497                         adds_vect = &rc->rc_alt_path.cep_adds_vect;

1499                         /* Set the common alternate address path fields */
1500                         status = hermon_set_addr_path(state, adds_vect,
1501                             qpc_path, HERMON_ADDRPATH_QP);
1502                         if (status != DDI_SUCCESS) {
1503                                 return (status);
1504                         }
1505                         qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;


1508                         /*
1509                          * Check for valid alternate path port number and fill
1510                          * it in
1511                          */
1512                         portnum = rc->rc_alt_path.cep_hca_port_num;
1513                         if (hermon_portnum_is_valid(state, portnum)) {
```



```
1514                                 qp->qp_portnum_alt = portnum - 1;
1515                                 qpc->alt_addr_path.sched_q =
1516                                     HERMON_QP_SCHEDQ_GET(portnum - 1,
1517                                     adds_vect->av_srvl, qp->qp_is_special);
1518                         } else {
1519                                 return (IBT_HCA_PORT_INVALID);
1520                         }
1521                         /*
1522                          * Check for valid alternate path PKey index and fill
1523                          * it in
1524                          */
1525                         pkeyindx = rc->rc_alt_path.cep_pkey_ix;
1526                         if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1527                                 qpc->alt_addr_path.pkey_indx = pkeyindx;
1528                         } else {
1529                                 return (IBT_PKEY_IX_ILLEGAL);
1530                         }
1531                         opmask |= HERMON_CMD_OP_ALT_PATH;
1532                 }

1534         } else if (qp->qp_serv_type == HERMON_QP_UC) {
1535                 uc = &info_p->qp_transport.uc;
1536                 qpc_path = &qpc->pri_addr_path;
1537                 adds_vect = &uc->uc_path.cep_adds_vect;

1539                 /*
1540                  * Set the common primary address path fields
1541                  */
1542                 status = hermon_set_addr_path(state, adds_vect, qpc_path,
1543                     HERMON_ADDRPATH_QP);
1544                 if (status != DDI_SUCCESS) {
1545                         return (status);
1546                 }

1548                 /* set the primary port num/schedq */
1549                 portnum = qp->qp_portnum + 1;
1550                 if (hermon_portnum_is_valid(state, portnum)) {
1551                         qpc->pri_addr_path.sched_q  =
1552                             HERMON_QP_SCHEDQ_GET(qp->qp_portnum,
1553                             adds_vect->av_srvl, qp->qp_is_special);
1554                 } else {
1555                         return (IBT_HCA_PORT_INVALID);
1556                 }

1558                 /*
1559                  * Setup the destination QP, recv PSN, MTU, max msg size,etc.
1560                  * Note max message size is defined to be the maximum IB
1561                  * allowed message size (which is 2^31 bytes).  Also max
1562                  * MTU is defined by HCA port properties.
1563                  */
1564                 qpc->rem_qpn      = uc->uc_dst_qpn;
1565                 qpc->next_rcv_psn = uc->uc_rq_psn;
1566                 qpc->msg_max      = HERMON_QP_LOG_MAX_MSGSZ;
1567                 mtu = uc->uc_path_mtu;
1568                 if (hermon_qp_validate_mtu(state, mtu) != DDI_SUCCESS) {
1569                         return (IBT_HCA_PORT_MTU_EXCEEDED);
1570                 }
1571                 qpc->mtu = mtu;

1573                 /*
1574                  * Save away the MTU value.  This is used in future sqd2sqd
1575                  * transitions, as the MTU must remain the same in future
1576                  * changes.
1577                  */
1578                 qp->qp_save_mtu = qpc->mtu;
```

```
1580                /*
1581                 * If we are attempting to modify the PKey index for this QP,
1582                 * then check for valid PKey index and fill it in.  Also set
1583                 * the appropriate flag in the "opmask" parameter.
1584                 */
1585                if (flags & IBT_CEP_SET_PKEY_IX) {
1586                        pkeyindx = uc->uc_path.cep_pkey_ix;
1587                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1588                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1589                                opmask |= HERMON_CMD_OP_PKEYINDX;
1590                        } else {
1591                                return (IBT_PKEY_IX_ILLEGAL);
1592                        }
1593                }

1595                /*
1596                 * Check if any of the flags indicate a change in the RDMA
1597                 * Write (recv) enable/disable and set the appropriate flag
1598                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
1599                 * not valid for UC transport.
1600                 */
1601                if (flags & IBT_CEP_SET_RDMA_W) {
1602                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1603                        opmask |= HERMON_CMD_OP_RWE;
1604                }

1606                /*
1607                 * Check for optional alternate path and fill in the
1608                 * appropriate QPC fields if one is specified
1609                 */
1610                if (flags & IBT_CEP_SET_ALT_PATH) {
1611                        qpc_path = &qpc->alt_addr_path;
1612                        adds_vect = &uc->uc_alt_path.cep_adds_vect;

1614                        /* Set the common alternate address path fields */
1615                        status = hermon_set_addr_path(state, adds_vect,
1616                            qpc_path, HERMON_ADDRPATH_QP);
1617                        if (status != DDI_SUCCESS) {
1618                                return (status);
1619                        }

1621                        qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

1623                        /*
1624                         * Check for valid alternate path port number and fill
1625                         * it in
1626                         */
1627                        portnum = uc->uc_alt_path.cep_hca_port_num;
1628                        if (hermon_portnum_is_valid(state, portnum)) {
1629                                qp->qp_portnum_alt = portnum - 1;
1630                                qpc->alt_addr_path.sched_q =
1631                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
1632                                    adds_vect->av_srvl, qp->qp_is_special);
1633                        } else {
1634                                return (IBT_HCA_PORT_INVALID);
1635                        }

1637                        /*
1638                         * Check for valid alternate path PKey index and fill
1639                         * it in
1640                         */
1641                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
1642                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1643                                qpc->alt_addr_path.pkey_indx = pkeyindx;
1644                        } else {
1645                                return (IBT_PKEY_IX_ILLEGAL);
```



```
1646                        }
1647                        opmask |= HERMON_CMD_OP_ALT_PATH;
1648                }
1649        } else {
1650                /*
1651                 * Invalid QP transport type. If we got here then it's a
1652                 * warning of a probably serious problem.  So print a message
1653                 * and return failure
1654                 */
1655                HERMON_WARNING(state, "unknown QP transport type in init2rtr");
1656                return (ibc_get_ci_failure(0));
1657        }

1659        /*
1660         * Post the INIT2RTR_QP command to the Hermon firmware
1661         *
1662         * We do a HERMON_NOSLEEP here because we are still holding the
1663         * "qp_lock".  If we got raised to interrupt level by priority
1664         * inversion, we do not want to block in this routine waiting for
1665         * success.
1666         */
1667        status = hermon_cmn_qp_cmd_post(state, INIT2RTR_QP, qpc, qp->qp_qpnum,
1668            opmask, HERMON_CMD_NOSLEEP_SPIN);
1669        if (status != HERMON_CMD_SUCCESS) {
1670                if (status != HERMON_CMD_BAD_QP_STATE) {
1671                        cmn_err(CE_NOTE, "hermon%d: INIT2RTR_QP command "
1672                            "failed: %08x\n", state->hs_instance, status);
1673                        if (status == HERMON_CMD_INVALID_STATUS) {
1674                                hermon_fm_ereport(state, HCA_SYS_ERR,
1675                                    HCA_ERR_SRV_LOST);
1676                        }
1677                        return (ibc_get_ci_failure(0));
1678                } else {
1679                        return (IBT_QP_STATE_INVALID);
1680                }
1681        }

1683        return (DDI_SUCCESS);
1684 }


1687 /*
1688  * hermon_qp_rtr2rts()
1689  *    Context: Can be called from interrupt or base context.
1690  */
1691 static int
1692 hermon_qp_rtr2rts(hermon_state_t *state, hermon_qphdl_t qp,
1693     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
1694 {
1695        hermon_hw_qpc_t         *qpc;
1696        ibt_qp_rc_attr_t        *rc;
1697        ibt_qp_ud_attr_t        *ud;
1698        ibt_qp_uc_attr_t        *uc;
1699        hermon_hw_addr_path_t   *qpc_path;
1700        ibt_adds_vect_t         *adds_vect;
1701        uint_t                  portnum, pkeyindx, sra_max;
1702        uint32_t                opmask = 0;
1703        int                     status;

1705        ASSERT(MUTEX_HELD(&qp->qp_lock));

1707        /*
1708         * Grab the temporary QPC entry from QP software state
1709         */
1710        qpc = &qp->qpc;
```

```
1712         /*
1713          * Now fill in the QPC fields which are specific to transport type
1714          */
1715         if (qp->qp_type == IBT_UD_RQP) {
1716                 ud = &info_p->qp_transport.ud;
1717
1718                 /* Set the send PSN */
1719                 qpc->next_snd_psn = ud->ud_sq_psn;
1720
1721                 /*
1722                  * If we are attempting to modify the QKey for this QP, then
1723                  * fill it in and set the appropriate flag in the "opmask"
1724                  * parameter.
1725                  */
1726                 if (flags & IBT_CEP_SET_QKEY) {
1727                         qpc->qkey = ud->ud_qkey;
1728                         opmask |= HERMON_CMD_OP_QKEY;
1729                 }
1730
1731         } else if (qp->qp_serv_type == HERMON_QP_RC) {
1732                 rc = &info_p->qp_transport.rc;
1733                 qpc_path = &qpc->pri_addr_path;
1734
1735                 /*
1736                  * Setup the send PSN, ACK timeout, and retry counts
1737                  */
1738                 qpc->next_snd_psn       = rc->rc_sq_psn;
1739                 qpc_path->ack_timeout   = rc->rc_path.cep_timeout;
1740                 qpc->rnr_retry          = rc->rc_rnr_retry_cnt;
1741                                         /* in qpc now, not path */
1742                 qpc->retry_cnt          = rc->rc_retry_cnt;
1743
1744                 /*
1745                  * Set "ack_req_freq" based on the configuration variable
1746                  */
1747                 qpc->ack_req_freq = state->hs_cfg_profile->cp_ackreq_freq;
1748
1749                 /*
1750                  * Check that the number of specified "outgoing RDMA resources"
1751                  * is valid.  And if it is, then setup the "sra_max"
1752                  * appropriately
1753                  */
1754                 if (hermon_qp_validate_init_depth(state, rc, &sra_max) !=
1755                     DDI_SUCCESS) {
1756                         return (IBT_INVALID_PARAM);
1757                 }
1758                 qpc->sra_max = sra_max;
1759
1760
1761                 /*
1762                  * Check if any of the flags indicate a change in the RDMA
1763                  * (recv) enable/disable flags and set the appropriate flag in
1764                  * the "opmask" parameter
1765                  */
1766                 opmask |= hermon_check_rdma_enable_flags(flags, info_p, qpc);
1767
1768                 /*
1769                  * If we are attempting to modify the path migration state for
1770                  * this QP, then check for valid state and fill it in.  Also
1771                  * set the appropriate flag in the "opmask" parameter.
1772                  */
1773                 if (flags & IBT_CEP_SET_MIG) {
1774                         if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
1775                                 qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
1776                         } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
1777                                 qpc->pm_state = HERMON_QP_PMSTATE_REARM;
```



```
1778                         } else {
1779                                 return (IBT_QP_APM_STATE_INVALID);
1780                         }
1781                         opmask |= HERMON_CMD_OP_PM_STATE;
1782                 }
1783
1784                 /*
1785                  * If we are attempting to modify the "Minimum RNR NAK" value
1786                  * for this QP, then fill it in and set the appropriate flag
1787                  * in the "opmask" parameter.
1788                  */
1789                 if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
1790                         qpc->min_rnr_nak = rc->rc_min_rnr_nak;
1791                         opmask |= HERMON_CMD_OP_MINRNRNAK;
1792                 }
1793
1794                 /*
1795                  * Check for optional alternate path and fill in the
1796                  * appropriate QPC fields if one is specified
1797                  */
1798                 if (flags & IBT_CEP_SET_ALT_PATH) {
1799                         qpc_path = &qpc->alt_addr_path;
1800                         adds_vect = &rc->rc_alt_path.cep_adds_vect;
1801
1802                         /* Set the common alternate address path fields */
1803                         status = hermon_set_addr_path(state, adds_vect,
1804                             qpc_path, HERMON_ADDRPATH_QP);
1805                         if (status != DDI_SUCCESS) {
1806                                 return (status);
1807                         }
1808
1809                         qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;
1810
1811                         /*
1812                          * Check for valid alternate path port number and fill
1813                          * it in
1814                          */
1815                         portnum = rc->rc_alt_path.cep_hca_port_num;
1816                         if (hermon_portnum_is_valid(state, portnum)) {
1817                                 qp->qp_portnum_alt = portnum - 1;
1818                                 qpc->alt_addr_path.sched_q =
1819                                     HERMON_QP_SCHEDQ_GET(portnum - 1,
1820                                     adds_vect->av_srvl, qp->qp_is_special);
1821                         } else {
1822                                 return (IBT_HCA_PORT_INVALID);
1823                         }
1824
1825                         /*
1826                          * Check for valid alternate path PKey index and fill
1827                          * it in
1828                          */
1829                         pkeyindx = rc->rc_alt_path.cep_pkey_ix;
1830                         if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1831                                 qpc->alt_addr_path.pkey_indx = pkeyindx;
1832                         } else {
1833                                 return (IBT_PKEY_IX_ILLEGAL);
1834                         }
1835                         opmask |= HERMON_CMD_OP_ALT_PATH;
1836                 }
1837
1838         } else if (qp->qp_serv_type == HERMON_QP_UC) {
1839                 uc = &info_p->qp_transport.uc;
1840
1841                 /* Set the send PSN */
1842                 qpc->next_snd_psn = uc->uc_sq_psn;
```

```
1844                /*
1845                 * Configure the QP to allow (sending of) all types of allowable
1846                 * UC traffic (i.e. RDMA Write).
1847                 */


1850                /*
1851                 * Check if any of the flags indicate a change in the RDMA
1852                 * Write (recv) enable/disable and set the appropriate flag
1853                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
1854                 * not valid for UC transport.
1855                 */
1856                if (flags & IBT_CEP_SET_RDMA_W) {
1857                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1858                        opmask |= HERMON_CMD_OP_RWE;
1859                }

1861                /*
1862                 * If we are attempting to modify the path migration state for
1863                 * this QP, then check for valid state and fill it in.  Also
1864                 * set the appropriate flag in the "opmask" parameter.
1865                 */
1866                if (flags & IBT_CEP_SET_MIG) {
1867                        if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
1868                                qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
1869                        } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
1870                                qpc->pm_state = HERMON_QP_PMSTATE_REARM;
1871                        } else {
1872                                return (IBT_QP_APM_STATE_INVALID);
1873                        }
1874                        opmask |= HERMON_CMD_OP_PM_STATE;
1875                }

1877                /*
1878                 * Check for optional alternate path and fill in the
1879                 * appropriate QPC fields if one is specified
1880                 */
1881                if (flags & IBT_CEP_SET_ALT_PATH) {
1882                        qpc_path = &qpc->alt_addr_path;
1883                        adds_vect = &uc->uc_alt_path.cep_adds_vect;

1885                        /* Set the common alternate address path fields */
1886                        status = hermon_set_addr_path(state, adds_vect,
1887                            qpc_path, HERMON_ADDRPATH_QP);
1888                        if (status != DDI_SUCCESS) {
1889                                return (status);
1890                        }
1891                        qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

1893                        /*
1894                         * Check for valid alternate path port number and fill
1895                         * it in
1896                         */
1897                        portnum = uc->uc_alt_path.cep_hca_port_num;
1898                        if (hermon_portnum_is_valid(state, portnum)) {
1899                                qpc->alt_addr_path.sched_q =
1900                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
1901                                    adds_vect->av_srvl, qp->qp_is_special);
1902                        } else {
1903                                return (IBT_HCA_PORT_INVALID);
1904                        }

1906                        /*
1907                         * Check for valid alternate path PKey index and fill
1908                         * it in
1909                         */
```



```
1910                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
1911                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
1912                                qpc->alt_addr_path.pkey_indx = pkeyindx;
1913                        } else {
1914                                return (IBT_PKEY_IX_ILLEGAL);
1915                        }
1916                        opmask |= HERMON_CMD_OP_ALT_PATH;
1917                }
1918        } else {
1919                /*
1920                 * Invalid QP transport type. If we got here then it's a
1921                 * warning of a probably serious problem.  So print a message
1922                 * and return failure
1923                 */
1924                HERMON_WARNING(state, "unknown QP transport type in rtr2rts");
1925                return (ibc_get_ci_failure(0));
1926        }

1928        /*
1929         * Post the RTR2RTS_QP command to the Hermon firmware
1930         *
1931         * We do a HERMON_NOSLEEP here because we are still holding the
1932         * "qp_lock".  If we got raised to interrupt level by priority
1933         * inversion, we do not want to block in this routine waiting for
1934         * success.
1935         */
1936        status = hermon_cmn_qp_cmd_post(state, RTR2RTS_QP, qpc, qp->qp_qpnum,
1937            opmask, HERMON_CMD_NOSLEEP_SPIN);
1938        if (status != HERMON_CMD_SUCCESS) {
1939                if (status != HERMON_CMD_BAD_QP_STATE) {
1940                        cmn_err(CE_NOTE, "hermon%d: RTR2RTS_QP command failed: "
1941                            "%08x\n", state->hs_instance, status);
1942                        if (status == HERMON_CMD_INVALID_STATUS) {
1943                                hermon_fm_ereport(state, HCA_SYS_ERR,
1944                                    HCA_ERR_SRV_LOST);
1945                        }
1946                        return (ibc_get_ci_failure(0));
1947                } else {
1948                        return (IBT_QP_STATE_INVALID);
1949                }
1950        }

1952        return (DDI_SUCCESS);
1953 }


1956 /*
1957  * hermon_qp_rts2rts()
1958  *    Context: Can be called from interrupt or base context.
1959  */
1960 static int
1961 hermon_qp_rts2rts(hermon_state_t *state, hermon_qphdl_t qp,
1962     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
1963 {
1964        hermon_hw_qpc_t         *qpc;
1965        ibt_qp_rc_attr_t        *rc;
1966        ibt_qp_ud_attr_t        *ud;
1967        ibt_qp_uc_attr_t        *uc;
1968        hermon_hw_addr_path_t   *qpc_path;
1969        ibt_adds_vect_t         *adds_vect;
1970        uint_t                  portnum, pkeyindx;
1971        uint32_t                opmask = 0;
1972        int                     status;

1974        ASSERT(MUTEX_HELD(&qp->qp_lock));
```

```
1976         /*
1977          * Grab the temporary QPC entry from QP software state
1978          */
1980         qpc = &qp->qpc;
1982         /*
1983          * Since there are no common fields to be filled in for this command,
1984          * we begin with the QPC fields which are specific to transport type.
1985          */
1986         if (qp->qp_type == IBT_UD_RQP) {
1987                 ud = &info_p->qp_transport.ud;
1989                 /*
1990                  * If we are attempting to modify the QKey for this QP, then
1991                  * fill it in and set the appropriate flag in the "opmask"
1992                  * parameter.
1993                  */
1994                 if (flags & IBT_CEP_SET_QKEY) {
1995                         qpc->qkey = ud->ud_qkey;
1996                         opmask |= HERMON_CMD_OP_QKEY;
1997                 }
1999         } else if (qp->qp_serv_type == HERMON_QP_RC) {
2000                 rc = &info_p->qp_transport.rc;
2002                 /*
2003                  * Check if any of the flags indicate a change in the RDMA
2004                  * (recv) enable/disable flags and set the appropriate flag in
2005                  * the "opmask" parameter
2006                  */
2007                 opmask |= hermon_check_rdma_enable_flags(flags, info_p, qpc);
2009                 /*
2010                  * If we are attempting to modify the path migration state for
2011                  * this QP, then check for valid state and fill it in.  Also
2012                  * set the appropriate flag in the "opmask" parameter.
2013                  */
2014                 if (flags & IBT_CEP_SET_MIG) {
2015                         if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
2016                                 qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
2017                         } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
2018                                 qpc->pm_state = HERMON_QP_PMSTATE_REARM;
2019                         } else {
2020                                 return (IBT_QP_APM_STATE_INVALID);
2021                         }
2022                         opmask |= HERMON_CMD_OP_PM_STATE;
2023                 }
2025                 /*
2026                  * If we are attempting to modify the "Minimum RNR NAK" value
2027                  * for this QP, then fill it in and set the appropriate flag
2028                  * in the "opmask" parameter.
2029                  */
2030                 if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
2031                         qpc->min_rnr_nak = rc->rc_min_rnr_nak;
2032                         opmask |= HERMON_CMD_OP_MINRNRNAK;
2033                 }
2035                 /*
2036                  * Check for optional alternate path and fill in the
2037                  * appropriate QPC fields if one is specified
2038                  */
2039                 if (flags & IBT_CEP_SET_ALT_PATH) {
2040                         qpc_path = &qpc->alt_addr_path;
2041                         adds_vect = &rc->rc_alt_path.cep_adds_vect;
```



```
2043                         /* Set the common alternate address path fields */
2044                         status = hermon_set_addr_path(state, adds_vect,
2045                             qpc_path, HERMON_ADDRPATH_QP);
2046                         if (status != DDI_SUCCESS) {
2047                                 return (status);
2048                         }
2049                         qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;
2051                         /*
2052                          * Check for valid alternate path port number and fill
2053                          * it in
2054                          */
2055                         portnum = rc->rc_alt_path.cep_hca_port_num;
2056                         if (hermon_portnum_is_valid(state, portnum)) {
2057                                 qp->qp_portnum_alt = portnum - 1;
2058                                 qpc->alt_addr_path.sched_q =
2059                                     HERMON_QP_SCHEDQ_GET(portnum - 1,
2060                                     adds_vect->av_srvl, qp->qp_is_special);
2061                         } else {
2062                                 return (IBT_HCA_PORT_INVALID);
2063                         }
2065                         /*
2066                          * Check for valid alternate path PKey index and fill
2067                          * it in
2068                          */
2069                         pkeyindx = rc->rc_alt_path.cep_pkey_ix;
2070                         if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2071                                 qpc->alt_addr_path.pkey_indx = pkeyindx;
2072                         } else {
2073                                 return (IBT_PKEY_IX_ILLEGAL);
2074                         }
2075                         opmask |= HERMON_CMD_OP_ALT_PATH;
2076                 }
2078         } else if (qp->qp_serv_type == HERMON_QP_UC) {
2079                 uc = &info_p->qp_transport.uc;
2081                 /*
2082                  * Check if any of the flags indicate a change in the RDMA
2083                  * Write (recv) enable/disable and set the appropriate flag
2084                  * in the "opmask" parameter. Note: RDMA Read and Atomic are
2085                  * not valid for UC transport.
2086                  */
2087                 if (flags & IBT_CEP_SET_RDMA_W) {
2088                         qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
2089                         opmask |= HERMON_CMD_OP_RWE;
2090                 }
2092                 /*
2093                  * If we are attempting to modify the path migration state for
2094                  * this QP, then check for valid state and fill it in.  Also
2095                  * set the appropriate flag in the "opmask" parameter.
2096                  */
2097                 if (flags & IBT_CEP_SET_MIG) {
2098                         if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
2099                                 qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
2100                         } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
2101                                 qpc->pm_state = HERMON_QP_PMSTATE_REARM;
2102                         } else {
2103                                 return (IBT_QP_APM_STATE_INVALID);
2104                         }
2105                         opmask |= HERMON_CMD_OP_PM_STATE;
2106                 }
```

```
2108                /*
2109                 * Check for optional alternate path and fill in the
2110                 * appropriate QPC fields if one is specified
2111                 */
2112                if (flags & IBT_CEP_SET_ALT_PATH) {
2113                        qpc_path = &qpc->alt_addr_path;
2114                        adds_vect = &uc->uc_alt_path.cep_adds_vect;

2116                        /* Set the common alternate address path fields */
2117                        status = hermon_set_addr_path(state, adds_vect,
2118                            qpc_path, HERMON_ADDRPATH_QP);
2119                        if (status != DDI_SUCCESS) {
2120                                return (status);
2121                        }

2123                        /*
2124                         * Check for valid alternate path port number and fill
2125                         * it in
2126                         */
2127                        portnum = uc->uc_alt_path.cep_hca_port_num;
2128                        if (hermon_portnum_is_valid(state, portnum)) {
2129                                qp->qp_portnum_alt = portnum - 1;
2130                                qpc->alt_addr_path.sched_q =
2131                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
2132                                    adds_vect->av_srvl, qp->qp_is_special);
2133                        } else {
2134                                return (IBT_HCA_PORT_INVALID);
2135                        }

2137                        /*
2138                         * Check for valid alternate path PKey index and fill
2139                         * it in
2140                         */
2141                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
2142                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2143                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2144                        } else {
2145                                return (IBT_PKEY_IX_ILLEGAL);
2146                        }
2147                        opmask |= HERMON_CMD_OP_ALT_PATH;
2148                }
2149        } else {
2150                /*
2151                 * Invalid QP transport type. If we got here then it's a
2152                 * warning of a probably serious problem.  So print a message
2153                 * and return failure
2154                 */
2155                HERMON_WARNING(state, "unknown QP transport type in rts2rts");
2156                return (ibc_get_ci_failure(0));
2157        }

2159        /*
2160         * Post the RTS2RTS_QP command to the Hermon firmware
2161         *
2162         * We do a HERMON_NOSLEEP here because we are still holding the
2163         * "qp_lock".  If we got raised to interrupt level by priority
2164         * inversion, we do not want to block in this routine waiting for
2165         * success.
2166         */
2167        status = hermon_cmn_qp_cmd_post(state, RTS2RTS_QP, qpc, qp->qp_qpnum,
2168            opmask, HERMON_CMD_NOSLEEP_SPIN);
2169        if (status != HERMON_CMD_SUCCESS) {
2170                if (status != HERMON_CMD_BAD_QP_STATE) {
2171                        cmn_err(CE_NOTE, "hermon%d: RTS2RTS_QP command failed: "
2172                            "%08x\n", state->hs_instance, status);
2173                        if (status == HERMON_CMD_INVALID_STATUS) {
```



```
2174                                hermon_fm_ereport(state, HCA_SYS_ERR,
2175                                    HCA_ERR_SRV_LOST);
2176                        }
2177                        return (ibc_get_ci_failure(0));
2178                } else {
2179                        return (IBT_QP_STATE_INVALID);
2180                }
2181        }

2183        return (DDI_SUCCESS);
2184 }


2187 #ifdef HERMON_NOTNOW
2188 /*
2189  * hermon_qp_rts2sqd()
2190  *    Context: Can be called from interrupt or base context.
2191  */
2192 static int
2193 hermon_qp_rts2sqd(hermon_state_t *state, hermon_qphdl_t qp,
2194     ibt_cep_modify_flags_t flags)
2195 {
2196        int                     status;

2198        ASSERT(MUTEX_HELD(&qp->qp_lock));

2200        /*
2201         * Set a flag to indicate whether or not the consumer is interested
2202         * in receiving the SQ drained event.  Since we are going to always
2203         * request hardware generation of the SQD event, we use the value in
2204         * "qp_forward_sqd_event" to determine whether or not to pass the event
2205         * to the IBTF or to silently consume it.
2206         */
2207        qp->qp_forward_sqd_event = (flags & IBT_CEP_SET_SQD_EVENT) ? 1 : 0;

2209        /*
2210         * Post the RTS2SQD_QP command to the Hermon firmware
2211         *
2212         * We do a HERMON_NOSLEEP here because we are still holding the
2213         * "qp_lock".  If we got raised to interrupt level by priority
2214         * inversion, we do not want to block in this routine waiting for
2215         * success.
2216         */
2217        status = hermon_cmn_qp_cmd_post(state, RTS2SQD_QP, NULL, qp->qp_qpnum,
2218            0, HERMON_CMD_NOSLEEP_SPIN);
2219        if (status != HERMON_CMD_SUCCESS) {
2220                if (status != HERMON_CMD_BAD_QP_STATE) {
2221                        cmn_err(CE_NOTE, "hermon%d: RTS2SQD_QP command failed: "
2222                            "%08x\n", state->hs_instance, status);
2223                        if (status == HERMON_CMD_INVALID_STATUS) {
2224                                hermon_fm_ereport(state, HCA_SYS_ERR,
2225                                    HCA_ERR_SRV_LOST);
2226                        }
2227                        return (ibc_get_ci_failure(0));
2228                } else {
2229                        return (IBT_QP_STATE_INVALID);
2230                }
2231        }

2233        /*
2234         * Mark the current QP state as "SQ Draining".  This allows us to
2235         * distinguish between the two underlying states in SQD. (see QueryQP()
2236         * code in hermon_qp.c)
2237         */
2238        qp->qp_sqd_still_draining = 1;
```

```
2240         return (DDI_SUCCESS);
2241 }
2242 #endif


2245 /*
2246  * hermon_qp_sqd2rts()
2247  *    Context: Can be called from interrupt or base context.
2248  */
2249 static int
2250 hermon_qp_sqd2rts(hermon_state_t *state, hermon_qphdl_t qp,
2251     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
2252 {
2253         hermon_hw_qpc_t         *qpc;
2254         ibt_qp_rc_attr_t        *rc;
2255         ibt_qp_ud_attr_t        *ud;
2256         ibt_qp_uc_attr_t        *uc;
2257         hermon_hw_addr_path_t   *qpc_path;
2258         ibt_adds_vect_t         *adds_vect;
2259         uint_t                  portnum, pkeyindx;
2260         uint_t                  rra_max, sra_max;
2261         uint32_t                opmask = 0;
2262         int                     status;

2264         ASSERT(MUTEX_HELD(&qp->qp_lock));

2266         /*
2267          * Grab the temporary QPC entry from QP software state
2268          */
2269         qpc = &qp->qpc;

2271         /*
2272          * Fill in the common fields in the QPC
2273          */

2275         /*
2276          * Now fill in the QPC fields which are specific to transport type
2277          */
2278         if (qp->qp_type == IBT_UD_RQP) {
2279                 ud = &info_p->qp_transport.ud;

2281                 /*
2282                  * If we are attempting to modify the port for this QP, then
2283                  * check for valid port number and fill it in.  Also set the
2284                  * appropriate flag in the "opmask" parameter.
2285                  */
2286                 if (flags & IBT_CEP_SET_PORT) {
2287                         portnum = ud->ud_port;
2288                         if (hermon_portnum_is_valid(state, portnum)) {
2289                                 qp->qp_portnum = portnum - 1;
2290                                 qpc->pri_addr_path.sched_q =
2291                                     HERMON_QP_SCHEDQ_GET(portnum - 1,
2292                                     0, qp->qp_is_special);
2293                         } else {
2294                                 return (IBT_HCA_PORT_INVALID);
2295                         }
2296                         opmask |= HERMON_CMD_OP_PRIM_PORT;
2297                 }

2299                 /*
2300                  * If we are attempting to modify the PKey index for this QP,
2301                  * then check for valid PKey index and fill it in.  Also set
2302                  * the appropriate flag in the "opmask" parameter.
2303                  */
2304                 if (flags & IBT_CEP_SET_PKEY_IX) {
2305                         pkeyindx = ud->ud_pkey_ix;
```



```
2306                         if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2307                                 qpc->pri_addr_path.pkey_indx = pkeyindx;
2308                                 opmask |= HERMON_CMD_OP_PKEYINDX;
2309                                 qp->qp_pkeyindx = pkeyindx;
2310                         } else {
2311                                 return (IBT_PKEY_IX_ILLEGAL);
2312                         }
2313                 }

2315                 /*
2316                  * If we are attempting to modify the QKey for this QP, then
2317                  * fill it in and set the appropriate flag in the "opmask"
2318                  * parameter.
2319                  */
2320                 if (flags & IBT_CEP_SET_QKEY) {
2321                         qpc->qkey = ud->ud_qkey;
2322                         opmask |= HERMON_CMD_OP_QKEY;
2323                 }

2325         } else if (qp->qp_serv_type == HERMON_QP_RC) {
2326                 rc = &info_p->qp_transport.rc;

2328                 /*
2329                  * Check if any of the flags indicate a change in the RDMA
2330                  * (recv) enable/disable flags and set the appropriate flag in
2331                  * the "opmask" parameter
2332                  */
2333                 opmask |= hermon_check_rdma_enable_flags(flags, info_p, qpc);

2335                 qpc->retry_cnt = rc->rc_retry_cnt;

2337                 /*
2338                  * If we are attempting to modify the path migration state for
2339                  * this QP, then check for valid state and fill it in.  Also
2340                  * set the appropriate flag in the "opmask" parameter.
2341                  */
2342                 if (flags & IBT_CEP_SET_MIG) {
2343                         if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
2344                                 qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
2345                         } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
2346                                 qpc->pm_state = HERMON_QP_PMSTATE_REARM;
2347                         } else {
2348                                 return (IBT_QP_APM_STATE_INVALID);
2349                         }
2350                         opmask |= HERMON_CMD_OP_PM_STATE;
2351                 }

2353                 /*
2354                  * Check for optional alternate path and fill in the
2355                  * appropriate QPC fields if one is specified
2356                  */
2357                 if (flags & IBT_CEP_SET_ALT_PATH) {
2358                         qpc_path = &qpc->alt_addr_path;
2359                         adds_vect = &rc->rc_alt_path.cep_adds_vect;

2361                         /* Set the common alternate address path fields */
2362                         status = hermon_set_addr_path(state, adds_vect,
2363                             qpc_path, HERMON_ADDRPATH_QP);
2364                         if (status != DDI_SUCCESS) {
2365                                 return (status);
2366                         }
2367                         qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;
2368                         /*
2369                          * Check for valid alternate path port number and fill
2370                          * it in
2371                          */
```

```
2372                        portnum = rc->rc_alt_path.cep_hca_port_num;
2373                        if (hermon_portnum_is_valid(state, portnum)) {
2374                                qp->qp_portnum_alt = portnum - 1;
2375                                qpc->alt_addr_path.sched_q =
2376                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
2377                                    adds_vect->av_srvl, qp->qp_is_special);
2378                        } else {
2379                                return (IBT_HCA_PORT_INVALID);
2380                        }

2382                        /*
2383                         * Check for valid alternate path PKey index and fill
2384                         * it in
2385                         */
2386                        pkeyindx = rc->rc_alt_path.cep_pkey_ix;
2387                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2388                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2389                        } else {
2390                                return (IBT_PKEY_IX_ILLEGAL);
2391                        }
2392                        opmask |= HERMON_CMD_OP_ALT_PATH;
2393                }

2395                /*
2396                 * If we are attempting to modify the number of "outgoing
2397                 * RDMA resources" for this QP, then check for valid value and
2398                 * fill it in.  Also set the appropriate flag in the "opmask"
2399                 * parameter.
2400                 */
2401                if (flags & IBT_CEP_SET_RDMARA_OUT) {
2402                        if (hermon_qp_validate_init_depth(state, rc,
2403                            &sra_max) != DDI_SUCCESS) {
2404                                return (IBT_INVALID_PARAM);
2405                        }
2406                        qpc->sra_max = sra_max;
2407                        opmask |= HERMON_CMD_OP_SRA_SET;
2408                }

2410                /*
2411                 * If we are attempting to modify the number of "incoming
2412                 * RDMA resources" for this QP, then check for valid value and
2413                 * update the "rra_max" and "ra_buf_index" fields in the QPC to
2414                 * point to the pre-allocated RDB resources (in DDR).  Also set
2415                 * the appropriate flag in the "opmask" parameter.
2416                 */
2417                if (flags & IBT_CEP_SET_RDMARA_IN) {
2418                        if (hermon_qp_validate_resp_rsrc(state, rc,
2419                            &rra_max) != DDI_SUCCESS) {
2420                                return (IBT_INVALID_PARAM);
2421                        }
2422                        qpc->rra_max = rra_max;
2423                        opmask |= HERMON_CMD_OP_RRA_SET;
2424                }


2427                /*
2428                 * If we are attempting to modify the "Minimum RNR NAK" value
2429                 * for this QP, then fill it in and set the appropriate flag
2430                 * in the "opmask" parameter.
2431                 */
2432                if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
2433                        qpc->min_rnr_nak = rc->rc_min_rnr_nak;
2434                        opmask |= HERMON_CMD_OP_MINRNRNAK;
2435                }

2437        } else if (qp->qp_serv_type == HERMON_QP_UC) {
```



```
2438                uc = &info_p->qp_transport.uc;

2440                /*
2441                 * Check if any of the flags indicate a change in the RDMA
2442                 * Write (recv) enable/disable and set the appropriate flag
2443                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
2444                 * not valid for UC transport.
2445                 */
2446                if (flags & IBT_CEP_SET_RDMA_W) {
2447                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
2448                        opmask |= HERMON_CMD_OP_RWE;
2449                }

2451                /*
2452                 * If we are attempting to modify the path migration state for
2453                 * this QP, then check for valid state and fill it in.  Also
2454                 * set the appropriate flag in the "opmask" parameter.
2455                 */
2456                if (flags & IBT_CEP_SET_MIG) {
2457                        if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
2458                                qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
2459                        } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
2460                                qpc->pm_state = HERMON_QP_PMSTATE_REARM;
2461                        } else {
2462                                return (IBT_QP_APM_STATE_INVALID);
2463                        }
2464                        opmask |= HERMON_CMD_OP_PM_STATE;
2465                }

2467                /*
2468                 * Check for optional alternate path and fill in the
2469                 * appropriate QPC fields if one is specified
2470                 */
2471                if (flags & IBT_CEP_SET_ALT_PATH) {
2472                        qpc_path = &qpc->alt_addr_path;
2473                        adds_vect = &uc->uc_alt_path.cep_adds_vect;

2475                        /* Set the common alternate address path fields */
2476                        status = hermon_set_addr_path(state, adds_vect,
2477                            qpc_path, HERMON_ADDRPATH_QP);
2478                        if (status != DDI_SUCCESS) {
2479                                return (status);
2480                        }

2482                        /*
2483                         * Check for valid alternate path port number and fill
2484                         * it in
2485                         */
2486                        portnum = uc->uc_alt_path.cep_hca_port_num;
2487                        if (hermon_portnum_is_valid(state, portnum)) {
2488                                qp->qp_portnum_alt = portnum - 1;
2489                                qpc->alt_addr_path.sched_q =
2490                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
2491                                    adds_vect->av_srvl, qp->qp_is_special);
2492                        } else {
2493                                return (IBT_HCA_PORT_INVALID);
2494                        }

2496                        /*
2497                         * Check for valid alternate path PKey index and fill
2498                         * it in
2499                         */
2500                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
2501                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2502                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2503                        } else {
```

```
2504                                return (IBT_PKEY_IX_ILLEGAL);
2505                        }
2506                        opmask |= HERMON_CMD_OP_ALT_PATH;
2507                }
2508        } else {
2509                /*
2510                 * Invalid QP transport type. If we got here then it's a
2511                 * warning of a probably serious problem.  So print a message
2512                 * and return failure
2513                 */
2514                HERMON_WARNING(state, "unknown QP transport type in sqd2rts");
2515                return (ibc_get_ci_failure(0));
2516        }

2518        /*
2519         * Post the SQD2RTS_QP command to the Hermon firmware
2520         *
2521         * We do a HERMON_NOSLEEP here because we are still holding the
2522         * "qp_lock".  If we got raised to interrupt level by priority
2523         * inversion, we do not want to block in this routine waiting for
2524         * success.
2525         */
2526        status = hermon_cmn_qp_cmd_post(state, SQD2RTS_QP, qpc, qp->qp_qpnum,
2527            opmask, HERMON_CMD_NOSLEEP_SPIN);
2528        if (status != HERMON_CMD_SUCCESS) {
2529                if (status != HERMON_CMD_BAD_QP_STATE) {
2530                        cmn_err(CE_NOTE, "hermon%d: SQD2RTS_QP command failed: "
2531                            "%08x\n", state->hs_instance, status);
2532                        if (status == HERMON_CMD_INVALID_STATUS) {
2533                                hermon_fm_ereport(state, HCA_SYS_ERR,
2534                                    HCA_ERR_SRV_LOST);
2535                        }
2536                        return (ibc_get_ci_failure(0));
2537                } else {
2538                        return (IBT_QP_STATE_INVALID);
2539                }
2540        }

2542        return (DDI_SUCCESS);
2543 }


2546 /*
2547  * hermon_qp_sqd2sqd()
2548  *    Context: Can be called from interrupt or base context.
2549  */
2550 static int
2551 hermon_qp_sqd2sqd(hermon_state_t *state, hermon_qphdl_t qp,
2552     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
2553 {
2554        hermon_hw_qpc_t         *qpc;
2555        ibt_qp_rc_attr_t        *rc;
2556        ibt_qp_ud_attr_t        *ud;
2557        ibt_qp_uc_attr_t        *uc;
2558        hermon_hw_addr_path_t   *qpc_path;
2559        ibt_adds_vect_t         *adds_vect;
2560        uint_t                  portnum, pkeyindx;
2561        uint_t                  rra_max, sra_max;
2562        uint32_t                opmask = 0;
2563        int                     status;

2565        ASSERT(MUTEX_HELD(&qp->qp_lock));

2567        /*
2568         * Grab the temporary QPC entry from QP software state
2569         */
```



```
2570        qpc = &qp->qpc;

2572        /*
2573         * Fill in the common fields in the QPC
2574         */

2576        /*
2577         * Now fill in the QPC fields which are specific to transport type
2578         */
2579        if (qp->qp_type == IBT_UD_RQP) {
2580                ud = &info_p->qp_transport.ud;

2582                /*
2583                 * If we are attempting to modify the port for this QP, then
2584                 * check for valid port number and fill it in.  Also set the
2585                 * appropriate flag in the "opmask" parameter.
2586                 */
2587                if (flags & IBT_CEP_SET_PORT) {
2588                        portnum = ud->ud_port;
2589                        if (hermon_portnum_is_valid(state, portnum)) {
2590                                qp->qp_portnum = portnum - 1;
2591                                qpc->pri_addr_path.sched_q =
2592                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
2593                                    0, qp->qp_is_special);
2594                        } else {
2595                                return (IBT_HCA_PORT_INVALID);
2596                        }
2597                        opmask |= HERMON_CMD_OP_SCHEDQUEUE;
2598                }

2600                /*
2601                 * If we are attempting to modify the PKey index for this QP,
2602                 * then check for valid PKey index and fill it in.  Also set
2603                 * the appropriate flag in the "opmask" parameter.
2604                 */
2605                if (flags & IBT_CEP_SET_PKEY_IX) {
2606                        pkeyindx = ud->ud_pkey_ix;
2607                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2608                                qpc->pri_addr_path.pkey_indx = pkeyindx;
2609                                opmask |= HERMON_CMD_OP_PKEYINDX;
2610                                qp->qp_pkeyindx = pkeyindx;
2611                        } else {
2612                                return (IBT_PKEY_IX_ILLEGAL);
2613                        }
2614                }

2616                /*
2617                 * If we are attempting to modify the QKey for this QP, then
2618                 * fill it in and set the appropriate flag in the "opmask"
2619                 * parameter.
2620                 */
2621                if (flags & IBT_CEP_SET_QKEY) {
2622                        qpc->qkey = ud->ud_qkey;
2623                        opmask |= HERMON_CMD_OP_QKEY;
2624                }

2626        } else if (qp->qp_serv_type == HERMON_QP_RC) {
2627                rc = &info_p->qp_transport.rc;

2629                /*
2630                 * Check if any of the flags indicate a change in the RDMA
2631                 * (recv) enable/disable flags and set the appropriate flag in
2632                 * the "opmask" parameter
2633                 */
2634                opmask |= hermon_check_rdma_enable_flags(flags, info_p, qpc);
```

```
2636                /*
2637                 * Check for optional primary path and fill in the
2638                 * appropriate QPC fields if one is specified
2639                 */
2640                if (flags & IBT_CEP_SET_ADDS_VECT) {
2641                        qpc_path = &qpc->pri_addr_path;
2642                        adds_vect = &rc->rc_path.cep_adds_vect;

2644                        /* Set the common primary address path fields */
2645                        status = hermon_set_addr_path(state, adds_vect,
2646                            qpc_path, HERMON_ADDRPATH_QP);
2647                        if (status != DDI_SUCCESS) {
2648                                return (status);
2649                        }
2650                        qpc->rnr_retry = rc->rc_rnr_retry_cnt;
2651                        qpc_path->ack_timeout = rc->rc_path.cep_timeout;
2652                        qpc->retry_cnt = rc->rc_retry_cnt;

2654                        portnum = qp->qp_portnum + 1;
2655                        if (hermon_portnum_is_valid(state, portnum)) {
2656                                qpc->pri_addr_path.sched_q  =
2657                                    HERMON_QP_SCHEDQ_GET(qp->qp_portnum,
2658                                    adds_vect->av_srvl, qp->qp_is_special);
2659                        } else {
2660                                return (IBT_HCA_PORT_INVALID);
2661                        }

2663                        /*
2664                         * MTU changes as part of sqd2sqd are not allowed.
2665                         * Simply keep the same MTU value here, stored in the
2666                         * qphdl from init2rtr time.
2667                         */
2668                        qpc->mtu = qp->qp_save_mtu;

2670                        opmask |= (HERMON_CMD_OP_PRIM_PATH |
2671                            HERMON_CMD_OP_RETRYCNT | HERMON_CMD_OP_ACKTIMEOUT |
2672                            HERMON_CMD_OP_PRIM_RNRRETRY);
2673                }

2675                /*
2676                 * If we are attempting to modify the path migration state for
2677                 * this QP, then check for valid state and fill it in.  Also
2678                 * set the appropriate flag in the "opmask" parameter.
2679                 */
2680                if (flags & IBT_CEP_SET_MIG) {
2681                        if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
2682                                qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
2683                        } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
2684                                qpc->pm_state = HERMON_QP_PMSTATE_REARM;
2685                        } else {
2686                                return (IBT_QP_APM_STATE_INVALID);
2687                        }
2688                        opmask |= HERMON_CMD_OP_PM_STATE;
2689                }

2691                /*
2692                 * If we are attempting to modify the PKey index for this QP,
2693                 * then check for valid PKey index and fill it in.  Also set
2694                 * the appropriate flag in the "opmask" parameter.
2695                 */
2696                if (flags & IBT_CEP_SET_PKEY_IX) {
2697                        pkeyindx = rc->rc_path.cep_pkey_ix;
2698                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2699                                qpc->pri_addr_path.pkey_indx = pkeyindx;
2700                                opmask |= HERMON_CMD_OP_PKEYINDX;
2701                        } else {
```



```
2702                                return (IBT_PKEY_IX_ILLEGAL);
2703                        }
2704                }

2706                /*
2707                 * If we are attempting to modify the port for this QP, then
2708                 * check for valid port number and fill it in.  Also set the
2709                 * appropriate flag in the "opmask" parameter.
2710                 */
2711                if (flags & IBT_CEP_SET_PORT) {
2712                        portnum = rc->rc_path.cep_hca_port_num;
2713                        if (hermon_portnum_is_valid(state, portnum)) {
2714                                qp->qp_portnum = portnum - 1;
2715                                qpc->pri_addr_path.sched_q =
2716                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
2717                                    adds_vect->av_srvl, qp->qp_is_special);
2718                        } else {
2719                                return (IBT_HCA_PORT_INVALID);
2720                        }
2721                        opmask |= HERMON_CMD_OP_SCHEDQUEUE;
2722                }

2724                /*
2725                 * Check for optional alternate path and fill in the
2726                 * appropriate QPC fields if one is specified
2727                 */
2728                if (flags & IBT_CEP_SET_ALT_PATH) {
2729                        qpc_path = &qpc->alt_addr_path;
2730                        adds_vect = &rc->rc_alt_path.cep_adds_vect;

2732                        /* Set the common alternate address path fields */
2733                        status = hermon_set_addr_path(state, adds_vect,
2734                            qpc_path, HERMON_ADDRPATH_QP);
2735                        if (status != DDI_SUCCESS) {
2736                                return (status);
2737                        }
2738                        qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

2740                        /*
2741                         * Check for valid alternate path port number and fill
2742                         * it in
2743                         */
2744                        portnum = rc->rc_alt_path.cep_hca_port_num;
2745                        if (hermon_portnum_is_valid(state, portnum)) {
2746                                qp->qp_portnum_alt = portnum - 1;
2747                                qpc->alt_addr_path.sched_q =
2748                                    HERMON_QP_SCHEDQ_GET(portnum - 1,
2749                                    adds_vect->av_srvl, qp->qp_is_special);
2750                        } else {
2751                                return (IBT_HCA_PORT_INVALID);
2752                        }

2754                        /*
2755                         * Check for valid alternate path PKey index and fill
2756                         * it in
2757                         */
2758                        pkeyindx = rc->rc_alt_path.cep_pkey_ix;
2759                        if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2760                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2761                        } else {
2762                                return (IBT_PKEY_IX_ILLEGAL);
2763                        }
2764                        opmask |= HERMON_CMD_OP_ALT_PATH;
2765                }

2767                /*
```

```
2768                 * If we are attempting to modify the number of "outgoing
2769                 * RDMA resources" for this QP, then check for valid value and
2770                 * fill it in.  Also set the appropriate flag in the "opmask"
2771                 * parameter.
2772                 */
2773                if (flags & IBT_CEP_SET_RDMARA_OUT) {
2774                        if (hermon_qp_validate_init_depth(state, rc,
2775                            &sra_max) != DDI_SUCCESS) {
2776                                return (IBT_INVALID_PARAM);
2777                        }
2778                        qpc->sra_max = sra_max;
2779                        opmask |= HERMON_CMD_OP_SRA_SET;
2780                }
2781
2782                /*
2783                 * If we are attempting to modify the number of "incoming
2784                 * RDMA resources" for this QP, then check for valid value and
2785                 * update the "rra_max" and "ra_buf_index" fields in the QPC to
2786                 * point to the pre-allocated RDB resources (in DDR).  Also set
2787                 * the appropriate flag in the "opmask" parameter.
2788                 */
2789                if (flags & IBT_CEP_SET_RDMARA_IN) {
2790                        if (hermon_qp_validate_resp_rsrc(state, rc,
2791                            &rra_max) != DDI_SUCCESS) {
2792                                return (IBT_INVALID_PARAM);
2793                        }
2794                        qpc->rra_max = rra_max;
2795                        opmask |= HERMON_CMD_OP_RRA_SET;
2796                }
2797
2798                /*
2799                 * If we are attempting to modify the "Local Ack Timeout" value
2800                 * for this QP, then fill it in and set the appropriate flag in
2801                 * the "opmask" parameter.
2802                 */
2803                if (flags & IBT_CEP_SET_TIMEOUT) {
2804                        qpc_path = &qpc->pri_addr_path;
2805                        qpc_path->ack_timeout = rc->rc_path.cep_timeout;
2806                        opmask |= HERMON_CMD_OP_ACKTIMEOUT;
2807                }
2808
2809                /*
2810                 * If we are attempting to modify the "Retry Count" for this QP,
2811                 * then fill it in and set the appropriate flag in the "opmask"
2812                 * parameter.
2813                 */
2814                if (flags & IBT_CEP_SET_RETRY) {
2815                        qpc->retry_cnt = rc->rc_retry_cnt;
2816                        opmask |= HERMON_CMD_OP_PRIM_RNRRETRY;
2817                }
2818
2819                /*
2820                 * If we are attempting to modify the "RNR Retry Count" for this
2821                 * QP, then fill it in and set the appropriate flag in the
2822                 * "opmask" parameter.
2823                 */
2824                if (flags & IBT_CEP_SET_RNR_NAK_RETRY) {
2825                        qpc_path = &qpc->pri_addr_path;
2826                        qpc->rnr_retry = rc->rc_rnr_retry_cnt;
2827                        opmask |= HERMON_CMD_OP_RETRYCNT;
2828                }
2829
2830                /*
2831                 * If we are attempting to modify the "Minimum RNR NAK" value
2832                 * for this QP, then fill it in and set the appropriate flag
2833                 * in the "opmask" parameter.
```



```
2834                 */
2835                if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
2836                        qpc->min_rnr_nak = rc->rc_min_rnr_nak;
2837                        opmask |= HERMON_CMD_OP_MINRNRNAK;
2838                }
2839
2840        } else if (qp->qp_serv_type == HERMON_QP_UC) {
2841                uc = &info_p->qp_transport.uc;
2842
2843                /*
2844                 * Check if any of the flags indicate a change in the RDMA
2845                 * Write (recv) enable/disable and set the appropriate flag
2846                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
2847                 * not valid for UC transport.
2848                 */
2849                if (flags & IBT_CEP_SET_RDMA_W) {
2850                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
2851                        opmask |= HERMON_CMD_OP_RWE;
2852                }
2853
2854                /*
2855                 * Check for optional primary path and fill in the
2856                 * appropriate QPC fields if one is specified
2857                 */
2858                if (flags & IBT_CEP_SET_ADDS_VECT) {
2859                        qpc_path = &qpc->pri_addr_path;
2860                        adds_vect = &uc->uc_path.cep_adds_vect;
2861
2862                        /* Set the common primary address path fields */
2863                        status = hermon_set_addr_path(state, adds_vect,
2864                            qpc_path, HERMON_ADDRPATH_QP);
2865                        if (status != DDI_SUCCESS) {
2866                                return (status);
2867                        }
2868                        portnum = qp->qp_portnum + 1;
2869                        if (hermon_portnum_is_valid(state, portnum)) {
2870                                qpc->pri_addr_path.sched_q =
2871                                    HERMON_QP_SCHEDQ_GET(qp->qp_portnum,
2872                                    adds_vect->av_srvl, qp->qp_is_special);
2873                        } else {
2874                                return (IBT_HCA_PORT_INVALID);
2875                        }
2876
2877                        /*
2878                         * MTU changes as part of sqd2sqd are not allowed.
2879                         * Simply keep the same MTU value here, stored in the
2880                         * qphdl from init2rtr time.
2881                         */
2882                        qpc->mtu = qp->qp_save_mtu;
2883
2884                        opmask |= HERMON_CMD_OP_PRIM_PATH;
2885                }
2886
2887                /*
2888                 * If we are attempting to modify the path migration state for
2889                 * this QP, then check for valid state and fill it in.  Also
2890                 * set the appropriate flag in the "opmask" parameter.
2891                 */
2892                if (flags & IBT_CEP_SET_MIG) {
2893                        if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
2894                                qpc->pm_state = HERMON_QP_PMSTATE_MIGRATED;
2895                        } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
2896                                qpc->pm_state = HERMON_QP_PMSTATE_REARM;
2897                        } else {
2898                                return (IBT_QP_APM_STATE_INVALID);
2899                        }
```

```
2900                    opmask |= HERMON_CMD_OP_PM_STATE;
2901            }
2902
2903            /*
2904             * If we are attempting to modify the PKey index for this QP,
2905             * then check for valid PKey index and fill it in.  Also set
2906             * the appropriate flag in the "opmask" parameter.
2907             */
2908            if (flags & IBT_CEP_SET_PKEY_IX) {
2909                    pkeyindx = uc->uc_path.cep_pkey_ix;
2910                    if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2911                            qpc->pri_addr_path.pkey_indx = pkeyindx;
2912                            opmask |= HERMON_CMD_OP_PKEYINDX;
2913                    } else {
2914                            return (IBT_PKEY_IX_ILLEGAL);
2915                    }
2916            }
2917
2918            /*
2919             * Check for optional alternate path and fill in the
2920             * appropriate QPC fields if one is specified
2921             */
2922            if (flags & IBT_CEP_SET_ALT_PATH) {
2923                    qpc_path = &qpc->alt_addr_path;
2924                    adds_vect = &uc->uc_alt_path.cep_adds_vect;
2925
2926                    /* Set the common alternate address path fields */
2927                    status = hermon_set_addr_path(state, adds_vect,
2928                        qpc_path, HERMON_ADDRPATH_QP);
2929                    if (status != DDI_SUCCESS) {
2930                            return (status);
2931                    }
2932
2933                    /*
2934                     * Check for valid alternate path port number and fill
2935                     * it in
2936                     */
2937                    portnum = uc->uc_alt_path.cep_hca_port_num;
2938                    if (hermon_portnum_is_valid(state, portnum)) {
2939                            qp->qp_portnum_alt = portnum - 1;
2940                            qpc->alt_addr_path.sched_q =
2941                                HERMON_QP_SCHEDQ_GET(portnum - 1,
2942                                adds_vect->av_srvl, qp->qp_is_special);
2943                    } else {
2944                            return (IBT_HCA_PORT_INVALID);
2945                    }
2946
2947                    /*
2948                     * Check for valid alternate path PKey index and fill
2949                     * it in
2950                     */
2951                    pkeyindx = uc->uc_alt_path.cep_pkey_ix;
2952                    if (hermon_pkeyindex_is_valid(state, pkeyindx)) {
2953                            qpc->alt_addr_path.pkey_indx = pkeyindx;
2954                    } else {
2955                            return (IBT_PKEY_IX_ILLEGAL);
2956                    }
2957                    opmask |= HERMON_CMD_OP_ALT_PATH;
2958            }
2959        } else {
2960                /*
2961                 * Invalid QP transport type. If we got here then it's a
2962                 * warning of a probably serious problem.  So print a message
2963                 * and return failure
2964                 */
2965                HERMON_WARNING(state, "unknown QP transport type in sqd2sqd");
```



```
2966                return (ibc_get_ci_failure(0));
2967        }
2968
2969        /*
2970         * Post the SQD2SQD_QP command to the Hermon firmware
2971         *
2972         * We do a HERMON_NOSLEEP here because we are still holding the
2973         * "qp_lock".  If we got raised to interrupt level by priority
2974         * inversion, we do not want to block in this routine waiting for
2975         * success.
2976         */
2977        status = hermon_cmn_qp_cmd_post(state, SQD2SQD_QP, qpc, qp->qp_qpnum,
2978            opmask, HERMON_CMD_NOSLEEP_SPIN);
2979        if (status != HERMON_CMD_SUCCESS) {
2980                if (status != HERMON_CMD_BAD_QP_STATE) {
2981                        cmn_err(CE_NOTE, "hermon%d: SQD2SQD_QP command failed: "
2982                            "%08x\n", state->hs_instance, status);
2983                        if (status == HERMON_CMD_INVALID_STATUS) {
2984                                hermon_fm_ereport(state, HCA_SYS_ERR,
2985                                    HCA_ERR_SRV_LOST);
2986                        }
2987                        return (ibc_get_ci_failure(0));
2988                } else {
2989                        return (IBT_QP_STATE_INVALID);
2990                }
2991        }
2992
2993        return (DDI_SUCCESS);
2994 }
2995
2996
2997 /*
2998  * hermon_qp_sqerr2rts()
2999  *    Context: Can be called from interrupt or base context.
3000  */
3001 static int
3002 hermon_qp_sqerr2rts(hermon_state_t *state, hermon_qphdl_t qp,
3003     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
3004 {
3005        hermon_hw_qpc_t         *qpc;
3006        ibt_qp_ud_attr_t        *ud;
3007        uint32_t                opmask = 0;
3008        int                     status;
3009
3010        ASSERT(MUTEX_HELD(&qp->qp_lock));
3011
3012        /*
3013         * Grab the temporary QPC entry from QP software state
3014         */
3015        qpc = &qp->qpc;
3016
3017        /*
3018         * Since there are no common fields to be filled in for this command,
3019         * we begin with the QPC fields which are specific to transport type.
3020         */
3021        if (qp->qp_type == IBT_UD_RQP) {
3022                ud = &info_p->qp_transport.ud;
3023
3024                /*
3025                 * If we are attempting to modify the QKey for this QP, then
3026                 * fill it in and set the appropriate flag in the "opmask"
3027                 * parameter.
3028                 */
3029                if (flags & IBT_CEP_SET_QKEY) {
3030                        qpc->qkey = ud->ud_qkey;
3031                        opmask |= HERMON_CMD_OP_QKEY;
```

```
3032                 }

3034         } else if (qp->qp_serv_type == HERMON_QP_UC) {

3036                 /*
3037                  * Check if any of the flags indicate a change in the RDMA
3038                  * Write (recv) enable/disable and set the appropriate flag
3039                  * in the "opmask" parameter. Note: RDMA Read and Atomic are
3040                  * not valid for UC transport.
3041                  */
3042                 if (flags & IBT_CEP_SET_RDMA_W) {
3043                         qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
3044                         opmask |= HERMON_CMD_OP_RWE;
3045                 }
3046         } else {
3047                 /*
3048                  * Invalid QP transport type. If we got here then it's a
3049                  * warning of a probably serious problem.  So print a message
3050                  * and return failure
3051                  */
3052                 HERMON_WARNING(state, "unknown QP transport type in sqerr2rts");
3053                 return (ibc_get_ci_failure(0));
3054         }

3056         /*
3057          * Post the SQERR2RTS_QP command to the Hermon firmware
3058          *
3059          * We do a HERMON_NOSLEEP here because we are still holding the
3060          * "qp_lock".  If we got raised to interrupt level by priority
3061          * inversion, we do not want to block in this routine waiting for
3062          * success.
3063          */
3064         status = hermon_cmn_qp_cmd_post(state, SQERR2RTS_QP, qpc, qp->qp_qpnum,
3065             opmask, HERMON_CMD_NOSLEEP_SPIN);
3066         if (status != HERMON_CMD_SUCCESS) {
3067                 if (status != HERMON_CMD_BAD_QP_STATE) {
3068                         cmn_err(CE_NOTE, "hermon%d: SQERR2RTS_QP command "
3069                             "failed: %08x\n", state->hs_instance, status);
3070                         if (status == HERMON_CMD_INVALID_STATUS) {
3071                                 hermon_fm_ereport(state, HCA_SYS_ERR,
3072                                     HCA_ERR_SRV_LOST);
3073                         }
3074                         return (ibc_get_ci_failure(0));
3075                 } else {
3076                         return (IBT_QP_STATE_INVALID);
3077                 }
3078         }

3080         return (DDI_SUCCESS);
3081 }


3084 /*
3085  * hermon_qp_to_error()
3086  *    Context: Can be called from interrupt or base context.
3087  */
3088 static int
3089 hermon_qp_to_error(hermon_state_t *state, hermon_qphdl_t qp)
3090 {
3091         int     status;

3093         ASSERT(MUTEX_HELD(&qp->qp_lock));

3095         /*
3096          * Post the TOERR_QP command to the Hermon firmware
3097          *
```



```
3098          * We do a HERMON_NOSLEEP here because we are still holding the
3099          * "qp_lock".  If we got raised to interrupt level by priority
3100          * inversion, we do not want to block in this routine waiting for
3101          * success.
3102          */
3103         status = hermon_cmn_qp_cmd_post(state, TOERR_QP, NULL, qp->qp_qpnum,
3104             0, HERMON_CMD_NOSLEEP_SPIN);
3105         if (status != HERMON_CMD_SUCCESS) {
3106                 cmn_err(CE_NOTE, "hermon%d: TOERR_QP command failed: %08x\n",
3107                     state->hs_instance, status);
3108                 if (status == HERMON_CMD_INVALID_STATUS) {
3109                         hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
3110                 }
3111                 return (ibc_get_ci_failure(0));
3112         }

3114         return (DDI_SUCCESS);
3115 }


3118 /*
3119  * hermon_qp_to_reset()
3120  *    Context: Can be called from interrupt or base context.
3121  */
3122 int
3123 hermon_qp_to_reset(hermon_state_t *state, hermon_qphdl_t qp)
3124 {
3125         hermon_hw_qpc_t *qpc;
3126         int             status;

3128         ASSERT(MUTEX_HELD(&qp->qp_lock));

3130         /*
3131          * Grab the temporary QPC entry from QP software state
3132          */
3133         qpc = &qp->qpc;

3135         /*
3136          * Post the TORST_QP command to the Hermon firmware
3137          *
3138          * We do a HERMON_NOSLEEP here because we are still holding the
3139          * "qp_lock".  If we got raised to interrupt level by priority
3140          * inversion, we do not want to block in this routine waiting for
3141          * success.
3142          */
3143         status = hermon_cmn_qp_cmd_post(state, TORST_QP, qpc, qp->qp_qpnum,
3144             0, HERMON_CMD_NOSLEEP_SPIN);
3145         if (status != HERMON_CMD_SUCCESS) {
3146                 cmn_err(CE_NOTE, "hermon%d: TORST_QP command failed: %08x\n",
3147                     state->hs_instance, status);
3148                 if (status == HERMON_CMD_INVALID_STATUS) {
3149                         hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
3150                 }
3151                 return (ibc_get_ci_failure(0));
3152         }
3153         if (qp->qp_serv_type == HERMON_QP_FEXCH) {
3154                 status = hermon_fcoib_fexch_mkey_fini(state, qp->qp_pdhdl,
3155                     qp->qp_qpnum, HERMON_CMD_NOSLEEP_SPIN);
3156                 if (status != DDI_SUCCESS)
3157                         cmn_err(CE_NOTE, "hermon%d: fexch_mkey_fini failed "
3158                             "%08x\n", state->hs_instance, status);
3159         }
3160         return (DDI_SUCCESS);
3161 }
```

```
3164 /*
3165  * hermon_qp_reset2err()
3166  *    Context: Can be called from interrupt or base context.
3167  */
3168 static int
3169 hermon_qp_reset2err(hermon_state_t *state, hermon_qphdl_t qp)
3170 {
3171         hermon_hw_qpc_t *qpc;
3172         int             status;
3173         uint32_t        cqnmask;

3175         ASSERT(MUTEX_HELD(&qp->qp_lock));

3177         /*
3178          * In order to implement the transition from "Reset" directly to the
3179          * "Error" state, it is necessary to first give ownership of the QP
3180          * context to the Hermon hardware.  This is accomplished by
3181          * transitioning the QP to "Init" as an intermediate step and then,
3182          * immediately transitioning to "Error".
3183          *
3184          * When this function returns success, the QP context will be owned by
3185          * the Hermon hardware and will be in the "Error" state.
3186          */

3188         /*
3189          * Grab the temporary QPC entry from QP software state
3190          */
3191         qpc = &qp->qpc;

3193         /*
3194          * Fill in the common fields in the QPC
3195          */
3196         if (qp->qp_is_special) {
3197                 qpc->serv_type  = HERMON_QP_MLX;
3198         } else {
3199                 qpc->serv_type  = qp->qp_serv_type;
3200         }
3201         qpc->pm_state           = HERMON_QP_PMSTATE_MIGRATED;
3202         qpc->usr_page           = qp->qp_uarpg;
3203         /* dbr is now an address, not an index */
3204         qpc->dbr_addrh          = ((uint64_t)qp->qp_rq_pdbr >> 32);
3205         qpc->dbr_addrl          = ((uint64_t)qp->qp_rq_pdbr & 0xFFFFFFFC) >> 2;
3206         qpc->pd                 = qp->qp_pdhdl->pd_pdnum;
3207         /*
3208          * HERMON:
3209          * qpc->wqe_baseaddr is replaced by LKey from the cMPT, and
3210          * page_offset, mtt_base_addr_h/l, and log2_page_size will
3211          * be used to map the WQE buffer
3212          * NOTE that the cMPT is created implicitly when the QP is
3213          * transitioned from reset to init
3214          */
3215         qpc->log2_pgsz          = qp->qp_mrhdl->mr_log2_pgsz;
3216         qpc->mtt_base_addrh     = (qp->qp_mrhdl->mr_mttaddr) >> 32 & 0xFF;
3217         qpc->mtt_base_addrl     = (qp->qp_mrhdl->mr_mttaddr) >> 3 & 0xFFFFFFFF;
3218         cqnmask = (1 << state->hs_cfg_profile->cp_log_num_cq) - 1;
3219         qpc->cqn_snd            =
3220             (qp->qp_sq_cqhdl == NULL) ? 0 : qp->qp_sq_cqhdl->cq_cqnum & cqnmask;
3221         qpc->page_offs          = qp->qp_wqinfo.qa_pgoffs >> 6;
3222         qpc->cqn_rcv            =
3223             (qp->qp_rq_cqhdl == NULL) ? 0 : qp->qp_rq_cqhdl->cq_cqnum & cqnmask;

3225         qpc->sq_wqe_counter     = 0;
3226         qpc->rq_wqe_counter     = 0;
3227         qpc->log_sq_stride      = qp->qp_sq_log_wqesz - 4;
3228         qpc->log_rq_stride      = qp->qp_rq_log_wqesz - 4;
3229         qpc->log_sq_size        = highbit(qp->qp_sq_bufsz) - 1;
```



```
3230         qpc->log_rq_size        = highbit(qp->qp_rq_bufsz) - 1;
3231         qpc->srq_en             = (qp->qp_alloc_flags & IBT_QP_USES_SRQ) != 0;
3232         qpc->sq_no_prefetch     = qp->qp_no_prefetch;

3234         if (qp->qp_alloc_flags & IBT_QP_USES_SRQ) {
3235                 qpc->srq_number = qp->qp_srqhdl->srq_srqnum;
3236         } else {
3237                 qpc->srq_number = 0;
3238         }

3240         qpc->fre                = 0; /* default disable fast registration WR */
3241         qpc->rlky               = 0; /* default disable reserved lkey */

3243         /*
3244          * Now fill in the QPC fields which are specific to transport type
3245          */
3246         if (qp->qp_type == IBT_UD_RQP) {
3247                 /* Set the UD parameters to an invalid default */
3248                 qpc->qkey = 0;
3249                 qpc->pri_addr_path.sched_q =
3250                     HERMON_QP_SCHEDQ_GET(0, 0, qp->qp_is_special);
3251                 qpc->pri_addr_path.pkey_indx = 0;

3253         } else if (qp->qp_serv_type == HERMON_QP_RC) {
3254                 /* Set the RC parameters to invalid default */
3255                 qpc->rre = 0;
3256                 qpc->rwe = 0;
3257                 qpc->rae = 0;
3258                 qpc->alt_addr_path.sched_q =
3259                     HERMON_QP_SCHEDQ_GET(0, 0, qp->qp_is_special);
3260                 qpc->pri_addr_path.pkey_indx = 0;

3262         } else if (qp->qp_serv_type == HERMON_QP_UC) {
3263                 /* Set the UC parameters to invalid default */
3264                 qpc->rwe = 0;
3265                 qpc->alt_addr_path.sched_q =
3266                     HERMON_QP_SCHEDQ_GET(0, 0, qp->qp_is_special);
3267                 qpc->pri_addr_path.pkey_indx = 0;

3269         } else {
3270                 /*
3271                  * Invalid QP transport type. If we got here then it's a
3272                  * warning of a probably serious problem.  So print a message
3273                  * and return failure
3274                  */
3275                 HERMON_WARNING(state, "unknown QP transport type in rst2err");
3276                 return (ibc_get_ci_failure(0));
3277         }

3279         /*
3280          * Post the RST2INIT_QP command to the Hermon firmware
3281          *
3282          * We do a HERMON_NOSLEEP here because we are still holding the
3283          * "qp_lock".  If we got raised to interrupt level by priority
3284          * inversion, we do not want to block in this routine waiting for
3285          * success.
3286          */
3287         status = hermon_cmn_qp_cmd_post(state, RST2INIT_QP, qpc, qp->qp_qpnum,
3288             0, HERMON_CMD_NOSLEEP_SPIN);
3289         if (status != HERMON_CMD_SUCCESS) {
3290                 cmn_err(CE_NOTE, "hermon%d: RST2INIT_QP command failed: %08x\n",
3291                     state->hs_instance, status);
3292                 if (status == HERMON_CMD_INVALID_STATUS) {
3293                         hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
3294                 }
3295                 return (ibc_get_ci_failure(0));
```

```
3296         }

3298         /*
3299          * Now post the TOERR_QP command to the Hermon firmware
3300          *
3301          * We still do a HERMON_NOSLEEP here because we are still holding the
3302          * "qp_lock".  Note:  If this fails (which it really never should),
3303          * it indicates a serious problem in the HW or SW.  We try to move
3304          * the QP back to the "Reset" state if possible and print a warning
3305          * message if not.  In any case, we return an error here.
3306          */
3307         status = hermon_cmn_qp_cmd_post(state, TOERR_QP, NULL, qp->qp_qpnum,
3308             0, HERMON_CMD_NOSLEEP_SPIN);
3309         if (status != HERMON_CMD_SUCCESS) {
3310                 cmn_err(CE_NOTE, "hermon%d: TOERR_QP command failed: %08x\n",
3311                     state->hs_instance, status);
3312                 if (status == HERMON_CMD_INVALID_STATUS) {
3313                         hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
3314                 }
3315                 if (hermon_qp_to_reset(state, qp) != DDI_SUCCESS) {
3316                         HERMON_WARNING(state, "failed to reset QP context");
3317                 }
3318                 return (ibc_get_ci_failure(0));
3319         }

3321         return (DDI_SUCCESS);
3322 }


3325 /*
3326  * hermon_check_rdma_enable_flags()
3327  *    Context: Can be called from interrupt or base context.
3328  */
3329 static uint_t
3330 hermon_check_rdma_enable_flags(ibt_cep_modify_flags_t flags,
3331     ibt_qp_info_t *info_p, hermon_hw_qpc_t *qpc)
3332 {
3333         uint_t  opmask = 0;

3335         if (flags & IBT_CEP_SET_RDMA_R) {
3336                 qpc->rre = (info_p->qp_flags & IBT_CEP_RDMA_RD) ? 1 : 0;
3337                 opmask |= HERMON_CMD_OP_RRE;
3338         }

3340         if (flags & IBT_CEP_SET_RDMA_W) {
3341                 qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
3342                 opmask |= HERMON_CMD_OP_RWE;
3343         }

3345         if (flags & IBT_CEP_SET_ATOMIC) {
3346                 qpc->rae = (info_p->qp_flags & IBT_CEP_ATOMIC) ? 1 : 0;
3347                 opmask |= HERMON_CMD_OP_RAE;
3348         }

3350         return (opmask);
3351 }

3353 /*
3354  * hermon_qp_validate_resp_rsrc()
3355  *    Context: Can be called from interrupt or base context.
3356  */
3357 static int
3358 hermon_qp_validate_resp_rsrc(hermon_state_t *state, ibt_qp_rc_attr_t *rc,
3359     uint_t *rra_max)
3360 {
3361         uint_t  rdma_ra_in;
```



```
3363         rdma_ra_in = rc->rc_rdma_ra_in;

3365         /*
3366          * Check if number of responder resources is too large.  Return an
3367          * error if it is
3368          */
3369         if (rdma_ra_in > state->hs_cfg_profile->cp_hca_max_rdma_in_qp) {
3370                 return (IBT_INVALID_PARAM);
3371         }

3373         /*
3374          * If the number of responder resources is too small, round it up.
3375          * Then find the next highest power-of-2
3376          */
3377         if (rdma_ra_in == 0) {
3378                 rdma_ra_in = 1;
3379         }
3380         if (ISP2(rdma_ra_in)) {
  36         if ((rdma_ra_in & (rdma_ra_in - 1)) == 0) {
3381                 *rra_max = highbit(rdma_ra_in) - 1;
3382         } else {
3383                 *rra_max = highbit(rdma_ra_in);
3384         }
3385         return (DDI_SUCCESS);
3386 }


3389 /*
3390  * hermon_qp_validate_init_depth()
3391  *    Context: Can be called from interrupt or base context.
3392  */
3393 static int
3394 hermon_qp_validate_init_depth(hermon_state_t *state, ibt_qp_rc_attr_t *rc,
3395     uint_t *sra_max)
3396 {
3397         uint_t  rdma_ra_out;

3399         rdma_ra_out = rc->rc_rdma_ra_out;

3401         /*
3402          * Check if requested initiator depth is too large.  Return an error
3403          * if it is
3404          */
3405         if (rdma_ra_out > state->hs_cfg_profile->cp_hca_max_rdma_out_qp) {
3406                 return (IBT_INVALID_PARAM);
3407         }

3409         /*
3410          * If the requested initiator depth is too small, round it up.
3411          * Then find the next highest power-of-2
3412          */
3413         if (rdma_ra_out == 0) {
3414                 rdma_ra_out = 1;
3415         }
3416         if (ISP2(rdma_ra_out)) {
  72         if ((rdma_ra_out & (rdma_ra_out - 1)) == 0) {
3417                 *sra_max = highbit(rdma_ra_out) - 1;
3418         } else {
3419                 *sra_max = highbit(rdma_ra_out);
3420         }
3421         return (DDI_SUCCESS);
3422 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   80113 Thu Oct 23 10:42:13 2014
new/usr/src/uts/common/io/ib/adapters/hermon/hermon_rsrc.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * hermon_rsrc.c
 *    Hermon Resource Management Routines
 *
 *    Implements all the routines necessary for setup, teardown, and
 *    alloc/free of all Hermon resources, including those that are managed
 *    by Hermon hardware or which live in Hermon's direct attached DDR memory.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/vmem.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/hermon/hermon.h>

int hermon_rsrc_verbose = 0;

/*
 * The following routines are used for initializing and destroying
 * the resource pools used by the Hermon resource allocation routines.
 * They consist of four classes of object:
 *
 * Mailboxes:  The "In" and "Out" mailbox types are used by the Hermon
 *    command interface routines.  Mailboxes are used to pass information
 *    back and forth to the Hermon firmware.  Either type of mailbox may
 *    be allocated from Hermon's direct attached DDR memory or from system
 *    memory (although currently all "In" mailboxes are in DDR and all "out"
 *    mailboxes come from system memory.
 *
 * HW entry objects:  These objects represent resources required by the Hermon
```



```
 *    hardware.  These objects include things like Queue Pair contexts (QPC),
 *    Completion Queue contexts (CQC), Event Queue contexts (EQC), RDB (for
 *    supporting RDMA Read/Atomic), Multicast Group entries (MCG), Memory
 *    Protection Table entries (MPT), Memory Translation Table entries (MTT).
 *
 *    What these objects all have in common is that they are each required
 *    to come from ICM memory, they are always allocated from tables, and
 *    they are not to be directly accessed (read or written) by driver
 *    software (Mellanox FMR access to MPT is an exception).
 *    The other notable exceptions are the UAR pages (UAR_PG) which are
 *    allocated from the UAR address space rather than DDR, and the UD
 *    address vectors (UDAV) which are similar to the common object types
 *    with the major difference being that UDAVs _are_ directly read and
 *    written by driver software.
 *
 * SW handle objects: These objects represent resources required by Hermon
 *    driver software.  They are primarily software tracking structures,
 *    which are allocated from system memory (using kmem_cache).  Several of
 *    the objects have both a "constructor" and "destructor" method
 *    associated with them (see below).
 *
 * Protection Domain (PD) handle objects:  These objects are very much like
 *    a SW handle object with the notable difference that all PD handle
 *    objects have an actual Protection Domain number (PD) associated with
 *    them (and the PD number is allocated/managed through a separate
 *    vmem_arena specifically set aside for this purpose.
 */

static int hermon_rsrc_mbox_init(hermon_state_t *state,
    hermon_rsrc_mbox_info_t *info);
static void hermon_rsrc_mbox_fini(hermon_state_t *state,
    hermon_rsrc_mbox_info_t *info);

static int hermon_rsrc_sw_handles_init(hermon_state_t *state,
    hermon_rsrc_sw_hdl_info_t *info);
static void hermon_rsrc_sw_handles_fini(hermon_state_t *state,
    hermon_rsrc_sw_hdl_info_t *info);

static int hermon_rsrc_pd_handles_init(hermon_state_t *state,
    hermon_rsrc_sw_hdl_info_t *info);
static void hermon_rsrc_pd_handles_fini(hermon_state_t *state,
    hermon_rsrc_sw_hdl_info_t *info);

/*
 * The following routines are used for allocating and freeing the specific
 * types of objects described above from their associated resource pools.
 */
static int hermon_rsrc_mbox_alloc(hermon_rsrc_pool_info_t *pool_info,
    uint_t num, hermon_rsrc_t *hdl);
static void hermon_rsrc_mbox_free(hermon_rsrc_t *hdl);

static int hermon_rsrc_hw_entry_alloc(hermon_rsrc_pool_info_t *pool_info,
    uint_t num, uint_t num_align, uint_t sleepflag, hermon_rsrc_t *hdl);
static void hermon_rsrc_hw_entry_free(hermon_rsrc_pool_info_t *pool_info,
    hermon_rsrc_t *hdl);
static int hermon_rsrc_hw_entry_reserve(hermon_rsrc_pool_info_t *pool_info,
    uint_t num, uint_t num_align, uint_t sleepflag, hermon_rsrc_t *hdl);

static int hermon_rsrc_hw_entry_icm_confirm(hermon_rsrc_pool_info_t *pool_info,
    uint_t num, hermon_rsrc_t *hdl, int num_to_hdl);
static int hermon_rsrc_hw_entry_icm_free(hermon_rsrc_pool_info_t *pool_info,
    hermon_rsrc_t *hdl, int num_to_hdl);

static int hermon_rsrc_swhdl_alloc(hermon_rsrc_pool_info_t *pool_info,
    uint_t sleepflag, hermon_rsrc_t *hdl);
static void hermon_rsrc_swhdl_free(hermon_rsrc_pool_info_t *pool_info,
```

```
128     hermon_rsrc_t *hdl);

130 static int hermon_rsrc_pdhdl_alloc(hermon_rsrc_pool_info_t *pool_info,
131     uint_t sleepflag, hermon_rsrc_t *hdl);
132 static void hermon_rsrc_pdhdl_free(hermon_rsrc_pool_info_t *pool_info,
133     hermon_rsrc_t *hdl);

135 static int hermon_rsrc_fexch_alloc(hermon_state_t *state,
136     hermon_rsrc_type_t rsrc, uint_t num, uint_t sleepflag, hermon_rsrc_t *hdl);
137 static void hermon_rsrc_fexch_free(hermon_state_t *state, hermon_rsrc_t *hdl);
138 static int hermon_rsrc_rfci_alloc(hermon_state_t *state,
139     hermon_rsrc_type_t rsrc, uint_t num, uint_t sleepflag, hermon_rsrc_t *hdl);
140 static void hermon_rsrc_rfci_free(hermon_state_t *state, hermon_rsrc_t *hdl);

142 /*
143  * The following routines are the constructors and destructors for several
144  * of the SW handle type objects.  For certain types of SW handles objects
145  * (all of which are implemented using kmem_cache), we need to do some
146  * special field initialization (specifically, mutex_init/destroy).  These
147  * routines enable that init and teardown.
148  */
149 static int hermon_rsrc_pdhdl_constructor(void *pd, void *priv, int flags);
150 static void hermon_rsrc_pdhdl_destructor(void *pd, void *state);
151 static int hermon_rsrc_cqhdl_constructor(void *cq, void *priv, int flags);
152 static void hermon_rsrc_cqhdl_destructor(void *cq, void *state);
153 static int hermon_rsrc_qphdl_constructor(void *cq, void *priv, int flags);
154 static void hermon_rsrc_qphdl_destructor(void *cq, void *state);
155 static int hermon_rsrc_srqhdl_constructor(void *srq, void *priv, int flags);
156 static void hermon_rsrc_srqhdl_destructor(void *srq, void *state);
157 static int hermon_rsrc_refcnt_constructor(void *rc, void *priv, int flags);
158 static void hermon_rsrc_refcnt_destructor(void *rc, void *state);
159 static int hermon_rsrc_ahhdl_constructor(void *ah, void *priv, int flags);
160 static void hermon_rsrc_ahhdl_destructor(void *ah, void *state);
161 static int hermon_rsrc_mrhdl_constructor(void *mr, void *priv, int flags);
162 static void hermon_rsrc_mrhdl_destructor(void *mr, void *state);

164 /*
165  * Special routine to calculate and return the size of a MCG object based
166  * on current driver configuration (specifically, the number of QP per MCG
167  * that has been configured.
168  */
169 static int hermon_rsrc_mcg_entry_get_size(hermon_state_t *state,
170     uint_t *mcg_size_shift);


173 /*
174  * hermon_rsrc_alloc()
175  *
176  *    Context: Can be called from interrupt or base context.
177  *    The "sleepflag" parameter is used by all object allocators to
178  *    determine whether to SLEEP for resources or not.
179  */
180 int
181 hermon_rsrc_alloc(hermon_state_t *state, hermon_rsrc_type_t rsrc, uint_t num,
182     uint_t sleepflag, hermon_rsrc_t **hdl)
183 {
184         hermon_rsrc_pool_info_t *rsrc_pool;
185         hermon_rsrc_t           *tmp_rsrc_hdl;
186         int                     flag, status = DDI_FAILURE;

188         ASSERT(state != NULL);
189         ASSERT(hdl != NULL);

191         rsrc_pool = &state->hs_rsrc_hdl[rsrc];
192         ASSERT(rsrc_pool != NULL);
```



```
194         /*
195          * Allocate space for the object used to track the resource handle
196          */
197         flag = (sleepflag == HERMON_SLEEP) ? KM_SLEEP : KM_NOSLEEP;
198         tmp_rsrc_hdl = kmem_cache_alloc(state->hs_rsrc_cache, flag);
199         if (tmp_rsrc_hdl == NULL) {
200                 return (DDI_FAILURE);
201         }
202         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*tmp_rsrc_hdl))

204         /*
205          * Set rsrc_hdl type.  This is later used by the hermon_rsrc_free call
206          * to know what type of resource is being freed.
207          */
208         tmp_rsrc_hdl->rsrc_type = rsrc;

210         /*
211          * Depending on resource type, call the appropriate alloc routine
212          */
213         switch (rsrc) {
214         case HERMON_IN_MBOX:
215         case HERMON_OUT_MBOX:
216         case HERMON_INTR_IN_MBOX:
217         case HERMON_INTR_OUT_MBOX:
218                 status = hermon_rsrc_mbox_alloc(rsrc_pool, num, tmp_rsrc_hdl);
219                 break;

221         case HERMON_DMPT:
222                 /* Allocate "num" (contiguous/aligned for FEXCH) DMPTs */
223         case HERMON_QPC:
224                 /* Allocate "num" (contiguous/aligned for RSS) QPCs */
225                 status = hermon_rsrc_hw_entry_alloc(rsrc_pool, num, num,
226                     sleepflag, tmp_rsrc_hdl);
227                 break;

229         case HERMON_QPC_FEXCH_PORT1:
230         case HERMON_QPC_FEXCH_PORT2:
231                 /* Allocate "num" contiguous/aligned QPCs for FEXCH */
232                 status = hermon_rsrc_fexch_alloc(state, rsrc, num,
233                     sleepflag, tmp_rsrc_hdl);
234                 break;

236         case HERMON_QPC_RFCI_PORT1:
237         case HERMON_QPC_RFCI_PORT2:
238                 /* Allocate "num" contiguous/aligned QPCs for RFCI */
239                 status = hermon_rsrc_rfci_alloc(state, rsrc, num,
240                     sleepflag, tmp_rsrc_hdl);
241                 break;

243         case HERMON_MTT:
244         case HERMON_CQC:
245         case HERMON_SRQC:
246         case HERMON_EQC:
247         case HERMON_MCG:
248         case HERMON_UARPG:
249                 /* Allocate "num" unaligned resources */
250                 status = hermon_rsrc_hw_entry_alloc(rsrc_pool, num, 1,
251                     sleepflag, tmp_rsrc_hdl);
252                 break;

254         case HERMON_MRHDL:
255         case HERMON_EQHDL:
256         case HERMON_CQHDL:
257         case HERMON_SRQHDL:
258         case HERMON_AHHDL:
259         case HERMON_QPHDL:
```

```
260         case HERMON_REFCNT:
261                 status = hermon_rsrc_swhdl_alloc(rsrc_pool, sleepflag,
262                     tmp_rsrc_hdl);
263                 break;
264
265         case HERMON_PDHDL:
266                 status = hermon_rsrc_pdhdl_alloc(rsrc_pool, sleepflag,
267                     tmp_rsrc_hdl);
268                 break;
269
270         case HERMON_RDB:        /* handled during HERMON_QPC */
271         case HERMON_ALTC:       /* handled during HERMON_QPC */
272         case HERMON_AUXC:       /* handled during HERMON_QPC */
273         case HERMON_CMPT_QPC:   /* handled during HERMON_QPC */
274         case HERMON_CMPT_SRQC:  /* handled during HERMON_SRQC */
275         case HERMON_CMPT_CQC:   /* handled during HERMON_CPC */
276         case HERMON_CMPT_EQC:   /* handled during HERMON_EPC */
277         default:
278                 HERMON_WARNING(state, "unexpected resource type in alloc ");
279                 cmn_err(CE_WARN, "Resource type %x \n", rsrc_pool->rsrc_type);
280                 break;
281         }
282
283         /*
284          * If the resource allocation failed, then free the special resource
285          * tracking structure and return failure.  Otherwise return the
286          * handle for the resource tracking structure.
287          */
288         if (status != DDI_SUCCESS) {
289                 kmem_cache_free(state->hs_rsrc_cache, tmp_rsrc_hdl);
290                 return (DDI_FAILURE);
291         } else {
292                 *hdl = tmp_rsrc_hdl;
293                 return (DDI_SUCCESS);
294         }
295 }


298 /*
299  * hermon_rsrc_reserve()
300  *
301  *    Context: Can only be called from attach.
302  *    The "sleepflag" parameter is used by all object allocators to
303  *    determine whether to SLEEP for resources or not.
304  */
305 int
306 hermon_rsrc_reserve(hermon_state_t *state, hermon_rsrc_type_t rsrc, uint_t num,
307     uint_t sleepflag, hermon_rsrc_t **hdl)
308 {
309         hermon_rsrc_pool_info_t *rsrc_pool;
310         hermon_rsrc_t           *tmp_rsrc_hdl;
311         int                     flag, status = DDI_FAILURE;
312
313         ASSERT(state != NULL);
314         ASSERT(hdl != NULL);
315
316         rsrc_pool = &state->hs_rsrc_hdl[rsrc];
317         ASSERT(rsrc_pool != NULL);
318
319         /*
320          * Allocate space for the object used to track the resource handle
321          */
322         flag = (sleepflag == HERMON_SLEEP) ? KM_SLEEP : KM_NOSLEEP;
323         tmp_rsrc_hdl = kmem_cache_alloc(state->hs_rsrc_cache, flag);
324         if (tmp_rsrc_hdl == NULL) {
325                 return (DDI_FAILURE);
```



```
326         }
327         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*tmp_rsrc_hdl))
328
329         /*
330          * Set rsrc_hdl type.  This is later used by the hermon_rsrc_free call
331          * to know what type of resource is being freed.
332          */
333         tmp_rsrc_hdl->rsrc_type = rsrc;
334
335         switch (rsrc) {
336         case HERMON_QPC:
337         case HERMON_DMPT:
338         case HERMON_MTT:
339                 /*
340                  * Reserve num resources, naturally aligned (N * num).
341                  */
342                 status = hermon_rsrc_hw_entry_reserve(rsrc_pool, num, num,
343                     sleepflag, tmp_rsrc_hdl);
344                 break;
345
346         default:
347                 HERMON_WARNING(state, "unexpected resource type in reserve ");
348                 cmn_err(CE_WARN, "Resource type %x \n", rsrc);
349                 break;
350         }
351
352         /*
353          * If the resource allocation failed, then free the special resource
354          * tracking structure and return failure.  Otherwise return the
355          * handle for the resource tracking structure.
356          */
357         if (status != DDI_SUCCESS) {
358                 kmem_cache_free(state->hs_rsrc_cache, tmp_rsrc_hdl);
359                 return (DDI_FAILURE);
360         } else {
361                 *hdl = tmp_rsrc_hdl;
362                 return (DDI_SUCCESS);
363         }
364 }


367 /*
368  * hermon_rsrc_fexch_alloc()
369  *
370  *    Context: Can only be called from base context.
371  *    The "sleepflag" parameter is used by all object allocators to
372  *    determine whether to SLEEP for resources or not.
373  */
374 static int
375 hermon_rsrc_fexch_alloc(hermon_state_t *state, hermon_rsrc_type_t rsrc,
376     uint_t num, uint_t sleepflag, hermon_rsrc_t *hdl)
377 {
378         hermon_fcoib_t          *fcoib;
379         void                    *addr;
380         uint32_t                fexch_qpn_base;
381         hermon_rsrc_pool_info_t *qpc_pool, *mpt_pool, *mtt_pool;
382         int                     flag, status;
383         hermon_rsrc_t           mpt_hdl; /* temporary, just for icm_confirm */
384         hermon_rsrc_t           mtt_hdl; /* temporary, just for icm_confirm */
385         uint_t                  portm1; /* hca_port_number - 1 */
386         uint_t                  nummtt;
387         vmem_t                  *vmp;
388
389         ASSERT(state != NULL);
390         ASSERT(hdl != NULL);
```

```
392         if ((state->hs_ibtfinfo.hca_attr->hca_flags2 & IBT_HCA2_FC) == 0)
393                 return (DDI_FAILURE);

395         portm1 = rsrc - HERMON_QPC_FEXCH_PORT1;
396         fcoib = &state->hs_fcoib;
397         flag = (sleepflag == HERMON_SLEEP) ? VM_SLEEP : VM_NOSLEEP;

399         /* Allocate from the FEXCH QP range */
400         vmp = fcoib->hfc_fexch_vmemp[portm1];
401         addr = vmem_xalloc(vmp, num, num, 0, 0, NULL, NULL, flag | VM_FIRSTFIT);
402         if (addr == NULL) {
403                 return (DDI_FAILURE);
404         }
405         fexch_qpn_base = (uint32_t)((uintptr_t)addr -
406             fcoib->hfc_vmemstart + fcoib->hfc_fexch_base[portm1]);

408         /* ICM confirm for the FEXCH QP range */
409         qpc_pool = &state->hs_rsrc_hdl[HERMON_QPC];
410         hdl->hr_len = num << qpc_pool->rsrc_shift;
411         hdl->hr_addr = addr;    /* used only for vmem_xfree */
412         hdl->hr_indx = fexch_qpn_base;

414         status = hermon_rsrc_hw_entry_icm_confirm(qpc_pool, num, hdl, 1);
415         if (status != DDI_SUCCESS) {
416                 vmem_xfree(vmp, addr, num);
417                 return (DDI_FAILURE);
418         }

420         /* ICM confirm for the Primary MKEYs (client side only) */
421         mpt_pool = &state->hs_rsrc_hdl[HERMON_DMPT];
422         mpt_hdl.hr_len = num << mpt_pool->rsrc_shift;
423         mpt_hdl.hr_addr = NULL;
424         mpt_hdl.hr_indx = fcoib->hfc_mpt_base[portm1] +
425             (fexch_qpn_base - fcoib->hfc_fexch_base[portm1]);

427         status = hermon_rsrc_hw_entry_icm_confirm(mpt_pool, num, &mpt_hdl, 0);
428         if (status != DDI_SUCCESS) {
429                 status = hermon_rsrc_hw_entry_icm_free(qpc_pool, hdl, 1);
430                 vmem_xfree(vmp, addr, num);
431                 return (DDI_FAILURE);
432         }

434         /* ICM confirm for the MTTs of the Primary MKEYs (client side only) */
435         nummtt = fcoib->hfc_mtts_per_mpt;
436         num *= nummtt;
437         mtt_pool = &state->hs_rsrc_hdl[HERMON_MTT];
438         mtt_hdl.hr_len = num << mtt_pool->rsrc_shift;
439         mtt_hdl.hr_addr = NULL;
440         mtt_hdl.hr_indx = fcoib->hfc_mtt_base[portm1] +
441             (fexch_qpn_base - fcoib->hfc_fexch_base[portm1]) *
442             nummtt;

444         status = hermon_rsrc_hw_entry_icm_confirm(mtt_pool, num, &mtt_hdl, 0);
445         if (status != DDI_SUCCESS) {
446                 vmem_xfree(vmp, addr, num);
447                 return (DDI_FAILURE);
448         }
449         return (DDI_SUCCESS);
450 }

452 static void
453 hermon_rsrc_fexch_free(hermon_state_t *state, hermon_rsrc_t *hdl)
454 {
455         hermon_fcoib_t          *fcoib;
456         uint_t                  portm1; /* hca_port_number - 1 */
```



```
458         ASSERT(state != NULL);
459         ASSERT(hdl != NULL);

461         portm1 = hdl->rsrc_type - HERMON_QPC_FEXCH_PORT1;
462         fcoib = &state->hs_fcoib;
463         vmem_xfree(fcoib->hfc_fexch_vmemp[portm1], hdl->hr_addr,
464             hdl->hr_len >> state->hs_rsrc_hdl[HERMON_QPC].rsrc_shift);
465 }

467 /*
468  * hermon_rsrc_rfci_alloc()
469  *
470  *    Context: Can only be called from base context.
471  *    The "sleepflag" parameter is used by all object allocators to
472  *    determine whether to SLEEP for resources or not.
473  */
474 static int
475 hermon_rsrc_rfci_alloc(hermon_state_t *state, hermon_rsrc_type_t rsrc,
476     uint_t num, uint_t sleepflag, hermon_rsrc_t *hdl)
477 {
478         hermon_fcoib_t          *fcoib;
479         void                    *addr;
480         uint32_t                rfci_qpn_base;
481         hermon_rsrc_pool_info_t *qpc_pool;
482         int                     flag, status;
483         uint_t                  portm1; /* hca_port_number - 1 */
484         vmem_t                  *vmp;

486         ASSERT(state != NULL);
487         ASSERT(hdl != NULL);

489         if ((state->hs_ibtfinfo.hca_attr->hca_flags2 & IBT_HCA2_FC) == 0)
490                 return (DDI_FAILURE);

492         portm1 = rsrc - HERMON_QPC_RFCI_PORT1;
493         fcoib = &state->hs_fcoib;
494         flag = (sleepflag == HERMON_SLEEP) ? VM_SLEEP : VM_NOSLEEP;

496         /* Allocate from the RFCI QP range */
497         vmp = fcoib->hfc_rfci_vmemp[portm1];
498         addr = vmem_xalloc(vmp, num, num, 0, 0, NULL, NULL, flag | VM_FIRSTFIT);
499         if (addr == NULL) {
500                 return (DDI_FAILURE);
501         }
502         rfci_qpn_base = (uint32_t)((uintptr_t)addr -
503             fcoib->hfc_vmemstart + fcoib->hfc_rfci_base[portm1]);

505         /* ICM confirm for the RFCI QP */
506         qpc_pool = &state->hs_rsrc_hdl[HERMON_QPC];
507         hdl->hr_len = num << qpc_pool->rsrc_shift;
508         hdl->hr_addr = addr;    /* used only for vmem_xfree */
509         hdl->hr_indx = rfci_qpn_base;

511         status = hermon_rsrc_hw_entry_icm_confirm(qpc_pool, num, hdl, 1);
512         if (status != DDI_SUCCESS) {
513                 vmem_xfree(vmp, addr, num);
514                 return (DDI_FAILURE);
515         }
516         return (DDI_SUCCESS);
517 }

519 static void
520 hermon_rsrc_rfci_free(hermon_state_t *state, hermon_rsrc_t *hdl)
521 {
522         hermon_fcoib_t          *fcoib;
523         uint_t                  portm1; /* hca_port_number - 1 */
```

```
525         ASSERT(state != NULL);
526         ASSERT(hdl != NULL);

528         portm1 = hdl->rsrc_type - HERMON_QPC_RFCI_PORT1;
529         fcoib = &state->hs_fcoib;
530         vmem_xfree(fcoib->hfc_rfci_vmemp[portm1], hdl->hr_addr,
531             hdl->hr_len >> state->hs_rsrc_hdl[HERMON_QPC].rsrc_shift);
532 }


535 /*
536  * hermon_rsrc_free()
537  *    Context: Can be called from interrupt or base context.
538  */
539 void
540 hermon_rsrc_free(hermon_state_t *state, hermon_rsrc_t **hdl)
541 {
542         hermon_rsrc_pool_info_t *rsrc_pool;

544         ASSERT(state != NULL);
545         ASSERT(hdl != NULL);

547         rsrc_pool = &state->hs_rsrc_hdl[(*hdl)->rsrc_type];
548         ASSERT(rsrc_pool != NULL);

550         /*
551          * Depending on resource type, call the appropriate free routine
552          */
553         switch (rsrc_pool->rsrc_type) {
554         case HERMON_IN_MBOX:
555         case HERMON_OUT_MBOX:
556         case HERMON_INTR_IN_MBOX:
557         case HERMON_INTR_OUT_MBOX:
558                 hermon_rsrc_mbox_free(*hdl);
559                 break;

561         case HERMON_QPC_FEXCH_PORT1:
562         case HERMON_QPC_FEXCH_PORT2:
563                 hermon_rsrc_fexch_free(state, *hdl);
564                 break;

566         case HERMON_QPC_RFCI_PORT1:
567         case HERMON_QPC_RFCI_PORT2:
568                 hermon_rsrc_rfci_free(state, *hdl);
569                 break;

571         case HERMON_QPC:
572         case HERMON_CQC:
573         case HERMON_SRQC:
574         case HERMON_EQC:
575         case HERMON_DMPT:
576         case HERMON_MCG:
577         case HERMON_MTT:
578         case HERMON_UARPG:
579                 hermon_rsrc_hw_entry_free(rsrc_pool, *hdl);
580                 break;

582         case HERMON_MRHDL:
583         case HERMON_EQHDL:
584         case HERMON_CQHDL:
585         case HERMON_SRQHDL:
586         case HERMON_AHHDL:
587         case HERMON_QPHDL:
588         case HERMON_REFCNT:
589                 hermon_rsrc_swhdl_free(rsrc_pool, *hdl);
```



```
590                 break;

592         case HERMON_PDHDL:
593                 hermon_rsrc_pdhdl_free(rsrc_pool, *hdl);
594                 break;

596         case HERMON_RDB:
597         case HERMON_ALTC:
598         case HERMON_AUXC:
599         case HERMON_CMPT_QPC:
600         case HERMON_CMPT_SRQC:
601         case HERMON_CMPT_CQC:
602         case HERMON_CMPT_EQC:
603         default:
604                 cmn_err(CE_CONT, "!rsrc_type = 0x%x\n", rsrc_pool->rsrc_type);
605                 break;
606         }

608         /*
609          * Free the special resource tracking structure, set the handle to
610          * NULL, and return.
611          */
612         kmem_cache_free(state->hs_rsrc_cache, *hdl);
613         *hdl = NULL;
614 }


617 /*
618  * hermon_rsrc_init_phase1()
619  *
620  *    Completes the first phase of Hermon resource/configuration init.
621  *    This involves creating the kmem_cache for the "hermon_rsrc_t"
622  *    structs, allocating the space for the resource pool handles,
623  *    and setting up the "Out" mailboxes.
624  *
625  *    When this function completes, the Hermon driver is ready to
626  *    post the following commands which return information only in the
627  *    "Out" mailbox: QUERY_DDR, QUERY_FW, QUERY_DEV_LIM, and QUERY_ADAPTER
628  *    If any of these commands are to be posted at this time, they must be
629  *    done so only when "spinning" (as the outstanding command list and
630  *    EQ setup code has not yet run)
631  *
632  *    Context: Only called from attach() path context
633  */
634 int
635 hermon_rsrc_init_phase1(hermon_state_t *state)
636 {
637         hermon_rsrc_pool_info_t         *rsrc_pool;
638         hermon_rsrc_mbox_info_t                 mbox_info;
639         hermon_rsrc_cleanup_level_t     cleanup;
640         hermon_cfg_profile_t            *cfgprof;
641         uint64_t                        num, size;
642         int                             status;
643         char                            *rsrc_name;

645         ASSERT(state != NULL);

647         /* This is where Phase 1 of resource initialization begins */
648         cleanup = HERMON_RSRC_CLEANUP_LEVEL0;

650         /* Build kmem cache name from Hermon instance */
651         rsrc_name = kmem_zalloc(HERMON_RSRC_NAME_MAXLEN, KM_SLEEP);
652         HERMON_RSRC_NAME(rsrc_name, HERMON_RSRC_CACHE);

654         /*
655          * Create the kmem_cache for "hermon_rsrc_t" structures
```

```
656          * (kmem_cache_create will SLEEP until successful)
657          */
658         state->hs_rsrc_cache = kmem_cache_create(rsrc_name,
659             sizeof (hermon_rsrc_t), 0, NULL, NULL, NULL, NULL, NULL, 0);
660
661         /*
662          * Allocate an array of hermon_rsrc_pool_info_t's (used in all
663          * subsequent resource allocations)
664          */
665         state->hs_rsrc_hdl = kmem_zalloc(HERMON_NUM_RESOURCES *
666             sizeof (hermon_rsrc_pool_info_t), KM_SLEEP);
667
668         /* Pull in the configuration profile */
669         cfgprof = state->hs_cfg_profile;
670
671         /* Initialize the resource pool for "out" mailboxes */
672         num  =  ((uint64_t)1 << cfgprof->cp_log_num_outmbox);
673         size =  ((uint64_t)1 << cfgprof->cp_log_outmbox_size);
674         rsrc_pool = &state->hs_rsrc_hdl[HERMON_OUT_MBOX];
675         rsrc_pool->rsrc_loc       = HERMON_IN_SYSMEM;
676         rsrc_pool->rsrc_pool_size = (size * num);
677         rsrc_pool->rsrc_shift     = cfgprof->cp_log_outmbox_size;
678         rsrc_pool->rsrc_quantum   = (uint_t)size;
679         rsrc_pool->rsrc_align     = HERMON_MBOX_ALIGN;
680         rsrc_pool->rsrc_state     = state;
681         mbox_info.mbi_num         = num;
682         mbox_info.mbi_size        = size;
683         mbox_info.mbi_rsrcpool    = rsrc_pool;
684         status = hermon_rsrc_mbox_init(state, &mbox_info);
685         if (status != DDI_SUCCESS) {
686                 hermon_rsrc_fini(state, cleanup);
687                 status = DDI_FAILURE;
688                 goto rsrcinitp1_fail;
689         }
690         cleanup = HERMON_RSRC_CLEANUP_LEVEL1;
691
692         /* Initialize the mailbox list */
693         status = hermon_outmbox_list_init(state);
694         if (status != DDI_SUCCESS) {
695                 hermon_rsrc_fini(state, cleanup);
696                 status = DDI_FAILURE;
697                 goto rsrcinitp1_fail;
698         }
699         cleanup = HERMON_RSRC_CLEANUP_LEVEL2;
700
701         /* Initialize the resource pool for "interrupt out" mailboxes */
702         num  =  ((uint64_t)1 << cfgprof->cp_log_num_intr_outmbox);
703         size =  ((uint64_t)1 << cfgprof->cp_log_outmbox_size);
704         rsrc_pool = &state->hs_rsrc_hdl[HERMON_INTR_OUT_MBOX];
705         rsrc_pool->rsrc_loc       = HERMON_IN_SYSMEM;
706         rsrc_pool->rsrc_pool_size = (size * num);
707         rsrc_pool->rsrc_shift     = cfgprof->cp_log_outmbox_size;
708         rsrc_pool->rsrc_quantum   = (uint_t)size;
709         rsrc_pool->rsrc_align     = HERMON_MBOX_ALIGN;
710         rsrc_pool->rsrc_state     = state;
711         mbox_info.mbi_num         = num;
712         mbox_info.mbi_size        = size;
713         mbox_info.mbi_rsrcpool    = rsrc_pool;
714         status = hermon_rsrc_mbox_init(state, &mbox_info);
715         if (status != DDI_SUCCESS) {
716                 hermon_rsrc_fini(state, cleanup);
717                 status = DDI_FAILURE;
718                 goto rsrcinitp1_fail;
719         }
720         cleanup = HERMON_RSRC_CLEANUP_LEVEL3;
```



```
722         /* Initialize the mailbox list */
723         status = hermon_intr_outmbox_list_init(state);
724         if (status != DDI_SUCCESS) {
725                 hermon_rsrc_fini(state, cleanup);
726                 status = DDI_FAILURE;
727                 goto rsrcinitp1_fail;
728         }
729         cleanup = HERMON_RSRC_CLEANUP_LEVEL4;
730
731         /* Initialize the resource pool for "in" mailboxes */
732         num  =  ((uint64_t)1 << cfgprof->cp_log_num_inmbox);
733         size =  ((uint64_t)1 << cfgprof->cp_log_inmbox_size);
734         rsrc_pool = &state->hs_rsrc_hdl[HERMON_IN_MBOX];
735         rsrc_pool->rsrc_loc       = HERMON_IN_SYSMEM;
736         rsrc_pool->rsrc_pool_size = (size * num);
737         rsrc_pool->rsrc_shift     = cfgprof->cp_log_inmbox_size;
738         rsrc_pool->rsrc_quantum   = (uint_t)size;
739         rsrc_pool->rsrc_align     = HERMON_MBOX_ALIGN;
740         rsrc_pool->rsrc_state     = state;
741         mbox_info.mbi_num         = num;
742         mbox_info.mbi_size        = size;
743         mbox_info.mbi_rsrcpool    = rsrc_pool;
744         status = hermon_rsrc_mbox_init(state, &mbox_info);
745         if (status != DDI_SUCCESS) {
746                 hermon_rsrc_fini(state, cleanup);
747                 status = DDI_FAILURE;
748                 goto rsrcinitp1_fail;
749         }
750         cleanup = HERMON_RSRC_CLEANUP_LEVEL5;
751
752         /* Initialize the mailbox list */
753         status = hermon_inmbox_list_init(state);
754         if (status != DDI_SUCCESS) {
755                 hermon_rsrc_fini(state, cleanup);
756                 status = DDI_FAILURE;
757                 goto rsrcinitp1_fail;
758         }
759         cleanup = HERMON_RSRC_CLEANUP_LEVEL6;
760
761         /* Initialize the resource pool for "interrupt in" mailboxes */
762         num  =  ((uint64_t)1 << cfgprof->cp_log_num_intr_inmbox);
763         size =  ((uint64_t)1 << cfgprof->cp_log_inmbox_size);
764         rsrc_pool = &state->hs_rsrc_hdl[HERMON_INTR_IN_MBOX];
765         rsrc_pool->rsrc_loc       = HERMON_IN_SYSMEM;
766         rsrc_pool->rsrc_pool_size = (size * num);
767         rsrc_pool->rsrc_shift     = cfgprof->cp_log_inmbox_size;
768         rsrc_pool->rsrc_quantum   = (uint_t)size;
769         rsrc_pool->rsrc_align     = HERMON_MBOX_ALIGN;
770         rsrc_pool->rsrc_state     = state;
771         mbox_info.mbi_num         = num;
772         mbox_info.mbi_size        = size;
773         mbox_info.mbi_rsrcpool    = rsrc_pool;
774         status = hermon_rsrc_mbox_init(state, &mbox_info);
775         if (status != DDI_SUCCESS) {
776                 hermon_rsrc_fini(state, cleanup);
777                 status = DDI_FAILURE;
778                 goto rsrcinitp1_fail;
779         }
780         cleanup = HERMON_RSRC_CLEANUP_LEVEL7;
781
782         /* Initialize the mailbox list */
783         status = hermon_intr_inmbox_list_init(state);
784         if (status != DDI_SUCCESS) {
785                 hermon_rsrc_fini(state, cleanup);
786                 status = DDI_FAILURE;
787                 goto rsrcinitp1_fail;
```

```
788         }
789         cleanup = HERMON_RSRC_CLEANUP_PHASE1_COMPLETE;
790         kmem_free(rsrc_name, HERMON_RSRC_NAME_MAXLEN);
791         return (DDI_SUCCESS);

793 rsrcinitp1_fail:
794         kmem_free(rsrc_name, HERMON_RSRC_NAME_MAXLEN);
795         return (status);
796 }


799 /*
800  * hermon_rsrc_init_phase2()
801  *    Context: Only called from attach() path context
802  */
803 int
804 hermon_rsrc_init_phase2(hermon_state_t *state)
805 {
806         hermon_rsrc_sw_hdl_info_t       hdl_info;
807         hermon_rsrc_hw_entry_info_t     entry_info;
808         hermon_rsrc_pool_info_t         *rsrc_pool;
809         hermon_rsrc_cleanup_level_t     cleanup, ncleanup;
810         hermon_cfg_profile_t            *cfgprof;
811         hermon_hw_querydevlim_t         *devlim;
812         uint64_t                        num, max, num_prealloc;
813         uint_t                          mcg_size, mcg_size_shift;
814         int                             i, status;
815         char                            *rsrc_name;

817         ASSERT(state != NULL);

819         /* Phase 2 initialization begins where Phase 1 left off */
820         cleanup = HERMON_RSRC_CLEANUP_PHASE1_COMPLETE;

822         /* Allocate the ICM resource name space */

824         /* Build the ICM vmem arena names from Hermon instance */
825         rsrc_name = kmem_zalloc(HERMON_RSRC_NAME_MAXLEN, KM_SLEEP);

827         /*
828          * Initialize the resource pools for all objects that exist in
829          * context memory (ICM). The ICM consists of context tables, each
830          * type of resource (QP, CQ, EQ, etc) having it's own context table
831          * (QPC, CQC, EQC, etc...).
832          */
833         cfgprof = state->hs_cfg_profile;
834         devlim  = &state->hs_devlim;

836         /*
837          * Initialize the resource pools for each of the driver resources.
838          * With a few exceptions, these resources fall into the two cateogories
839          * of either hw_entries or sw_entries.
840          */

842         /*
843          * Initialize the resource pools for ICM (hardware) types first.
844          * These resources are managed through vmem arenas, which are
845          * created via the rsrc pool initialization routine. Note that,
846          * due to further calculations, the MCG resource pool is
847          * initialized seperately.
848          */
849         for (i = 0; i < HERMON_NUM_ICM_RESOURCES; i++) {

851                 rsrc_pool = &state->hs_rsrc_hdl[i];
852                 rsrc_pool->rsrc_type = i;
853                 rsrc_pool->rsrc_state = state;
```



```
855                 /* Set the resource-specific attributes */
856                 switch (i) {
857                 case HERMON_MTT:
858                         max = ((uint64_t)1 << devlim->log_max_mtt);
859                         num_prealloc = ((uint64_t)1 << devlim->log_rsvd_mtt);
860                         HERMON_RSRC_NAME(rsrc_name, HERMON_MTT_VMEM);
861                         ncleanup = HERMON_RSRC_CLEANUP_LEVEL9;
862                         break;

864                 case HERMON_DMPT:
865                         max = ((uint64_t)1 << devlim->log_max_dmpt);
866                         num_prealloc = ((uint64_t)1 << devlim->log_rsvd_dmpt);
867                         HERMON_RSRC_NAME(rsrc_name, HERMON_DMPT_VMEM);
868                         ncleanup = HERMON_RSRC_CLEANUP_LEVEL10;
869                         break;

871                 case HERMON_QPC:
872                         max = ((uint64_t)1 << devlim->log_max_qp);
873                         num_prealloc = ((uint64_t)1 << devlim->log_rsvd_qp);
874                         HERMON_RSRC_NAME(rsrc_name, HERMON_QPC_VMEM);
875                         ncleanup = HERMON_RSRC_CLEANUP_LEVEL11;
876                         break;

878                 case HERMON_CQC:
879                         max = ((uint64_t)1 << devlim->log_max_cq);
880                         num_prealloc = ((uint64_t)1 << devlim->log_rsvd_cq);
881                         HERMON_RSRC_NAME(rsrc_name, HERMON_CQC_VMEM);
882                         ncleanup = HERMON_RSRC_CLEANUP_LEVEL13;
883                         break;

885                 case HERMON_SRQC:
886                         max = ((uint64_t)1 << devlim->log_max_srq);
887                         num_prealloc = ((uint64_t)1 << devlim->log_rsvd_srq);
888                         HERMON_RSRC_NAME(rsrc_name, HERMON_SRQC_VMEM);
889                         ncleanup = HERMON_RSRC_CLEANUP_LEVEL16;
890                         break;

892                 case HERMON_EQC:
893                         max = ((uint64_t)1 << devlim->log_max_eq);
894                         num_prealloc = state->hs_rsvd_eqs;
895                         HERMON_RSRC_NAME(rsrc_name, HERMON_EQC_VMEM);
896                         ncleanup = HERMON_RSRC_CLEANUP_LEVEL18;
897                         break;

899                 case HERMON_MCG:        /* handled below */
900                 case HERMON_AUXC:
901                 case HERMON_ALTC:
902                 case HERMON_RDB:
903                 case HERMON_CMPT_QPC:
904                 case HERMON_CMPT_SRQC:
905                 case HERMON_CMPT_CQC:
906                 case HERMON_CMPT_EQC:
907                 default:
908                         /* We don't need to initialize this rsrc here. */
909                         continue;
910                 }

912                 /* Set the common values for all resource pools */
913                 rsrc_pool->rsrc_state     = state;
914                 rsrc_pool->rsrc_loc       = HERMON_IN_ICM;
915                 rsrc_pool->rsrc_pool_size = state->hs_icm[i].table_size;
916                 rsrc_pool->rsrc_align     = state->hs_icm[i].table_size;
917                 rsrc_pool->rsrc_shift     = state->hs_icm[i].log_object_size;
918                 rsrc_pool->rsrc_quantum   = state->hs_icm[i].object_size;
```

```
920                 /* Now, initialize the entry_info and call the init routine */
921                 entry_info.hwi_num        = state->hs_icm[i].num_entries;
922                 entry_info.hwi_max        = max;
923                 entry_info.hwi_prealloc   = num_prealloc;
924                 entry_info.hwi_rsrcpool   = rsrc_pool;
925                 entry_info.hwi_rsrcname   = rsrc_name;
926                 status = hermon_rsrc_hw_entries_init(state, &entry_info);
927                 if (status != DDI_SUCCESS) {
928                         hermon_rsrc_fini(state, cleanup);
929                         status = DDI_FAILURE;
930                         goto rsrcinitp2_fail;
931                 }
932                 cleanup = ncleanup;
933         }

935         /*
936          * Initialize the Multicast Group (MCG) entries. First, calculate
937          * (and validate) the size of the MCGs.
938          */
939         status = hermon_rsrc_mcg_entry_get_size(state, &mcg_size_shift);
940         if (status != DDI_SUCCESS) {
941                 hermon_rsrc_fini(state, cleanup);
942                 status = DDI_FAILURE;
943                 goto rsrcinitp2_fail;
944         }
945         mcg_size = HERMON_MCGMEM_SZ(state);

947         /*
948          * Initialize the resource pool for the MCG table entries.  Notice
949          * that the number of MCGs is configurable. Note also that a certain
950          * number of MCGs must be set aside for Hermon firmware use (they
951          * correspond to the number of MCGs used by the internal hash
952          * function).
953          */
954         num                     = ((uint64_t)1 << cfgprof->cp_log_num_mcg);
955         max                     = ((uint64_t)1 << devlim->log_max_mcg);
956         num_prealloc      = ((uint64_t)1 << cfgprof->cp_log_num_mcg_hash);
957         rsrc_pool               = &state->hs_rsrc_hdl[HERMON_MCG];
958         rsrc_pool->rsrc_loc       = HERMON_IN_ICM;
959         rsrc_pool->rsrc_pool_size = (mcg_size * num);
960         rsrc_pool->rsrc_shift     = mcg_size_shift;
961         rsrc_pool->rsrc_quantum   = mcg_size;
962         rsrc_pool->rsrc_align     = (mcg_size * num);
963         rsrc_pool->rsrc_state     = state;
964         HERMON_RSRC_NAME(rsrc_name, HERMON_MCG_VMEM);
965         entry_info.hwi_num        = num;
966         entry_info.hwi_max        = max;
967         entry_info.hwi_prealloc   = num_prealloc;
968         entry_info.hwi_rsrcpool   = rsrc_pool;
969         entry_info.hwi_rsrcname   = rsrc_name;
970         status = hermon_rsrc_hw_entries_init(state, &entry_info);
971         if (status != DDI_SUCCESS) {
972                 hermon_rsrc_fini(state, cleanup);
973                 status = DDI_FAILURE;
974                 goto rsrcinitp2_fail;
975         }
976         cleanup = HERMON_RSRC_CLEANUP_LEVEL19;

978         /*
979          * Initialize the full range of ICM for the AUXC resource.
980          * This is done because its size is so small, about 1 byte per QP.
981          */

983         /*
984          * Initialize the Hermon command handling interfaces.  This step
985          * sets up the outstanding command tracking mechanism for easy access
```



```
986          * and fast allocation (see hermon_cmd.c for more details).
987          */
988         status = hermon_outstanding_cmdlist_init(state);
989         if (status != DDI_SUCCESS) {
990                 hermon_rsrc_fini(state, cleanup);
991                 status = DDI_FAILURE;
992                 goto rsrcinitp2_fail;
993         }
994         cleanup = HERMON_RSRC_CLEANUP_LEVEL20;

996         /* Initialize the resource pool and vmem arena for the PD handles */
997         rsrc_pool               = &state->hs_rsrc_hdl[HERMON_PDHDL];
998         rsrc_pool->rsrc_loc     = HERMON_IN_SYSMEM;
999         rsrc_pool->rsrc_quantum  = sizeof (struct hermon_sw_pd_s);
1000        rsrc_pool->rsrc_state    = state;
1001        HERMON_RSRC_NAME(rsrc_name, HERMON_PDHDL_CACHE);
1002        hdl_info.swi_num         = ((uint64_t)1 << cfgprof->cp_log_num_pd);
1003        hdl_info.swi_max         = ((uint64_t)1 << devlim->log_max_pd);
1004        hdl_info.swi_rsrcpool    = rsrc_pool;
1005        hdl_info.swi_constructor = hermon_rsrc_pdhdl_constructor;
1006        hdl_info.swi_destructor  = hermon_rsrc_pdhdl_destructor;
1007        hdl_info.swi_rsrcname    = rsrc_name;
1008        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1009        status = hermon_rsrc_pd_handles_init(state, &hdl_info);
1010        if (status != DDI_SUCCESS) {
1011                hermon_rsrc_fini(state, cleanup);
1012                status = DDI_FAILURE;
1013                goto rsrcinitp2_fail;
1014        }
1015        cleanup = HERMON_RSRC_CLEANUP_LEVEL21;

1017        /*
1018         * Initialize the resource pools for the rest of the software handles.
1019         * This includes MR handles, EQ handles, QP handles, etc.  These
1020         * objects are almost entirely managed using kmem_cache routines,
1021         * and do not utilize a vmem arena.
1022         */
1023        for (i = HERMON_NUM_ICM_RESOURCES; i < HERMON_NUM_RESOURCES; i++) {
1024                rsrc_pool = &state->hs_rsrc_hdl[i];
1025                rsrc_pool->rsrc_type = i;

1027                /* Set the resource-specific attributes */
1028                switch (i) {
1029                case HERMON_MRHDL:
1030                        rsrc_pool->rsrc_quantum =
1031                            sizeof (struct hermon_sw_mr_s);
1032                        HERMON_RSRC_NAME(rsrc_name, HERMON_MRHDL_CACHE);
1033                        hdl_info.swi_num =
1034                            ((uint64_t)1 << cfgprof->cp_log_num_dmpt) +
1035                            ((uint64_t)1 << cfgprof->cp_log_num_cmpt);
1036                        hdl_info.swi_max =
1037                            ((uint64_t)1 << cfgprof->cp_log_num_dmpt) +
1038                            ((uint64_t)1 << cfgprof->cp_log_num_cmpt);
1039                        hdl_info.swi_constructor =
1040                            hermon_rsrc_mrhdl_constructor;
1041                        hdl_info.swi_destructor  = hermon_rsrc_mrhdl_destructor;
1042                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1043                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL22;
1044                        break;

1046                case HERMON_EQHDL:
1047                        rsrc_pool->rsrc_quantum =
1048                            sizeof (struct hermon_sw_eq_s);
1049                        HERMON_RSRC_NAME(rsrc_name, HERMON_EQHDL_CACHE);
1050                        hdl_info.swi_num = HERMON_NUM_EQ;
1051                        hdl_info.swi_max = ((uint64_t)1 << devlim->log_max_eq);
```

```
1052                        hdl_info.swi_constructor = NULL;
1053                        hdl_info.swi_destructor  = NULL;
1054                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1055                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL23;
1056                        break;

1058                case HERMON_CQHDL:
1059                        rsrc_pool->rsrc_quantum =
1060                            sizeof (struct hermon_sw_cq_s);
1061                        HERMON_RSRC_NAME(rsrc_name, HERMON_CQHDL_CACHE);
1062                        hdl_info.swi_num =
1063                            (uint64_t)1 << cfgprof->cp_log_num_cq;
1064                        hdl_info.swi_max = (uint64_t)1 << devlim->log_max_cq;
1065                        hdl_info.swi_constructor =
1066                            hermon_rsrc_cqhdl_constructor;
1067                        hdl_info.swi_destructor  = hermon_rsrc_cqhdl_destructor;
1068                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1069                        hdl_info.swi_prealloc_sz = sizeof (hermon_cqhdl_t);
1070                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL24;
1071                        break;

1073                case HERMON_SRQHDL:
1074                        rsrc_pool->rsrc_quantum =
1075                            sizeof (struct hermon_sw_srq_s);
1076                        HERMON_RSRC_NAME(rsrc_name, HERMON_SRQHDL_CACHE);
1077                        hdl_info.swi_num =
1078                            (uint64_t)1 << cfgprof->cp_log_num_srq;
1079                        hdl_info.swi_max = (uint64_t)1 << devlim->log_max_srq;
1080                        hdl_info.swi_constructor =
1081                            hermon_rsrc_srqhdl_constructor;
1082                        hdl_info.swi_destructor = hermon_rsrc_srqhdl_destructor;
1083                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1084                        hdl_info.swi_prealloc_sz = sizeof (hermon_srqhdl_t);
1085                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL25;
1086                        break;

1088                case HERMON_AHHDL:
1089                        rsrc_pool->rsrc_quantum =
1090                            sizeof (struct hermon_sw_ah_s);
1091                        HERMON_RSRC_NAME(rsrc_name, HERMON_AHHDL_CACHE);
1092                        hdl_info.swi_num =
1093                            (uint64_t)1 << cfgprof->cp_log_num_ah;
1094                        hdl_info.swi_max = HERMON_NUM_AH;
1095                        hdl_info.swi_constructor =
1096                            hermon_rsrc_ahhdl_constructor;
1097                        hdl_info.swi_destructor  = hermon_rsrc_ahhdl_destructor;
1098                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1099                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL26;
1100                        break;

1102                case HERMON_QPHDL:
1103                        rsrc_pool->rsrc_quantum =
1104                            sizeof (struct hermon_sw_qp_s);
1105                        HERMON_RSRC_NAME(rsrc_name, HERMON_QPHDL_CACHE);
1106                        hdl_info.swi_num =
1107                            (uint64_t)1 << cfgprof->cp_log_num_qp;
1108                        hdl_info.swi_max = (uint64_t)1 << devlim->log_max_qp;
1109                        hdl_info.swi_constructor =
1110                            hermon_rsrc_qphdl_constructor;
1111                        hdl_info.swi_destructor = hermon_rsrc_qphdl_destructor;
1112                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1113                        hdl_info.swi_prealloc_sz = sizeof (hermon_qphdl_t);
1114                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL27;
1115                        break;

1117                case HERMON_REFCNT:
```



```
1118                        rsrc_pool->rsrc_quantum  = sizeof (hermon_sw_refcnt_t);
1119                        HERMON_RSRC_NAME(rsrc_name, HERMON_REFCNT_CACHE);
1120                        hdl_info.swi_num =
1121                            (uint64_t)1 << cfgprof->cp_log_num_dmpt;
1122                        hdl_info.swi_max = (uint64_t)1 << devlim->log_max_dmpt;
1123                        hdl_info.swi_constructor =
1124                            hermon_rsrc_refcnt_constructor;
1125                        hdl_info.swi_destructor = hermon_rsrc_refcnt_destructor;
1126                        hdl_info.swi_flags       = HERMON_SWHDL_KMEMCACHE_INIT;
1127                        ncleanup = HERMON_RSRC_CLEANUP_LEVEL28;
1128                        break;

1130                default:
1131                        continue;
1132                }

1134                /* Set the common values and call the init routine */
1135                rsrc_pool->rsrc_loc      = HERMON_IN_SYSMEM;
1136                rsrc_pool->rsrc_state    = state;
1137                hdl_info.swi_rsrcpool    = rsrc_pool;
1138                hdl_info.swi_rsrcname    = rsrc_name;
1139                status = hermon_rsrc_sw_handles_init(state, &hdl_info);
1140                if (status != DDI_SUCCESS) {
1141                        hermon_rsrc_fini(state, cleanup);
1142                        status = DDI_FAILURE;
1143                        goto rsrcinitp2_fail;
1144                }
1145                cleanup = ncleanup;
1146        }

1148        /*
1149         * Initialize a resource pool for the MCG handles.  Notice that for
1150         * these MCG handles, we are allocating a table of structures (used to
1151         * keep track of the MCG entries that are being written to hardware
1152         * and to speed up multicast attach/detach operations).
1153         */
1154        hdl_info.swi_num         = ((uint64_t)1 << cfgprof->cp_log_num_mcg);
1155        hdl_info.swi_max         = ((uint64_t)1 << devlim->log_max_mcg);
1156        hdl_info.swi_flags       = HERMON_SWHDL_TABLE_INIT;
1157        hdl_info.swi_prealloc_sz = sizeof (struct hermon_sw_mcg_list_s);
1158        status = hermon_rsrc_sw_handles_init(state, &hdl_info);
1159        if (status != DDI_SUCCESS) {
1160                hermon_rsrc_fini(state, cleanup);
1161                status = DDI_FAILURE;
1162                goto rsrcinitp2_fail;
1163        }
1164        state->hs_mcghdl = hdl_info.swi_table_ptr;
1165        cleanup = HERMON_RSRC_CLEANUP_LEVEL29;

1167        /*
1168         * Last, initialize the resource pool for the UAR pages, which contain
1169         * the hardware's doorbell registers. Each process supported in User
1170         * Mode is assigned a UAR page. Also coming from this pool are the
1171         * kernel-assigned UAR page, and any hardware-reserved pages. Note
1172         * that the number of UAR pages is configurable, the value must be less
1173         * than the maximum value (obtained from the QUERY_DEV_LIM command) or
1174         * the initialization will fail.  Note also that we assign the base
1175         * address of the UAR BAR to the rsrc_start parameter.
1176         */
1177        num                      = ((uint64_t)1 << cfgprof->cp_log_num_uar);
1178        max                      = num;
1179        num_prealloc             = max(devlim->num_rsvd_uar, 128);
1180        rsrc_pool                = &state->hs_rsrc_hdl[HERMON_UARPG];
1181        rsrc_pool->rsrc_loc      = HERMON_IN_UAR;
1182        rsrc_pool->rsrc_pool_size = (num << PAGESHIFT);
1183        rsrc_pool->rsrc_shift    = PAGESHIFT;
```

```
1184         rsrc_pool->rsrc_quantum   = (uint_t)PAGESIZE;
1185         rsrc_pool->rsrc_align     = PAGESIZE;
1186         rsrc_pool->rsrc_state     = state;
1187         rsrc_pool->rsrc_start     = (void *)state->hs_reg_uar_baseaddr;
1188         HERMON_RSRC_NAME(rsrc_name, HERMON_UAR_PAGE_VMEM_ATTCH);
1189         entry_info.hwi_num        = num;
1190         entry_info.hwi_max        = max;
1191         entry_info.hwi_prealloc   = num_prealloc;
1192         entry_info.hwi_rsrcpool   = rsrc_pool;
1193         entry_info.hwi_rsrcname   = rsrc_name;
1194         status = hermon_rsrc_hw_entries_init(state, &entry_info);
1195         if (status != DDI_SUCCESS) {
1196                 hermon_rsrc_fini(state, cleanup);
1197                 status = DDI_FAILURE;
1198                 goto rsrcinitp2_fail;
1199         }

1201         cleanup = HERMON_RSRC_CLEANUP_ALL;

1203         kmem_free(rsrc_name, HERMON_RSRC_NAME_MAXLEN);
1204         return (DDI_SUCCESS);

1206 rsrcinitp2_fail:
1207         kmem_free(rsrc_name, HERMON_RSRC_NAME_MAXLEN);
1208         return (status);
1209 }



1212 /*
1213  * hermon_rsrc_fini()
1214  *    Context: Only called from attach() and/or detach() path contexts
1215  */
1216 void
1217 hermon_rsrc_fini(hermon_state_t *state, hermon_rsrc_cleanup_level_t clean)
1218 {
1219         hermon_rsrc_sw_hdl_info_t       hdl_info;
1220         hermon_rsrc_hw_entry_info_t     entry_info;
1221         hermon_rsrc_mbox_info_t         mbox_info;
1222         hermon_cfg_profile_t            *cfgprof;

1224         ASSERT(state != NULL);

1226         cfgprof = state->hs_cfg_profile;

1228         /*
1229          * If init code above is shortened up (see comments), then we
1230          * need to establish how to safely and simply clean up from any
1231          * given failure point. Flags, maybe...
1232          */

1234         switch (clean) {
1235         /*
1236          * If we add more resources that need to be cleaned up here, we should
1237          * ensure that HERMON_RSRC_CLEANUP_ALL is still the first entry (i.e.
1238          * corresponds to the last resource allocated).
1239          */

1241         case HERMON_RSRC_CLEANUP_ALL:
1242         case HERMON_RSRC_CLEANUP_LEVEL31:
1243                 /* Cleanup the UAR page resource pool, first the dbr pages */
1244                 if (state->hs_kern_dbr) {
1245                         hermon_dbr_kern_free(state);
1246                         state->hs_kern_dbr = NULL;
1247                 }

1249                 /* NS then, the pool itself */
```



```
1250                 entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_UARPG];
1251                 hermon_rsrc_hw_entries_fini(state, &entry_info);

1253                 /* FALLTHROUGH */

1255         case HERMON_RSRC_CLEANUP_LEVEL30:
1256                 /* Cleanup the central MCG handle pointers list */
1257                 hdl_info.swi_rsrcpool  = NULL;
1258                 hdl_info.swi_table_ptr = state->hs_mcghdl;
1259                 hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_mcg);
1260                 hdl_info.swi_prealloc_sz = sizeof (struct hermon_sw_mcg_list_s);
1261                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1262                 /* FALLTHROUGH */

1264         case HERMON_RSRC_CLEANUP_LEVEL29:
1265                 /* Cleanup the reference count resource pool */
1266                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_REFCNT];
1267                 hdl_info.swi_table_ptr = NULL;
1268                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1269                 /* FALLTHROUGH */

1271         case HERMON_RSRC_CLEANUP_LEVEL28:
1272                 /* Cleanup the QP handle resource pool */
1273                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_QPHDL];
1274                 hdl_info.swi_table_ptr = NULL;
1275                 hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_qp);
1276                 hdl_info.swi_prealloc_sz = sizeof (hermon_qphdl_t);
1277                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1278                 /* FALLTHROUGH */
1279         case HERMON_RSRC_CLEANUP_LEVEL27:
1280                 /* Cleanup the address handle resrouce pool */
1281                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_AHHDL];
1282                 hdl_info.swi_table_ptr = NULL;
1283                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1284                 /* FALLTHROUGH */

1286         case HERMON_RSRC_CLEANUP_LEVEL26:
1287                 /* Cleanup the SRQ handle resource pool. */
1288                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_SRQHDL];
1289                 hdl_info.swi_table_ptr = NULL;
1290                 hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_srq);
1291                 hdl_info.swi_prealloc_sz = sizeof (hermon_srqhdl_t);
1292                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1293                 /* FALLTHROUGH */

1295         case HERMON_RSRC_CLEANUP_LEVEL25:
1296                 /* Cleanup the CQ handle resource pool */
1297                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_CQHDL];
1298                 hdl_info.swi_table_ptr = NULL;
1299                 hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_cq);
1300                 hdl_info.swi_prealloc_sz = sizeof (hermon_cqhdl_t);
1301                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1302                 /* FALLTHROUGH */

1304         case HERMON_RSRC_CLEANUP_LEVEL24:
1305                 /* Cleanup the EQ handle resource pool */
1306                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_EQHDL];
1307                 hdl_info.swi_table_ptr = NULL;
1308                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
1309                 /* FALLTHROUGH */

1311         case HERMON_RSRC_CLEANUP_LEVEL23:
1312                 /* Cleanup the MR handle resource pool */
1313                 hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_MRHDL];
1314                 hdl_info.swi_table_ptr = NULL;
1315                 hermon_rsrc_sw_handles_fini(state, &hdl_info);
```

```
1316                /* FALLTHROUGH */
1317
1318        case HERMON_RSRC_CLEANUP_LEVEL22:
1319                /* Cleanup the PD handle resource pool */
1320                hdl_info.swi_rsrcpool  = &state->hs_rsrc_hdl[HERMON_PDHDL];
1321                hdl_info.swi_table_ptr = NULL;
1322                hermon_rsrc_pd_handles_fini(state, &hdl_info);
1323                /* FALLTHROUGH */
1324
1325        case HERMON_RSRC_CLEANUP_LEVEL21:
1326                /* Currently unused - FALLTHROUGH */
1327
1328        case HERMON_RSRC_CLEANUP_LEVEL20:
1329                /* Cleanup the outstanding command list  */
1330                hermon_outstanding_cmdlist_fini(state);
1331                /* FALLTHROUGH */
1332
1333        case HERMON_RSRC_CLEANUP_LEVEL19:
1334                /* Cleanup the EQC table resource pool */
1335                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_EQC];
1336                hermon_rsrc_hw_entries_fini(state, &entry_info);
1337                /* FALLTHROUGH */
1338
1339        case HERMON_RSRC_CLEANUP_LEVEL18:
1340                /* Cleanup the MCG table resource pool */
1341                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_MCG];
1342                hermon_rsrc_hw_entries_fini(state, &entry_info);
1343                /* FALLTHROUGH */
1344
1345        case HERMON_RSRC_CLEANUP_LEVEL17:
1346                /* Currently Unused - fallthrough */
1347        case HERMON_RSRC_CLEANUP_LEVEL16:
1348                /* Cleanup the SRQC table resource pool */
1349                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_SRQC];
1350                hermon_rsrc_hw_entries_fini(state, &entry_info);
1351                /* FALLTHROUGH */
1352
1353        case HERMON_RSRC_CLEANUP_LEVEL15:
1354                /* Cleanup the AUXC table resource pool */
1355                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_AUXC];
1356                hermon_rsrc_hw_entries_fini(state, &entry_info);
1357                /* FALLTHROUGH */
1358
1359        case HERMON_RSRC_CLEANUP_LEVEL14:
1360                /* Cleanup the ALTCF table resource pool */
1361                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_ALTC];
1362                hermon_rsrc_hw_entries_fini(state, &entry_info);
1363                /* FALLTHROUGH */
1364
1365        case HERMON_RSRC_CLEANUP_LEVEL13:
1366                /* Cleanup the CQC table resource pool */
1367                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_CQC];
1368                hermon_rsrc_hw_entries_fini(state, &entry_info);
1369                /* FALLTHROUGH */
1370
1371        case HERMON_RSRC_CLEANUP_LEVEL12:
1372                /* Cleanup the RDB table resource pool */
1373                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_RDB];
1374                hermon_rsrc_hw_entries_fini(state, &entry_info);
1375                /* FALLTHROUGH */
1376
1377        case HERMON_RSRC_CLEANUP_LEVEL11:
1378                /* Cleanup the QPC table resource pool */
1379                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_QPC];
1380                hermon_rsrc_hw_entries_fini(state, &entry_info);
1381                /* FALLTHROUGH */
```



```
1383        case HERMON_RSRC_CLEANUP_LEVEL10EQ:
1384                /* Cleanup the cMPTs for the EQs, CQs, SRQs, and QPs */
1385                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_CMPT_EQC];
1386                hermon_rsrc_hw_entries_fini(state, &entry_info);
1387                /* FALLTHROUGH */
1388
1389        case HERMON_RSRC_CLEANUP_LEVEL10CQ:
1390                /* Cleanup the cMPTs for the EQs, CQs, SRQs, and QPs */
1391                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_CMPT_CQC];
1392                hermon_rsrc_hw_entries_fini(state, &entry_info);
1393                /* FALLTHROUGH */
1394
1395        case HERMON_RSRC_CLEANUP_LEVEL10SRQ:
1396                /* Cleanup the cMPTs for the EQs, CQs, SRQs, and QPs */
1397                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_CMPT_SRQC];
1398                hermon_rsrc_hw_entries_fini(state, &entry_info);
1399                /* FALLTHROUGH */
1400
1401        case HERMON_RSRC_CLEANUP_LEVEL10QP:
1402                /* Cleanup the cMPTs for the EQs, CQs, SRQs, and QPs */
1403                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_CMPT_QPC];
1404                hermon_rsrc_hw_entries_fini(state, &entry_info);
1405                /* FALLTHROUGH */
1406
1407        case HERMON_RSRC_CLEANUP_LEVEL10:
1408                /* Cleanup the dMPT table resource pool */
1409                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_DMPT];
1410                hermon_rsrc_hw_entries_fini(state, &entry_info);
1411                /* FALLTHROUGH */
1412
1413        case HERMON_RSRC_CLEANUP_LEVEL9:
1414                /* Cleanup the MTT table resource pool */
1415                entry_info.hwi_rsrcpool = &state->hs_rsrc_hdl[HERMON_MTT];
1416                hermon_rsrc_hw_entries_fini(state, &entry_info);
1417                break;
1418
1419        /*
1420         * The cleanup below comes from the "Phase 1" initialization step.
1421         * (see hermon_rsrc_init_phase1() above)
1422         */
1423        case HERMON_RSRC_CLEANUP_PHASE1_COMPLETE:
1424                /* Cleanup the "In" mailbox list  */
1425                hermon_intr_inmbox_list_fini(state);
1426                /* FALLTHROUGH */
1427
1428        case HERMON_RSRC_CLEANUP_LEVEL7:
1429                /* Cleanup the interrupt "In" mailbox resource pool */
1430                mbox_info.mbi_rsrcpool =
1431                    &state->hs_rsrc_hdl[HERMON_INTR_IN_MBOX];
1432                hermon_rsrc_mbox_fini(state, &mbox_info);
1433                /* FALLTHROUGH */
1434
1435        case HERMON_RSRC_CLEANUP_LEVEL6:
1436                /* Cleanup the "In" mailbox list  */
1437                hermon_inmbox_list_fini(state);
1438                /* FALLTHROUGH */
1439
1440        case HERMON_RSRC_CLEANUP_LEVEL5:
1441                /* Cleanup the "In" mailbox resource pool */
1442                mbox_info.mbi_rsrcpool = &state->hs_rsrc_hdl[HERMON_IN_MBOX];
1443                hermon_rsrc_mbox_fini(state, &mbox_info);
1444                /* FALLTHROUGH */
1445
1446        case HERMON_RSRC_CLEANUP_LEVEL4:
1447                /* Cleanup the interrupt "Out" mailbox list  */
```

```
1448                 hermon_intr_outmbox_list_fini(state);
1449                 /* FALLTHROUGH */

1451         case HERMON_RSRC_CLEANUP_LEVEL3:
1452                 /* Cleanup the "Out" mailbox resource pool */
1453                 mbox_info.mbi_rsrcpool =
1454                     &state->hs_rsrc_hdl[HERMON_INTR_OUT_MBOX];
1455                 hermon_rsrc_mbox_fini(state, &mbox_info);
1456                 /* FALLTHROUGH */

1458         case HERMON_RSRC_CLEANUP_LEVEL2:
1459                 /* Cleanup the "Out" mailbox list  */
1460                 hermon_outmbox_list_fini(state);
1461                 /* FALLTHROUGH */

1463         case HERMON_RSRC_CLEANUP_LEVEL1:
1464                 /* Cleanup the "Out" mailbox resource pool */
1465                 mbox_info.mbi_rsrcpool = &state->hs_rsrc_hdl[HERMON_OUT_MBOX];
1466                 hermon_rsrc_mbox_fini(state, &mbox_info);
1467                 /* FALLTHROUGH */

1469         case HERMON_RSRC_CLEANUP_LEVEL0:
1470                 /* Free the array of hermon_rsrc_pool_info_t's */

1472                 kmem_free(state->hs_rsrc_hdl, HERMON_NUM_RESOURCES *
1473                     sizeof (hermon_rsrc_pool_info_t));

1475                 kmem_cache_destroy(state->hs_rsrc_cache);
1476                 break;

1478         default:
1479                 HERMON_WARNING(state, "unexpected resource cleanup level");
1480                 break;
1481         }
1482 }



1485 /*
1486  * hermon_rsrc_mbox_init()
1487  *    Context: Only called from attach() path context
1488  */
1489 static int
1490 hermon_rsrc_mbox_init(hermon_state_t *state, hermon_rsrc_mbox_info_t *info)
1491 {
1492         hermon_rsrc_pool_info_t *rsrc_pool;
1493         hermon_rsrc_priv_mbox_t *priv;

1495         ASSERT(state != NULL);
1496         ASSERT(info != NULL);

1498         rsrc_pool = info->mbi_rsrcpool;
1499         ASSERT(rsrc_pool != NULL);

1501         /* Allocate and initialize mailbox private structure */
1502         priv = kmem_zalloc(sizeof (hermon_rsrc_priv_mbox_t), KM_SLEEP);
1503         priv->pmb_dip           = state->hs_dip;
1504         priv->pmb_devaccattr    = state->hs_reg_accattr;
1505         priv->pmb_xfer_mode     = DDI_DMA_CONSISTENT;

1507         /*
1508          * Initialize many of the default DMA attributes.  Then set alignment
1509          * and scatter-gather restrictions specific for mailbox memory.
1510          */
1511         hermon_dma_attr_init(state, &priv->pmb_dmaattr);
1512         priv->pmb_dmaattr.dma_attr_align  = HERMON_MBOX_ALIGN;
1513         priv->pmb_dmaattr.dma_attr_sgllen = 1;
```



```
1514         priv->pmb_dmaattr.dma_attr_flags = 0;
1515         rsrc_pool->rsrc_private = priv;

1517         ASSERT(rsrc_pool->rsrc_loc == HERMON_IN_SYSMEM);

1519         rsrc_pool->rsrc_start = NULL;
1520         rsrc_pool->rsrc_vmp = NULL;

1522         return (DDI_SUCCESS);
1523 }


1526 /*
1527  * hermon_rsrc_mbox_fini()
1528  *    Context: Only called from attach() and/or detach() path contexts
1529  */
1530 /* ARGSUSED */
1531 static void
1532 hermon_rsrc_mbox_fini(hermon_state_t *state, hermon_rsrc_mbox_info_t *info)
1533 {
1534         hermon_rsrc_pool_info_t *rsrc_pool;

1536         ASSERT(state != NULL);
1537         ASSERT(info != NULL);

1539         rsrc_pool = info->mbi_rsrcpool;
1540         ASSERT(rsrc_pool != NULL);

1542         /* Free up the private struct */
1543         kmem_free(rsrc_pool->rsrc_private, sizeof (hermon_rsrc_priv_mbox_t));
1544 }


1547 /*
1548  * hermon_rsrc_hw_entries_init()
1549  *    Context: Only called from attach() path context
1550  */
1551 int
1552 hermon_rsrc_hw_entries_init(hermon_state_t *state,
1553     hermon_rsrc_hw_entry_info_t *info)
1554 {
1555         hermon_rsrc_pool_info_t *rsrc_pool;
1556         hermon_rsrc_t           *rsvd_rsrc = NULL;
1557         vmem_t                  *vmp;
1558         uint64_t                num_hwentry, max_hwentry, num_prealloc;
1559         int                     status;

1561         ASSERT(state != NULL);
1562         ASSERT(info != NULL);

1564         rsrc_pool       = info->hwi_rsrcpool;
1565         ASSERT(rsrc_pool != NULL);
1566         num_hwentry     = info->hwi_num;
1567         max_hwentry     = info->hwi_max;
1568         num_prealloc    = info->hwi_prealloc;

1570         if (hermon_rsrc_verbose) {
1571                 IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entries_init: "
1572                     "rsrc_type (0x%x) num (%llx) max (0x%llx) prealloc "
1573                     "(0x%llx)", rsrc_pool->rsrc_type, (longlong_t)num_hwentry,
1574                     (longlong_t)max_hwentry, (longlong_t)num_prealloc);
1575         }

1577         /* Make sure number of HW entries makes sense */
1578         if (num_hwentry > max_hwentry) {
1579                 return (DDI_FAILURE);
```

```
1580         }
1582         /* Set this pool's rsrc_start from the initial ICM allocation */
1583         if (rsrc_pool->rsrc_start == 0) {
1585                 /* use a ROUND value that works on both 32 and 64-bit kernels */
1586                 rsrc_pool->rsrc_start = (void *)(uintptr_t)0x10000000;
1588                 if (hermon_rsrc_verbose) {
1589                         IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entries_init:"
1590                             " rsrc_type (0x%x) rsrc_start set (0x%lx)",
1591                             rsrc_pool->rsrc_type, rsrc_pool->rsrc_start);
1592                 }
1593         }
1595         /*
1596          * Create new vmem arena for the HW entries table if rsrc_quantum
1597          * is non-zero.  Otherwise if rsrc_quantum is zero, then these HW
1598          * entries are not going to be dynamically allocatable (i.e. they
1599          * won't be allocated/freed through hermon_rsrc_alloc/free).  This
1600          * latter option is used for both ALTC and CMPT resources which
1601          * are managed by hardware.
1602          */
1603         if (rsrc_pool->rsrc_quantum != 0) {
1604                 vmp = vmem_create(info->hwi_rsrcname,
1605                     (void *)(uintptr_t)rsrc_pool->rsrc_start,
1606                     rsrc_pool->rsrc_pool_size, rsrc_pool->rsrc_quantum,
1607                     NULL, NULL, NULL, 0, VM_SLEEP);
1608                 if (vmp == NULL) {
1609                         /* failed to create vmem arena */
1610                         return (DDI_FAILURE);
1611                 }
1612                 rsrc_pool->rsrc_vmp = vmp;
1613                 if (hermon_rsrc_verbose) {
1614                         IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entries_init:"
1615                             " rsrc_type (0x%x) created vmem arena for rsrc",
1616                             rsrc_pool->rsrc_type);
1617                 }
1618         } else {
1619                 /* we do not require a vmem arena */
1620                 rsrc_pool->rsrc_vmp = NULL;
1621                 if (hermon_rsrc_verbose) {
1622                         IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entries_init:"
1623                             " rsrc_type (0x%x) vmem arena not required",
1624                             rsrc_pool->rsrc_type);
1625                 }
1626         }
1628         /* Allocate hardware reserved resources, if any */
1629         if (num_prealloc != 0) {
1630                 status = hermon_rsrc_alloc(state, rsrc_pool->rsrc_type,
1631                     num_prealloc, HERMON_SLEEP, &rsvd_rsrc);
1632                 if (status != DDI_SUCCESS) {
1633                         /* unable to preallocate the reserved entries */
1634                         if (rsrc_pool->rsrc_vmp != NULL) {
1635                                 vmem_destroy(rsrc_pool->rsrc_vmp);
1636                         }
1637                         return (DDI_FAILURE);
1638                 }
1639         }
1640         rsrc_pool->rsrc_private = rsvd_rsrc;
1642         return (DDI_SUCCESS);
1643 }
```



```
1646 /*
1647  * hermon_rsrc_hw_entries_fini()
1648  *    Context: Only called from attach() and/or detach() path contexts
1649  */
1650 void
1651 hermon_rsrc_hw_entries_fini(hermon_state_t *state,
1652     hermon_rsrc_hw_entry_info_t *info)
1653 {
1654         hermon_rsrc_pool_info_t *rsrc_pool;
1655         hermon_rsrc_t           *rsvd_rsrc;
1657         ASSERT(state != NULL);
1658         ASSERT(info != NULL);
1660         rsrc_pool = info->hwi_rsrcpool;
1661         ASSERT(rsrc_pool != NULL);
1663         /* Free up any "reserved" (i.e. preallocated) HW entries */
1664         rsvd_rsrc = (hermon_rsrc_t *)rsrc_pool->rsrc_private;
1665         if (rsvd_rsrc != NULL) {
1666                 hermon_rsrc_free(state, &rsvd_rsrc);
1667         }
1669         /*
1670          * If we've actually setup a vmem arena for the HW entries, then
1671          * destroy it now
1672          */
1673         if (rsrc_pool->rsrc_vmp != NULL) {
1674                 vmem_destroy(rsrc_pool->rsrc_vmp);
1675         }
1676 }
1679 /*
1680  * hermon_rsrc_sw_handles_init()
1681  *    Context: Only called from attach() path context
1682  */
1683 /* ARGSUSED */
1684 static int
1685 hermon_rsrc_sw_handles_init(hermon_state_t *state,
1686     hermon_rsrc_sw_hdl_info_t *info)
1687 {
1688         hermon_rsrc_pool_info_t *rsrc_pool;
1689         uint64_t                num_swhdl, max_swhdl, prealloc_sz;
1691         ASSERT(state != NULL);
1692         ASSERT(info != NULL);
1694         rsrc_pool       = info->swi_rsrcpool;
1695         ASSERT(rsrc_pool != NULL);
1696         num_swhdl       = info->swi_num;
1697         max_swhdl       = info->swi_max;
1698         prealloc_sz     = info->swi_prealloc_sz;
1701         /* Make sure number of SW handles makes sense */
1702         if (num_swhdl > max_swhdl) {
1703                 return (DDI_FAILURE);
1704         }
1706         /*
1707          * Depending on the flags parameter, create a kmem_cache for some
1708          * number of software handle structures.  Note: kmem_cache_create()
1709          * will SLEEP until successful.
1710          */
1711         if (info->swi_flags & HERMON_SWHDL_KMEMCACHE_INIT) {
```

```
1712                 rsrc_pool->rsrc_private = kmem_cache_create(
1713                     info->swi_rsrcname, rsrc_pool->rsrc_quantum, 0,
1714                     info->swi_constructor, info->swi_destructor, NULL,
1715                     rsrc_pool->rsrc_state, NULL, 0);
1716         }


1719         /* Allocate the central list of SW handle pointers */
1720         if (info->swi_flags & HERMON_SWHDL_TABLE_INIT) {
1721                 info->swi_table_ptr = kmem_zalloc(num_swhdl * prealloc_sz,
1722                     KM_SLEEP);
1723         }

1725         return (DDI_SUCCESS);
1726 }


1729 /*
1730  * hermon_rsrc_sw_handles_fini()
1731  *    Context: Only called from attach() and/or detach() path contexts
1732  */
1733 /* ARGSUSED */
1734 static void
1735 hermon_rsrc_sw_handles_fini(hermon_state_t *state,
1736     hermon_rsrc_sw_hdl_info_t *info)
1737 {
1738         hermon_rsrc_pool_info_t *rsrc_pool;
1739         uint64_t                num_swhdl, prealloc_sz;

1741         ASSERT(state != NULL);
1742         ASSERT(info != NULL);

1744         rsrc_pool       = info->swi_rsrcpool;
1745         num_swhdl       = info->swi_num;
1746         prealloc_sz     = info->swi_prealloc_sz;

1748         /*
1749          * If a "software handle" kmem_cache exists for this resource, then
1750          * destroy it now
1751          */
1752         if (rsrc_pool != NULL) {
1753                 kmem_cache_destroy(rsrc_pool->rsrc_private);
1754         }

1756         /* Free up this central list of SW handle pointers */
1757         if (info->swi_table_ptr != NULL) {
1758                 kmem_free(info->swi_table_ptr, num_swhdl * prealloc_sz);
1759         }
1760 }


1763 /*
1764  * hermon_rsrc_pd_handles_init()
1765  *    Context: Only called from attach() path context
1766  */
1767 static int
1768 hermon_rsrc_pd_handles_init(hermon_state_t *state,
1769     hermon_rsrc_sw_hdl_info_t *info)
1770 {
1771         hermon_rsrc_pool_info_t *rsrc_pool;
1772         vmem_t                  *vmp;
1773         char                    vmem_name[HERMON_RSRC_NAME_MAXLEN];
1774         int                     status;

1776         ASSERT(state != NULL);
1777         ASSERT(info != NULL);
```



```
1779         rsrc_pool = info->swi_rsrcpool;
1780         ASSERT(rsrc_pool != NULL);

1782         /* Initialize the resource pool for software handle table */
1783         status = hermon_rsrc_sw_handles_init(state, info);
1784         if (status != DDI_SUCCESS) {
1785                 return (DDI_FAILURE);
1786         }

1788         /* Build vmem arena name from Hermon instance */
1789         HERMON_RSRC_NAME(vmem_name, HERMON_PDHDL_VMEM);

1791         /* Create new vmem arena for PD numbers */
1792         vmp = vmem_create(vmem_name, (caddr_t)1, info->swi_num, 1, NULL,
1793             NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
1794         if (vmp == NULL) {
1795                 /* Unable to create vmem arena */
1796                 info->swi_table_ptr = NULL;
1797                 hermon_rsrc_sw_handles_fini(state, info);
1798                 return (DDI_FAILURE);
1799         }
1800         rsrc_pool->rsrc_vmp = vmp;

1802         return (DDI_SUCCESS);
1803 }


1806 /*
1807  * hermon_rsrc_pd_handles_fini()
1808  *    Context: Only called from attach() and/or detach() path contexts
1809  */
1810 static void
1811 hermon_rsrc_pd_handles_fini(hermon_state_t *state,
1812     hermon_rsrc_sw_hdl_info_t *info)
1813 {
1814         hermon_rsrc_pool_info_t *rsrc_pool;

1816         ASSERT(state != NULL);
1817         ASSERT(info != NULL);

1819         rsrc_pool = info->swi_rsrcpool;

1821         /* Destroy the specially created UAR scratch table vmem arena */
1822         vmem_destroy(rsrc_pool->rsrc_vmp);

1824         /* Destroy the "hermon_sw_pd_t" kmem_cache */
1825         hermon_rsrc_sw_handles_fini(state, info);
1826 }


1829 /*
1830  * hermon_rsrc_mbox_alloc()
1831  *    Context: Only called from attach() path context
1832  */
1833 static int
1834 hermon_rsrc_mbox_alloc(hermon_rsrc_pool_info_t *pool_info, uint_t num,
1835     hermon_rsrc_t *hdl)
1836 {
1837         hermon_rsrc_priv_mbox_t *priv;
1838         caddr_t                 kaddr;
1839         size_t                  real_len, temp_len;
1840         int                     status;

1842         ASSERT(pool_info != NULL);
1843         ASSERT(hdl != NULL);
```

```
1845         /* Get the private pointer for the mailboxes */
1846         priv = pool_info->rsrc_private;
1847         ASSERT(priv != NULL);

1849         /* Allocate a DMA handle for the mailbox */
1850         status = ddi_dma_alloc_handle(priv->pmb_dip, &priv->pmb_dmaattr,
1851             DDI_DMA_SLEEP, NULL, &hdl->hr_dmahdl);
1852         if (status != DDI_SUCCESS) {
1853                 return (DDI_FAILURE);
1854         }

1856         /* Allocate memory for the mailbox */
1857         temp_len = (num << pool_info->rsrc_shift);
1858         status = ddi_dma_mem_alloc(hdl->hr_dmahdl, temp_len,
1859             &priv->pmb_devaccattr, priv->pmb_xfer_mode, DDI_DMA_SLEEP,
1860             NULL, &kaddr, &real_len, &hdl->hr_acchdl);
1861         if (status != DDI_SUCCESS) {
1862                 /* No more memory available for mailbox entries */
1863                 ddi_dma_free_handle(&hdl->hr_dmahdl);
1864                 return (DDI_FAILURE);
1865         }

1867         hdl->hr_addr = (void *)kaddr;
1868         hdl->hr_len  = (uint32_t)real_len;

1870         return (DDI_SUCCESS);
1871 }


1874 /*
1875  * hermon_rsrc_mbox_free()
1876  *    Context: Can be called from interrupt or base context.
1877  */
1878 static void
1879 hermon_rsrc_mbox_free(hermon_rsrc_t *hdl)
1880 {
1881         ASSERT(hdl != NULL);

1883         /* Use ddi_dma_mem_free() to free up sys memory for mailbox */
1884         ddi_dma_mem_free(&hdl->hr_acchdl);

1886         /* Free the DMA handle for the mailbox */
1887         ddi_dma_free_handle(&hdl->hr_dmahdl);
1888 }


1891 /*
1892  * hermon_rsrc_hw_entry_alloc()
1893  *    Context: Can be called from interrupt or base context.
1894  */
1895 static int
1896 hermon_rsrc_hw_entry_alloc(hermon_rsrc_pool_info_t *pool_info, uint_t num,
1897     uint_t num_align, uint_t sleepflag, hermon_rsrc_t *hdl)
1898 {
1899         void                    *addr;
1900         uint64_t                offset;
1901         uint32_t                align;
1902         int                     status;
1903         int                     flag;

1905         ASSERT(pool_info != NULL);
1906         ASSERT(hdl != NULL);

1908         /*
1909          * Use vmem_xalloc() to get a properly aligned pointer (based on
```



```
1910          * the number requested) to the HW entry(ies).  This handles the
1911          * cases (for special QPCs and for RDB entries) where we need more
1912          * than one and need to ensure that they are properly aligned.
1913          */
1914         flag = (sleepflag == HERMON_SLEEP) ? VM_SLEEP : VM_NOSLEEP;
1915         hdl->hr_len = (num << pool_info->rsrc_shift);
1916         align = (num_align << pool_info->rsrc_shift);

1918         addr = vmem_xalloc(pool_info->rsrc_vmp, hdl->hr_len,
1919             align, 0, 0, NULL, NULL, flag | VM_FIRSTFIT);

1921         if (addr == NULL) {
1922                 /* No more HW entries available */
1923                 return (DDI_FAILURE);
1924         }

1926         hdl->hr_acchdl = NULL;  /* only used for mbox resources */

1928         /* Calculate vaddr and HW table index */
1929         offset = (uintptr_t)addr - (uintptr_t)pool_info->rsrc_start;
1930         hdl->hr_addr = addr;    /* only used for mbox and uarpg resources */
1931         hdl->hr_indx = offset >> pool_info->rsrc_shift;

1933         if (pool_info->rsrc_loc == HERMON_IN_ICM) {
1934                 int num_to_hdl;
1935                 hermon_rsrc_type_t rsrc_type = pool_info->rsrc_type;

1937                 num_to_hdl = (rsrc_type == HERMON_QPC ||
1938                     rsrc_type == HERMON_CQC || rsrc_type == HERMON_SRQC);

1940                 /* confirm ICM is mapped, and allocate if necessary */
1941                 status = hermon_rsrc_hw_entry_icm_confirm(pool_info, num, hdl,
1942                     num_to_hdl);
1943                 if (status != DDI_SUCCESS) {
1944                         return (DDI_FAILURE);
1945                 }
1946                 hdl->hr_addr = NULL;    /* not used for ICM resources */
1947         }

1949         return (DDI_SUCCESS);
1950 }


1953 /*
1954  * hermon_rsrc_hw_entry_reserve()
1955  *    Context: Can be called from interrupt or base context.
1956  */
1957 int
1958 hermon_rsrc_hw_entry_reserve(hermon_rsrc_pool_info_t *pool_info, uint_t num,
1959     uint_t num_align, uint_t sleepflag, hermon_rsrc_t *hdl)
1960 {
1961         void                    *addr;
1962         uint64_t                offset;
1963         uint32_t                align;
1964         int                     flag;

1966         ASSERT(pool_info != NULL);
1967         ASSERT(hdl != NULL);
1968         ASSERT(pool_info->rsrc_loc == HERMON_IN_ICM);

1970         /*
1971          * Use vmem_xalloc() to get a properly aligned pointer (based on
1972          * the number requested) to the HW entry(ies).  This handles the
1973          * cases (for special QPCs and for RDB entries) where we need more
1974          * than one and need to ensure that they are properly aligned.
1975          */
```

```
1976         flag = (sleepflag == HERMON_SLEEP) ? VM_SLEEP : VM_NOSLEEP;
1977         hdl->hr_len = (num << pool_info->rsrc_shift);
1978         align = (num_align << pool_info->rsrc_shift);

1980         addr = vmem_xalloc(pool_info->rsrc_vmp, hdl->hr_len,
1981             align, 0, 0, NULL, NULL, flag | VM_FIRSTFIT);

1983         if (addr == NULL) {
1984                 /* No more HW entries available */
1985                 return (DDI_FAILURE);
1986         }

1988         hdl->hr_acchdl = NULL;  /* only used for mbox resources */

1990         /* Calculate vaddr and HW table index */
1991         offset = (uintptr_t)addr - (uintptr_t)pool_info->rsrc_start;
1992         hdl->hr_addr = NULL;
1993         hdl->hr_indx = offset >> pool_info->rsrc_shift;

1995         /* ICM will be allocated and mapped if and when it gets used */

1997         return (DDI_SUCCESS);
1998 }


2001 /*
2002  * hermon_rsrc_hw_entry_free()
2003  *    Context: Can be called from interrupt or base context.
2004  */
2005 static void
2006 hermon_rsrc_hw_entry_free(hermon_rsrc_pool_info_t *pool_info,
2007     hermon_rsrc_t *hdl)
2008 {
2009         void                    *addr;
2010         uint64_t                offset;
2011         int                     status;

2013         ASSERT(pool_info != NULL);
2014         ASSERT(hdl != NULL);

2016         /* Calculate the allocated address */
2017         offset = hdl->hr_indx << pool_info->rsrc_shift;
2018         addr = (void *)(uintptr_t)(offset + (uintptr_t)pool_info->rsrc_start);

2020         /* Use vmem_xfree() to free up the HW table entry */
2021         vmem_xfree(pool_info->rsrc_vmp, addr, hdl->hr_len);

2023         if (pool_info->rsrc_loc == HERMON_IN_ICM) {
2024                 int num_to_hdl;
2025                 hermon_rsrc_type_t rsrc_type = pool_info->rsrc_type;

2027                 num_to_hdl = (rsrc_type == HERMON_QPC ||
2028                     rsrc_type == HERMON_CQC || rsrc_type == HERMON_SRQC);

2030                 /* free ICM references, and free ICM if required */
2031                 status = hermon_rsrc_hw_entry_icm_free(pool_info, hdl,
2032                     num_to_hdl);
2033                 if (status != DDI_SUCCESS)
2034                         HERMON_WARNING(pool_info->rsrc_state,
2035                             "failure in hw_entry_free");
2036         }
2037 }

2039 /*
2040  * hermon_rsrc_hw_entry_icm_confirm()
2041  *    Context: Can be called from interrupt or base context.
```



```
2042  */
2043 static int
2044 hermon_rsrc_hw_entry_icm_confirm(hermon_rsrc_pool_info_t *pool_info, uint_t num,
2045     hermon_rsrc_t *hdl, int num_to_hdl)
2046 {
2047         hermon_state_t          *state;
2048         hermon_icm_table_t      *icm_table;
2049         uint8_t                 *bitmap;
2050         hermon_dma_info_t       *dma_info;
2051         hermon_rsrc_type_t      type;
2052         uint32_t                rindx, span_offset;
2053         uint32_t                span_avail;
2054         int                     num_backed;
2055         int                     status;
2056         uint32_t                index1, index2;

2058         /*
2059          * Utility routine responsible for ensuring that there is memory
2060          * backing the ICM resources allocated via hermon_rsrc_hw_entry_alloc().
2061          * Confirm existing ICM mapping(s) or allocate ICM memory for the
2062          * given hardware resources being allocated, and increment the
2063          * ICM DMA structure(s) reference count.
2064          *
2065          * We may be allocating more objects than can fit in a single span,
2066          * or more than will fit in the remaining contiguous memory (from
2067          * the offset indicated by hdl->ar_indx) in the span in question.
2068          * In either of these cases, we'll be breaking up our allocation
2069          * into multiple spans.
2070          */
2071         state = pool_info->rsrc_state;
2072         type  = pool_info->rsrc_type;
2073         icm_table = &state->hs_icm[type];

2075         rindx = hdl->hr_indx;
2076         hermon_index(index1, index2, rindx, icm_table, span_offset);

2078         if (hermon_rsrc_verbose) {
2079                 IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entry_icm_confirm: "
2080                     "type (0x%x) num (0x%x) length (0x%x) index (0x%x, 0x%x): ",
2081                     type, num, hdl->hr_len, index1, index2);
2082         }

2084         mutex_enter(&icm_table->icm_table_lock);
2085         hermon_bitmap(bitmap, dma_info, icm_table, index1, num_to_hdl);
2086         while (num) {
2087 #ifndef __lock_lint
2088                 while (icm_table->icm_busy) {
2089                         cv_wait(&icm_table->icm_table_cv,
2090                             &icm_table->icm_table_lock);
2091                 }
2092 #endif
2093                 if (!HERMON_BMAP_BIT_ISSET(bitmap, index2)) {
2094                         /* Allocate ICM for this span */
2095                         icm_table->icm_busy = 1;
2096                         mutex_exit(&icm_table->icm_table_lock);
2097                         status = hermon_icm_alloc(state, type, index1, index2);
2098                         mutex_enter(&icm_table->icm_table_lock);
2099                         icm_table->icm_busy = 0;
2100                         cv_broadcast(&icm_table->icm_table_cv);
2101                         if (status != DDI_SUCCESS) {
2102                                 goto fail_alloc;
2103                         }
2104                         if (hermon_rsrc_verbose) {
2105                                 IBTF_DPRINTF_L2("hermon", "hermon_rsrc_"
2106                                     "hw_entry_icm_confirm: ALLOCATED ICM: "
2107                                     "type (0x%x) index (0x%x, 0x%x)",
```

```
2108                                    type, index1, index2);
2109                            }
2110                    }

2112                    /*
2113                     * We need to increment the refcnt of this span by the
2114                     * number of objects in this resource allocation that are
2115                     * backed by this span. Given that the rsrc allocation is
2116                     * contiguous, this value will be the number of objects in
2117                     * the span from 'span_offset' onward, either up to a max
2118                     * of the total number of objects, or the end of the span.
2119                     * So, determine the number of objects that can be backed
2120                     * by this span ('span_avail'), then determine the number
2121                     * of backed resources.
2122                     */
2123                    span_avail = icm_table->span - span_offset;
2124                    if (num > span_avail) {
2125                            num_backed = span_avail;
2126                    } else {
2127                            num_backed = num;
2128                    }

2130                    /*
2131                     * Now that we know 'num_backed', increment the refcnt,
2132                     * decrement the total number, and set 'span_offset' to
2133                     * 0 in case we roll over into the next span.
2134                     */
2135                    dma_info[index2].icm_refcnt += num_backed;
2136                    rindx += num_backed;
2137                    num -= num_backed;

2139                    if (hermon_rsrc_verbose) {
2140                            IBTF_DPRINTF_L2("ALLOC", "ICM type (0x%x) index "
2141                                "(0x%x, 0x%x) num_backed (0x%x)",
2142                                type, index1, index2, num_backed);
2143                            IBTF_DPRINTF_L2("ALLOC", "ICM type (0x%x) refcnt now "
2144                                "(0x%x) num_remaining (0x%x)", type,
2145                                dma_info[index2].icm_refcnt, num);
2146                    }
2147                    if (num == 0)
2148                            break;

2150                    hermon_index(index1, index2, rindx, icm_table, span_offset);
2151                    hermon_bitmap(bitmap, dma_info, icm_table, index1, num_to_hdl);
2152            }
2153            mutex_exit(&icm_table->icm_table_lock);

2155            return (DDI_SUCCESS);

2157    fail_alloc:
2158            /* JBDB */
2159            if (hermon_rsrc_verbose) {
2160                    IBTF_DPRINTF_L2("hermon", "hermon_rsrc_"
2161                        "hw_entry_icm_confirm: FAILED ICM ALLOC: "
2162                        "type (0x%x) num remaind (0x%x) index (0x%x, 0x%x)"
2163                        "refcnt (0x%x)", type, num, index1, index2,
2164                        icm_table->icm_dma[index1][index2].icm_refcnt);
2165            }
2166            IBTF_DPRINTF_L2("hermon", "WARNING: "
2167                "unimplemented fail code in hermon_rsrc_hw_entry_icm_alloc\n");

2169    #if needs_work
2170            /* free refcnt's and any spans we've allocated */
2171            while (index-- != start) {
2172                    /*
2173                     * JBDB - This is a bit tricky.  We need to
```



```
2174                     * free refcnt's on any spans that we've
2175                     * incremented them on, and completely free
2176                     * spans that we've allocated. How do we do
2177                     * this here? Does it need to be as involved
2178                     * as the core of icm_free() below, or can
2179                     * we leverage breadcrumbs somehow?
2180                     */
2181                    HERMON_WARNING(state, "unable to allocate ICM memory: "
2182                        "UNIMPLEMENTED HANDLING!!");
2183            }
2184    #else
2185            cmn_err(CE_WARN,
2186                "unimplemented fail code in hermon_rsrc_hw_entry_icm_alloc\n");
2187    #endif
2188            mutex_exit(&icm_table->icm_table_lock);

2190            HERMON_WARNING(state, "unable to allocate ICM memory");
2191            return (DDI_FAILURE);
2192    }

2194    /*
2195     * hermon_rsrc_hw_entry_icm_free()
2196     *    Context: Can be called from interrupt or base context.
2197     */
2198    static int
2199    hermon_rsrc_hw_entry_icm_free(hermon_rsrc_pool_info_t *pool_info,
2200        hermon_rsrc_t *hdl, int num_to_hdl)
2201    {
2202            hermon_state_t          *state;
2203            hermon_icm_table_t      *icm_table;
2204            uint8_t                 *bitmap;
2205            hermon_dma_info_t       *dma_info;
2206            hermon_rsrc_type_t      type;
2207            uint32_t                span_offset;
2208            uint32_t                span_remain;
2209            int                     num_freed;
2210            int                     num;
2211            uint32_t                index1, index2, rindx;

2213            /*
2214             * Utility routine responsible for freeing references to ICM
2215             * DMA spans, and freeing the ICM memory if necessary.
2216             *
2217             * We may have allocated objects in a single contiguous resource
2218             * allocation that reside in a number of spans, at any given
2219             * starting offset within a span. We therefore must determine
2220             * where this allocation starts, and then determine if we need
2221             * to free objects in more than one span.
2222             */
2223            state = pool_info->rsrc_state;
2224            type  = pool_info->rsrc_type;
2225            icm_table = &state->hs_icm[type];

2227            rindx = hdl->hr_indx;
2228            hermon_index(index1, index2, rindx, icm_table, span_offset);
2229            hermon_bitmap(bitmap, dma_info, icm_table, index1, num_to_hdl);

2231            /* determine the number of ICM objects in this allocation */
2232            num = hdl->hr_len >> pool_info->rsrc_shift;

2234            if (hermon_rsrc_verbose) {
2235                    IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entry_icm_free: "
2236                        "type (0x%x) num (0x%x) length (0x%x) index (0x%x, 0x%x)",
2237                        type, num, hdl->hr_len, index1, index2);
2238            }
2239            mutex_enter(&icm_table->icm_table_lock);
```

```
2240         while (num) {
2241                 /*
2242                  * As with the ICM confirm code above, we need to
2243                  * decrement the ICM span(s) by the number of
2244                  * resources being freed. So, determine the number
2245                  * of objects that are backed in this span from
2246                  * 'span_offset' onward, and set 'num_freed' to
2247                  * the smaller of either that number ('span_remain'),
2248                  * or the total number of objects being freed.
2249                  */
2250                 span_remain = icm_table->span - span_offset;
2251                 if (num > span_remain) {
2252                         num_freed = span_remain;
2253                 } else {
2254                         num_freed = num;
2255                 }
2256
2257                 /*
2258                  * Now that we know 'num_freed', decrement the refcnt,
2259                  * decrement the total number, and set 'span_offset' to
2260                  * 0 in case we roll over into the next span.
2261                  */
2262                 dma_info[index2].icm_refcnt -= num_freed;
2263                 num -= num_freed;
2264                 rindx += num_freed;
2265
2266                 if (hermon_rsrc_verbose) {
2267                         IBTF_DPRINTF_L2("FREE", "ICM type (0x%x) index "
2268                             "(0x%x, 0x%x) num_freed (0x%x)", type,
2269                             index1, index2, num_freed);
2270                         IBTF_DPRINTF_L2("FREE", "ICM type (0x%x) refcnt now "
2271                             "(0x%x) num remaining (0x%x)", type,
2272                             icm_table->icm_dma[index1][index2].icm_refcnt, num);
2273                 }
2274
2275 #if HERMON_ICM_FREE_ENABLED
2276                 /* If we've freed the last object in this span, free it */
2277                 if ((index1 != 0 || index2 != 0) &&
2278                     (dma_info[index2].icm_refcnt == 0)) {
2279                         if (hermon_rsrc_verbose) {
2280                                 IBTF_DPRINTF_L2("hermon", "hermon_rsrc_hw_entry"
2281                                     "_icm_free: freeing ICM type (0x%x) index"
2282                                     " (0x%x, 0x%x)", type, index1, index2);
2283                         }
2284                         hermon_icm_free(state, type, index1, index2);
2285                 }
2286 #endif
2287                 if (num == 0)
2288                         break;
2289
2290                 hermon_index(index1, index2, rindx, icm_table, span_offset);
2291                 hermon_bitmap(bitmap, dma_info, icm_table, index1, num_to_hdl);
2292         }
2293         mutex_exit(&icm_table->icm_table_lock);
2294
2295         return (DDI_SUCCESS);
2296 }



2300 /*
2301  * hermon_rsrc_swhdl_alloc()
2302  *    Context: Can be called from interrupt or base context.
2303  */
2304 static int
2305 hermon_rsrc_swhdl_alloc(hermon_rsrc_pool_info_t *pool_info, uint_t sleepflag,
```



```
2306     hermon_rsrc_t *hdl)
2307 {
2308         void    *addr;
2309         int     flag;
2310
2311         ASSERT(pool_info != NULL);
2312         ASSERT(hdl != NULL);
2313
2314         /* Allocate the software handle structure */
2315         flag = (sleepflag == HERMON_SLEEP) ? KM_SLEEP : KM_NOSLEEP;
2316         addr = kmem_cache_alloc(pool_info->rsrc_private, flag);
2317         if (addr == NULL) {
2318                 return (DDI_FAILURE);
2319         }
2320         hdl->hr_len  = pool_info->rsrc_quantum;
2321         hdl->hr_addr = addr;
2322
2323         return (DDI_SUCCESS);
2324 }



2327 /*
2328  * hermon_rsrc_swhdl_free()
2329  *    Context: Can be called from interrupt or base context.
2330  */
2331 static void
2332 hermon_rsrc_swhdl_free(hermon_rsrc_pool_info_t *pool_info, hermon_rsrc_t *hdl)
2333 {
2334         ASSERT(pool_info != NULL);
2335         ASSERT(hdl != NULL);
2336
2337         /* Free the software handle structure */
2338         kmem_cache_free(pool_info->rsrc_private, hdl->hr_addr);
2339 }



2342 /*
2343  * hermon_rsrc_pdhdl_alloc()
2344  *    Context: Can be called from interrupt or base context.
2345  */
2346 static int
2347 hermon_rsrc_pdhdl_alloc(hermon_rsrc_pool_info_t *pool_info, uint_t sleepflag,
2348     hermon_rsrc_t *hdl)
2349 {
2350         hermon_pdhdl_t  addr;
2351         void            *tmpaddr;
2352         int             flag, status;
2353
2354         ASSERT(pool_info != NULL);
2355         ASSERT(hdl != NULL);
2356
2357         /* Allocate the software handle */
2358         status = hermon_rsrc_swhdl_alloc(pool_info, sleepflag, hdl);
2359         if (status != DDI_SUCCESS) {
2360                 return (DDI_FAILURE);
2361         }
2362         addr = (hermon_pdhdl_t)hdl->hr_addr;
2363         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*addr))
2364
2365         /* Allocate a PD number for the handle */
2366         flag = (sleepflag == HERMON_SLEEP) ? VM_SLEEP : VM_NOSLEEP;
2367         tmpaddr = vmem_alloc(pool_info->rsrc_vmp, 1, flag);
2368         if (tmpaddr == NULL) {
2369                 /* No more PD number entries available */
2370                 hermon_rsrc_swhdl_free(pool_info, hdl);
2371                 return (DDI_FAILURE);
```

```
2372         }
2373         addr->pd_pdnum = (uint32_t)(uintptr_t)tmpaddr;
2374         addr->pd_rsrcp = hdl;
2375         hdl->hr_indx   = addr->pd_pdnum;

2377         return (DDI_SUCCESS);
2378 }


2381 /*
2382  * hermon_rsrc_pdhdl_free()
2383  *    Context: Can be called from interrupt or base context.
2384  */
2385 static void
2386 hermon_rsrc_pdhdl_free(hermon_rsrc_pool_info_t *pool_info, hermon_rsrc_t *hdl)
2387 {
2388         ASSERT(pool_info != NULL);
2389         ASSERT(hdl != NULL);

2391         /* Use vmem_free() to free up the PD number */
2392         vmem_free(pool_info->rsrc_vmp, (void *)(uintptr_t)hdl->hr_indx, 1);

2394         /* Free the software handle structure */
2395         hermon_rsrc_swhdl_free(pool_info, hdl);
2396 }


2399 /*
2400  * hermon_rsrc_pdhdl_constructor()
2401  *    Context: Can be called from interrupt or base context.
2402  */
2403 /* ARGSUSED */
2404 static int
2405 hermon_rsrc_pdhdl_constructor(void *pd, void *priv, int flags)
2406 {
2407         hermon_pdhdl_t  pdhdl;
2408         hermon_state_t  *state;

2410         pdhdl = (hermon_pdhdl_t)pd;
2411         state = (hermon_state_t *)priv;

2413         mutex_init(&pdhdl->pd_lock, NULL, MUTEX_DRIVER,
2414             DDI_INTR_PRI(state->hs_intrmsi_pri));

2416         return (DDI_SUCCESS);
2417 }


2420 /*
2421  * hermon_rsrc_pdhdl_destructor()
2422  *    Context: Can be called from interrupt or base context.
2423  */
2424 /* ARGSUSED */
2425 static void
2426 hermon_rsrc_pdhdl_destructor(void *pd, void *priv)
2427 {
2428         hermon_pdhdl_t  pdhdl;

2430         pdhdl = (hermon_pdhdl_t)pd;

2432         mutex_destroy(&pdhdl->pd_lock);
2433 }


2436 /*
2437  * hermon_rsrc_cqhdl_constructor()
```



```
2438  *    Context: Can be called from interrupt or base context.
2439  */
2440 /* ARGSUSED */
2441 static int
2442 hermon_rsrc_cqhdl_constructor(void *cq, void *priv, int flags)
2443 {
2444         hermon_cqhdl_t  cqhdl;
2445         hermon_state_t  *state;

2447         cqhdl = (hermon_cqhdl_t)cq;
2448         state = (hermon_state_t *)priv;

2450         mutex_init(&cqhdl->cq_lock, NULL, MUTEX_DRIVER,
2451             DDI_INTR_PRI(state->hs_intrmsi_pri));

2453         return (DDI_SUCCESS);
2454 }


2457 /*
2458  * hermon_rsrc_cqhdl_destructor()
2459  *    Context: Can be called from interrupt or base context.
2460  */
2461 /* ARGSUSED */
2462 static void
2463 hermon_rsrc_cqhdl_destructor(void *cq, void *priv)
2464 {
2465         hermon_cqhdl_t  cqhdl;

2467         cqhdl = (hermon_cqhdl_t)cq;

2469         mutex_destroy(&cqhdl->cq_lock);
2470 }


2473 /*
2474  * hermon_rsrc_qphdl_constructor()
2475  *    Context: Can be called from interrupt or base context.
2476  */
2477 /* ARGSUSED */
2478 static int
2479 hermon_rsrc_qphdl_constructor(void *qp, void *priv, int flags)
2480 {
2481         hermon_qphdl_t  qphdl;
2482         hermon_state_t  *state;

2484         qphdl = (hermon_qphdl_t)qp;
2485         state = (hermon_state_t *)priv;

2487         mutex_init(&qphdl->qp_lock, NULL, MUTEX_DRIVER,
2488             DDI_INTR_PRI(state->hs_intrmsi_pri));

2490         return (DDI_SUCCESS);
2491 }


2494 /*
2495  * hermon_rsrc_qphdl_destructor()
2496  *    Context: Can be called from interrupt or base context.
2497  */
2498 /* ARGSUSED */
2499 static void
2500 hermon_rsrc_qphdl_destructor(void *qp, void *priv)
2501 {
2502         hermon_qphdl_t  qphdl;
```

```
2504         qphdl = (hermon_qphdl_t)qp;

2506         mutex_destroy(&qphdl->qp_lock);
2507 }


2510 /*
2511  * hermon_rsrc_srqhdl_constructor()
2512  *    Context: Can be called from interrupt or base context.
2513  */
2514 /* ARGSUSED */
2515 static int
2516 hermon_rsrc_srqhdl_constructor(void *srq, void *priv, int flags)
2517 {
2518         hermon_srqhdl_t srqhdl;
2519         hermon_state_t  *state;

2521         srqhdl = (hermon_srqhdl_t)srq;
2522         state = (hermon_state_t *)priv;

2524         mutex_init(&srqhdl->srq_lock, NULL, MUTEX_DRIVER,
2525             DDI_INTR_PRI(state->hs_intrmsi_pri));

2527         return (DDI_SUCCESS);
2528 }


2531 /*
2532  * hermon_rsrc_srqhdl_destructor()
2533  *    Context: Can be called from interrupt or base context.
2534  */
2535 /* ARGSUSED */
2536 static void
2537 hermon_rsrc_srqhdl_destructor(void *srq, void *priv)
2538 {
2539         hermon_srqhdl_t srqhdl;

2541         srqhdl = (hermon_srqhdl_t)srq;

2543         mutex_destroy(&srqhdl->srq_lock);
2544 }


2547 /*
2548  * hermon_rsrc_refcnt_constructor()
2549  *    Context: Can be called from interrupt or base context.
2550  */
2551 /* ARGSUSED */
2552 static int
2553 hermon_rsrc_refcnt_constructor(void *rc, void *priv, int flags)
2554 {
2555         hermon_sw_refcnt_t      *refcnt;
2556         hermon_state_t          *state;

2558         refcnt = (hermon_sw_refcnt_t *)rc;
2559         state  = (hermon_state_t *)priv;

2561         mutex_init(&refcnt->swrc_lock, NULL, MUTEX_DRIVER,
2562             DDI_INTR_PRI(state->hs_intrmsi_pri));

2564         return (DDI_SUCCESS);
2565 }


2568 /*
2569  * hermon_rsrc_refcnt_destructor()
```



```
2570  *    Context: Can be called from interrupt or base context.
2571  */
2572 /* ARGSUSED */
2573 static void
2574 hermon_rsrc_refcnt_destructor(void *rc, void *priv)
2575 {
2576         hermon_sw_refcnt_t      *refcnt;

2578         refcnt = (hermon_sw_refcnt_t *)rc;

2580         mutex_destroy(&refcnt->swrc_lock);
2581 }


2584 /*
2585  * hermon_rsrc_ahhdl_constructor()
2586  *    Context: Can be called from interrupt or base context.
2587  */
2588 /* ARGSUSED */
2589 static int
2590 hermon_rsrc_ahhdl_constructor(void *ah, void *priv, int flags)
2591 {
2592         hermon_ahhdl_t  ahhdl;
2593         hermon_state_t  *state;

2595         ahhdl = (hermon_ahhdl_t)ah;
2596         state = (hermon_state_t *)priv;

2598         mutex_init(&ahhdl->ah_lock, NULL, MUTEX_DRIVER,
2599             DDI_INTR_PRI(state->hs_intrmsi_pri));
2600         return (DDI_SUCCESS);
2601 }


2604 /*
2605  * hermon_rsrc_ahhdl_destructor()
2606  *    Context: Can be called from interrupt or base context.
2607  */
2608 /* ARGSUSED */
2609 static void
2610 hermon_rsrc_ahhdl_destructor(void *ah, void *priv)
2611 {
2612         hermon_ahhdl_t  ahhdl;

2614         ahhdl = (hermon_ahhdl_t)ah;

2616         mutex_destroy(&ahhdl->ah_lock);
2617 }


2620 /*
2621  * hermon_rsrc_mrhdl_constructor()
2622  *    Context: Can be called from interrupt or base context.
2623  */
2624 /* ARGSUSED */
2625 static int
2626 hermon_rsrc_mrhdl_constructor(void *mr, void *priv, int flags)
2627 {
2628         hermon_mrhdl_t  mrhdl;
2629         hermon_state_t  *state;

2631         mrhdl = (hermon_mrhdl_t)mr;
2632         state = (hermon_state_t *)priv;

2634         mutex_init(&mrhdl->mr_lock, NULL, MUTEX_DRIVER,
2635             DDI_INTR_PRI(state->hs_intrmsi_pri));
```

```
2637         return (DDI_SUCCESS);
2638 }


2641 /*
2642  * hermon_rsrc_mrhdl_destructor()
2643  *    Context: Can be called from interrupt or base context.
2644  */
2645 /* ARGSUSED */
2646 static void
2647 hermon_rsrc_mrhdl_destructor(void *mr, void *priv)
2648 {
2649         hermon_mrhdl_t  mrhdl;

2651         mrhdl = (hermon_mrhdl_t)mr;

2653         mutex_destroy(&mrhdl->mr_lock);
2654 }


2657 /*
2658  * hermon_rsrc_mcg_entry_get_size()
2659  */
2660 static int
2661 hermon_rsrc_mcg_entry_get_size(hermon_state_t *state, uint_t *mcg_size_shift)
2662 {
2663         uint_t  num_qp_per_mcg, max_qp_per_mcg, log2;

2665         /*
2666          * Round the configured number of QP per MCG to next larger
2667          * power-of-2 size and update.
2668          */
2669         num_qp_per_mcg = state->hs_cfg_profile->cp_num_qp_per_mcg + 8;
2670         log2 = highbit(num_qp_per_mcg);
2671         if (ISP2(num_qp_per_mcg)) {
  35         if ((num_qp_per_mcg & (num_qp_per_mcg - 1)) == 0) {
2672                 log2 = log2 - 1;
2673         }
2674         state->hs_cfg_profile->cp_num_qp_per_mcg = (1 << log2) - 8;

2676         /* Now make sure number of QP per MCG makes sense */
2677         num_qp_per_mcg = state->hs_cfg_profile->cp_num_qp_per_mcg;
2678         max_qp_per_mcg = (1 << state->hs_devlim.log_max_qp_mcg);
2679         if (num_qp_per_mcg > max_qp_per_mcg) {
2680                 return (DDI_FAILURE);
2681         }

2683         /* Return the (shift) size of an individual MCG HW entry */
2684         *mcg_size_shift = log2 + 2;

2686         return (DDI_SUCCESS);
2687 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   32449 Thu Oct 23 10:42:13 2014
new/usr/src/uts/common/io/ib/adapters/hermon/hermon_srq.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * hermon_srq.c
 *    Hermon Shared Receive Queue Processing Routines
 *
 *    Implements all the routines necessary for allocating, freeing, querying,
 *    modifying and posting shared receive queues.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/hermon/hermon.h>

static void hermon_srq_sgl_to_logwqesz(hermon_state_t *state, uint_t num_sgl,
    hermon_qp_wq_type_t wq_type, uint_t *logwqesz, uint_t *max_sgl);

/*
 * hermon_srq_alloc()
 *    Context: Can be called only from user or kernel context.
 */
int
hermon_srq_alloc(hermon_state_t *state, hermon_srq_info_t *srqinfo,
    uint_t sleepflag)
{
        ibt_srq_hdl_t           ibt_srqhdl;
        hermon_pdhdl_t          pd;
        ibt_srq_sizes_t         *sizes;
        ibt_srq_sizes_t         *real_sizes;
        hermon_srqhdl_t         *srqhdl;
        ibt_srq_flags_t         flags;
```



```
        hermon_rsrc_t           *srqc, *rsrc;
        hermon_hw_srqc_t        srqc_entry;
        uint32_t                *buf;
        hermon_srqhdl_t         srq;
        hermon_umap_db_entry_t  *umapdb;
        ibt_mr_attr_t           mr_attr;
        hermon_mr_options_t     mr_op;
        hermon_mrhdl_t          mr;
        uint64_t                value, srq_desc_off;
        uint32_t                log_srq_size;
        uint32_t                uarpg;
        uint_t                  srq_is_umap;
        int                     flag, status;
        uint_t                  max_sgl;
        uint_t                  wqesz;
        uint_t                  srq_wr_sz;
        _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*sizes))

        /*
         * options-->wq_location used to be for location, now explicitly
         * LOCATION_NORMAL
         */

        /*
         * Extract the necessary info from the hermon_srq_info_t structure
         */
        real_sizes = srqinfo->srqi_real_sizes;
        sizes      = srqinfo->srqi_sizes;
        pd         = srqinfo->srqi_pd;
        ibt_srqhdl = srqinfo->srqi_ibt_srqhdl;
        flags      = srqinfo->srqi_flags;
        srqhdl     = srqinfo->srqi_srqhdl;

        /*
         * Determine whether SRQ is being allocated for userland access or
         * whether it is being allocated for kernel access.  If the SRQ is
         * being allocated for userland access, then lookup the UAR doorbell
         * page number for the current process.  Note:  If this is not found
         * (e.g. if the process has not previously open()'d the Hermon driver),
         * then an error is returned.
         */
        srq_is_umap = (flags & IBT_SRQ_USER_MAP) ? 1 : 0;
        if (srq_is_umap) {
                status = hermon_umap_db_find(state->hs_instance, ddi_get_pid(),
                    MLNX_UMAP_UARPG_RSRC, &value, 0, NULL);
                if (status != DDI_SUCCESS) {
                        status = IBT_INVALID_PARAM;
                        goto srqalloc_fail3;
                }
                uarpg = ((hermon_rsrc_t *)(uintptr_t)value)->hr_indx;
        } else {
                uarpg = state->hs_kernel_uar_index;
        }

        /* Increase PD refcnt */
        hermon_pd_refcnt_inc(pd);

        /* Allocate an SRQ context entry */
        status = hermon_rsrc_alloc(state, HERMON_SRQC, 1, sleepflag, &srqc);
        if (status != DDI_SUCCESS) {
                status = IBT_INSUFF_RESOURCE;
                goto srqalloc_fail1;
        }

        /* Allocate the SRQ Handle entry */
        status = hermon_rsrc_alloc(state, HERMON_SRQHDL, 1, sleepflag, &rsrc);
```

```
128         if (status != DDI_SUCCESS) {
129                 status = IBT_INSUFF_RESOURCE;
130                 goto srqalloc_fail2;
131         }
132
133         srq = (hermon_srqhdl_t)rsrc->hr_addr;
134         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*srq))
135
136         bzero(srq, sizeof (struct hermon_sw_srq_s));
137         /* Calculate the SRQ number */
138
139         /* just use the index, implicit in Hermon */
140         srq->srq_srqnum = srqc->hr_indx;
141
142         /*
143          * If this will be a user-mappable SRQ, then allocate an entry for
144          * the "userland resources database".  This will later be added to
145          * the database (after all further SRQ operations are successful).
146          * If we fail here, we must undo the reference counts and the
147          * previous resource allocation.
148          */
149         if (srq_is_umap) {
150                 umapdb = hermon_umap_db_alloc(state->hs_instance,
151                     srq->srq_srqnum, MLNX_UMAP_SRQMEM_RSRC,
152                     (uint64_t)(uintptr_t)rsrc);
153                 if (umapdb == NULL) {
154                         status = IBT_INSUFF_RESOURCE;
155                         goto srqalloc_fail3;
156                 }
157         }
158
159         /*
160          * Allocate the doorbell record.  Hermon just needs one for the
161          * SRQ, and use uarpg (above) as the uar index
162          */
163
164         status = hermon_dbr_alloc(state, uarpg, &srq->srq_wq_dbr_acchdl,
165             &srq->srq_wq_vdbr, &srq->srq_wq_pdbr, &srq->srq_rdbr_mapoffset);
166         if (status != DDI_SUCCESS) {
167                 status = IBT_INSUFF_RESOURCE;
168                 goto srqalloc_fail4;
169         }
170
171         /*
172          * Calculate the appropriate size for the SRQ.
173          * Note:  All Hermon SRQs must be a power-of-2 in size.  Also
174          * they may not be any smaller than HERMON_SRQ_MIN_SIZE.  This step
175          * is to round the requested size up to the next highest power-of-2
176          */
177         srq_wr_sz = max(sizes->srq_wr_sz + 1, HERMON_SRQ_MIN_SIZE);
178         log_srq_size = highbit(srq_wr_sz);
179         if (ISP2(srq_wr_sz)) {
 34         if ((srq_wr_sz & (srq_wr_sz - 1)) == 0) {
180                 log_srq_size = log_srq_size - 1;
181         }
182
183         /*
184          * Next we verify that the rounded-up size is valid (i.e. consistent
185          * with the device limits and/or software-configured limits).  If not,
186          * then obviously we have a lot of cleanup to do before returning.
187          */
188         if (log_srq_size > state->hs_cfg_profile->cp_log_max_srq_sz) {
189                 status = IBT_HCA_WR_EXCEEDED;
190                 goto srqalloc_fail4a;
191         }
```



```
193         /*
194          * Next we verify that the requested number of SGL is valid (i.e.
195          * consistent with the device limits and/or software-configured
196          * limits).  If not, then obviously the same cleanup needs to be done.
197          */
198         max_sgl = state->hs_ibtfinfo.hca_attr->hca_max_srq_sgl;
199         if (sizes->srq_sgl_sz > max_sgl) {
200                 status = IBT_HCA_SGL_EXCEEDED;
201                 goto srqalloc_fail4a;
202         }
203
204         /*
205          * Determine the SRQ's WQE sizes.  This depends on the requested
206          * number of SGLs.  Note: This also has the side-effect of
207          * calculating the real number of SGLs (for the calculated WQE size)
208          */
209         hermon_srq_sgl_to_logwqesz(state, sizes->srq_sgl_sz,
210             HERMON_QP_WQ_TYPE_RECVQ, &srq->srq_wq_log_wqesz,
211             &srq->srq_wq_sgl);
212
213         /*
214          * Allocate the memory for SRQ work queues.  Note:  The location from
215          * which we will allocate these work queues is always
216          * QUEUE_LOCATION_NORMAL.  Since Hermon work queues are not
217          * allowed to cross a 32-bit (4GB) boundary, the alignment of the work
218          * queue memory is very important.  We used to allocate work queues
219          * (the combined receive and send queues) so that they would be aligned
220          * on their combined size.  That alignment guaranteed that they would
221          * never cross the 4GB boundary (Hermon work queues are on the order of
222          * MBs at maximum).  Now we are able to relax this alignment constraint
223          * by ensuring that the IB address assigned to the queue memory (as a
224          * result of the hermon_mr_register() call) is offset from zero.
225          * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
226          * guarantee the alignment, but when attempting to use IOMMU bypass
227          * mode we found that we were not allowed to specify any alignment that
228          * was more restrictive than the system page size.  So we avoided this
229          * constraint by passing two alignment values, one for the memory
230          * allocation itself and the other for the DMA handle (for later bind).
231          * This used to cause more memory than necessary to be allocated (in
232          * order to guarantee the more restrictive alignment contraint).  But
233          * be guaranteeing the zero-based IB virtual address for the queue, we
234          * are able to conserve this memory.
235          *
236          * Note: If SRQ is not user-mappable, then it may come from either
237          * kernel system memory or from HCA-attached local DDR memory.
238          *
239          * Note2: We align this queue on a pagesize boundary.  This is required
240          * to make sure that all the resulting IB addresses will start at 0, for
241          * a zero-based queue.  By making sure we are aligned on at least a
242          * page, any offset we use into our queue will be the same as when we
243          * perform hermon_srq_modify() operations later.
244          */
245         wqesz = (1 << srq->srq_wq_log_wqesz);
246         srq->srq_wqinfo.qa_size = (1 << log_srq_size) * wqesz;
247         srq->srq_wqinfo.qa_alloc_align = PAGESIZE;
248         srq->srq_wqinfo.qa_bind_align = PAGESIZE;
249         if (srq_is_umap) {
250                 srq->srq_wqinfo.qa_location = HERMON_QUEUE_LOCATION_USERLAND;
251         } else {
252                 srq->srq_wqinfo.qa_location = HERMON_QUEUE_LOCATION_NORMAL;
253         }
254         status = hermon_queue_alloc(state, &srq->srq_wqinfo, sleepflag);
255         if (status != DDI_SUCCESS) {
256                 status = IBT_INSUFF_RESOURCE;
257                 goto srqalloc_fail4a;
258         }
```

```
259         buf = (uint32_t *)srq->srq_wqinfo.qa_buf_aligned;
260         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*buf))

262         /*
263          * Register the memory for the SRQ work queues.  The memory for the SRQ
264          * must be registered in the Hermon cMPT tables.  This gives us the LKey
265          * to specify in the SRQ context later.  Note: If the work queue is to
266          * be allocated from DDR memory, then only a "bypass" mapping is
267          * appropriate.  And if the SRQ memory is user-mappable, then we force
268          * DDI_DMA_CONSISTENT mapping.  Also, in order to meet the alignment
269          * restriction, we pass the "mro_bind_override_addr" flag in the call
270          * to hermon_mr_register().  This guarantees that the resulting IB vaddr
271          * will be zero-based (modulo the offset into the first page).  If we
272          * fail here, we still have the bunch of resource and reference count
273          * cleanup to do.
274          */
275         flag = (sleepflag == HERMON_SLEEP) ? IBT_MR_SLEEP :
276             IBT_MR_NOSLEEP;
277         mr_attr.mr_vaddr = (uint64_t)(uintptr_t)buf;
278         mr_attr.mr_len   = srq->srq_wqinfo.qa_size;
279         mr_attr.mr_as    = NULL;
280         mr_attr.mr_flags = flag | IBT_MR_ENABLE_LOCAL_WRITE;
281         mr_op.mro_bind_type   = state->hs_cfg_profile->cp_iommu_bypass;
282         mr_op.mro_bind_dmahdl = srq->srq_wqinfo.qa_dmahdl;
283         mr_op.mro_bind_override_addr = 1;
284         status = hermon_mr_register(state, pd, &mr_attr, &mr,
285             &mr_op, HERMON_SRQ_CMPT);
286         if (status != DDI_SUCCESS) {
287                 status = IBT_INSUFF_RESOURCE;
288                 goto srqalloc_fail5;
289         }
290         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*mr))

292         /*
293          * Calculate the offset between the kernel virtual address space
294          * and the IB virtual address space.  This will be used when
295          * posting work requests to properly initialize each WQE.
296          */
297         srq_desc_off = (uint64_t)(uintptr_t)srq->srq_wqinfo.qa_buf_aligned -
298             (uint64_t)mr->mr_bindinfo.bi_addr;

300         srq->srq_wq_wqhdr = hermon_wrid_wqhdr_create(1 << log_srq_size);

302         /*
303          * Fill in all the return arguments (if necessary).  This includes
304          * real queue size and real SGLs.
305          */
306         if (real_sizes != NULL) {
307                 real_sizes->srq_wr_sz = (1 << log_srq_size) - 1;
308                 real_sizes->srq_sgl_sz = srq->srq_wq_sgl;
309         }

311         /*
312          * Fill in the SRQC entry.  This is the final step before passing
313          * ownership of the SRQC entry to the Hermon hardware.  We use all of
314          * the information collected/calculated above to fill in the
315          * requisite portions of the SRQC.  Note: If this SRQ is going to be
316          * used for userland access, then we need to set the UAR page number
317          * appropriately (otherwise it's a "don't care")
318          */
319         bzero(&srqc_entry, sizeof (hermon_hw_srqc_t));
320         srqc_entry.state           = HERMON_SRQ_STATE_HW_OWNER;
321         srqc_entry.log_srq_size    = log_srq_size;
322         srqc_entry.srqn            = srq->srq_srqnum;
323         srqc_entry.log_rq_stride   = srq->srq_wq_log_wqesz - 4;
324                                      /* 16-byte chunks */
```



```
326         srqc_entry.page_offs       = srq->srq_wqinfo.qa_pgoffs >> 6;
327         srqc_entry.log2_pgsz       = mr->mr_log2_pgsz;
328         srqc_entry.mtt_base_addrh  = (uint32_t)((mr->mr_mttaddr >> 32) & 0xFF);
329         srqc_entry.mtt_base_addrl  = mr->mr_mttaddr >> 3;
330         srqc_entry.pd              = pd->pd_pdnum;
331         srqc_entry.dbr_addrh = (uint32_t)((uint64_t)srq->srq_wq_pdbr >> 32);
332         srqc_entry.dbr_addrl = (uint32_t)((uint64_t)srq->srq_wq_pdbr >> 2);

334         /*
335          * all others - specifically, xrcd, cqn_xrc, lwm, wqe_cnt, and wqe_cntr
336          * are zero thanks to the bzero of the structure
337          */

339         /*
340          * Write the SRQC entry to hardware.  Lastly, we pass ownership of
341          * the entry to the hardware (using the Hermon SW2HW_SRQ firmware
342          * command).  Note: In general, this operation shouldn't fail.  But
343          * if it does, we have to undo everything we've done above before
344          * returning error.
345          */
346         status = hermon_cmn_ownership_cmd_post(state, SW2HW_SRQ, &srqc_entry,
347             sizeof (hermon_hw_srqc_t), srq->srq_srqnum,
348             sleepflag);
349         if (status != HERMON_CMD_SUCCESS) {
350                 cmn_err(CE_CONT, "Hermon: SW2HW_SRQ command failed: %08x\n",
351                     status);
352                 if (status == HERMON_CMD_INVALID_STATUS) {
353                         hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
354                 }
355                 status = ibc_get_ci_failure(0);
356                 goto srqalloc_fail8;
357         }

359         /*
360          * Fill in the rest of the Hermon SRQ handle.  We can update
361          * the following fields for use in further operations on the SRQ.
362          */
363         srq->srq_srqcrsrcp = srqc;
364         srq->srq_rsrcp     = rsrc;
365         srq->srq_mrhdl     = mr;
366         srq->srq_refcnt    = 0;
367         srq->srq_is_umap   = srq_is_umap;
368         srq->srq_uarpg     = uarpg;
369         srq->srq_umap_dhp  = (devmap_cookie_t)NULL;
370         srq->srq_pdhdl     = pd;
371         srq->srq_wq_bufsz  = (1 << log_srq_size);
372         srq->srq_wq_buf    = buf;
373         srq->srq_desc_off  = srq_desc_off;
374         srq->srq_hdlrarg   = (void *)ibt_srqhdl;
375         srq->srq_state     = 0;
376         srq->srq_real_sizes.srq_wr_sz = (1 << log_srq_size);
377         srq->srq_real_sizes.srq_sgl_sz = srq->srq_wq_sgl;

379         /*
380          * Put SRQ handle in Hermon SRQNum-to-SRQhdl list.  Then fill in the
381          * "srqhdl" and return success
382          */
383         hermon_icm_set_num_to_hdl(state, HERMON_SRQC, srqc->hr_indx, srq);

385         /*
386          * If this is a user-mappable SRQ, then we need to insert the
387          * previously allocated entry into the "userland resources database".
388          * This will allow for later lookup during devmap() (i.e. mmap())
389          * calls.
390          */
```

```
391         if (srq->srq_is_umap) {
392                 hermon_umap_db_add(umapdb);
393         } else {        /* initialize work queue for kernel SRQs */
394                 int i, len, last;
395                 uint16_t *desc;

397                 desc = (uint16_t *)buf;
398                 len = wqesz / sizeof (*desc);
399                 last = srq->srq_wq_bufsz - 1;
400                 for (i = 0; i < last; i++) {
401                         desc[1] = htons(i + 1);
402                         desc += len;
403                 }
404                 srq->srq_wq_wqhdr->wq_tail = last;
405                 srq->srq_wq_wqhdr->wq_head = 0;
406         }

408         *srqhdl = srq;

410         return (status);

412 /*
413  * The following is cleanup for all possible failure cases in this routine
414  */
415 srqalloc_fail8:
416         hermon_wrid_wqhdr_destroy(srq->srq_wq_wqhdr);
417 srqalloc_fail7:
418         if (hermon_mr_deregister(state, &mr, HERMON_MR_DEREG_ALL,
419             HERMON_SLEEPFLAG_FOR_CONTEXT()) != DDI_SUCCESS) {
420                 HERMON_WARNING(state, "failed to deregister SRQ memory");
421         }
422 srqalloc_fail5:
423         hermon_queue_free(&srq->srq_wqinfo);
424 srqalloc_fail4a:
425         hermon_dbr_free(state, uarpg, srq->srq_wq_vdbr);
426 srqalloc_fail4:
427         if (srq_is_umap) {
428                 hermon_umap_db_free(umapdb);
429         }
430 srqalloc_fail3:
431         hermon_rsrc_free(state, &rsrc);
432 srqalloc_fail2:
433         hermon_rsrc_free(state, &srqc);
434 srqalloc_fail1:
435         hermon_pd_refcnt_dec(pd);
436 srqalloc_fail:
437         return (status);
438 }
```

unchanged_portion_omitted_

```
581 /*
582  * hermon_srq_modify()
583  *    Context: Can be called only from user or kernel context.
584  */
585 int
586 hermon_srq_modify(hermon_state_t *state, hermon_srqhdl_t srq, uint_t size,
587     uint_t *real_size, uint_t sleepflag)
588 {
589         hermon_qalloc_info_t    new_srqinfo, old_srqinfo;
590         hermon_rsrc_t           *mtt, *old_mtt;
591         hermon_bind_info_t      bind;
592         hermon_bind_info_t      old_bind;
593         hermon_mrhdl_t          mr;
594         hermon_hw_srqc_t        srqc_entry;
595         hermon_hw_dmpt_t        mpt_entry;
```



```
596         uint64_t                *wre_new, *wre_old;
597         uint64_t                mtt_addr;
598         uint64_t                srq_pgoffs;
599         uint64_t                srq_desc_off;
600         uint32_t                *buf, srq_old_bufsz;
601         uint32_t                wqesz;
602         uint_t                  max_srq_size;
603         uint_t                  mtt_pgsize_bits;
604         uint_t                  log_srq_size, maxprot;
605         int                     status;

607         if ((state->hs_devlim.mod_wr_srq == 0) ||
608             (state->hs_cfg_profile->cp_srq_resize_enabled == 0))
609                 return (IBT_NOT_SUPPORTED);

611         /*
612          * If size requested is larger than device capability, return
613          * Insufficient Resources
614          */
615         max_srq_size = (1 << state->hs_cfg_profile->cp_log_max_srq_sz);
616         if (size > max_srq_size) {
617                 return (IBT_HCA_WR_EXCEEDED);
618         }

620         /*
621          * Calculate the appropriate size for the SRQ.
622          * Note:  All Hermon SRQs must be a power-of-2 in size.  Also
623          * they may not be any smaller than HERMON_SRQ_MIN_SIZE.  This step
624          * is to round the requested size up to the next highest power-of-2
625          */
626         size = max(size, HERMON_SRQ_MIN_SIZE);
627         log_srq_size = highbit(size);
628         if (ISP2(size)) {
483         if ((size & (size - 1)) == 0) {
629                 log_srq_size = log_srq_size - 1;
630         }

632         /*
633          * Next we verify that the rounded-up size is valid (i.e. consistent
634          * with the device limits and/or software-configured limits).
635          */
636         if (log_srq_size > state->hs_cfg_profile->cp_log_max_srq_sz) {
637                 status = IBT_HCA_WR_EXCEEDED;
638                 goto srqmodify_fail;
639         }

641         /*
642          * Allocate the memory for newly resized Shared Receive Queue.
643          *
644          * Note: If SRQ is not user-mappable, then it may come from either
645          * kernel system memory or from HCA-attached local DDR memory.
646          *
647          * Note2: We align this queue on a pagesize boundary.  This is required
648          * to make sure that all the resulting IB addresses will start at 0,
649          * for a zero-based queue.  By making sure we are aligned on at least a
650          * page, any offset we use into our queue will be the same as it was
651          * when we allocated it at hermon_srq_alloc() time.
652          */
653         wqesz = (1 << srq->srq_wq_log_wqesz);
654         new_srqinfo.qa_size = (1 << log_srq_size) * wqesz;
655         new_srqinfo.qa_alloc_align = PAGESIZE;
656         new_srqinfo.qa_bind_align  = PAGESIZE;
657         if (srq->srq_is_umap) {
658                 new_srqinfo.qa_location = HERMON_QUEUE_LOCATION_USERLAND;
659         } else {
660                 new_srqinfo.qa_location = HERMON_QUEUE_LOCATION_NORMAL;
```

```
661         }
662         status = hermon_queue_alloc(state, &new_srqinfo, sleepflag);
663         if (status != DDI_SUCCESS) {
664                 status = IBT_INSUFF_RESOURCE;
665                 goto srqmodify_fail;
666         }
667         buf = (uint32_t *)new_srqinfo.qa_buf_aligned;
668         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*buf))

670         /*
671          * Allocate the memory for the new WRE list.  This will be used later
672          * when we resize the wridlist based on the new SRQ size.
673          */
674         wre_new = kmem_zalloc((1 << log_srq_size) * sizeof (uint64_t),
675             sleepflag);
676         if (wre_new == NULL) {
677                 status = IBT_INSUFF_RESOURCE;
678                 goto srqmodify_fail;
679         }

681         /*
682          * Fill in the "bind" struct.  This struct provides the majority
683          * of the information that will be used to distinguish between an
684          * "addr" binding (as is the case here) and a "buf" binding (see
685          * below).  The "bind" struct is later passed to hermon_mr_mem_bind()
686          * which does most of the "heavy lifting" for the Hermon memory
687          * registration routines.
688          */
689         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(bind))
690         bzero(&bind, sizeof (hermon_bind_info_t));
691         bind.bi_type  = HERMON_BINDHDL_VADDR;
692         bind.bi_addr  = (uint64_t)(uintptr_t)buf;
693         bind.bi_len   = new_srqinfo.qa_size;
694         bind.bi_as    = NULL;
695         bind.bi_flags = sleepflag == HERMON_SLEEP ? IBT_MR_SLEEP :
696             IBT_MR_NOSLEEP | IBT_MR_ENABLE_LOCAL_WRITE;
697         bind.bi_bypass = state->hs_cfg_profile->cp_iommu_bypass;

699         status = hermon_mr_mtt_bind(state, &bind, new_srqinfo.qa_dmahdl, &mtt,
700             &mtt_pgsize_bits, 0); /* no relaxed ordering */
701         if (status != DDI_SUCCESS) {
702                 status = status;
703                 kmem_free(wre_new, (1 << log_srq_size) *
704                     sizeof (uint64_t));
705                 hermon_queue_free(&new_srqinfo);
706                 goto srqmodify_fail;
707         }

709         /*
710          * Calculate the offset between the kernel virtual address space
711          * and the IB virtual address space.  This will be used when
712          * posting work requests to properly initialize each WQE.
713          *
714          * Note: bind addr is zero-based (from alloc) so we calculate the
715          * correct new offset here.
716          */
717         bind.bi_addr = bind.bi_addr & ((1 << mtt_pgsize_bits) - 1);
718         srq_desc_off = (uint64_t)(uintptr_t)new_srqinfo.qa_buf_aligned -
719             (uint64_t)bind.bi_addr;
720         srq_pgoffs   = (uint_t)
721             ((uintptr_t)new_srqinfo.qa_buf_aligned & HERMON_PAGEOFFSET);

723         /*
724          * Fill in the MPT entry.  This is the final step before passing
725          * ownership of the MPT entry to the Hermon hardware.  We use all of
726          * the information collected/calculated above to fill in the
```



```
727          * requisite portions of the MPT.
728          */
729         bzero(&mpt_entry, sizeof (hermon_hw_dmpt_t));
730         mpt_entry.reg_win_len   = bind.bi_len;
731         mtt_addr = (mtt->hr_indx << HERMON_MTT_SIZE_SHIFT);
732         mpt_entry.mtt_addr_h = mtt_addr >> 32;
733         mpt_entry.mtt_addr_l = mtt_addr >> 3;

735         /*
736          * for hermon we build up a new srqc and pass that (partially filled
737          * to resize SRQ instead of modifying the (d)mpt directly
738          */



742         /*
743          * Now we grab the SRQ lock.  Since we will be updating the actual
744          * SRQ location and the producer/consumer indexes, we should hold
745          * the lock.
746          *
747          * We do a HERMON_NOSLEEP here (and below), though, because we are
748          * holding the "srq_lock" and if we got raised to interrupt level
749          * by priority inversion, we would not want to block in this routine
750          * waiting for success.
751          */
752         mutex_enter(&srq->srq_lock);

754         /*
755          * Copy old entries to new buffer
756          */
757         srq_old_bufsz = srq->srq_wq_bufsz;
758         bcopy(srq->srq_wq_buf, buf, srq_old_bufsz * wqesz);

760         /*
761          * Setup MPT information for use in the MODIFY_MPT command
762          */
763         mr = srq->srq_mrhdl;
764         mutex_enter(&mr->mr_lock);

766         /*
767          * now, setup the srqc information needed for resize - limit the
768          * values, but use the same structure as the srqc
769          */

771         srqc_entry.log_srq_size   = log_srq_size;
772         srqc_entry.page_offs      = srq_pgoffs >> 6;
773         srqc_entry.log2_pgsz      = mr->mr_log2_pgsz;
774         srqc_entry.mtt_base_addrl = (uint64_t)mtt_addr >> 32;
775         srqc_entry.mtt_base_addrh = mtt_addr >> 3;

777         /*
778          * RESIZE_SRQ
779          *
780          * If this fails for any reason, then it is an indication that
781          * something (either in HW or SW) has gone seriously wrong.  So we
782          * print a warning message and return.
783          */
784         status = hermon_resize_srq_cmd_post(state, &srqc_entry,
785             srq->srq_srqnum, sleepflag);
786         if (status != HERMON_CMD_SUCCESS) {
787                 cmn_err(CE_CONT, "Hermon: RESIZE_SRQ command failed: %08x\n",
788                     status);
789                 if (status == HERMON_CMD_INVALID_STATUS) {
790                         hermon_fm_ereport(state, HCA_SYS_ERR, HCA_ERR_SRV_LOST);
791                 }
792                 (void) hermon_mr_mtt_unbind(state, &bind, mtt);
```

```
793                 kmem_free(wre_new, (1 << log_srq_size) *
794                     sizeof (uint64_t));
795                 hermon_queue_free(&new_srqinfo);
796                 mutex_exit(&mr->mr_lock);
797                 mutex_exit(&srq->srq_lock);
798                 return (ibc_get_ci_failure(0));
799         }
800         /*
801          * Update the Hermon Shared Receive Queue handle with all the new
802          * information.  At the same time, save away all the necessary
803          * information for freeing up the old resources
804          */
805         old_srqinfo        = srq->srq_wqinfo;
806         old_mtt            = srq->srq_mrhdl->mr_mttrsrcp;
807         bcopy(&srq->srq_mrhdl->mr_bindinfo, &old_bind,
808             sizeof (hermon_bind_info_t));

810         /* Now set the new info */
811         srq->srq_wqinfo    = new_srqinfo;
812         srq->srq_wq_buf    = buf;
813         srq->srq_wq_bufsz  = (1 << log_srq_size);
814         bcopy(&bind, &srq->srq_mrhdl->mr_bindinfo, sizeof (hermon_bind_info_t));
815         srq->srq_mrhdl->mr_mttrsrcp = mtt;
816         srq->srq_desc_off  = srq_desc_off;
817         srq->srq_real_sizes.srq_wr_sz = (1 << log_srq_size);

819         /* Update MR mtt pagesize */
820         mr->mr_logmttpgsz = mtt_pgsize_bits;
821         mutex_exit(&mr->mr_lock);

823         /*
824          * Initialize new wridlist, if needed.
825          *
826          * If a wridlist already is setup on an SRQ (the QP associated with an
827          * SRQ has moved "from_reset") then we must update this wridlist based
828          * on the new SRQ size.  We allocate the new size of Work Request ID
829          * Entries, copy over the old entries to the new list, and
830          * re-initialize the srq wridlist in non-umap case
831          */
832         wre_old = srq->srq_wq_wqhdr->wq_wrid;

834         bcopy(wre_old, wre_new, srq_old_bufsz * sizeof (uint64_t));

836         /* Setup new sizes in wre */
837         srq->srq_wq_wqhdr->wq_wrid = wre_new;

839         /*
840          * If "old" SRQ was a user-mappable SRQ that is currently mmap()'d out
841          * to a user process, then we need to call devmap_devmem_remap() to
842          * invalidate the mapping to the SRQ memory.  We also need to
843          * invalidate the SRQ tracking information for the user mapping.
844          *
845          * Note: On failure, the remap really shouldn't ever happen.  So, if it
846          * does, it is an indication that something has gone seriously wrong.
847          * So we print a warning message and return error (knowing, of course,
848          * that the "old" SRQ memory will be leaked)
849          */
850         if ((srq->srq_is_umap) && (srq->srq_umap_dhp != NULL)) {
851                 maxprot = (PROT_READ | PROT_WRITE | PROT_USER);
852                 status = devmap_devmem_remap(srq->srq_umap_dhp,
853                     state->hs_dip, 0, 0, srq->srq_wqinfo.qa_size, maxprot,
854                     DEVMAP_MAPPING_INVALID, NULL);
855                 if (status != DDI_SUCCESS) {
856                         mutex_exit(&srq->srq_lock);
857                         HERMON_WARNING(state, "failed in SRQ memory "
858                             "devmap_devmem_remap()");
```



```
859                         /* We can, however, free the memory for old wre */
860                         kmem_free(wre_old, srq_old_bufsz * sizeof (uint64_t));
861                         return (ibc_get_ci_failure(0));
862                 }
863                 srq->srq_umap_dhp = (devmap_cookie_t)NULL;
864         }

866         /*
867          * Drop the SRQ lock now.  The only thing left to do is to free up
868          * the old resources.
869          */
870         mutex_exit(&srq->srq_lock);

872         /*
873          * Unbind the MTT entries.
874          */
875         status = hermon_mr_mtt_unbind(state, &old_bind, old_mtt);
876         if (status != DDI_SUCCESS) {
877                 HERMON_WARNING(state, "failed to unbind old SRQ memory");
878                 status = ibc_get_ci_failure(0);
879                 goto srqmodify_fail;
880         }

882         /* Free the memory for old wre */
883         kmem_free(wre_old, srq_old_bufsz * sizeof (uint64_t));

885         /* Free the memory for the old SRQ */
886         hermon_queue_free(&old_srqinfo);

888         /*
889          * Fill in the return arguments (if necessary).  This includes the
890          * real new completion queue size.
891          */
892         if (real_size != NULL) {
893                 *real_size = (1 << log_srq_size);
894         }

896         return (DDI_SUCCESS);

898 srqmodify_fail:
899         return (status);
900 }
```

*unchanged_portion_omitted_*

```
960 /*
961  * hermon_srq_sgl_to_logwqesz()
962  *    Context: Can be called from interrupt or base context.
963  */
964 static void
965 hermon_srq_sgl_to_logwqesz(hermon_state_t *state, uint_t num_sgl,
966     hermon_qp_wq_type_t wq_type, uint_t *logwqesz, uint_t *max_sgl)
967 {
968         uint_t  max_size, log2, actual_sgl;

970         switch (wq_type) {
971         case HERMON_QP_WQ_TYPE_RECVQ:
972                 /*
973                  * Use requested maximum SGL to calculate max descriptor size
974                  * (while guaranteeing that the descriptor size is a
975                  * power-of-2 cachelines).
976                  */
977                 max_size = (HERMON_QP_WQE_MLX_SRQ_HDRS + (num_sgl << 4));
978                 log2 = highbit(max_size);
979                 if (ISP2(max_size)) {
834                 if ((max_size & (max_size - 1)) == 0) {
```

```
980                         log2 = log2 - 1;
981                 }

983                 /* Make sure descriptor is at least the minimum size */
984                 log2 = max(log2, HERMON_QP_WQE_LOG_MINIMUM);

986                 /* Calculate actual number of SGL (given WQE size) */
987                 actual_sgl = ((1 << log2) - HERMON_QP_WQE_MLX_SRQ_HDRS) >> 4;
988                 break;

990         default:
991                 HERMON_WARNING(state, "unexpected work queue type");
992                 break;
993         }

995         /* Fill in the return values */
996         *logwqesz = log2;
997         *max_sgl  = min(state->hs_cfg_profile->cp_srq_max_sgl, actual_sgl);
998 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28866 Thu Oct 23 10:42:13 2014
new/usr/src/uts/common/io/ib/adapters/tavor/tavor_cfg.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * tavor_cfg.c
 *    Tavor Configuration Profile Routines
 *
 *    Implements the routines necessary for initializing and (later) tearing
 *    down the list of Tavor configuration information.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/tavor/tavor.h>

/* Set to enable alternative configurations: 0 = automatic config, 1 = manual */
uint32_t tavor_alt_config_enable        = 0;

/* Number of supported QPs and their maximum size */
uint32_t tavor_log_num_qp               = TAVOR_NUM_QP_SHIFT_128;
uint32_t tavor_log_max_qp_sz            = TAVOR_QP_SZ_SHIFT;

/* Number of supported SGL per WQE */
uint32_t tavor_wqe_max_sgl              = TAVOR_NUM_WQE_SGL;

/* Number of supported CQs and their maximum size */
uint32_t tavor_log_num_cq               = TAVOR_NUM_CQ_SHIFT_128;
uint32_t tavor_log_max_cq_sz            = TAVOR_CQ_SZ_SHIFT;

/* Select to enable SRQ or not; NOTE: 0 for disabled, 1 for enabled */
uint32_t tavor_srq_enable               = 1;
```



```
/* Number of supported SRQs and their maximum size */
uint32_t tavor_log_num_srq              = TAVOR_NUM_SRQ_SHIFT_128;
uint32_t tavor_log_max_srq_sz           = TAVOR_SRQ_SZ_SHIFT;
uint32_t tavor_srq_max_sgl              = TAVOR_SRQ_MAX_SGL;

/* Default size for all EQs */
uint32_t tavor_log_default_eq_sz        = TAVOR_DEFAULT_EQ_SZ_SHIFT;

/* Number of supported RDB (for incoming RDMA Read/Atomic) */
uint32_t tavor_log_num_rdb              = TAVOR_NUM_RDB_SHIFT_128;

/*
 * Number of support multicast groups, number of QP per multicast group, and
 * the number of entries (from the total number) in the multicast group "hash
 * table"
 */
uint32_t tavor_log_num_mcg              = TAVOR_NUM_MCG_SHIFT;
uint32_t tavor_num_qp_per_mcg           = TAVOR_NUM_QP_PER_MCG;
uint32_t tavor_log_num_mcg_hash         = TAVOR_NUM_MCG_HASH_SHIFT;

/*
 * Number of supported MPTs (memory regions and windows) and their maximum
 * size.  Also the number of MTT per "MTT segment" (see tavor_mr.h for more
 * details)
 */
uint32_t tavor_log_num_mpt              = TAVOR_NUM_MPT_SHIFT_128;
uint32_t tavor_log_max_mrw_sz           = TAVOR_MAX_MEM_MPT_SHIFT_128;
uint32_t tavor_log_num_mttseg           = TAVOR_NUM_MTTSEG_SHIFT;

/*
 * Number of supported Tavor mailboxes ("In" and "Out") and their maximum
 * sizes, respectively
 */
uint32_t tavor_log_num_inmbox           = TAVOR_NUM_MAILBOXES_SHIFT;
uint32_t tavor_log_num_outmbox          = TAVOR_NUM_MAILBOXES_SHIFT;
uint32_t tavor_log_num_intr_inmbox      = TAVOR_NUM_INTR_MAILBOXES_SHIFT;
uint32_t tavor_log_num_intr_outmbox     = TAVOR_NUM_INTR_MAILBOXES_SHIFT;
uint32_t tavor_log_inmbox_size          = TAVOR_MBOX_SIZE_SHIFT;
uint32_t tavor_log_outmbox_size         = TAVOR_MBOX_SIZE_SHIFT;

/* Number of supported UAR pages */
uint32_t tavor_log_num_uar              = TAVOR_NUM_UAR_SHIFT;

/* Number of supported Protection Domains (PD) */
uint32_t tavor_log_num_pd               = TAVOR_NUM_PD_SHIFT;

/* Number of supported Address Handles (AH) */
uint32_t tavor_log_num_ah               = TAVOR_NUM_AH_SHIFT;

/*
 * Number of total supported PKeys per PKey table (i.e.
 * per port).  Also the number of SGID per GID table.
 */
uint32_t tavor_log_max_pkeytbl          = TAVOR_NUM_PKEYTBL_SHIFT;
uint32_t tavor_log_max_gidtbl           = TAVOR_NUM_GIDTBL_SHIFT;

/* Maximum "responder resources" (in) and "initiator depth" (out) per QP */
uint32_t tavor_hca_max_rdma_in_qp       = TAVOR_HCA_MAX_RDMA_IN_QP;
uint32_t tavor_hca_max_rdma_out_qp      = TAVOR_HCA_MAX_RDMA_OUT_QP;

/* Maximum supported MTU and portwidth */
uint32_t tavor_max_mtu                  = TAVOR_MAX_MTU;
uint32_t tavor_max_port_width           = TAVOR_MAX_PORT_WIDTH;

/* Number of supported Virtual Lanes (VL) */
```

```
128 uint32_t tavor_max_vlcap                = TAVOR_MAX_VLCAP;

130 /* Number of supported ports (1 or 2) */
131 uint32_t tavor_num_ports                = TAVOR_NUM_PORTS;

133 /*
134  * Whether or not to use the built-in (i.e. in firmware) agents for QP0 and
135  * QP1, respectively.
136  */
137 uint32_t tavor_qp0_agents_in_fw         = 1;
138 uint32_t tavor_qp1_agents_in_fw         = 0;

140 /*
141  * Whether DMA mappings should be made with DDI_DMA_STREAMING or with
142  * DDI_DMA_CONSISTENT mode.  Note: 0 for "streaming", 1 for "consistent"
143  */
144 uint32_t tavor_streaming_consistent     = 1;

146 /*
147  * For DMA mappings made with DDI_DMA_CONSISTENT, this flag determines
148  * whether to override the necessity for calls to ddi_dma_sync().
149  */
150 uint32_t tavor_consistent_syncoverride  = 0;

152 /*
153  * Whether DMA mappings should bypass the PCI IOMMU or not.
154  * tavor_iommu_bypass is a global setting for all memory addresses.  However,
155  * if set to BYPASS, memory attempted to be registered for streaming (ie:
156  * NON-COHERENT) will necessarily turn off BYPASS for that registration.  To
157  * instead disable streaming in this situation the
158  * 'tavor_disable_streaming_on_bypass' can be set to 1.  This setting will
159  * change the memory mapping to be implicitly consistent (ie: COHERENT), and
160  * will still perform the iommu BYPASS operation.
161  */
162 uint32_t tavor_iommu_bypass             = 1;
163 uint32_t tavor_disable_streaming_on_bypass = 0;

165 /*
166  * Whether QP work queues should be allocated from system memory or
167  * from Tavor DDR memory.  Note: 0 for system memory, 1 for DDR memory
168  */
169 uint32_t tavor_qp_wq_inddr              = 0;

171 /*
172  * Whether SRQ work queues should be allocated from system memory or
173  * from Tavor DDR memory.  Note: 0 for system memory, 1 for DDR memory
174  */
175 uint32_t tavor_srq_wq_inddr             = 0;

177 /*
178  * Whether Tavor should use MSI (Message Signaled Interrupts), if available.
179  * Note: 0 indicates 'legacy interrupt', 1 indicates MSI (if available)
180  */
181 uint32_t tavor_use_msi_if_avail         = 1;

183 /*
184  * This is a patchable variable that determines the time we will wait after
185  * initiating SW reset before we do our first read from Tavor config space.
186  * If this value is set too small (less than the default 100ms), it is
187  * possible for Tavor hardware to be unready to respond to the config cycle
188  * reads.  This could cause master abort on the PCI bridge.  Note: If
189  * "tavor_sw_reset_delay" is set to zero, then no software reset of the Tavor
190  * device will be attempted.
191  */
192 uint32_t tavor_sw_reset_delay           = TAVOR_SW_RESET_DELAY;
```



```
194 /*
195  * These are patchable variables for tavor command polling. The poll_delay is
196  * the number of usec to wait in-between calls to poll the 'go' bit.  The
197  * poll_max is the total number of usec to loop in waiting for the 'go' bit to
198  * clear.
199  */
200 uint32_t tavor_cmd_poll_delay           = TAVOR_CMD_POLL_DELAY;
201 uint32_t tavor_cmd_poll_max             = TAVOR_CMD_POLL_MAX;

203 /*
204  * This is a patchable variable that determines the frequency with which
205  * the AckReq bit will be set in outgoing RC packets.  The AckReq bit will be
206  * set in at least every 2^tavor_qp_ackreq_freq packets (but at least once
207  * per message, i.e. in the last packet).  Tuning this value can increase
208  * IB fabric utilization by cutting down on the number of unnecessary ACKs.
209  */
210 uint32_t tavor_qp_ackreq_freq           = TAVOR_QP_ACKREQ_FREQ;

212 /*
213  * This is a patchable variable that determines the default value for the
214  * maximum number of outstanding split transactions.  The number of
215  * outstanding split transations (i.e. PCI reads) has an affect on device
216  * throughput.  The value here should not be modified as it defines the
217  * default (least common denominator - one (1) PCI read) behavior that is
218  * guaranteed to work, regardless of how the Tavor firmware has been
219  * initialized.  The format for this variable is the same as the corresponding
220  * field in the "PCI-X Command Register".
221  */
222 #ifdef  __sparc
223 /*
224  * Default SPARC platforms to be 1 outstanding PCI read.
225  */
226 int tavor_max_out_splt_trans    = 0;
227 #else
228 /*
229  * Default non-SPARC platforms to be the default as set in tavor firmware
230  * number of outstanding PCI reads.
231  */
232 int tavor_max_out_splt_trans    = -1;
233 #endif

235 /*
236  * This is a patchable variable that determines the default value for the
237  * maximum size of PCI read burst.  This maximum size has an affect on
238  * device throughput.  The value here should not be modified as it defines
239  * the default (least common denominator - 512B read) behavior that is
240  * guaranteed to work, regardless of how the Tavor device has been
241  * initialized.  The format for this variable is the same as the corresponding
242  * field in the "PCI-X Command Register".
243  */
244 #ifdef  __sparc
245 /*
246  * Default SPARC platforms to be 512B read.
247  */
248 int tavor_max_mem_rd_byte_cnt   = 0;
249 static void tavor_check_iommu_bypass(tavor_state_t *state,
250     tavor_cfg_profile_t *cp);
251 #else
252 /*
253  * Default non-SPARC platforms to be the default as set in tavor firmware.
254  *
255  */
256 int tavor_max_mem_rd_byte_cnt   = -1;
257 #endif

259 static void tavor_cfg_wqe_sizes(tavor_cfg_profile_t *cp);
```

```
260 static void tavor_cfg_prop_lookup(tavor_state_t *state,
261     tavor_cfg_profile_t *cp);

263 /*
264  * tavor_cfg_profile_init_phase1()
265  *    Context: Only called from attach() path context
266  */
267 int
268 tavor_cfg_profile_init_phase1(tavor_state_t *state)
269 {
270         tavor_cfg_profile_t     *cp;

272         TAVOR_TNF_ENTER(tavor_cfg_profile_init_phase1);

274         /*
275          * Allocate space for the configuration profile structure
276          */
277         cp = (tavor_cfg_profile_t *)kmem_zalloc(sizeof (tavor_cfg_profile_t),
278             KM_SLEEP);

280         cp->cp_qp0_agents_in_fw         = tavor_qp0_agents_in_fw;
281         cp->cp_qp1_agents_in_fw         = tavor_qp1_agents_in_fw;
282         cp->cp_sw_reset_delay           = tavor_sw_reset_delay;
283         cp->cp_cmd_poll_delay           = tavor_cmd_poll_delay;
284         cp->cp_cmd_poll_max             = tavor_cmd_poll_max;
285         cp->cp_ackreq_freq              = tavor_qp_ackreq_freq;
286         cp->cp_max_out_splt_trans       = tavor_max_out_splt_trans;
287         cp->cp_max_mem_rd_byte_cnt      = tavor_max_mem_rd_byte_cnt;
288         cp->cp_srq_enable               = tavor_srq_enable;
289         cp->cp_fmr_enable               = 0;
290         cp->cp_fmr_max_remaps           = 0;

292         /*
293          * Although most of the configuration is enabled in "phase2" of the
294          * cfg_profile_init, we have to setup the OUT mailboxes here, since
295          * they are used immediately after this "phase1" completes.  Check for
296          * alt_config_enable, and set the values appropriately.  Otherwise, the
297          * config profile is setup using the values based on the dimm size.
298          * While it is expected that the mailbox size and number will remain
299          * the same independent of dimm size, we separate it out here anyway
300          * for completeness.
301          *
302          * We have to setup SRQ settings here because MOD_STAT_CFG must be
303          * called before our call to QUERY_DEVLIM.  If SRQ is enabled, then we
304          * must enable it in the firmware so that the phase2 settings will have
305          * the right device limits.
306          */
307         if (tavor_alt_config_enable) {
308                 cp->cp_log_num_outmbox          = tavor_log_num_outmbox;
309                 cp->cp_log_num_intr_outmbox     = tavor_log_num_intr_outmbox;
310                 cp->cp_log_outmbox_size         = tavor_log_outmbox_size;
311                 cp->cp_log_num_inmbox           = tavor_log_num_inmbox;
312                 cp->cp_log_num_intr_inmbox      = tavor_log_num_intr_inmbox;
313                 cp->cp_log_inmbox_size          = tavor_log_inmbox_size;
314                 cp->cp_log_num_srq              = tavor_log_num_srq;
315                 cp->cp_log_max_srq_sz           = tavor_log_max_srq_sz;

317         } else if (state->ts_cfg_profile_setting >= TAVOR_DDR_SIZE_256) {
318                 cp->cp_log_num_outmbox          = TAVOR_NUM_MAILBOXES_SHIFT;
319                 cp->cp_log_num_intr_outmbox     =
320                     TAVOR_NUM_INTR_MAILBOXES_SHIFT;
321                 cp->cp_log_outmbox_size         = TAVOR_MBOX_SIZE_SHIFT;
322                 cp->cp_log_num_inmbox           = TAVOR_NUM_MAILBOXES_SHIFT;
323                 cp->cp_log_num_intr_inmbox      =
324                     TAVOR_NUM_INTR_MAILBOXES_SHIFT;
325                 cp->cp_log_inmbox_size          = TAVOR_MBOX_SIZE_SHIFT;
```



```
326                 cp->cp_log_num_srq              = TAVOR_NUM_SRQ_SHIFT_256;
327                 cp->cp_log_max_srq_sz           = TAVOR_SRQ_SZ_SHIFT;

329         } else if (state->ts_cfg_profile_setting == TAVOR_DDR_SIZE_128) {
330                 cp->cp_log_num_outmbox          = TAVOR_NUM_MAILBOXES_SHIFT;
331                 cp->cp_log_num_intr_outmbox     =
332                     TAVOR_NUM_INTR_MAILBOXES_SHIFT;
333                 cp->cp_log_outmbox_size         = TAVOR_MBOX_SIZE_SHIFT;
334                 cp->cp_log_num_inmbox           = TAVOR_NUM_MAILBOXES_SHIFT;
335                 cp->cp_log_num_intr_inmbox      =
336                     TAVOR_NUM_INTR_MAILBOXES_SHIFT;
337                 cp->cp_log_inmbox_size          = TAVOR_MBOX_SIZE_SHIFT;
338                 cp->cp_log_num_srq              = TAVOR_NUM_SRQ_SHIFT_128;
339                 cp->cp_log_max_srq_sz           = TAVOR_SRQ_SZ_SHIFT;

341         } else if (state->ts_cfg_profile_setting == TAVOR_DDR_SIZE_MIN) {
342                 cp->cp_log_num_outmbox          = TAVOR_NUM_MAILBOXES_SHIFT;
343                 cp->cp_log_num_intr_outmbox     =
344                     TAVOR_NUM_INTR_MAILBOXES_SHIFT;
345                 cp->cp_log_outmbox_size         = TAVOR_MBOX_SIZE_SHIFT;
346                 cp->cp_log_num_inmbox           = TAVOR_NUM_MAILBOXES_SHIFT;
347                 cp->cp_log_num_intr_inmbox      =
348                     TAVOR_NUM_INTR_MAILBOXES_SHIFT;
349                 cp->cp_log_inmbox_size          = TAVOR_MBOX_SIZE_SHIFT;
350                 cp->cp_log_num_srq              = TAVOR_NUM_SRQ_SHIFT_MIN;
351                 cp->cp_log_max_srq_sz           = TAVOR_SRQ_SZ_SHIFT_MIN;

353         } else {
354                 TNF_PROBE_0(tavor_cfg_profile_invalid_dimmsz_fail,
355                     TAVOR_TNF_ERROR, "");
356                 return (DDI_FAILURE);
357         }

359         /*
360          * Set default DMA mapping mode.  Ensure consistency of flags
361          * with both architecture type and other configuration flags.
362          */
363         if (tavor_streaming_consistent == 0) {
364 #ifdef  __sparc
365                 cp->cp_streaming_consistent = DDI_DMA_STREAMING;

367                 /* Can't do both "streaming" and IOMMU bypass */
368                 if (tavor_iommu_bypass != 0) {
369                         TNF_PROBE_0(tavor_cfg_profile_streamingbypass_fail,
370                             TAVOR_TNF_ERROR, "");
371                         kmem_free(cp, sizeof (tavor_cfg_profile_t));
372                         return (DDI_FAILURE);
373                 }
374 #else
375                 cp->cp_streaming_consistent = DDI_DMA_CONSISTENT;
376 #endif
377         } else {
378                 cp->cp_streaming_consistent = DDI_DMA_CONSISTENT;
379         }

381         /* Determine whether to override ddi_dma_sync() */
382         cp->cp_consistent_syncoverride = tavor_consistent_syncoverride;

384         /* Attach the configuration profile to Tavor softstate */
385         state->ts_cfg_profile = cp;

387         TAVOR_TNF_EXIT(tavor_cfg_profile_init_phase1);
388         return (DDI_SUCCESS);
389 }

391 /*
```

```
 * tavor_cfg_profile_init_phase2()
 *    Context: Only called from attach() path context
 */
int
tavor_cfg_profile_init_phase2(tavor_state_t *state)
{
        tavor_cfg_profile_t     *cp;

        TAVOR_TNF_ENTER(tavor_cfg_profile_init_phase2);

        /* Read the configuration profile from Tavor softstate */
        cp = state->ts_cfg_profile;

        /*
         * Verify the config profile setting.  The 'setting' should already be
         * set, during a call to ddi_dev_regsize() to get the size of DDR
         * memory, or during a fallback to a smaller supported size.  If it is
         * not set, we should not have reached this 'phase2'.  So we assert
         * here.
         */
        ASSERT(state->ts_cfg_profile_setting != 0);

        /*
         * The automatic configuration override is the
         * 'tavor_alt_config_enable' variable.  If this is set, we no longer
         * use the DIMM size to enable the correct profile.  Instead, all of
         * the tavor config options at the top of this file are used directly.
         *
         * This allows customization for a user who knows what they are doing
         * to set tavor configuration values manually.
         *
         * If this variable is 0, we do automatic config for both 128MB and
         * 256MB DIMM sizes.
         */
        if (tavor_alt_config_enable) {
                /*
                 * Initialize the configuration profile
                 */
                cp->cp_log_num_qp               = tavor_log_num_qp;
                cp->cp_log_max_qp_sz            = tavor_log_max_qp_sz;

                /* Determine WQE sizes from requested max SGLs */
                tavor_cfg_wqe_sizes(cp);

                cp->cp_log_num_cq               = tavor_log_num_cq;
                cp->cp_log_max_cq_sz            = tavor_log_max_cq_sz;
                cp->cp_log_default_eq_sz        = tavor_log_default_eq_sz;
                cp->cp_log_num_rdb              = tavor_log_num_rdb;
                cp->cp_log_num_mcg              = tavor_log_num_mcg;
                cp->cp_num_qp_per_mcg           = tavor_num_qp_per_mcg;
                cp->cp_log_num_mcg_hash         = tavor_log_num_mcg_hash;
                cp->cp_log_num_mpt              = tavor_log_num_mpt;
                cp->cp_log_max_mrw_sz           = tavor_log_max_mrw_sz;
                cp->cp_log_num_mttseg           = tavor_log_num_mttseg;
                cp->cp_log_num_uar              = tavor_log_num_uar;
                cp->cp_log_num_pd               = tavor_log_num_pd;
                cp->cp_log_num_ah               = tavor_log_num_ah;
                cp->cp_log_max_pkeytbl          = tavor_log_max_pkeytbl;
                cp->cp_log_max_gidtbl           = tavor_log_max_gidtbl;
                cp->cp_hca_max_rdma_in_qp       = tavor_hca_max_rdma_in_qp;
                cp->cp_hca_max_rdma_out_qp      = tavor_hca_max_rdma_out_qp;
                cp->cp_max_mtu                  = tavor_max_mtu;
                cp->cp_max_port_width           = tavor_max_port_width;
                cp->cp_max_vlcap                = tavor_max_vlcap;
                cp->cp_num_ports                = tavor_num_ports;
                cp->cp_qp0_agents_in_fw         = tavor_qp0_agents_in_fw;
```



```
                cp->cp_qp1_agents_in_fw         = tavor_qp1_agents_in_fw;
                cp->cp_sw_reset_delay           = tavor_sw_reset_delay;
                cp->cp_ackreq_freq              = tavor_qp_ackreq_freq;
                cp->cp_max_out_splt_trans       = tavor_max_out_splt_trans;
                cp->cp_max_mem_rd_byte_cnt      = tavor_max_mem_rd_byte_cnt;

        } else if (state->ts_cfg_profile_setting >= TAVOR_DDR_SIZE_256) {
                /*
                 * Initialize the configuration profile
                 */
                cp->cp_log_num_qp               = TAVOR_NUM_QP_SHIFT_256;
                cp->cp_log_max_qp_sz            = TAVOR_QP_SZ_SHIFT;

                /* Determine WQE sizes from requested max SGLs */
                tavor_cfg_wqe_sizes(cp);

                cp->cp_log_num_cq               = TAVOR_NUM_CQ_SHIFT_256;
                cp->cp_log_max_cq_sz            = TAVOR_CQ_SZ_SHIFT;
                cp->cp_log_default_eq_sz        = TAVOR_DEFAULT_EQ_SZ_SHIFT;
                cp->cp_log_num_rdb              = TAVOR_NUM_RDB_SHIFT_256;
                cp->cp_log_num_mcg              = TAVOR_NUM_MCG_SHIFT;
                cp->cp_num_qp_per_mcg           = TAVOR_NUM_QP_PER_MCG;
                cp->cp_log_num_mcg_hash         = TAVOR_NUM_MCG_HASH_SHIFT;
                cp->cp_log_num_mpt              = TAVOR_NUM_MPT_SHIFT_256;
                cp->cp_log_max_mrw_sz           = TAVOR_MAX_MEM_MPT_SHIFT_256;
                cp->cp_log_num_mttseg           = TAVOR_NUM_MTTSEG_SHIFT;
                cp->cp_log_num_uar              = TAVOR_NUM_UAR_SHIFT;
                cp->cp_log_num_pd               = TAVOR_NUM_PD_SHIFT;
                cp->cp_log_num_ah               = TAVOR_NUM_AH_SHIFT;
                cp->cp_log_max_pkeytbl          = TAVOR_NUM_PKEYTBL_SHIFT;
                cp->cp_log_max_gidtbl           = TAVOR_NUM_GIDTBL_SHIFT;
                cp->cp_hca_max_rdma_in_qp       = TAVOR_HCA_MAX_RDMA_IN_QP;
                cp->cp_hca_max_rdma_out_qp      = TAVOR_HCA_MAX_RDMA_OUT_QP;
                cp->cp_max_mtu                  = TAVOR_MAX_MTU;
                cp->cp_max_port_width           = TAVOR_MAX_PORT_WIDTH;
                cp->cp_max_vlcap                = TAVOR_MAX_VLCAP;
                cp->cp_num_ports                = TAVOR_NUM_PORTS;
                cp->cp_qp0_agents_in_fw         = tavor_qp0_agents_in_fw;
                cp->cp_qp1_agents_in_fw         = tavor_qp1_agents_in_fw;
                cp->cp_sw_reset_delay           = tavor_sw_reset_delay;
                cp->cp_ackreq_freq              = tavor_qp_ackreq_freq;
                cp->cp_max_out_splt_trans       = tavor_max_out_splt_trans;
                cp->cp_max_mem_rd_byte_cnt      = tavor_max_mem_rd_byte_cnt;

        } else if (state->ts_cfg_profile_setting == TAVOR_DDR_SIZE_128) {
                /*
                 * Initialize the configuration profile
                 */
                cp->cp_log_num_qp               = TAVOR_NUM_QP_SHIFT_128;
                cp->cp_log_max_qp_sz            = TAVOR_QP_SZ_SHIFT;

                /* Determine WQE sizes from requested max SGLs */
                tavor_cfg_wqe_sizes(cp);

                cp->cp_log_num_cq               = TAVOR_NUM_CQ_SHIFT_128;
                cp->cp_log_max_cq_sz            = TAVOR_CQ_SZ_SHIFT;
                cp->cp_log_default_eq_sz        = TAVOR_DEFAULT_EQ_SZ_SHIFT;
                cp->cp_log_num_rdb              = TAVOR_NUM_RDB_SHIFT_128;
                cp->cp_log_num_mcg              = TAVOR_NUM_MCG_SHIFT;
                cp->cp_num_qp_per_mcg           = TAVOR_NUM_QP_PER_MCG;
                cp->cp_log_num_mcg_hash         = TAVOR_NUM_MCG_HASH_SHIFT;
                cp->cp_log_num_mpt              = TAVOR_NUM_MPT_SHIFT_128;
                cp->cp_log_max_mrw_sz           = TAVOR_MAX_MEM_MPT_SHIFT_128;
                cp->cp_log_num_mttseg           = TAVOR_NUM_MTTSEG_SHIFT;
                cp->cp_log_num_uar              = TAVOR_NUM_UAR_SHIFT;
                cp->cp_log_num_pd               = TAVOR_NUM_PD_SHIFT;
```

```
524                 cp->cp_log_num_ah               = TAVOR_NUM_AH_SHIFT;
525                 cp->cp_log_max_pkeytbl          = TAVOR_NUM_PKEYTBL_SHIFT;
526                 cp->cp_log_max_gidtbl           = TAVOR_NUM_GIDTBL_SHIFT;
527                 cp->cp_hca_max_rdma_in_qp       = TAVOR_HCA_MAX_RDMA_IN_QP;
528                 cp->cp_hca_max_rdma_out_qp      = TAVOR_HCA_MAX_RDMA_OUT_QP;
529                 cp->cp_max_mtu                  = TAVOR_MAX_MTU;
530                 cp->cp_max_port_width           = TAVOR_MAX_PORT_WIDTH;
531                 cp->cp_max_vlcap                = TAVOR_MAX_VLCAP;
532                 cp->cp_num_ports                = TAVOR_NUM_PORTS;
533                 cp->cp_qp0_agents_in_fw         = tavor_qp0_agents_in_fw;
534                 cp->cp_qp1_agents_in_fw         = tavor_qp1_agents_in_fw;
535                 cp->cp_sw_reset_delay           = tavor_sw_reset_delay;
536                 cp->cp_ackreq_freq              = tavor_qp_ackreq_freq;
537                 cp->cp_max_out_splt_trans       = tavor_max_out_splt_trans;
538                 cp->cp_max_mem_rd_byte_cnt      = tavor_max_mem_rd_byte_cnt;

540         } else if (state->ts_cfg_profile_setting == TAVOR_DDR_SIZE_MIN) {
541                 /*
542                  * Initialize the configuration profile for minimal footprint.
543                  */

545                 cp->cp_log_num_qp               = TAVOR_NUM_QP_SHIFT_MIN;
546                 cp->cp_log_max_qp_sz            = TAVOR_QP_SZ_SHIFT_MIN;

548                 /* Determine WQE sizes from requested max SGLs */
549                 tavor_cfg_wqe_sizes(cp);

551                 cp->cp_log_num_cq               = TAVOR_NUM_CQ_SHIFT_MIN;
552                 cp->cp_log_max_cq_sz            = TAVOR_CQ_SZ_SHIFT_MIN;
553                 cp->cp_log_default_eq_sz        = TAVOR_DEFAULT_EQ_SZ_SHIFT;
554                 cp->cp_log_num_rdb              = TAVOR_NUM_RDB_SHIFT_MIN;
555                 cp->cp_log_num_mcg              = TAVOR_NUM_MCG_SHIFT_MIN;
556                 cp->cp_num_qp_per_mcg           = TAVOR_NUM_QP_PER_MCG_MIN;
557                 cp->cp_log_num_mcg_hash         = TAVOR_NUM_MCG_HASH_SHIFT_MIN;
558                 cp->cp_log_num_mpt              = TAVOR_NUM_MPT_SHIFT_MIN;
559                 cp->cp_log_max_mrw_sz           = TAVOR_MAX_MEM_MPT_SHIFT_MIN;
560                 cp->cp_log_num_mttseg           = TAVOR_NUM_MTTSEG_SHIFT_MIN;
561                 cp->cp_log_num_uar              = TAVOR_NUM_UAR_SHIFT_MIN;
562                 cp->cp_log_num_pd               = TAVOR_NUM_PD_SHIFT;
563                 cp->cp_log_num_ah               = TAVOR_NUM_AH_SHIFT_MIN;
564                 cp->cp_log_max_pkeytbl          = TAVOR_NUM_PKEYTBL_SHIFT;
565                 cp->cp_log_max_gidtbl           = TAVOR_NUM_GIDTBL_SHIFT;
566                 cp->cp_hca_max_rdma_in_qp       = TAVOR_HCA_MAX_RDMA_IN_QP;
567                 cp->cp_hca_max_rdma_out_qp      = TAVOR_HCA_MAX_RDMA_OUT_QP;
568                 cp->cp_max_mtu                  = TAVOR_MAX_MTU;
569                 cp->cp_max_port_width           = TAVOR_MAX_PORT_WIDTH;
570                 cp->cp_max_vlcap                = TAVOR_MAX_VLCAP;
571                 cp->cp_num_ports                = TAVOR_NUM_PORTS;
572                 cp->cp_qp0_agents_in_fw         = tavor_qp0_agents_in_fw;
573                 cp->cp_qp1_agents_in_fw         = tavor_qp1_agents_in_fw;
574                 cp->cp_sw_reset_delay           = tavor_sw_reset_delay;
575                 cp->cp_ackreq_freq              = tavor_qp_ackreq_freq;
576                 cp->cp_max_out_splt_trans       = tavor_max_out_splt_trans;
577                 cp->cp_max_mem_rd_byte_cnt      = tavor_max_mem_rd_byte_cnt;

579         } else {
580                 TNF_PROBE_0(tavor_cfg_profile_invalid_dimmsz_fail,
581                     TAVOR_TNF_ERROR, "");
582                 return (DDI_FAILURE);
583         }

585         /*
586          * Set IOMMU bypass or not.  Ensure consistency of flags with
587          * architecture type.
588          */
589 #ifdef __sparc
```



```
590         if (tavor_iommu_bypass == 1) {
591                 tavor_check_iommu_bypass(state, cp);
592         } else {
593                 cp->cp_iommu_bypass = TAVOR_BINDMEM_NORMAL;
594                 cp->cp_disable_streaming_on_bypass = 0;
595         }
596 #else
597         cp->cp_iommu_bypass = TAVOR_BINDMEM_NORMAL;
598         cp->cp_disable_streaming_on_bypass = 0;
599 #endif
600         /* Set whether QP WQEs will be in DDR or not */
601         cp->cp_qp_wq_inddr = (tavor_qp_wq_inddr == 0) ?
602             TAVOR_QUEUE_LOCATION_NORMAL : TAVOR_QUEUE_LOCATION_INDDR;

604         /* Set whether SRQ WQEs will be in DDR or not */
605         cp->cp_srq_wq_inddr = (tavor_srq_wq_inddr == 0) ?
606             TAVOR_QUEUE_LOCATION_NORMAL : TAVOR_QUEUE_LOCATION_INDDR;

608         cp->cp_use_msi_if_avail = tavor_use_msi_if_avail;

610         /* Determine additional configuration from optional properties */
611         tavor_cfg_prop_lookup(state, cp);

613         TAVOR_TNF_EXIT(tavor_cfg_profile_init_phase2);
614         return (DDI_SUCCESS);
615 }


618 /*
619  * tavor_cfg_profile_fini()
620  *    Context: Only called from attach() and/or detach() path contexts
621  */
622 void
623 tavor_cfg_profile_fini(tavor_state_t *state)
624 {
625         TAVOR_TNF_ENTER(tavor_cfg_profile_fini);

627         /*
628          * Free up the space for configuration profile
629          */
630         kmem_free(state->ts_cfg_profile, sizeof (tavor_cfg_profile_t));

632         TAVOR_TNF_EXIT(tavor_cfg_profile_fini);
633 }


636 /*
637  * tavor_cfg_wqe_sizes()
638  *    Context: Only called from attach() path context
639  */
640 static void
641 tavor_cfg_wqe_sizes(tavor_cfg_profile_t *cp)
642 {
643         uint_t  max_size, log2;
644         uint_t  max_sgl, real_max_sgl;

646         /*
647          * Get the requested maximum number SGL per WQE from the Tavor
648          * patchable variable
649          */
650         max_sgl = tavor_wqe_max_sgl;

652         /*
653          * Use requested maximum number of SGL to calculate the max descriptor
654          * size (while guaranteeing that the descriptor size is a power-of-2
655          * cachelines).  We have to use the calculation for QP1 MLX transport
```

```
656          * because the possibility that we might need to inline a GRH, along
657          * with all the other headers and alignment restrictions, sets the
658          * maximum for the number of SGLs that we can advertise support for.
659          */
660         max_size = (TAVOR_QP_WQE_MLX_QP1_HDRS + (max_sgl << 4));
661         log2 = highbit(max_size);
662         if (ISP2(max_size)) {
 35         if ((max_size & (max_size - 1)) == 0) {
663                 log2 = log2 - 1;
664         }
665         max_size = (1 << log2);

667         /*
668          * Now clip the maximum descriptor size based on Tavor HW maximum
669          */
670         max_size = min(max_size, TAVOR_QP_WQE_MAX_SIZE);

672         /*
673          * Then use the calculated max descriptor size to determine the "real"
674          * maximum SGL (the number beyond which we would roll over to the next
675          * power-of-2).
676          */
677         real_max_sgl = (max_size - TAVOR_QP_WQE_MLX_QP1_HDRS) >> 4;

679         /* Then save away this configuration information */
680         cp->cp_wqe_max_sgl      = max_sgl;
681         cp->cp_wqe_real_max_sgl = real_max_sgl;

683         /* SRQ SGL gets set to it's own patchable variable value */
684         cp->cp_srq_max_sgl              = tavor_srq_max_sgl;
685 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   67087 Thu Oct 23 10:42:14 2014
new/usr/src/uts/common/io/ib/adapters/tavor/tavor_qp.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * tavor_qp.c
 *    Tavor Queue Pair Processing Routines
 *
 *    Implements all the routines necessary for allocating, freeing, and
 *    querying the Tavor queue pairs.
 */

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>
#include <sys/sysmacros.h>

#include <sys/ib/adapters/tavor/tavor.h>
#include <sys/ib/ib_pkt_hdrs.h>

static int tavor_qp_create_qpn(tavor_state_t *state, tavor_qphdl_t qp,
    tavor_rsrc_t *qpc);
static int tavor_qpn_avl_compare(const void *q, const void *e);
static int tavor_special_qp_rsrc_alloc(tavor_state_t *state,
    ibt_sqp_type_t type, uint_t port, tavor_rsrc_t **qp_rsrc);
static int tavor_special_qp_rsrc_free(tavor_state_t *state, ibt_sqp_type_t type,
    uint_t port);
static void tavor_qp_sgl_to_logwqesz(tavor_state_t *state, uint_t num_sgl,
    tavor_qp_wq_type_t wq_type, uint_t *logwqesz, uint_t *max_sgl);

/*
 * tavor_qp_alloc()
 *    Context: Can be called only from user or kernel context.
 */
int
tavor_qp_alloc(tavor_state_t *state, tavor_qp_info_t *qpinfo,
```



```
    uint_t sleepflag, tavor_qp_options_t *op)
{
        tavor_rsrc_pool_info_t  *rsrc_pool;
        tavor_rsrc_t            *qpc, *rsrc, *rdb;
        tavor_umap_db_entry_t   *umapdb;
        tavor_qphdl_t           qp;
        ibt_qp_alloc_attr_t     *attr_p;
        ibt_qp_type_t           type;
        ibtl_qp_hdl_t           ibt_qphdl;
        ibt_chan_sizes_t        *queuesz_p;
        ib_qpn_t                *qpn;
        tavor_qphdl_t           *qphdl;
        ibt_mr_attr_t           mr_attr;
        tavor_mr_options_t      mr_op;
        tavor_srqhdl_t          srq;
        tavor_pdhdl_t           pd;
        tavor_cqhdl_t           sq_cq, rq_cq;
        tavor_mrhdl_t           mr;
        uint64_t                value, qp_desc_off;
        uint32_t                *sq_buf, *rq_buf;
        uint32_t                log_qp_sq_size, log_qp_rq_size;
        uint32_t                sq_size, rq_size;
        uint32_t                sq_wqe_size, rq_wqe_size;
        uint32_t                max_rdb, max_sgl, uarpg;
        uint_t                  wq_location, dma_xfer_mode, qp_is_umap;
        uint_t                  qp_srq_en;
        int                     status, flag;
        char                    *errormsg;

        TAVOR_TNF_ENTER(tavor_qp_alloc);

        _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*attr_p, *queuesz_p))

        /*
         * Check the "options" flag.  Currently this flag tells the driver
         * whether or not the QP's work queues should be come from normal
         * system memory or whether they should be allocated from DDR memory.
         */
        if (op == NULL) {
                wq_location = TAVOR_QUEUE_LOCATION_NORMAL;
        } else {
                wq_location = op->qpo_wq_loc;
        }

        /*
         * Extract the necessary info from the tavor_qp_info_t structure
         */
        attr_p    = qpinfo->qpi_attrp;
        type      = qpinfo->qpi_type;
        ibt_qphdl = qpinfo->qpi_ibt_qphdl;
        queuesz_p = qpinfo->qpi_queueszp;
        qpn       = qpinfo->qpi_qpn;
        qphdl     = &qpinfo->qpi_qphdl;

        /*
         * Determine whether QP is being allocated for userland access or
         * whether it is being allocated for kernel access.  If the QP is
         * being allocated for userland access, then lookup the UAR doorbell
         * page number for the current process.  Note:  If this is not found
         * (e.g. if the process has not previously open()'d the Tavor driver),
         * then an error is returned.
         */
        qp_is_umap = (attr_p->qp_alloc_flags & IBT_QP_USER_MAP) ? 1 : 0;
        if (qp_is_umap) {
                status = tavor_umap_db_find(state->ts_instance, ddi_get_pid(),
                    MLNX_UMAP_UARPG_RSRC, &value, 0, NULL);
```

```
128                 if (status != DDI_SUCCESS) {
129                         /* Set "status" and "errormsg" and goto failure */
130                         TAVOR_TNF_FAIL(IBT_INVALID_PARAM, "failed UAR page");
131                         goto qpalloc_fail;
132                 }
133                 uarpg = ((tavor_rsrc_t *)(uintptr_t)value)->tr_indx;
134         }
135
136         /*
137          * Determine whether QP is being associated with an SRQ
138          */
139         qp_srq_en = (attr_p->qp_alloc_flags & IBT_QP_USES_SRQ) ? 1 : 0;
140         if (qp_srq_en) {
141                 /*
142                  * Check for valid SRQ handle pointers
143                  */
144                 if (attr_p->qp_ibc_srq_hdl == NULL) {
145                         /* Set "status" and "errormsg" and goto failure */
146                         TAVOR_TNF_FAIL(IBT_SRQ_HDL_INVALID,
147                             "invalid SRQ handle");
148                         goto qpalloc_fail;
149                 }
150                 srq = (tavor_srqhdl_t)attr_p->qp_ibc_srq_hdl;
151         }
152
153         /*
154          * Check for valid QP service type (only UD/RC/UC supported)
155          */
156         if (((type != IBT_UD_RQP) && (type != IBT_RC_RQP) &&
157             (type != IBT_UC_RQP))) {
158                 /* Set "status" and "errormsg" and goto failure */
159                 TAVOR_TNF_FAIL(IBT_QP_SRV_TYPE_INVALID, "invalid serv type");
160                 goto qpalloc_fail;
161         }
162
163         /*
164          * Only RC is supported on an SRQ -- This is a Tavor hardware
165          * limitation.  Arbel native mode will not have this shortcoming.
166          */
167         if (qp_srq_en && type != IBT_RC_RQP) {
168                 /* Set "status" and "errormsg" and goto failure */
169                 TAVOR_TNF_FAIL(IBT_INVALID_PARAM, "invalid serv type with SRQ");
170                 goto qpalloc_fail;
171         }
172
173         /*
174          * Check for valid PD handle pointer
175          */
176         if (attr_p->qp_pd_hdl == NULL) {
177                 /* Set "status" and "errormsg" and goto failure */
178                 TAVOR_TNF_FAIL(IBT_PD_HDL_INVALID, "invalid PD handle");
179                 goto qpalloc_fail;
180         }
181         pd = (tavor_pdhdl_t)attr_p->qp_pd_hdl;
182
183         /*
184          * If on an SRQ, check to make sure the PD is the same
185          */
186         if (qp_srq_en && (pd->pd_pdnum != srq->srq_pdhdl->pd_pdnum)) {
187                 /* Set "status" and "errormsg" and goto failure */
188                 TAVOR_TNF_FAIL(IBT_PD_HDL_INVALID, "invalid PD handle");
189                 goto qpalloc_fail;
190         }
191
192         /* Increment the reference count on the protection domain (PD) */
193         tavor_pd_refcnt_inc(pd);
```



```
195         /*
196          * Check for valid CQ handle pointers
197          */
198         if ((attr_p->qp_ibc_scq_hdl == NULL) ||
199             (attr_p->qp_ibc_rcq_hdl == NULL)) {
200                 /* Set "status" and "errormsg" and goto failure */
201                 TAVOR_TNF_FAIL(IBT_CQ_HDL_INVALID, "invalid CQ handle");
202                 goto qpalloc_fail1;
203         }
204         sq_cq = (tavor_cqhdl_t)attr_p->qp_ibc_scq_hdl;
205         rq_cq = (tavor_cqhdl_t)attr_p->qp_ibc_rcq_hdl;
206
207         /*
208          * Increment the reference count on the CQs.  One or both of these
209          * could return error if we determine that the given CQ is already
210          * being used with a special (SMI/GSI) QP.
211          */
212         status = tavor_cq_refcnt_inc(sq_cq, TAVOR_CQ_IS_NORMAL);
213         if (status != DDI_SUCCESS) {
214                 /* Set "status" and "errormsg" and goto failure */
215                 TAVOR_TNF_FAIL(IBT_CQ_HDL_INVALID, "invalid CQ handle");
216                 goto qpalloc_fail1;
217         }
218         status = tavor_cq_refcnt_inc(rq_cq, TAVOR_CQ_IS_NORMAL);
219         if (status != DDI_SUCCESS) {
220                 /* Set "status" and "errormsg" and goto failure */
221                 TAVOR_TNF_FAIL(IBT_CQ_HDL_INVALID, "invalid CQ handle");
222                 goto qpalloc_fail2;
223         }
224
225         /*
226          * Allocate an QP context entry.  This will be filled in with all
227          * the necessary parameters to define the Queue Pair.  Unlike
228          * other Tavor hardware resources, ownership is not immediately
229          * given to hardware in the final step here.  Instead, we must
230          * wait until the QP is later transitioned to the "Init" state before
231          * passing the QP to hardware.  If we fail here, we must undo all
232          * the reference count (CQ and PD).
233          */
234         status = tavor_rsrc_alloc(state, TAVOR_QPC, 1, sleepflag, &qpc);
235         if (status != DDI_SUCCESS) {
236                 /* Set "status" and "errormsg" and goto failure */
237                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed QP context");
238                 goto qpalloc_fail3;
239         }
240
241         /*
242          * Allocate the software structure for tracking the queue pair
243          * (i.e. the Tavor Queue Pair handle).  If we fail here, we must
244          * undo the reference counts and the previous resource allocation.
245          */
246         status = tavor_rsrc_alloc(state, TAVOR_QPHDL, 1, sleepflag, &rsrc);
247         if (status != DDI_SUCCESS) {
248                 /* Set "status" and "errormsg" and goto failure */
249                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed QP handle");
250                 goto qpalloc_fail4;
251         }
252         qp = (tavor_qphdl_t)rsrc->tr_addr;
253         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*qp))
254
255         /*
256          * Calculate the QP number from QPC index.  This routine handles
257          * all of the operations necessary to keep track of used, unused,
258          * and released QP numbers.
259          */
```

```
260         status = tavor_qp_create_qpn(state, qp, qpc);
261         if (status != DDI_SUCCESS) {
262                 /* Set "status" and "errormsg" and goto failure */
263                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed QPN create");
264                 goto qpalloc_fail5;
265         }
266
267         /*
268          * If this will be a user-mappable QP, then allocate an entry for
269          * the "userland resources database".  This will later be added to
270          * the database (after all further QP operations are successful).
271          * If we fail here, we must undo the reference counts and the
272          * previous resource allocation.
273          */
274         if (qp_is_umap) {
275                 umapdb = tavor_umap_db_alloc(state->ts_instance, qp->qp_qpnum,
276                     MLNX_UMAP_QPMEM_RSRC, (uint64_t)(uintptr_t)rsrc);
277                 if (umapdb == NULL) {
278                         /* Set "status" and "errormsg" and goto failure */
279                         TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed umap add");
280                         goto qpalloc_fail6;
281                 }
282         }
283
284         /*
285          * If this is an RC QP, then pre-allocate the maximum number of RDB
286          * entries.  This allows us to ensure that we can later cover all
287          * the resources needed by hardware for handling multiple incoming
288          * RDMA Reads.  Note: These resources are obviously not always
289          * necessary.  They are allocated here anyway.  Someday maybe this
290          * can be modified to allocate these on-the-fly (i.e. only if RDMA
291          * Read or Atomic operations are enabled) XXX
292          * If we fail here, we have a bunch of resource and reference count
293          * cleanup to do.
294          */
295         if (type == IBT_RC_RQP) {
296                 max_rdb = state->ts_cfg_profile->cp_hca_max_rdma_in_qp;
297                 status = tavor_rsrc_alloc(state, TAVOR_RDB, max_rdb,
298                     sleepflag, &rdb);
299                 if (status != DDI_SUCCESS) {
300                         /* Set "status" and "errormsg" and goto failure */
301                         TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed RDB");
302                         goto qpalloc_fail7;
303                 }
304                 qp->qp_rdbrsrcp = rdb;
305                 /* Calculate offset (into DDR memory) of RDB entries */
306                 rsrc_pool = &state->ts_rsrc_hdl[TAVOR_RDB];
307                 qp->qp_rdb_ddraddr = (uintptr_t)rsrc_pool->rsrc_ddr_offset +
308                     (rdb->tr_indx << TAVOR_RDB_SIZE_SHIFT);
309         }
310
311         /*
312          * Calculate the appropriate size for the work queues.
313          * Note:  All Tavor QP work queues must be a power-of-2 in size.  Also
314          * they may not be any smaller than TAVOR_QP_MIN_SIZE.  This step is
315          * to round the requested size up to the next highest power-of-2
316          */
317         attr_p->qp_sizes.cs_sq = max(attr_p->qp_sizes.cs_sq, TAVOR_QP_MIN_SIZE);
318         attr_p->qp_sizes.cs_rq = max(attr_p->qp_sizes.cs_rq, TAVOR_QP_MIN_SIZE);
319         log_qp_sq_size = highbit(attr_p->qp_sizes.cs_sq);
320         if (ISP2(attr_p->qp_sizes.cs_sq)) {
320         if ((attr_p->qp_sizes.cs_sq & (attr_p->qp_sizes.cs_sq - 1)) == 0) {
321                 log_qp_sq_size = log_qp_sq_size - 1;
322         }
323         log_qp_rq_size = highbit(attr_p->qp_sizes.cs_rq);
324         if (ISP2(attr_p->qp_sizes.cs_rq)) {
```



```
324         if ((attr_p->qp_sizes.cs_rq & (attr_p->qp_sizes.cs_rq - 1)) == 0) {
325                 log_qp_rq_size = log_qp_rq_size - 1;
326         }
327
328         /*
329          * Next we verify that the rounded-up size is valid (i.e. consistent
330          * with the device limits and/or software-configured limits).  If not,
331          * then obviously we have a lot of cleanup to do before returning.
332          */
333         if ((log_qp_sq_size > state->ts_cfg_profile->cp_log_max_qp_sz) ||
334             (!qp_srq_en && (log_qp_rq_size >
335             state->ts_cfg_profile->cp_log_max_qp_sz))) {
336                 /* Set "status" and "errormsg" and goto failure */
337                 TAVOR_TNF_FAIL(IBT_HCA_WR_EXCEEDED, "max QP size");
338                 goto qpalloc_fail8;
339         }
340
341         /*
342          * Next we verify that the requested number of SGL is valid (i.e.
343          * consistent with the device limits and/or software-configured
344          * limits).  If not, then obviously the same cleanup needs to be done.
345          */
346         max_sgl = state->ts_cfg_profile->cp_wqe_real_max_sgl;
347         if ((attr_p->qp_sizes.cs_sq_sgl > max_sgl) ||
348             (!qp_srq_en && (attr_p->qp_sizes.cs_rq_sgl > max_sgl))) {
349                 /* Set "status" and "errormsg" and goto failure */
350                 TAVOR_TNF_FAIL(IBT_HCA_SGL_EXCEEDED, "max QP SGL");
351                 goto qpalloc_fail8;
352         }
353
354         /*
355          * Determine this QP's WQE sizes (for both the Send and Recv WQEs).
356          * This will depend on the requested number of SGLs.  Note: this
357          * has the side-effect of also calculating the real number of SGLs
358          * (for the calculated WQE size).
359          *
360          * For QP's on an SRQ, we set these to 0.
361          */
362         if (qp_srq_en) {
363                 qp->qp_rq_log_wqesz = 0;
364                 qp->qp_rq_sgl = 0;
365         } else {
366                 tavor_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_rq_sgl,
367                     TAVOR_QP_WQ_TYPE_RECVQ, &qp->qp_rq_log_wqesz,
368                     &qp->qp_rq_sgl);
369         }
370         tavor_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
371             TAVOR_QP_WQ_TYPE_SENDQ, &qp->qp_sq_log_wqesz, &qp->qp_sq_sgl);
372
373         /*
374          * Allocate the memory for QP work queues.  Note:  The location from
375          * which we will allocate these work queues has been passed in
376          * through the tavor_qp_options_t structure.  Since Tavor work queues
377          * are not allowed to cross a 32-bit (4GB) boundary, the alignment of
378          * the work queue memory is very important.  We used to allocate
379          * work queues (the combined receive and send queues) so that they
380          * would be aligned on their combined size.  That alignment guaranteed
381          * that they would never cross the 4GB boundary (Tavor work queues
382          * are on the order of MBs at maximum).  Now we are able to relax
383          * this alignment constraint by ensuring that the IB address assigned
384          * to the queue memory (as a result of the tavor_mr_register() call)
385          * is offset from zero.
386          * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
387          * guarantee the alignment, but when attempting to use IOMMU bypass
388          * mode we found that we were not allowed to specify any alignment
389          * that was more restrictive than the system page size.
```

```
390          * So we avoided this constraint by passing two alignment values,
391          * one for the memory allocation itself and the other for the DMA
392          * handle (for later bind).  This used to cause more memory than
393          * necessary to be allocated (in order to guarantee the more
394          * restrictive alignment contraint).  But be guaranteeing the
395          * zero-based IB virtual address for the queue, we are able to
396          * conserve this memory.
397          * Note: If QP is not user-mappable, then it may come from either
398          * kernel system memory or from HCA-attached local DDR memory.
399          */
400         sq_wqe_size = 1 << qp->qp_sq_log_wqesz;
401         sq_size     = (1 << log_qp_sq_size) * sq_wqe_size;

403         /* QP on SRQ sets these to 0 */
404         if (qp_srq_en) {
405                 rq_wqe_size = 0;
406                 rq_size     = 0;
407         } else {
408                 rq_wqe_size = 1 << qp->qp_rq_log_wqesz;
409                 rq_size     = (1 << log_qp_rq_size) * rq_wqe_size;
410         }

412         qp->qp_wqinfo.qa_size = sq_size + rq_size;
413         qp->qp_wqinfo.qa_alloc_align = max(sq_wqe_size, rq_wqe_size);
414         qp->qp_wqinfo.qa_bind_align  = max(sq_wqe_size, rq_wqe_size);
415         if (qp_is_umap) {
416                 qp->qp_wqinfo.qa_location = TAVOR_QUEUE_LOCATION_USERLAND;
417         } else {
418                 qp->qp_wqinfo.qa_location = wq_location;
419         }
420         status = tavor_queue_alloc(state, &qp->qp_wqinfo, sleepflag);
421         if (status != DDI_SUCCESS) {
422                 /* Set "status" and "errormsg" and goto failure */
423                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed work queue");
424                 goto qpalloc_fail8;
425         }
426         if (sq_wqe_size > rq_wqe_size) {
427                 sq_buf = qp->qp_wqinfo.qa_buf_aligned;

429                 /*
430                  * If QP's on an SRQ, we set the rq_buf to NULL
431                  */
432                 if (qp_srq_en)
433                         rq_buf = NULL;
434                 else
435                         rq_buf = (uint32_t *)((uintptr_t)sq_buf + sq_size);
436         } else {
437                 rq_buf = qp->qp_wqinfo.qa_buf_aligned;
438                 sq_buf = (uint32_t *)((uintptr_t)rq_buf + rq_size);
439         }

441         /*
442          * Register the memory for the QP work queues.  The memory for the
443          * QP must be registered in the Tavor TPT tables.  This gives us the
444          * LKey to specify in the QP context later.  Note: The memory for
445          * Tavor work queues (both Send and Recv) must be contiguous and
446          * registered as a single memory region.  Note also: If the work
447          * queue is to be allocated from DDR memory, then only a "bypass"
448          * mapping is appropriate.  And if the QP memory is user-mappable,
449          * then we force DDI_DMA_CONSISTENT mapping.
450          * Also, in order to meet the alignment restriction, we pass the
451          * "mro_bind_override_addr" flag in the call to tavor_mr_register().
452          * This guarantees that the resulting IB vaddr will be zero-based
453          * (modulo the offset into the first page).
454          * If we fail here, we still have the bunch of resource and reference
455          * count cleanup to do.
```



```
456          */
457         flag = (sleepflag == TAVOR_SLEEP) ? IBT_MR_SLEEP :
458             IBT_MR_NOSLEEP;
459         mr_attr.mr_vaddr    = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned;
460         mr_attr.mr_len      = qp->qp_wqinfo.qa_size;
461         mr_attr.mr_as       = NULL;
462         mr_attr.mr_flags    = flag;
463         if (qp_is_umap) {
464                 mr_op.mro_bind_type = state->ts_cfg_profile->cp_iommu_bypass;
465         } else {
466                 if (wq_location == TAVOR_QUEUE_LOCATION_NORMAL) {
467                         mr_op.mro_bind_type =
468                             state->ts_cfg_profile->cp_iommu_bypass;
469                         dma_xfer_mode =
470                             state->ts_cfg_profile->cp_streaming_consistent;
471                         if (dma_xfer_mode == DDI_DMA_STREAMING) {
472                                 mr_attr.mr_flags |= IBT_MR_NONCOHERENT;
473                         }
474                 } else {
475                         mr_op.mro_bind_type = TAVOR_BINDMEM_BYPASS;
476                 }
477         }
478         mr_op.mro_bind_dmahdl = qp->qp_wqinfo.qa_dmahdl;
479         mr_op.mro_bind_override_addr = 1;
480         status = tavor_mr_register(state, pd, &mr_attr, &mr, &mr_op);
481         if (status != DDI_SUCCESS) {
482                 /* Set "status" and "errormsg" and goto failure */
483                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed register mr");
484                 goto qpalloc_fail9;
485         }

487         /*
488          * Calculate the offset between the kernel virtual address space
489          * and the IB virtual address space.  This will be used when
490          * posting work requests to properly initialize each WQE.
491          */
492         qp_desc_off = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned -
493             (uint64_t)mr->mr_bindinfo.bi_addr;

495         /*
496          * Fill in all the return arguments (if necessary).  This includes
497          * real work queue sizes, real SGLs, and QP number
498          */
499         if (queuesz_p != NULL) {
500                 queuesz_p->cs_sq        = (1 << log_qp_sq_size);
501                 queuesz_p->cs_sq_sgl    = qp->qp_sq_sgl;

503                 /* QP on an SRQ set these to 0 */
504                 if (qp_srq_en) {
505                         queuesz_p->cs_rq        = 0;
506                         queuesz_p->cs_rq_sgl    = 0;
507                 } else {
508                         queuesz_p->cs_rq        = (1 << log_qp_rq_size);
509                         queuesz_p->cs_rq_sgl    = qp->qp_rq_sgl;
510                 }
511         }
512         if (qpn != NULL) {
513                 *qpn = (ib_qpn_t)qp->qp_qpnum;
514         }

516         /*
517          * Fill in the rest of the Tavor Queue Pair handle.  We can update
518          * the following fields for use in further operations on the QP.
519          */
520         qp->qp_qpcrsrcp         = qpc;
521         qp->qp_rsrcp            = rsrc;
```

```
522         qp->qp_state            = TAVOR_QP_RESET;
523         qp->qp_pdhdl            = pd;
524         qp->qp_mrhdl            = mr;
525         qp->qp_sq_sigtype       = (attr_p->qp_flags & IBT_WR_SIGNALED) ?
526             TAVOR_QP_SQ_WR_SIGNALED : TAVOR_QP_SQ_ALL_SIGNALED;
527         qp->qp_is_special       = 0;
528         qp->qp_is_umap          = qp_is_umap;
529         qp->qp_uarpg            = (qp->qp_is_umap) ? uarpg : 0;
530         qp->qp_umap_dhp         = (devmap_cookie_t)NULL;
531         qp->qp_sq_cqhdl         = sq_cq;
532         qp->qp_sq_lastwqeaddr   = NULL;
533         qp->qp_sq_bufsz         = (1 << log_qp_sq_size);
534         qp->qp_sq_buf           = sq_buf;
535         qp->qp_desc_off         = qp_desc_off;
536         qp->qp_rq_cqhdl         = rq_cq;
537         qp->qp_rq_lastwqeaddr   = NULL;
538         qp->qp_rq_buf           = rq_buf;

540         /* QP on an SRQ sets this to 0 */
541         if (qp_srq_en) {
542                 qp->qp_rq_bufsz         = 0;
543         } else {
544                 qp->qp_rq_bufsz         = (1 << log_qp_rq_size);
545         }

547         qp->qp_forward_sqd_event  = 0;
548         qp->qp_sqd_still_draining = 0;
549         qp->qp_hdlrarg          = (void *)ibt_qphdl;
550         qp->qp_mcg_refcnt       = 0;

552         /*
553          * If this QP is to be associated with an SRQ, then set the SRQ handle
554          * appropriately.
555          */
556         if (qp_srq_en) {
557                 qp->qp_srqhdl = srq;
558                 qp->qp_srq_en = TAVOR_QP_SRQ_ENABLED;
559                 tavor_srq_refcnt_inc(qp->qp_srqhdl);
560         } else {
561                 qp->qp_srqhdl = NULL;
562                 qp->qp_srq_en = TAVOR_QP_SRQ_DISABLED;
563         }

565         /* Determine if later ddi_dma_sync will be necessary */
566         qp->qp_sync = TAVOR_QP_IS_SYNC_REQ(state, qp->qp_wqinfo);

568         /* Determine the QP service type */
569         if (type == IBT_RC_RQP) {
570                 qp->qp_serv_type = TAVOR_QP_RC;
571         } else if (type == IBT_UD_RQP) {
572                 qp->qp_serv_type = TAVOR_QP_UD;
573         } else {
574                 qp->qp_serv_type = TAVOR_QP_UC;
575         }

577         /* Zero out the QP context */
578         bzero(&qp->qpc, sizeof (tavor_hw_qpc_t));

580         /*
581          * Put QP handle in Tavor QPNum-to-QPHdl list.  Then fill in the
582          * "qphdl" and return success
583          */
584         ASSERT(state->ts_qphdl[qpc->tr_indx] == NULL);
585         state->ts_qphdl[qpc->tr_indx] = qp;

587         /*
```



```
588          * If this is a user-mappable QP, then we need to insert the previously
589          * allocated entry into the "userland resources database".  This will
590          * allow for later lookup during devmap() (i.e. mmap()) calls.
591          */
592         if (qp_is_umap) {
593                 tavor_umap_db_add(umapdb);
594         }

596         *qphdl = qp;

598         TAVOR_TNF_EXIT(tavor_qp_alloc);
599         return (DDI_SUCCESS);

601 /*
602  * The following is cleanup for all possible failure cases in this routine
603  */
604 qpalloc_fail9:
605         tavor_queue_free(state, &qp->qp_wqinfo);
606 qpalloc_fail8:
607         if (type == IBT_RC_RQP) {
608                 tavor_rsrc_free(state, &rdb);
609         }
610 qpalloc_fail7:
611         if (qp_is_umap) {
612                 tavor_umap_db_free(umapdb);
613         }
614 qpalloc_fail6:
615         /*
616          * Releasing the QPN will also free up the QPC context.  Update
617          * the QPC context pointer to indicate this.
618          */
619         tavor_qp_release_qpn(state, qp->qp_qpn_hdl, TAVOR_QPN_RELEASE);
620         qpc = NULL;
621 qpalloc_fail5:
622         tavor_rsrc_free(state, &rsrc);
623 qpalloc_fail4:
624         if (qpc) {
625                 tavor_rsrc_free(state, &qpc);
626         }
627 qpalloc_fail3:
628         tavor_cq_refcnt_dec(rq_cq);
629 qpalloc_fail2:
630         tavor_cq_refcnt_dec(sq_cq);
631 qpalloc_fail1:
632         tavor_pd_refcnt_dec(pd);
633 qpalloc_fail:
634         TNF_PROBE_1(tavor_qp_alloc_fail, TAVOR_TNF_ERROR, "",
635             tnf_string, msg, errormsg);
636         TAVOR_TNF_EXIT(tavor_qp_alloc);
637         return (status);
638 }



642 /*
643  * tavor_special_qp_alloc()
644  *    Context: Can be called only from user or kernel context.
645  */
646 int
647 tavor_special_qp_alloc(tavor_state_t *state, tavor_qp_info_t *qpinfo,
648     uint_t sleepflag, tavor_qp_options_t *op)
649 {
650         tavor_rsrc_t            *qpc, *rsrc;
651         tavor_qphdl_t           qp;
652         ibt_qp_alloc_attr_t     *attr_p;
653         ibt_sqp_type_t          type;
```

```
654         uint8_t                 port;
655         ibtl_qp_hdl_t           ibt_qphdl;
656         ibt_chan_sizes_t        *queuesz_p;
657         tavor_qphdl_t           *qphdl;
658         ibt_mr_attr_t           mr_attr;
659         tavor_mr_options_t      mr_op;
660         tavor_pdhdl_t           pd;
661         tavor_cqhdl_t           sq_cq, rq_cq;
662         tavor_mrhdl_t           mr;
663         uint64_t                qp_desc_off;
664         uint32_t                *sq_buf, *rq_buf;
665         uint32_t                log_qp_sq_size, log_qp_rq_size;
666         uint32_t                sq_size, rq_size, max_sgl;
667         uint32_t                sq_wqe_size, rq_wqe_size;
668         uint_t                  wq_location, dma_xfer_mode;
669         int                     status, flag;
670         char                    *errormsg;

672         TAVOR_TNF_ENTER(tavor_special_qp_alloc);

674         /*
675          * Check the "options" flag.  Currently this flag tells the driver
676          * whether or not the QP's work queues should be come from normal
677          * system memory or whether they should be allocated from DDR memory.
678          */
679         if (op == NULL) {
680                 wq_location = TAVOR_QUEUE_LOCATION_NORMAL;
681         } else {
682                 wq_location = op->qpo_wq_loc;
683         }

685         /*
686          * Extract the necessary info from the tavor_qp_info_t structure
687          */
688         attr_p    = qpinfo->qpi_attrp;
689         type      = qpinfo->qpi_type;
690         port      = qpinfo->qpi_port;
691         ibt_qphdl = qpinfo->qpi_ibt_qphdl;
692         queuesz_p = qpinfo->qpi_queueszp;
693         qphdl     = &qpinfo->qpi_qphdl;

695         /*
696          * Check for valid special QP type (only SMI & GSI supported)
697          */
698         if ((type != IBT_SMI_SQP) && (type != IBT_GSI_SQP)) {
699                 /* Set "status" and "errormsg" and goto failure */
700                 TAVOR_TNF_FAIL(IBT_QP_SPECIAL_TYPE_INVALID, "invalid QP type");
701                 goto spec_qpalloc_fail;
702         }

704         /*
705          * Check for valid port number
706          */
707         if (!tavor_portnum_is_valid(state, port)) {
708                 /* Set "status" and "errormsg" and goto failure */
709                 TAVOR_TNF_FAIL(IBT_HCA_PORT_INVALID, "invalid port num");
710                 goto spec_qpalloc_fail;
711         }
712         port = port - 1;

714         /*
715          * Check for valid PD handle pointer
716          */
717         if (attr_p->qp_pd_hdl == NULL) {
718                 /* Set "status" and "errormsg" and goto failure */
719                 TAVOR_TNF_FAIL(IBT_PD_HDL_INVALID, "invalid PD handle");
```



```
720                 goto spec_qpalloc_fail;
721         }
722         pd = (tavor_pdhdl_t)attr_p->qp_pd_hdl;

724         /* Increment the reference count on the PD */
725         tavor_pd_refcnt_inc(pd);

727         /*
728          * Check for valid CQ handle pointers
729          */
730         if ((attr_p->qp_ibc_scq_hdl == NULL) ||
731             (attr_p->qp_ibc_rcq_hdl == NULL)) {
732                 /* Set "status" and "errormsg" and goto failure */
733                 TAVOR_TNF_FAIL(IBT_CQ_HDL_INVALID, "invalid CQ handle");
734                 goto spec_qpalloc_fail1;
735         }
736         sq_cq = (tavor_cqhdl_t)attr_p->qp_ibc_scq_hdl;
737         rq_cq = (tavor_cqhdl_t)attr_p->qp_ibc_rcq_hdl;

739         /*
740          * Increment the reference count on the CQs.  One or both of these
741          * could return error if we determine that the given CQ is already
742          * being used with a non-special QP (i.e. a normal QP).
743          */
744         status = tavor_cq_refcnt_inc(sq_cq, TAVOR_CQ_IS_SPECIAL);
745         if (status != DDI_SUCCESS) {
746                 /* Set "status" and "errormsg" and goto failure */
747                 TAVOR_TNF_FAIL(IBT_CQ_HDL_INVALID, "invalid CQ handle");
748                 goto spec_qpalloc_fail1;
749         }
750         status = tavor_cq_refcnt_inc(rq_cq, TAVOR_CQ_IS_SPECIAL);
751         if (status != DDI_SUCCESS) {
752                 /* Set "status" and "errormsg" and goto failure */
753                 TAVOR_TNF_FAIL(IBT_CQ_HDL_INVALID, "invalid CQ handle");
754                 goto spec_qpalloc_fail2;
755         }

757         /*
758          * Allocate the special QP resources.  Essentially, this allocation
759          * amounts to checking if the request special QP has already been
760          * allocated.  If successful, the QP context return is an actual
761          * QP context that has been "aliased" to act as a special QP of the
762          * appropriate type (and for the appropriate port).  Just as in
763          * tavor_qp_alloc() above, ownership for this QP context is not
764          * immediately given to hardware in the final step here.  Instead, we
765          * wait until the QP is later transitioned to the "Init" state before
766          * passing the QP to hardware.  If we fail here, we must undo all
767          * the reference count (CQ and PD).
768          */
769         status = tavor_special_qp_rsrc_alloc(state, type, port, &qpc);
770         if (status != DDI_SUCCESS) {
771                 /* Set "status" and "errormsg" and goto failure */
772                 TAVOR_TNF_FAIL(status, "failed special QP rsrc");
773                 goto spec_qpalloc_fail3;
774         }

776         /*
777          * Allocate the software structure for tracking the special queue
778          * pair (i.e. the Tavor Queue Pair handle).  If we fail here, we
779          * must undo the reference counts and the previous resource allocation.
780          */
781         status = tavor_rsrc_alloc(state, TAVOR_QPHDL, 1, sleepflag, &rsrc);
782         if (status != DDI_SUCCESS) {
783                 /* Set "status" and "errormsg" and goto failure */
784                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed QP handle");
785                 goto spec_qpalloc_fail4;
```

```
786         }
787         qp = (tavor_qphdl_t)rsrc->tr_addr;
788         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*qp))

790         /*
791          * Actual QP number is a combination of the index of the QPC and
792          * the port number.  This is because the special QP contexts must
793          * be allocated two-at-a-time.
794          */
795         qp->qp_qpnum = qpc->tr_indx + port;

797         /*
798          * Calculate the appropriate size for the work queues.
799          * Note:  All Tavor QP work queues must be a power-of-2 in size.  Also
800          * they may not be any smaller than TAVOR_QP_MIN_SIZE.  This step is
801          * to round the requested size up to the next highest power-of-2
802          */
803         attr_p->qp_sizes.cs_sq = max(attr_p->qp_sizes.cs_sq, TAVOR_QP_MIN_SIZE);
804         attr_p->qp_sizes.cs_rq = max(attr_p->qp_sizes.cs_rq, TAVOR_QP_MIN_SIZE);
805         log_qp_sq_size = highbit(attr_p->qp_sizes.cs_sq);
806         if (ISP2(attr_p->qp_sizes.cs_sq)) {
806         if ((attr_p->qp_sizes.cs_sq & (attr_p->qp_sizes.cs_sq - 1)) == 0) {
807                 log_qp_sq_size = log_qp_sq_size - 1;
808         }
809         log_qp_rq_size = highbit(attr_p->qp_sizes.cs_rq);
810         if (ISP2(attr_p->qp_sizes.cs_rq)) {
810         if ((attr_p->qp_sizes.cs_rq & (attr_p->qp_sizes.cs_rq - 1)) == 0) {
811                 log_qp_rq_size = log_qp_rq_size - 1;
812         }

814         /*
815          * Next we verify that the rounded-up size is valid (i.e. consistent
816          * with the device limits and/or software-configured limits).  If not,
817          * then obviously we have a bit of cleanup to do before returning.
818          */
819         if ((log_qp_sq_size > state->ts_cfg_profile->cp_log_max_qp_sz) ||
820             (log_qp_rq_size > state->ts_cfg_profile->cp_log_max_qp_sz)) {
821                 /* Set "status" and "errormsg" and goto failure */
822                 TAVOR_TNF_FAIL(IBT_HCA_WR_EXCEEDED, "max QP size");
823                 goto spec_qpalloc_fail5;
824         }

826         /*
827          * Next we verify that the requested number of SGL is valid (i.e.
828          * consistent with the device limits and/or software-configured
829          * limits).  If not, then obviously the same cleanup needs to be done.
830          */
831         max_sgl = state->ts_cfg_profile->cp_wqe_real_max_sgl;
832         if ((attr_p->qp_sizes.cs_sq_sgl > max_sgl) ||
833             (attr_p->qp_sizes.cs_rq_sgl > max_sgl)) {
834                 /* Set "status" and "errormsg" and goto failure */
835                 TAVOR_TNF_FAIL(IBT_HCA_SGL_EXCEEDED, "max QP SGL");
836                 goto spec_qpalloc_fail5;
837         }

839         /*
840          * Determine this QP's WQE sizes (for both the Send and Recv WQEs).
841          * This will depend on the requested number of SGLs.  Note: this
842          * has the side-effect of also calculating the real number of SGLs
843          * (for the calculated WQE size).
844          */
845         tavor_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_rq_sgl,
846             TAVOR_QP_WQ_TYPE_RECVQ, &qp->qp_rq_log_wqesz, &qp->qp_rq_sgl);
847         if (type == IBT_SMI_SQP) {
848                 tavor_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
849                     TAVOR_QP_WQ_TYPE_SENDMLX_QP0, &qp->qp_sq_log_wqesz,
```



```
850                     &qp->qp_sq_sgl);
851         } else {
852                 tavor_qp_sgl_to_logwqesz(state, attr_p->qp_sizes.cs_sq_sgl,
853                     TAVOR_QP_WQ_TYPE_SENDMLX_QP1, &qp->qp_sq_log_wqesz,
854                     &qp->qp_sq_sgl);
855         }

857         /*
858          * Allocate the memory for QP work queues.  Note:  The location from
859          * which we will allocate these work queues has been passed in
860          * through the tavor_qp_options_t structure.  Since Tavor work queues
861          * are not allowed to cross a 32-bit (4GB) boundary, the alignment of
862          * the work queue memory is very important.  We used to allocate
863          * work queues (the combined receive and send queues) so that they
864          * would be aligned on their combined size.  That alignment guaranteed
865          * that they would never cross the 4GB boundary (Tavor work queues
866          * are on the order of MBs at maximum).  Now we are able to relax
867          * this alignment constraint by ensuring that the IB address assigned
868          * to the queue memory (as a result of the tavor_mr_register() call)
869          * is offset from zero.
870          * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
871          * guarantee the alignment, but when attempting to use IOMMU bypass
872          * mode we found that we were not allowed to specify any alignment
873          * that was more restrictive than the system page size.
874          * So we avoided this constraint by passing two alignment values,
875          * one for the memory allocation itself and the other for the DMA
876          * handle (for later bind).  This used to cause more memory than
877          * necessary to be allocated (in order to guarantee the more
878          * restrictive alignment contraint).  But be guaranteeing the
879          * zero-based IB virtual address for the queue, we are able to
880          * conserve this memory.
881          */
882         sq_wqe_size = 1 << qp->qp_sq_log_wqesz;
883         rq_wqe_size = 1 << qp->qp_rq_log_wqesz;
884         sq_size     = (1 << log_qp_sq_size) * sq_wqe_size;
885         rq_size     = (1 << log_qp_rq_size) * rq_wqe_size;
886         qp->qp_wqinfo.qa_size     = sq_size + rq_size;
887         qp->qp_wqinfo.qa_alloc_align = max(sq_wqe_size, rq_wqe_size);
888         qp->qp_wqinfo.qa_bind_align  = max(sq_wqe_size, rq_wqe_size);
889         qp->qp_wqinfo.qa_location = wq_location;
890         status = tavor_queue_alloc(state, &qp->qp_wqinfo, sleepflag);
891         if (status != NULL) {
892                 /* Set "status" and "errormsg" and goto failure */
893                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed work queue");
894                 goto spec_qpalloc_fail5;
895         }
896         if (sq_wqe_size > rq_wqe_size) {
897                 sq_buf = qp->qp_wqinfo.qa_buf_aligned;
898                 rq_buf = (uint32_t *)((uintptr_t)sq_buf + sq_size);
899         } else {
900                 rq_buf = qp->qp_wqinfo.qa_buf_aligned;
901                 sq_buf = (uint32_t *)((uintptr_t)rq_buf + rq_size);
902         }

904         /*
905          * Register the memory for the special QP work queues.  The memory for
906          * the special QP must be registered in the Tavor TPT tables.  This
907          * gives us the LKey to specify in the QP context later.  Note: The
908          * memory for Tavor work queues (both Send and Recv) must be contiguous
909          * and registered as a single memory region.  Note also: If the work
910          * queue is to be allocated from DDR memory, then only a "bypass"
911          * mapping is appropriate.
912          * Also, in order to meet the alignment restriction, we pass the
913          * "mro_bind_override_addr" flag in the call to tavor_mr_register().
914          * This guarantees that the resulting IB vaddr will be zero-based
915          * (modulo the offset into the first page).
```

```
916          * If we fail here, we have a bunch of resource and reference count
917          * cleanup to do.
918          */
919         flag = (sleepflag == TAVOR_SLEEP) ? IBT_MR_SLEEP :
920             IBT_MR_NOSLEEP;
921         mr_attr.mr_vaddr    = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned;
922         mr_attr.mr_len      = qp->qp_wqinfo.qa_size;
923         mr_attr.mr_as       = NULL;
924         mr_attr.mr_flags    = flag;
925         if (wq_location == TAVOR_QUEUE_LOCATION_NORMAL) {
926                 mr_op.mro_bind_type = state->ts_cfg_profile->cp_iommu_bypass;

928                 dma_xfer_mode = state->ts_cfg_profile->cp_streaming_consistent;
929                 if (dma_xfer_mode == DDI_DMA_STREAMING) {
930                         mr_attr.mr_flags |= IBT_MR_NONCOHERENT;
931                 }
932         } else {
933                 mr_op.mro_bind_type = TAVOR_BINDMEM_BYPASS;
934         }
935         mr_op.mro_bind_dmahdl = qp->qp_wqinfo.qa_dmahdl;
936         mr_op.mro_bind_override_addr = 1;
937         status = tavor_mr_register(state, pd, &mr_attr, &mr, &mr_op);
938         if (status != DDI_SUCCESS) {
939                 /* Set "status" and "errormsg" and goto failure */
940                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed register mr");
941                 goto spec_qpalloc_fail6;
942         }

944         /*
945          * Calculate the offset between the kernel virtual address space
946          * and the IB virtual address space.  This will be used when
947          * posting work requests to properly initialize each WQE.
948          */
949         qp_desc_off = (uint64_t)(uintptr_t)qp->qp_wqinfo.qa_buf_aligned -
950             (uint64_t)mr->mr_bindinfo.bi_addr;

952         /*
953          * Fill in all the return arguments (if necessary).  This includes
954          * real work queue sizes, real SGLs, and QP number (which will be
955          * either zero or one, depending on the special QP type)
956          */
957         if (queuesz_p != NULL) {
958                 queuesz_p->cs_sq        = (1 << log_qp_sq_size);
959                 queuesz_p->cs_sq_sgl    = qp->qp_sq_sgl;
960                 queuesz_p->cs_rq        = (1 << log_qp_rq_size);
961                 queuesz_p->cs_rq_sgl    = qp->qp_rq_sgl;
962         }

964         /*
965          * Fill in the rest of the Tavor Queue Pair handle.  We can update
966          * the following fields for use in further operations on the QP.
967          */
968         qp->qp_qpcrsrcp         = qpc;
969         qp->qp_rsrcp            = rsrc;
970         qp->qp_state            = TAVOR_QP_RESET;
971         qp->qp_pdhdl            = pd;
972         qp->qp_mrhdl            = mr;
973         qp->qp_sq_sigtype       = (attr_p->qp_flags & IBT_WR_SIGNALED) ?
974             TAVOR_QP_SQ_WR_SIGNALED : TAVOR_QP_SQ_ALL_SIGNALED;
975         qp->qp_is_special       = (type == IBT_SMI_SQP) ?
976             TAVOR_QP_SMI : TAVOR_QP_GSI;
977         qp->qp_is_umap          = 0;
978         qp->qp_uarpg            = 0;
979         qp->qp_sq_cqhdl         = sq_cq;
980         qp->qp_sq_lastwqeaddr   = NULL;
981         qp->qp_sq_bufsz         = (1 << log_qp_sq_size);
```



```
982         qp->qp_sq_buf           = sq_buf;
983         qp->qp_desc_off         = qp_desc_off;
984         qp->qp_rq_cqhdl         = rq_cq;
985         qp->qp_rq_lastwqeaddr   = NULL;
986         qp->qp_rq_bufsz         = (1 << log_qp_rq_size);
987         qp->qp_rq_buf           = rq_buf;
988         qp->qp_portnum          = port;
989         qp->qp_pkeyindx         = 0;
990         qp->qp_hdlrarg          = (void *)ibt_qphdl;
991         qp->qp_mcg_refcnt       = 0;
992         qp->qp_srq_en           = 0;
993         qp->qp_srqhdl           = NULL;

995         /* Determine if later ddi_dma_sync will be necessary */
996         qp->qp_sync = TAVOR_QP_IS_SYNC_REQ(state, qp->qp_wqinfo);

998         /* All special QPs are UD QP service type */
999         qp->qp_serv_type = TAVOR_QP_UD;

1001        /* Zero out the QP context */
1002        bzero(&qp->qpc, sizeof (tavor_hw_qpc_t));

1004        /*
1005         * Put QP handle in Tavor QPNum-to-QPHdl list.  Then fill in the
1006         * "qphdl" and return success
1007         */
1008        ASSERT(state->ts_qphdl[qpc->tr_indx + port] == NULL);
1009        state->ts_qphdl[qpc->tr_indx + port] = qp;

1011        *qphdl = qp;

1013        TAVOR_TNF_EXIT(tavor_special_qp_alloc);
1014        return (DDI_SUCCESS);

1016 /*
1017  * The following is cleanup for all possible failure cases in this routine
1018  */
1019 spec_qpalloc_fail6:
1020        tavor_queue_free(state, &qp->qp_wqinfo);
1021 spec_qpalloc_fail5:
1022        tavor_rsrc_free(state, &rsrc);
1023 spec_qpalloc_fail4:
1024        if (tavor_special_qp_rsrc_free(state, type, port) != DDI_SUCCESS) {
1025                TAVOR_WARNING(state, "failed to free special QP rsrc");
1026        }
1027 spec_qpalloc_fail3:
1028        tavor_cq_refcnt_dec(rq_cq);
1029 spec_qpalloc_fail2:
1030        tavor_cq_refcnt_dec(sq_cq);
1031 spec_qpalloc_fail1:
1032        tavor_pd_refcnt_dec(pd);
1033 spec_qpalloc_fail:
1034        TNF_PROBE_1(tavor_special_qp_alloc_fail, TAVOR_TNF_ERROR, "",
1035            tnf_string, msg, errormsg);
1036        TAVOR_TNF_EXIT(tavor_special_qp_alloc);
1037        return (status);
1038 }
```

unchanged_portion_omitted_

```
1973 /*
1974  * tavor_qp_sgl_to_logwqesz()
1975  *    Context: Can be called from interrupt or base context.
1976  */
1977 static void
1978 tavor_qp_sgl_to_logwqesz(tavor_state_t *state, uint_t num_sgl,
```

```
1979     tavor_qp_wq_type_t wq_type, uint_t *logwqesz, uint_t *max_sgl)
1980 {
1981         uint_t  max_size, log2, actual_sgl;

1983         TAVOR_TNF_ENTER(tavor_qp_sgl_to_logwqesz);

1985         switch (wq_type) {
1986         case TAVOR_QP_WQ_TYPE_SENDQ:
1987                 /*
1988                  * Use requested maximum SGL to calculate max descriptor size
1989                  * (while guaranteeing that the descriptor size is a
1990                  * power-of-2 cachelines).
1991                  */
1992                 max_size = (TAVOR_QP_WQE_MLX_SND_HDRS + (num_sgl << 4));
1993                 log2 = highbit(max_size);
1994                 if (ISP2(max_size)) {
1994                 if ((max_size & (max_size - 1)) == 0) {
1995                         log2 = log2 - 1;
1996                 }

1998                 /* Make sure descriptor is at least the minimum size */
1999                 log2 = max(log2, TAVOR_QP_WQE_LOG_MINIMUM);

2001                 /* Calculate actual number of SGL (given WQE size) */
2002                 actual_sgl = ((1 << log2) - TAVOR_QP_WQE_MLX_SND_HDRS) >> 4;
2003                 break;

2005         case TAVOR_QP_WQ_TYPE_RECVQ:
2006                 /*
2007                  * Same as above (except for Recv WQEs)
2008                  */
2009                 max_size = (TAVOR_QP_WQE_MLX_RCV_HDRS + (num_sgl << 4));
2010                 log2 = highbit(max_size);
2011                 if (ISP2(max_size)) {
2011                 if ((max_size & (max_size - 1)) == 0) {
2012                         log2 = log2 - 1;
2013                 }

2015                 /* Make sure descriptor is at least the minimum size */
2016                 log2 = max(log2, TAVOR_QP_WQE_LOG_MINIMUM);

2018                 /* Calculate actual number of SGL (given WQE size) */
2019                 actual_sgl = ((1 << log2) - TAVOR_QP_WQE_MLX_RCV_HDRS) >> 4;
2020                 break;

2022         case TAVOR_QP_WQ_TYPE_SENDMLX_QP0:
2023                 /*
2024                  * Same as above (except for MLX transport WQEs).  For these
2025                  * WQEs we have to account for the space consumed by the
2026                  * "inline" packet headers.  (This is smaller than for QP1
2027                  * below because QP0 is not allowed to send packets with a GRH.
2028                  */
2029                 max_size = (TAVOR_QP_WQE_MLX_QP0_HDRS + (num_sgl << 4));
2030                 log2 = highbit(max_size);
2031                 if (ISP2(max_size)) {
2031                 if ((max_size & (max_size - 1)) == 0) {
2032                         log2 = log2 - 1;
2033                 }

2035                 /* Make sure descriptor is at least the minimum size */
2036                 log2 = max(log2, TAVOR_QP_WQE_LOG_MINIMUM);

2038                 /* Calculate actual number of SGL (given WQE size) */
2039                 actual_sgl = ((1 << log2) - TAVOR_QP_WQE_MLX_QP0_HDRS) >> 4;
2040                 break;
```



```
2042         case TAVOR_QP_WQ_TYPE_SENDMLX_QP1:
2043                 /*
2044                  * Same as above.  For these WQEs we again have to account for
2045                  * the space consumed by the "inline" packet headers.  (This
2046                  * is larger than for QP0 above because we have to account for
2047                  * the possibility of a GRH in each packet - and this
2048                  * introduces an alignment issue that causes us to consume
2049                  * an additional 8 bytes).
2050                  */
2051                 max_size = (TAVOR_QP_WQE_MLX_QP1_HDRS + (num_sgl << 4));
2052                 log2 = highbit(max_size);
2053                 if (ISP2(max_size)) {
2053                 if ((max_size & (max_size - 1)) == 0) {
2054                         log2 = log2 - 1;
2055                 }

2057                 /* Make sure descriptor is at least the minimum size */
2058                 log2 = max(log2, TAVOR_QP_WQE_LOG_MINIMUM);

2060                 /* Calculate actual number of SGL (given WQE size) */
2061                 actual_sgl = ((1 << log2) - TAVOR_QP_WQE_MLX_QP1_HDRS) >> 4;
2062                 break;

2064         default:
2065                 TAVOR_WARNING(state, "unexpected work queue type");
2066                 TNF_PROBE_0(tavor_qp_sgl_to_logwqesz_inv_wqtype_fail,
2067                     TAVOR_TNF_ERROR, "");
2068                 break;
2069         }

2071         /* Fill in the return values */
2072         *logwqesz = log2;
2073         *max_sgl  = min(state->ts_cfg_profile->cp_wqe_real_max_sgl, actual_sgl);

2075         TAVOR_TNF_EXIT(tavor_qp_sgl_to_logwqesz);
2076 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  104422 Thu Oct 23 10:42:14 2014
new/usr/src/uts/common/io/ib/adapters/tavor/tavor_qpmod.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * tavor_qpmod.c
 *    Tavor Queue Pair Modify Routines
 *
 *    This contains all the routines necessary to implement the Tavor
 *    ModifyQP() verb.  This includes all the code for legal transitions to
 *    and from Reset, Init, RTR, RTS, SQD, SQErr, and Error.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/tavor/tavor.h>
#include <sys/ib/ib_pkt_hdrs.h>

static int tavor_qp_reset2init(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_qp_info_t *info_p);
static int tavor_qp_init2init(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int tavor_qp_init2rtr(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int tavor_qp_rtr2rts(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int tavor_qp_rts2rts(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int tavor_qp_rts2sqd(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags);
static int tavor_qp_sqd2rts(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
```



```
static int tavor_qp_sqd2sqd(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int tavor_qp_sqerr2rts(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p);
static int tavor_qp_to_error(tavor_state_t *state, tavor_qphdl_t qp);
static int tavor_qp_reset2err(tavor_state_t *state, tavor_qphdl_t qp);

static uint_t tavor_check_rdma_enable_flags(ibt_cep_modify_flags_t flags,
    ibt_qp_info_t *info_p, tavor_hw_qpc_t *qpc);
static int tavor_qp_validate_resp_rsrc(tavor_state_t *state,
    ibt_qp_rc_attr_t *rc, uint_t *rra_max);
static int tavor_qp_validate_init_depth(tavor_state_t *state,
    ibt_qp_rc_attr_t *rc, uint_t *sra_max);
static int tavor_qp_validate_mtu(tavor_state_t *state, uint_t mtu);

/*
 * tavor_qp_modify()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
int
tavor_qp_modify(tavor_state_t *state, tavor_qphdl_t qp,
    ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p,
    ibt_queue_sizes_t *actual_sz)
{
        ibt_cep_state_t         cur_state, mod_state;
        ibt_cep_modify_flags_t  okflags;
        int                     status;
        char                    *errormsg;

        TAVOR_TNF_ENTER(tavor_qp_modify);

        /*
         * Lock the QP so that we can modify it atomically.  After grabbing
         * the lock, get the current QP state.  We will use this current QP
         * state to determine the legal transitions (and the checks that need
         * to be performed.)
         * Below you will find a case for every possible QP state.  In each
         * case we check that no flags are set which are not valid for the
         * possible transitions from that state.  If these tests pass (and if
         * the state transition we are attempting is legal), then we call
         * one of the helper functions.  Each of these functions does some
         * additional setup before posting a Tavor firmware command for the
         * appropriate state transition.
         */
        mutex_enter(&qp->qp_lock);

        /*
         * Verify that the transport type matches between the serv_type and the
         * qp_trans.  A caller to IBT must specify the qp_trans field as
         * IBT_UD_SRV, IBT_RC_SRV, or IBT_UC_SRV, depending on the QP.  We
         * check here that the correct value was specified, based on our
         * understanding of the QP serv type.
         *
         * Because callers specify part of a 'union' based on what QP type they
         * think they're working with, this ensures that we do not pickup bogus
         * data if the caller thought they were working with a different QP
         * type.
         */
        if (!(TAVOR_QP_TYPE_VALID(info_p->qp_trans, qp->qp_serv_type))) {
                mutex_exit(&qp->qp_lock);
                TNF_PROBE_1(tavor_qp_modify_inv_qp_trans_fail,
                    TAVOR_TNF_ERROR, "", tnf_uint, qptrans,
                    info_p->qp_trans);
                TAVOR_TNF_EXIT(tavor_qp_modify);
                return (IBT_QP_SRV_TYPE_INVALID);
```

```
128         }
129
130         /*
131          * If this is a transition to RTS (which is valid from RTR, RTS,
132          * SQError, and SQ Drain) then we should honor the "current QP state"
133          * specified by the consumer.  This means converting the IBTF QP state
134          * in "info_p->qp_current_state" to a Tavor QP state.  Otherwise, we
135          * assume that we already know the current state (i.e. whatever it was
136          * last modified to or queried as - in "qp->qp_state").
137          */
138         mod_state = info_p->qp_state;
139
140         if (flags & IBT_CEP_SET_RTR_RTS) {
141                 cur_state = TAVOR_QP_RTR;               /* Ready to Receive */
142
143         } else if ((flags & IBT_CEP_SET_STATE) &&
144             (mod_state == IBT_STATE_RTS)) {
145
146                 /* Convert the current IBTF QP state to a Tavor QP state */
147                 switch (info_p->qp_current_state) {
148                 case IBT_STATE_RTR:
149                         cur_state = TAVOR_QP_RTR;       /* Ready to Receive */
150                         break;
151                 case IBT_STATE_RTS:
152                         cur_state = TAVOR_QP_RTS;       /* Ready to Send */
153                         break;
154                 case IBT_STATE_SQE:
155                         cur_state = TAVOR_QP_SQERR;     /* Send Queue Error */
156                         break;
157                 case IBT_STATE_SQD:
158                         cur_state = TAVOR_QP_SQD;       /* SQ Drained */
159                         break;
160                 default:
161                         mutex_exit(&qp->qp_lock);
162                         TNF_PROBE_1(tavor_qp_modify_inv_currqpstate_fail,
163                             TAVOR_TNF_ERROR, "", tnf_uint, qpstate,
164                             info_p->qp_current_state);
165                         TAVOR_TNF_EXIT(tavor_qp_modify);
166                         return (IBT_QP_STATE_INVALID);
167                 }
168         } else {
169                 cur_state = qp->qp_state;
170         }
171
172         switch (cur_state) {
173         case TAVOR_QP_RESET:
174                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RESET_INIT |
175                     IBT_CEP_SET_RDMA_R | IBT_CEP_SET_RDMA_W |
176                     IBT_CEP_SET_ATOMIC | IBT_CEP_SET_PKEY_IX |
177                     IBT_CEP_SET_PORT | IBT_CEP_SET_QKEY);
178
179                 /*
180                  * Check for attempts to modify invalid attributes from the
181                  * "Reset" state
182                  */
183                 if (flags & ~okflags) {
184                         mutex_exit(&qp->qp_lock);
185                         /* Set "status" and "errormsg" and goto failure */
186                         TAVOR_TNF_FAIL(IBT_QP_ATTR_RO, "reset: invalid flag");
187                         goto qpmod_fail;
188                 }
189
190                 /*
191                  * Verify state transition is to either "Init", back to
192                  * "Reset", or to "Error".
193                  */
```



```
194                 if ((flags & IBT_CEP_SET_RESET_INIT) &&
195                     (flags & IBT_CEP_SET_STATE) &&
196                     (mod_state != IBT_STATE_INIT)) {
197                         /* Invalid transition - ambiguous flags */
198                         mutex_exit(&qp->qp_lock);
199                         /* Set "status" and "errormsg" and goto failure */
200                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
201                             "reset: ambiguous flags");
202                         goto qpmod_fail;
203
204                 } else if ((flags & IBT_CEP_SET_RESET_INIT) ||
205                     ((flags & IBT_CEP_SET_STATE) &&
206                     (mod_state == IBT_STATE_INIT))) {
207                         /*
208                          * Attempt to transition from "Reset" to "Init"
209                          */
210                         status = tavor_qp_reset2init(state, qp, info_p);
211                         if (status != DDI_SUCCESS) {
212                                 mutex_exit(&qp->qp_lock);
213                                 /* Set "status"/"errormsg", goto failure */
214                                 TAVOR_TNF_FAIL(status, "reset to init");
215                                 goto qpmod_fail;
216                         }
217                         qp->qp_state = TAVOR_QP_INIT;
218
219                 } else if ((flags & IBT_CEP_SET_STATE) &&
220                     (mod_state == IBT_STATE_RESET)) {
221                         /*
222                          * Attempt to transition from "Reset" back to "Reset"
223                          *    Nothing to do here really... just drop the lock
224                          *    and return success.  The qp->qp_state should
225                          *    already be set to TAVOR_QP_RESET.
226                          *
227                          * Note: We return here because we do not want to fall
228                          *    through to the tavor_wrid_from_reset_handling()
229                          *    routine below (since we are not really moving
230                          *    _out_ of the "Reset" state.
231                          */
232                         mutex_exit(&qp->qp_lock);
233                         TNF_PROBE_0_DEBUG(tavor_qp_modify_rst2rst,
234                             TAVOR_TNF_TRACE, "");
235                         TAVOR_TNF_EXIT(tavor_qp_modify);
236                         return (DDI_SUCCESS);
237
238                 } else if ((flags & IBT_CEP_SET_STATE) &&
239                     (mod_state == IBT_STATE_ERROR)) {
240                         /*
241                          * Attempt to transition from "Reset" to "Error"
242                          */
243                         status = tavor_qp_reset2err(state, qp);
244                         if (status != DDI_SUCCESS) {
245                                 mutex_exit(&qp->qp_lock);
246                                 /* Set "status"/"errormsg", goto failure */
247                                 TAVOR_TNF_FAIL(status, "reset to error");
248                                 goto qpmod_fail;
249                         }
250                         qp->qp_state = TAVOR_QP_ERR;
251
252                 } else {
253                         /* Invalid transition - return error */
254                         mutex_exit(&qp->qp_lock);
255                         /* Set "status" and "errormsg" and goto failure */
256                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
257                             "reset: invalid transition");
258                         goto qpmod_fail;
259                 }
```

```
261                 /*
262                  * Do any additional handling necessary here for the transition
263                  * from the "Reset" state (e.g. re-initialize the workQ WRID
264                  * lists).  Note: If tavor_wrid_from_reset_handling() fails,
265                  * then we attempt to transition the QP back to the "Reset"
266                  * state.  If that fails, then it is an indication of a serious
267                  * problem (either HW or SW).  So we print out a warning
268                  * message and return failure.
269                  */
270                 status = tavor_wrid_from_reset_handling(state, qp);
271                 if (status != DDI_SUCCESS) {
272                         if (tavor_qp_to_reset(state, qp) != DDI_SUCCESS) {
273                                 TAVOR_WARNING(state, "failed to reset QP");
274                         }
275                         qp->qp_state = TAVOR_QP_RESET;
276
277                         mutex_exit(&qp->qp_lock);
278                         /* Set "status" and "errormsg" and goto failure */
279                         TAVOR_TNF_FAIL(status, "reset: wrid_from_reset hdl");
280                         goto qpmod_fail;
281                 }
282                 break;
283
284         case TAVOR_QP_INIT:
285                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_INIT_RTR |
286                     IBT_CEP_SET_ADDS_VECT | IBT_CEP_SET_RDMARA_IN |
287                     IBT_CEP_SET_MIN_RNR_NAK | IBT_CEP_SET_ALT_PATH |
288                     IBT_CEP_SET_RDMA_R | IBT_CEP_SET_RDMA_W |
289                     IBT_CEP_SET_ATOMIC | IBT_CEP_SET_PKEY_IX |
290                     IBT_CEP_SET_QKEY | IBT_CEP_SET_PORT);
291
292                 /*
293                  * Check for attempts to modify invalid attributes from the
294                  * "Init" state
295                  */
296                 if (flags & ~okflags) {
297                         mutex_exit(&qp->qp_lock);
298                         /* Set "status" and "errormsg" and goto failure */
299                         TAVOR_TNF_FAIL(IBT_QP_ATTR_RO, "init: invalid flag");
300                         goto qpmod_fail;
301                 }
302
303                 /*
304                  * Verify state transition is to either "RTR", back to "Init",
305                  * to "Reset", or to "Error"
306                  */
307                 if ((flags & IBT_CEP_SET_INIT_RTR) &&
308                     (flags & IBT_CEP_SET_STATE) &&
309                     (mod_state != IBT_STATE_RTR)) {
310                         /* Invalid transition - ambiguous flags */
311                         mutex_exit(&qp->qp_lock);
312                         /* Set "status" and "errormsg" and goto failure */
313                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
314                             "init: ambiguous flags");
315                         goto qpmod_fail;
316
317                 } else if ((flags & IBT_CEP_SET_INIT_RTR) ||
318                     ((flags & IBT_CEP_SET_STATE) &&
319                     (mod_state == IBT_STATE_RTR))) {
320                         /*
321                          * Attempt to transition from "Init" to "RTR"
322                          */
323                         status = tavor_qp_init2rtr(state, qp, flags, info_p);
324                         if (status != DDI_SUCCESS) {
325                                 mutex_exit(&qp->qp_lock);
```



```
326                                 /* Set "status"/"errormsg", goto failure */
327                                 TAVOR_TNF_FAIL(status, "init to rtr");
328                                 goto qpmod_fail;
329                         }
330                         qp->qp_state = TAVOR_QP_RTR;
331
332                 } else if ((flags & IBT_CEP_SET_STATE) &&
333                     (mod_state == IBT_STATE_INIT)) {
334                         /*
335                          * Attempt to transition from "Init" to "Init"
336                          */
337                         status = tavor_qp_init2init(state, qp, flags, info_p);
338                         if (status != DDI_SUCCESS) {
339                                 mutex_exit(&qp->qp_lock);
340                                 /* Set "status"/"errormsg", goto failure */
341                                 TAVOR_TNF_FAIL(status, "init to init");
342                                 goto qpmod_fail;
343                         }
344                         qp->qp_state = TAVOR_QP_INIT;
345
346                 } else if ((flags & IBT_CEP_SET_STATE) &&
347                     (mod_state == IBT_STATE_RESET)) {
348                         /*
349                          * Attempt to transition from "Init" to "Reset"
350                          */
351                         status = tavor_qp_to_reset(state, qp);
352                         if (status != DDI_SUCCESS) {
353                                 mutex_exit(&qp->qp_lock);
354                                 /* Set "status"/"errormsg", goto failure */
355                                 TAVOR_TNF_FAIL(status, "init to reset");
356                                 goto qpmod_fail;
357                         }
358                         qp->qp_state = TAVOR_QP_RESET;
359
360                         /*
361                          * Do any additional handling necessary for the
362                          * transition _to_ the "Reset" state (e.g. update the
363                          * workQ WRID lists)
364                          */
365                         tavor_wrid_to_reset_handling(state, qp);
366
367                 } else if ((flags & IBT_CEP_SET_STATE) &&
368                     (mod_state == IBT_STATE_ERROR)) {
369                         /*
370                          * Attempt to transition from "Init" to "Error"
371                          */
372                         status = tavor_qp_to_error(state, qp);
373                         if (status != DDI_SUCCESS) {
374                                 mutex_exit(&qp->qp_lock);
375                                 /* Set "status"/"errormsg", goto failure */
376                                 TAVOR_TNF_FAIL(status, "init to error");
377                                 goto qpmod_fail;
378                         }
379                         qp->qp_state = TAVOR_QP_ERR;
380
381                 } else {
382                         /* Invalid transition - return error */
383                         mutex_exit(&qp->qp_lock);
384                         /* Set "status" and "errormsg" and goto failure */
385                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
386                             "init: invalid transition");
387                         goto qpmod_fail;
388                 }
389                 break;
390
391         case TAVOR_QP_RTR:
```

```
392                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RTR_RTS |
393                     IBT_CEP_SET_TIMEOUT | IBT_CEP_SET_RETRY |
394                     IBT_CEP_SET_RNR_NAK_RETRY | IBT_CEP_SET_RDMARA_OUT |
395                     IBT_CEP_SET_RDMA_R | IBT_CEP_SET_RDMA_W |
396                     IBT_CEP_SET_ATOMIC | IBT_CEP_SET_QKEY |
397                     IBT_CEP_SET_ALT_PATH | IBT_CEP_SET_MIG |
398                     IBT_CEP_SET_MIN_RNR_NAK);
399
400                 /*
401                  * Check for attempts to modify invalid attributes from the
402                  * "RTR" state
403                  */
404                 if (flags & ~okflags) {
405                         mutex_exit(&qp->qp_lock);
406                         /* Set "status" and "errormsg" and goto failure */
407                         TAVOR_TNF_FAIL(IBT_QP_ATTR_RO, "rtr: invalid flag");
408                         goto qpmod_fail;
409                 }
410
411                 /*
412                  * Verify state transition is to either "RTS", "Reset",
413                  * or "Error"
414                  */
415                 if ((flags & IBT_CEP_SET_RTR_RTS) &&
416                     (flags & IBT_CEP_SET_STATE) &&
417                     (mod_state != IBT_STATE_RTS)) {
418                         /* Invalid transition - ambiguous flags */
419                         mutex_exit(&qp->qp_lock);
420                         /* Set "status" and "errormsg" and goto failure */
421                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
422                             "reset: ambiguous flags");
423                         goto qpmod_fail;
424
425                 } else if ((flags & IBT_CEP_SET_RTR_RTS) ||
426                     ((flags & IBT_CEP_SET_STATE) &&
427                     (mod_state == IBT_STATE_RTS))) {
428                         /*
429                          * Attempt to transition from "RTR" to "RTS"
430                          */
431                         status = tavor_qp_rtr2rts(state, qp, flags, info_p);
432                         if (status != DDI_SUCCESS) {
433                                 mutex_exit(&qp->qp_lock);
434                                 /* Set "status"/"errormsg", goto failure */
435                                 TAVOR_TNF_FAIL(status, "rtr to rts");
436                                 goto qpmod_fail;
437                         }
438                         qp->qp_state = TAVOR_QP_RTS;
439
440                 } else if ((flags & IBT_CEP_SET_STATE) &&
441                     (mod_state == IBT_STATE_RESET)) {
442                         /*
443                          * Attempt to transition from "RTR" to "Reset"
444                          */
445                         status = tavor_qp_to_reset(state, qp);
446                         if (status != DDI_SUCCESS) {
447                                 mutex_exit(&qp->qp_lock);
448                                 /* Set "status"/"errormsg", goto failure */
449                                 TAVOR_TNF_FAIL(status, "rtr to reset");
450                                 goto qpmod_fail;
451                         }
452                         qp->qp_state = TAVOR_QP_RESET;
453
454                         /*
455                          * Do any additional handling necessary for the
456                          * transition _to_ the "Reset" state (e.g. update the
457                          * workQ WRID lists)
```



```
458                          */
459                         tavor_wrid_to_reset_handling(state, qp);
460
461                 } else if ((flags & IBT_CEP_SET_STATE) &&
462                     (mod_state == IBT_STATE_ERROR)) {
463                         /*
464                          * Attempt to transition from "RTR" to "Error"
465                          */
466                         status = tavor_qp_to_error(state, qp);
467                         if (status != DDI_SUCCESS) {
468                                 mutex_exit(&qp->qp_lock);
469                                 /* Set "status"/"errormsg", goto failure */
470                                 TAVOR_TNF_FAIL(status, "rtr to error");
471                                 goto qpmod_fail;
472                         }
473                         qp->qp_state = TAVOR_QP_ERR;
474
475                 } else {
476                         /* Invalid transition - return error */
477                         mutex_exit(&qp->qp_lock);
478                         /* Set "status" and "errormsg" and goto failure */
479                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
480                             "rtr: invalid transition");
481                         goto qpmod_fail;
482                 }
483                 break;
484
485         case TAVOR_QP_RTS:
486                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RDMA_R |
487                     IBT_CEP_SET_RDMA_W | IBT_CEP_SET_ATOMIC |
488                     IBT_CEP_SET_QKEY | IBT_CEP_SET_ALT_PATH |
489                     IBT_CEP_SET_MIG | IBT_CEP_SET_MIN_RNR_NAK |
490                     IBT_CEP_SET_SQD_EVENT);
491
492                 /*
493                  * Check for attempts to modify invalid attributes from the
494                  * "RTS" state
495                  */
496                 if (flags & ~okflags) {
497                         mutex_exit(&qp->qp_lock);
498                         /* Set "status" and "errormsg" and goto failure */
499                         TAVOR_TNF_FAIL(IBT_QP_ATTR_RO, "rts: invalid flag");
500                         goto qpmod_fail;
501                 }
502
503                 /*
504                  * Verify state transition is to either "RTS", "SQD", "Reset",
505                  * or "Error"
506                  */
507                 if ((flags & IBT_CEP_SET_STATE) &&
508                     (mod_state == IBT_STATE_RTS)) {
509                         /*
510                          * Attempt to transition from "RTS" to "RTS"
511                          */
512                         status = tavor_qp_rts2rts(state, qp, flags, info_p);
513                         if (status != DDI_SUCCESS) {
514                                 mutex_exit(&qp->qp_lock);
515                                 /* Set "status"/"errormsg", goto failure */
516                                 TAVOR_TNF_FAIL(status, "rts to rts");
517                                 goto qpmod_fail;
518                         }
519                         /* qp->qp_state = TAVOR_QP_RTS; */
520
521                 } else if ((flags & IBT_CEP_SET_STATE) &&
522                     (mod_state == IBT_STATE_SQD)) {
523                         /*
```

```
524                          * Attempt to transition from "RTS" to "SQD"
525                          */
526                         status = tavor_qp_rts2sqd(state, qp, flags);
527                         if (status != DDI_SUCCESS) {
528                                 mutex_exit(&qp->qp_lock);
529                                 /* Set "status"/"errormsg", goto failure */
530                                 TAVOR_TNF_FAIL(status, "rts to sqd");
531                                 goto qpmod_fail;
532                         }
533                         qp->qp_state = TAVOR_QP_SQD;
534
535                 } else if ((flags & IBT_CEP_SET_STATE) &&
536                     (mod_state == IBT_STATE_RESET)) {
537                         /*
538                          * Attempt to transition from "RTS" to "Reset"
539                          */
540                         status = tavor_qp_to_reset(state, qp);
541                         if (status != DDI_SUCCESS) {
542                                 mutex_exit(&qp->qp_lock);
543                                 /* Set "status"/"errormsg", goto failure */
544                                 TAVOR_TNF_FAIL(status, "rts to reset");
545                                 goto qpmod_fail;
546                         }
547                         qp->qp_state = TAVOR_QP_RESET;
548
549                         /*
550                          * Do any additional handling necessary for the
551                          * transition _to_ the "Reset" state (e.g. update the
552                          * workQ WRID lists)
553                          */
554                         tavor_wrid_to_reset_handling(state, qp);
555
556                 } else if ((flags & IBT_CEP_SET_STATE) &&
557                     (mod_state == IBT_STATE_ERROR)) {
558                         /*
559                          * Attempt to transition from "RTS" to "Error"
560                          */
561                         status = tavor_qp_to_error(state, qp);
562                         if (status != DDI_SUCCESS) {
563                                 mutex_exit(&qp->qp_lock);
564                                 /* Set "status"/"errormsg", goto failure */
565                                 TAVOR_TNF_FAIL(status, "rts to error");
566                                 goto qpmod_fail;
567                         }
568                         qp->qp_state = TAVOR_QP_ERR;
569
570                 } else {
571                         /* Invalid transition - return error */
572                         mutex_exit(&qp->qp_lock);
573                         /* Set "status" and "errormsg" and goto failure */
574                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
575                             "rts: invalid transition");
576                         goto qpmod_fail;
577                 }
578                 break;
579
580         case TAVOR_QP_SQERR:
581                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_RDMA_R |
582                     IBT_CEP_SET_RDMA_W | IBT_CEP_SET_ATOMIC |
583                     IBT_CEP_SET_QKEY | IBT_CEP_SET_MIN_RNR_NAK);
584
585                 /*
586                  * Check for attempts to modify invalid attributes from the
587                  * "SQErr" state
588                  */
589                 if (flags & ~okflags) {
```



```
590                         mutex_exit(&qp->qp_lock);
591                         /* Set "status" and "errormsg" and goto failure */
592                         TAVOR_TNF_FAIL(IBT_QP_ATTR_RO, "sqerr: invalid flag");
593                         goto qpmod_fail;
594                 }
595
596                 /*
597                  * Verify state transition is to either "RTS", "Reset", or
598                  * "Error"
599                  */
600                 if ((flags & IBT_CEP_SET_STATE) &&
601                     (mod_state == IBT_STATE_RTS)) {
602                         /*
603                          * Attempt to transition from "SQErr" to "RTS"
604                          */
605                         status = tavor_qp_sqerr2rts(state, qp, flags, info_p);
606                         if (status != DDI_SUCCESS) {
607                                 mutex_exit(&qp->qp_lock);
608                                 /* Set "status"/"errormsg", goto failure */
609                                 TAVOR_TNF_FAIL(status, "sqerr to rts");
610                                 goto qpmod_fail;
611                         }
612                         qp->qp_state = TAVOR_QP_RTS;
613
614                 } else if ((flags & IBT_CEP_SET_STATE) &&
615                     (mod_state == IBT_STATE_RESET)) {
616                         /*
617                          * Attempt to transition from "SQErr" to "Reset"
618                          */
619                         status = tavor_qp_to_reset(state, qp);
620                         if (status != DDI_SUCCESS) {
621                                 mutex_exit(&qp->qp_lock);
622                                 /* Set "status"/"errormsg", goto failure */
623                                 TAVOR_TNF_FAIL(status, "sqerr to reset");
624                                 goto qpmod_fail;
625                         }
626                         qp->qp_state = TAVOR_QP_RESET;
627
628                         /*
629                          * Do any additional handling necessary for the
630                          * transition _to_ the "Reset" state (e.g. update the
631                          * workQ WRID lists)
632                          */
633                         tavor_wrid_to_reset_handling(state, qp);
634
635                 } else if ((flags & IBT_CEP_SET_STATE) &&
636                     (mod_state == IBT_STATE_ERROR)) {
637                         /*
638                          * Attempt to transition from "SQErr" to "Error"
639                          */
640                         status = tavor_qp_to_error(state, qp);
641                         if (status != DDI_SUCCESS) {
642                                 mutex_exit(&qp->qp_lock);
643                                 /* Set "status"/"errormsg", goto failure */
644                                 TAVOR_TNF_FAIL(status, "sqerr to error");
645                                 goto qpmod_fail;
646                         }
647                         qp->qp_state = TAVOR_QP_ERR;
648
649                 } else {
650                         /* Invalid transition - return error */
651                         mutex_exit(&qp->qp_lock);
652                         /* Set "status" and "errormsg" and goto failure */
653                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
654                             "sqerr: invalid transition");
655                         goto qpmod_fail;
```

```
656                 }
657                 break;

659         case TAVOR_QP_SQD:
660                 okflags = (IBT_CEP_SET_STATE | IBT_CEP_SET_ADDS_VECT |
661                     IBT_CEP_SET_ALT_PATH | IBT_CEP_SET_MIG |
662                     IBT_CEP_SET_RDMARA_OUT | IBT_CEP_SET_RDMARA_IN |
663                     IBT_CEP_SET_QKEY | IBT_CEP_SET_PKEY_IX |
664                     IBT_CEP_SET_TIMEOUT | IBT_CEP_SET_RETRY |
665                     IBT_CEP_SET_RNR_NAK_RETRY | IBT_CEP_SET_PORT |
666                     IBT_CEP_SET_MIN_RNR_NAK | IBT_CEP_SET_RDMA_R |
667                     IBT_CEP_SET_RDMA_W | IBT_CEP_SET_ATOMIC);

669                 /*
670                  * Check for attempts to modify invalid attributes from the
671                  * "SQD" state
672                  */
673                 if (flags & ~okflags) {
674                         mutex_exit(&qp->qp_lock);
675                         /* Set "status" and "errormsg" and goto failure */
676                         TAVOR_TNF_FAIL(IBT_QP_ATTR_RO, "sqd: invalid flag");
677                         goto qpmod_fail;
678                 }

680                 /*
681                  * Verify state transition is to either "SQD", "RTS", "Reset",
682                  * or "Error"
683                  */

685                 if ((flags & IBT_CEP_SET_STATE) &&
686                     (mod_state == IBT_STATE_SQD)) {
687                         /*
688                          * Attempt to transition from "SQD" to "SQD"
689                          */
690                         status = tavor_qp_sqd2sqd(state, qp, flags, info_p);
691                         if (status != DDI_SUCCESS) {
692                                 mutex_exit(&qp->qp_lock);
693                                 /* Set "status"/"errormsg", goto failure */
694                                 TAVOR_TNF_FAIL(status, "sqd to sqd");
695                                 goto qpmod_fail;
696                         }
697                         qp->qp_state = TAVOR_QP_SQD;

699                 } else if ((flags & IBT_CEP_SET_STATE) &&
700                     (mod_state == IBT_STATE_RTS)) {
701                         /*
702                          * If still draining SQ, then fail transition attempt
703                          * to RTS.
704                          */
705                         if (qp->qp_sqd_still_draining) {
706                                 mutex_exit(&qp->qp_lock);
707                                 /* Set "status"/"errormsg", goto failure */
708                                 status = IBT_QP_STATE_INVALID;
709                                 TAVOR_TNF_FAIL(status, "sqd to rts; draining");
710                                 goto qpmod_fail;
711                         }

713                         /*
714                          * Attempt to transition from "SQD" to "RTS"
715                          */
716                         status = tavor_qp_sqd2rts(state, qp, flags, info_p);
717                         if (status != DDI_SUCCESS) {
718                                 mutex_exit(&qp->qp_lock);
719                                 /* Set "status"/"errormsg", goto failure */
720                                 TAVOR_TNF_FAIL(status, "sqd to rts");
721                                 goto qpmod_fail;
```



```
722                         }
723                         qp->qp_state = TAVOR_QP_RTS;

725                 } else if ((flags & IBT_CEP_SET_STATE) &&
726                     (mod_state == IBT_STATE_RESET)) {
727                         /*
728                          * Attempt to transition from "SQD" to "Reset"
729                          */
730                         status = tavor_qp_to_reset(state, qp);
731                         if (status != DDI_SUCCESS) {
732                                 mutex_exit(&qp->qp_lock);
733                                 /* Set "status"/"errormsg", goto failure */
734                                 TAVOR_TNF_FAIL(status, "sqd to reset");
735                                 goto qpmod_fail;
736                         }
737                         qp->qp_state = TAVOR_QP_RESET;

739                         /*
740                          * Do any additional handling necessary for the
741                          * transition _to_ the "Reset" state (e.g. update the
742                          * workQ WRID lists)
743                          */
744                         tavor_wrid_to_reset_handling(state, qp);

746                 } else if ((flags & IBT_CEP_SET_STATE) &&
747                     (mod_state == IBT_STATE_ERROR)) {
748                         /*
749                          * Attempt to transition from "SQD" to "Error"
750                          */
751                         status = tavor_qp_to_error(state, qp);
752                         if (status != DDI_SUCCESS) {
753                                 mutex_exit(&qp->qp_lock);
754                                 /* Set "status"/"errormsg", goto failure */
755                                 TAVOR_TNF_FAIL(status, "sqd to error");
756                                 goto qpmod_fail;
757                         }
758                         qp->qp_state = TAVOR_QP_ERR;

760                 } else {
761                         /* Invalid transition - return error */
762                         mutex_exit(&qp->qp_lock);
763                         /* Set "status" and "errormsg" and goto failure */
764                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
765                             "sqd: invalid transition");
766                         goto qpmod_fail;
767                 }
768                 break;

770         case TAVOR_QP_ERR:
771                 /*
772                  * Verify state transition is to either "Reset" or back to
773                  * "Error"
774                  */
775                 if ((flags & IBT_CEP_SET_STATE) &&
776                     (mod_state == IBT_STATE_RESET)) {
777                         /*
778                          * Attempt to transition from "Error" to "Reset"
779                          */
780                         status = tavor_qp_to_reset(state, qp);
781                         if (status != DDI_SUCCESS) {
782                                 mutex_exit(&qp->qp_lock);
783                                 /* Set "status"/"errormsg", goto failure */
784                                 TAVOR_TNF_FAIL(status, "error to reset");
785                                 goto qpmod_fail;
786                         }
787                         qp->qp_state = TAVOR_QP_RESET;
```

```
789                         /*
790                          * Do any additional handling necessary for the
791                          * transition _to_ the "Reset" state (e.g. update the
792                          * workQ WRID lists)
793                          */
794                         tavor_wrid_to_reset_handling(state, qp);
795
796                 } else if ((flags & IBT_CEP_SET_STATE) &&
797                     (mod_state == IBT_STATE_ERROR)) {
798                         /*
799                          * Attempt to transition from "Error" back to "Error"
800                          *    Nothing to do here really... just drop the lock
801                          *    and return success.  The qp->qp_state should
802                          *    already be set to TAVOR_QP_ERR.
803                          *
804                          */
805                         mutex_exit(&qp->qp_lock);
806                         TNF_PROBE_0_DEBUG(tavor_qp_modify_err2err,
807                             TAVOR_TNF_TRACE, "");
808                         TAVOR_TNF_EXIT(tavor_qp_modify);
809                         return (DDI_SUCCESS);
810
811                 } else {
812                         /* Invalid transition - return error */
813                         mutex_exit(&qp->qp_lock);
814                         /* Set "status" and "errormsg" and goto failure */
815                         TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID,
816                             "error: invalid transition");
817                         goto qpmod_fail;
818                 }
819                 break;
820
821         default:
822                 /*
823                  * Invalid QP state.  If we got here then it's a warning of
824                  * a probably serious problem.  So print a message and return
825                  * failure
826                  */
827                 mutex_exit(&qp->qp_lock);
828                 TAVOR_WARNING(state, "unknown QP state in modify");
829                 /* Set "status" and "errormsg" and goto failure */
830                 TAVOR_TNF_FAIL(IBT_QP_STATE_INVALID, "invalid curr QP state");
831                 goto qpmod_fail;
832         }
833
834         mutex_exit(&qp->qp_lock);
835         TAVOR_TNF_EXIT(tavor_qp_modify);
836         return (DDI_SUCCESS);
837
838 qpmod_fail:
839         TNF_PROBE_1(tavor_qp_modify_fail, TAVOR_TNF_ERROR, "",
840             tnf_string, msg, errormsg);
841         TAVOR_TNF_EXIT(tavor_qp_modify);
842         return (status);
843 }
844
845
846 /*
847  * tavor_qp_reset2init()
848  *    Context: Can be called from interrupt or base context.
849  */
850 static int
851 tavor_qp_reset2init(tavor_state_t *state, tavor_qphdl_t qp,
852     ibt_qp_info_t *info_p)
853 {
```



```
854         tavor_hw_qpc_t          *qpc;
855         ibt_qp_rc_attr_t        *rc;
856         ibt_qp_ud_attr_t        *ud;
857         ibt_qp_uc_attr_t        *uc;
858         uint_t                  portnum, pkeyindx;
859         int                     status;
860
861         TAVOR_TNF_ENTER(tavor_qp_reset2init);
862
863         ASSERT(MUTEX_HELD(&qp->qp_lock));
864
865         /*
866          * Grab the temporary QPC entry from QP software state
867          */
868         qpc = &qp->qpc;
869
870         /*
871          * Fill in the common and/or Tavor-specific fields in the QPC
872          */
873         if (qp->qp_is_special) {
874                 qpc->serv_type  = TAVOR_QP_MLX;
875         } else {
876                 qpc->serv_type  = qp->qp_serv_type;
877         }
878         qpc->pm_state           = TAVOR_QP_PMSTATE_MIGRATED;
879         qpc->de                 = TAVOR_QP_DESC_EVT_ENABLED;
880         qpc->sched_q            = TAVOR_QP_SCHEDQ_GET(qp->qp_qpnum);
881         if (qp->qp_is_umap) {
882                 qpc->usr_page = qp->qp_uarpg;
883         } else {
884                 qpc->usr_page = 0;
885         }
886         qpc->pd                 = qp->qp_pdhdl->pd_pdnum;
887         qpc->wqe_baseaddr       = 0;
888         qpc->wqe_lkey           = qp->qp_mrhdl->mr_lkey;
889         qpc->ssc                = qp->qp_sq_sigtype;
890         qpc->cqn_snd            = qp->qp_sq_cqhdl->cq_cqnum;
891         qpc->rsc                = TAVOR_QP_RQ_ALL_SIGNALED;
892         qpc->cqn_rcv            = qp->qp_rq_cqhdl->cq_cqnum;
893         qpc->srq_en             = qp->qp_srq_en;
894
895         if (qp->qp_srq_en == TAVOR_QP_SRQ_ENABLED) {
896                 qpc->srq_number = qp->qp_srqhdl->srq_srqnum;
897         } else {
898                 qpc->srq_number = 0;
899         }
900
901         /*
902          * Now fill in the QPC fields which are specific to transport type
903          */
904         if (qp->qp_serv_type == TAVOR_QP_UD) {
905                 ud = &info_p->qp_transport.ud;
906
907                 /* Set the QKey */
908                 qpc->qkey = ud->ud_qkey;
909
910                 /* Check for valid port number and fill it in */
911                 portnum = ud->ud_port;
912                 if (tavor_portnum_is_valid(state, portnum)) {
913                         qpc->pri_addr_path.portnum = portnum;
914                 } else {
915                         TNF_PROBE_1(tavor_qp_reset2init_inv_port_fail,
916                             TAVOR_TNF_ERROR, "", tnf_uint, port, portnum);
917                         TAVOR_TNF_EXIT(tavor_qp_reset2init);
918                         return (IBT_HCA_PORT_INVALID);
919                 }
```

```
921                 /* Check for valid PKey index and fill it in */
922                 pkeyindx = ud->ud_pkey_ix;
923                 if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
924                         qpc->pri_addr_path.pkey_indx = pkeyindx;
925                         qp->qp_pkeyindx = pkeyindx;
926                 } else {
927                         TNF_PROBE_1(tavor_qp_reset2init_inv_pkey_fail,
928                             TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx, pkeyindx);
929                         TAVOR_TNF_EXIT(tavor_qp_reset2init);
930                         return (IBT_PKEY_IX_ILLEGAL);
931                 }
932
933         } else if (qp->qp_serv_type == TAVOR_QP_RC) {
934                 rc = &info_p->qp_transport.rc;
935
936                 /* Set the RDMA (recv) enable/disable flags */
937                 qpc->rre = (info_p->qp_flags & IBT_CEP_RDMA_RD) ? 1 : 0;
938                 qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
939                 qpc->rae = (info_p->qp_flags & IBT_CEP_ATOMIC)  ? 1 : 0;
940
941                 /* Check for valid port number and fill it in */
942                 portnum = rc->rc_path.cep_hca_port_num;
943                 if (tavor_portnum_is_valid(state, portnum)) {
944                         qpc->pri_addr_path.portnum = portnum;
945                 } else {
946                         TNF_PROBE_1(tavor_qp_reset2init_inv_port_fail,
947                             TAVOR_TNF_ERROR, "", tnf_uint, port, portnum);
948                         TAVOR_TNF_EXIT(tavor_qp_reset2init);
949                         return (IBT_HCA_PORT_INVALID);
950                 }
951
952                 /* Check for valid PKey index and fill it in */
953                 pkeyindx = rc->rc_path.cep_pkey_ix;
954                 if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
955                         qpc->pri_addr_path.pkey_indx = pkeyindx;
956                 } else {
957                         TNF_PROBE_1(tavor_qp_reset2init_inv_pkey_fail,
958                             TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx, pkeyindx);
959                         TAVOR_TNF_EXIT(tavor_qp_reset2init);
960                         return (IBT_PKEY_IX_ILLEGAL);
961                 }
962
963         } else if (qp->qp_serv_type == TAVOR_QP_UC) {
964                 uc = &info_p->qp_transport.uc;
965
966                 /*
967                  * Set the RDMA (recv) enable/disable flags.  Note: RDMA Read
968                  * and Atomic are ignored by default.
969                  */
970                 qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
971
972                 /* Check for valid port number and fill it in */
973                 portnum = uc->uc_path.cep_hca_port_num;
974                 if (tavor_portnum_is_valid(state, portnum)) {
975                         qpc->pri_addr_path.portnum = portnum;
976                 } else {
977                         TNF_PROBE_1(tavor_qp_reset2init_inv_port_fail,
978                             TAVOR_TNF_ERROR, "", tnf_uint, port, portnum);
979                         TAVOR_TNF_EXIT(tavor_qp_reset2init);
980                         return (IBT_HCA_PORT_INVALID);
981                 }
982
983                 /* Check for valid PKey index and fill it in */
984                 pkeyindx = uc->uc_path.cep_pkey_ix;
985                 if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
```



```
986                         qpc->pri_addr_path.pkey_indx = pkeyindx;
987                 } else {
988                         TNF_PROBE_1(tavor_qp_reset2init_inv_pkey_fail,
989                             TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx, pkeyindx);
990                         TAVOR_TNF_EXIT(tavor_qp_reset2init);
991                         return (IBT_PKEY_IX_ILLEGAL);
992                 }
993         } else {
994                 /*
995                  * Invalid QP transport type. If we got here then it's a
996                  * warning of a probably serious problem.  So print a message
997                  * and return failure
998                  */
999                 TAVOR_WARNING(state, "unknown QP transport type in rst2init");
1000                TNF_PROBE_0(tavor_qp_reset2init_inv_transtype_fail,
1001                    TAVOR_TNF_ERROR, "");
1002                TAVOR_TNF_EXIT(tavor_qp_reset2init);
1003                return (ibc_get_ci_failure(0));
1004        }
1005
1006        /*
1007         * Post the RST2INIT_QP command to the Tavor firmware
1008         *
1009         * We do a TAVOR_NOSLEEP here because we are still holding the
1010         * "qp_lock".  If we got raised to interrupt level by priority
1011         * inversion, we do not want to block in this routine waiting for
1012         * success.
1013         */
1014        status = tavor_cmn_qp_cmd_post(state, RST2INIT_QP, qpc, qp->qp_qpnum,
1015            0, TAVOR_CMD_NOSLEEP_SPIN);
1016        if (status != TAVOR_CMD_SUCCESS) {
1017                cmn_err(CE_CONT, "Tavor: RST2INIT_QP command failed: %08x\n",
1018                    status);
1019                TNF_PROBE_1(tavor_qp_reset2init_cmd_fail, TAVOR_TNF_ERROR, "",
1020                    tnf_uint, status, status);
1021                TAVOR_TNF_EXIT(tavor_qp_reset2init);
1022                return (ibc_get_ci_failure(0));
1023        }
1024
1025        TAVOR_TNF_EXIT(tavor_qp_reset2init);
1026        return (DDI_SUCCESS);
1027 }
1028
1029
1030 /*
1031  * tavor_qp_init2init()
1032  *    Context: Can be called from interrupt or base context.
1033  */
1034 static int
1035 tavor_qp_init2init(tavor_state_t *state, tavor_qphdl_t qp,
1036     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
1037 {
1038        tavor_hw_qpc_t          *qpc;
1039        ibt_qp_rc_attr_t        *rc;
1040        ibt_qp_ud_attr_t        *ud;
1041        ibt_qp_uc_attr_t        *uc;
1042        uint_t                  portnum, pkeyindx;
1043        uint32_t                opmask = 0;
1044        int                     status;
1045
1046        TAVOR_TNF_ENTER(tavor_qp_init2init);
1047
1048        ASSERT(MUTEX_HELD(&qp->qp_lock));
1049
1050        /*
1051         * Grab the temporary QPC entry from QP software state
```

```
1052          */
1053         qpc = &qp->qpc;
1054
1055         /*
1056          * Since there are no common and/or Tavor-specific fields to be filled
1057          * in for this command, we begin with the QPC fields which are
1058          * specific to transport type.
1059          */
1060         if (qp->qp_serv_type == TAVOR_QP_UD) {
1061                 ud = &info_p->qp_transport.ud;
1062
1063                 /*
1064                  * If we are attempting to modify the port for this QP, then
1065                  * check for valid port number and fill it in.  Also set the
1066                  * appropriate flag in the "opmask" parameter.
1067                  */
1068                 if (flags & IBT_CEP_SET_PORT) {
1069                         portnum = ud->ud_port;
1070                         if (tavor_portnum_is_valid(state, portnum)) {
1071                                 qpc->pri_addr_path.portnum = portnum;
1072                         } else {
1073                                 TNF_PROBE_1(tavor_qp_init2init_inv_port_fail,
1074                                     TAVOR_TNF_ERROR, "", tnf_uint, port,
1075                                     portnum);
1076                                 TAVOR_TNF_EXIT(tavor_qp_init2init);
1077                                 return (IBT_HCA_PORT_INVALID);
1078                         }
1079                         opmask |= TAVOR_CMD_OP_PRIM_PORT;
1080                 }
1081
1082                 /*
1083                  * If we are attempting to modify the PKey index for this QP,
1084                  * then check for valid PKey index and fill it in.  Also set
1085                  * the appropriate flag in the "opmask" parameter.
1086                  */
1087                 if (flags & IBT_CEP_SET_PKEY_IX) {
1088                         pkeyindx = ud->ud_pkey_ix;
1089                         if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1090                                 qpc->pri_addr_path.pkey_indx = pkeyindx;
1091                                 opmask |= TAVOR_CMD_OP_PKEYINDX;
1092                                 qp->qp_pkeyindx = pkeyindx;
1093                         } else {
1094                                 TNF_PROBE_1(tavor_qp_init2init_inv_pkey_fail,
1095                                     TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
1096                                     pkeyindx);
1097                                 TAVOR_TNF_EXIT(tavor_qp_init2init);
1098                                 return (IBT_PKEY_IX_ILLEGAL);
1099                         }
1100                 }
1101
1102                 /*
1103                  * If we are attempting to modify the QKey for this QP, then
1104                  * fill it in and set the appropriate flag in the "opmask"
1105                  * parameter.
1106                  */
1107                 if (flags & IBT_CEP_SET_QKEY) {
1108                         qpc->qkey = ud->ud_qkey;
1109                         opmask |= TAVOR_CMD_OP_QKEY;
1110                 }
1111
1112         } else if (qp->qp_serv_type == TAVOR_QP_RC) {
1113                 rc = &info_p->qp_transport.rc;
1114
1115                 /*
1116                  * If we are attempting to modify the port for this QP, then
1117                  * check for valid port number and fill it in.  Also set the
```



```
1118                  * appropriate flag in the "opmask" parameter.
1119                  */
1120                 if (flags & IBT_CEP_SET_PORT) {
1121                         portnum = rc->rc_path.cep_hca_port_num;
1122                         if (tavor_portnum_is_valid(state, portnum)) {
1123                                 qpc->pri_addr_path.portnum = portnum;
1124                         } else {
1125                                 TNF_PROBE_1(tavor_qp_init2init_inv_port_fail,
1126                                     TAVOR_TNF_ERROR, "", tnf_uint, port,
1127                                     portnum);
1128                                 TAVOR_TNF_EXIT(tavor_qp_init2init);
1129                                 return (IBT_HCA_PORT_INVALID);
1130                         }
1131                         opmask |= TAVOR_CMD_OP_PRIM_PORT;
1132                 }
1133
1134                 /*
1135                  * If we are attempting to modify the PKey index for this QP,
1136                  * then check for valid PKey index and fill it in.  Also set
1137                  * the appropriate flag in the "opmask" parameter.
1138                  */
1139                 if (flags & IBT_CEP_SET_PKEY_IX) {
1140                         pkeyindx = rc->rc_path.cep_pkey_ix;
1141                         if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1142                                 qpc->pri_addr_path.pkey_indx = pkeyindx;
1143                                 opmask |= TAVOR_CMD_OP_PKEYINDX;
1144                         } else {
1145                                 TNF_PROBE_1(tavor_qp_init2init_inv_pkey_fail,
1146                                     TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
1147                                     pkeyindx);
1148                                 TAVOR_TNF_EXIT(tavor_qp_init2init);
1149                                 return (IBT_PKEY_IX_ILLEGAL);
1150                         }
1151                 }
1152
1153                 /*
1154                  * Check if any of the flags indicate a change in the RDMA
1155                  * (recv) enable/disable flags and set the appropriate flag in
1156                  * the "opmask" parameter
1157                  */
1158                 opmask |= tavor_check_rdma_enable_flags(flags, info_p, qpc);
1159
1160         } else if (qp->qp_serv_type == TAVOR_QP_UC) {
1161                 uc = &info_p->qp_transport.uc;
1162
1163                 /*
1164                  * If we are attempting to modify the port for this QP, then
1165                  * check for valid port number and fill it in.  Also set the
1166                  * appropriate flag in the "opmask" parameter.
1167                  */
1168                 if (flags & IBT_CEP_SET_PORT) {
1169                         portnum = uc->uc_path.cep_hca_port_num;
1170                         if (tavor_portnum_is_valid(state, portnum)) {
1171                                 qpc->pri_addr_path.portnum = portnum;
1172                         } else {
1173                                 TNF_PROBE_1(tavor_qp_init2init_inv_port_fail,
1174                                     TAVOR_TNF_ERROR, "", tnf_uint, port,
1175                                     portnum);
1176                                 TAVOR_TNF_EXIT(tavor_qp_init2init);
1177                                 return (IBT_HCA_PORT_INVALID);
1178                         }
1179                         opmask |= TAVOR_CMD_OP_PRIM_PORT;
1180                 }
1181
1182                 /*
1183                  * If we are attempting to modify the PKey index for this QP,
```

```
1184                 * then check for valid PKey index and fill it in.  Also set
1185                 * the appropriate flag in the "opmask" parameter.
1186                 */
1187                if (flags & IBT_CEP_SET_PKEY_IX) {
1188                        pkeyindx = uc->uc_path.cep_pkey_ix;
1189                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1190                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1191                                opmask |= TAVOR_CMD_OP_PKEYINDX;
1192                        } else {
1193                                TNF_PROBE_1(tavor_qp_init2init_inv_pkey_fail,
1194                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
1195                                    pkeyindx);
1196                                TAVOR_TNF_EXIT(tavor_qp_init2init);
1197                                return (IBT_PKEY_IX_ILLEGAL);
1198                        }
1199                }

1201                /*
1202                 * Check if any of the flags indicate a change in the RDMA
1203                 * Write (recv) enable/disable and set the appropriate flag
1204                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
1205                 * not valid for UC transport.
1206                 */
1207                if (flags & IBT_CEP_SET_RDMA_W) {
1208                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1209                        opmask |= TAVOR_CMD_OP_RWE;
1210                }
1211        } else {
1212                /*
1213                 * Invalid QP transport type. If we got here then it's a
1214                 * warning of a probably serious problem.  So print a message
1215                 * and return failure
1216                 */
1217                TAVOR_WARNING(state, "unknown QP transport type in init2init");
1218                TNF_PROBE_0(tavor_qp_init2init_inv_transtype_fail,
1219                    TAVOR_TNF_ERROR, "");
1220                TAVOR_TNF_EXIT(tavor_qp_init2init);
1221                return (ibc_get_ci_failure(0));
1222        }

1224        /*
1225         * Post the INIT2INIT_QP command to the Tavor firmware
1226         *
1227         * We do a TAVOR_NOSLEEP here because we are still holding the
1228         * "qp_lock".  If we got raised to interrupt level by priority
1229         * inversion, we do not want to block in this routine waiting for
1230         * success.
1231         */
1232        status = tavor_cmn_qp_cmd_post(state, INIT2INIT_QP, qpc, qp->qp_qpnum,
1233            opmask, TAVOR_CMD_NOSLEEP_SPIN);
1234        if (status != TAVOR_CMD_SUCCESS) {
1235                if (status != TAVOR_CMD_BAD_QP_STATE) {
1236                        cmn_err(CE_CONT, "Tavor: INIT2INIT_QP command failed: "
1237                            "%08x\n", status);
1238                        TNF_PROBE_1(tavor_qp_init2init_cmd_fail,
1239                            TAVOR_TNF_ERROR, "", tnf_uint, status, status);
1240                        TAVOR_TNF_EXIT(tavor_qp_init2init);
1241                        return (ibc_get_ci_failure(0));
1242                } else {
1243                        TNF_PROBE_0(tavor_qp_init2init_inv_qpstate_fail,
1244                            TAVOR_TNF_ERROR, "");
1245                        TAVOR_TNF_EXIT(tavor_qp_init2init);
1246                        return (IBT_QP_STATE_INVALID);
1247                }
1248        }
```



```
1250        TAVOR_TNF_EXIT(tavor_qp_init2init);
1251        return (DDI_SUCCESS);
1252 }


1255 /*
1256  * tavor_qp_init2rtr()
1257  *    Context: Can be called from interrupt or base context.
1258  */
1259 static int
1260 tavor_qp_init2rtr(tavor_state_t *state, tavor_qphdl_t qp,
1261     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
1262 {
1263        tavor_hw_qpc_t          *qpc;
1264        ibt_qp_rc_attr_t        *rc;
1265        ibt_qp_ud_attr_t        *ud;
1266        ibt_qp_uc_attr_t        *uc;
1267        tavor_hw_addr_path_t    *qpc_path;
1268        ibt_adds_vect_t         *adds_vect;
1269        uint_t                  portnum, pkeyindx, rdma_ra_in, rra_max;
1270        uint_t                  mtu;
1271        uint32_t                opmask = 0;
1272        int                     status;

1274        TAVOR_TNF_ENTER(tavor_qp_init2rtr);

1276        ASSERT(MUTEX_HELD(&qp->qp_lock));

1278        /*
1279         * Grab the temporary QPC entry from QP software state
1280         */
1281        qpc = &qp->qpc;

1283        /*
1284         * Since there are no common and/or Tavor-specific fields to be filled
1285         * in for this command, we begin with the QPC fields which are
1286         * specific to transport type.
1287         */
1288        if (qp->qp_serv_type == TAVOR_QP_UD) {
1289                ud = &info_p->qp_transport.ud;

1291                /*
1292                 * If this UD QP is also a "special QP" (QP0 or QP1), then
1293                 * the MTU is 256 bytes.  However, Tavor HW requires us to
1294                 * set the MTU to 4 (which is the IB code for a 2K MTU).
1295                 * If this is not a special QP, then we set the MTU to the
1296                 * configured maximum (which defaults to 2K).  Note: the
1297                 * QPC "msg_max" must also be set so as to correspond with
1298                 * the specified MTU value.
1299                 */
1300                if (qp->qp_is_special) {
1301                        qpc->mtu = 4;
1302                } else {
1303                        qpc->mtu = state->ts_cfg_profile->cp_max_mtu;
1304                }
1305                qpc->msg_max = qpc->mtu + 7;  /* must equal MTU plus seven */

1307                /*
1308                 * Save away the MTU value.  This is used in future sqd2sqd
1309                 * transitions, as the MTU must remain the same in future
1310                 * changes.
1311                 */
1312                qp->qp_save_mtu = qpc->mtu;

1314                /*
1315                 * If we are attempting to modify the PKey index for this QP,
```

```
1316                 * then check for valid PKey index and fill it in.  Also set
1317                 * the appropriate flag in the "opmask" parameter.
1318                 */
1319                if (flags & IBT_CEP_SET_PKEY_IX) {
1320                        pkeyindx = ud->ud_pkey_ix;
1321                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1322                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1323                                opmask |= TAVOR_CMD_OP_PKEYINDX;
1324                                qp->qp_pkeyindx = pkeyindx;
1325                        } else {
1326                                TNF_PROBE_1(tavor_qp_init2rtr_inv_pkey_fail,
1327                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
1328                                    pkeyindx);
1329                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1330                                return (IBT_PKEY_IX_ILLEGAL);
1331                        }
1332                }

1334                /*
1335                 * If we are attempting to modify the QKey for this QP, then
1336                 * fill it in and set the appropriate flag in the "opmask"
1337                 * parameter.
1338                 */
1339                if (flags & IBT_CEP_SET_QKEY) {
1340                        qpc->qkey = ud->ud_qkey;
1341                        opmask |= TAVOR_CMD_OP_QKEY;
1342                }

1344        } else if (qp->qp_serv_type == TAVOR_QP_RC) {
1345                rc = &info_p->qp_transport.rc;
1346                qpc_path = &qpc->pri_addr_path;
1347                adds_vect = &rc->rc_path.cep_adds_vect;

1349                /*
1350                 * Set the common primary address path fields
1351                 */
1352                status = tavor_set_addr_path(state, adds_vect, qpc_path,
1353                    TAVOR_ADDRPATH_QP, qp);
1354                if (status != DDI_SUCCESS) {
1355                        TNF_PROBE_0(tavor_qp_init2rtr_setaddrpath_fail,
1356                            TAVOR_TNF_ERROR, "");
1357                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1358                        return (status);
1359                }

1361                /*
1362                 * The following values are apparently "required" here (as
1363                 * they are part of the IBA-defined "Remote Node Address
1364                 * Vector").  However, they are also going to be "required"
1365                 * later - at RTR2RTS_QP time.  Not sure why.  But we set
1366                 * them here anyway.
1367                 */
1368                qpc_path->rnr_retry     = rc->rc_rnr_retry_cnt;
1369                qpc->retry_cnt          = rc->rc_retry_cnt;
1370                qpc_path->ack_timeout   = rc->rc_path.cep_timeout;

1372                /*
1373                 * Setup the destination QP, recv PSN, MTU, max msg size,etc.
1374                 * Note max message size is defined to be the maximum IB
1375                 * allowed message size (which is 2^31 bytes).  Also max
1376                 * MTU is defined by HCA port properties.
1377                 */
1378                qpc->rem_qpn      = rc->rc_dst_qpn;
1379                qpc->next_rcv_psn = rc->rc_rq_psn;
1380                qpc->msg_max      = TAVOR_QP_LOG_MAX_MSGSZ;
```



```
1382                /*
1383                 * If this QP is using an SRQ, 'ric' must be set to 1.
1384                 */
1385                qpc->ric = (qp->qp_srq_en == TAVOR_QP_SRQ_ENABLED) ? 1 : 0;
1386                mtu = rc->rc_path_mtu;
1387                if (tavor_qp_validate_mtu(state, mtu) != DDI_SUCCESS) {
1388                        TNF_PROBE_1(tavor_qp_init2rtr_inv_mtu_fail,
1389                            TAVOR_TNF_ERROR, "", tnf_uint, mtu, mtu);
1390                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1391                        return (IBT_HCA_PORT_MTU_EXCEEDED);
1392                }
1393                qpc->mtu = mtu;

1395                /*
1396                 * Save away the MTU value.  This is used in future sqd2sqd
1397                 * transitions, as the MTU must remain the same in future
1398                 * changes.
1399                 */
1400                qp->qp_save_mtu = qpc->mtu;

1402                /*
1403                 * Though it is a "required" parameter, "min_rnr_nak" is
1404                 * optionally specifiable in Tavor.  So we hardcode the
1405                 * optional flag here.
1406                 */
1407                qpc->min_rnr_nak = rc->rc_min_rnr_nak;
1408                opmask |= TAVOR_CMD_OP_MINRNRNAK;

1410                /*
1411                 * Check that the number of specified "incoming RDMA resources"
1412                 * is valid.  And if it is, then setup the "rra_max" and
1413                 * "ra_buf_index" fields in the QPC to point to the
1414                 * pre-allocated RDB resources (in DDR)
1415                 */
1416                rdma_ra_in = rc->rc_rdma_ra_in;
1417                if (tavor_qp_validate_resp_rsrc(state, rc, &rra_max) !=
1418                    DDI_SUCCESS) {
1419                        TNF_PROBE_1(tavor_qp_init2rtr_inv_rdma_in_fail,
1420                            TAVOR_TNF_ERROR, "", tnf_uint, rdma_ra_in,
1421                            rdma_ra_in);
1422                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1423                        return (IBT_INVALID_PARAM);
1424                }
1425                qpc->rra_max = rra_max;
1426                qpc->ra_buff_indx = qp->qp_rdb_ddraddr >> TAVOR_RDB_SIZE_SHIFT;

1428                /*
1429                 * If we are attempting to modify the PKey index for this QP,
1430                 * then check for valid PKey index and fill it in.  Also set
1431                 * the appropriate flag in the "opmask" parameter.
1432                 */
1433                if (flags & IBT_CEP_SET_PKEY_IX) {
1434                        pkeyindx = rc->rc_path.cep_pkey_ix;
1435                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1436                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1437                                opmask |= TAVOR_CMD_OP_PKEYINDX;
1438                        } else {
1439                                TNF_PROBE_1(tavor_qp_init2rtr_inv_pkey_fail,
1440                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
1441                                    pkeyindx);
1442                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1443                                return (IBT_PKEY_IX_ILLEGAL);
1444                        }
1445                }

1447                /*
```

```
1448                 * Check if any of the flags indicate a change in the RDMA
1449                 * (recv) enable/disable flags and set the appropriate flag in
1450                 * the "opmask" parameter
1451                 */
1452                opmask |= tavor_check_rdma_enable_flags(flags, info_p, qpc);

1454                /*
1455                 * Check for optional alternate path and fill in the
1456                 * appropriate QPC fields if one is specified
1457                 */
1458                if (flags & IBT_CEP_SET_ALT_PATH) {
1459                        qpc_path = &qpc->alt_addr_path;
1460                        adds_vect = &rc->rc_alt_path.cep_adds_vect;

1462                        /* Set the common alternate address path fields */
1463                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
1464                            TAVOR_ADDRPATH_QP, qp);
1465                        if (status != DDI_SUCCESS) {
1466                                TNF_PROBE_0(tavor_qp_init2rtr_setaddrpath_fail,
1467                                    TAVOR_TNF_ERROR, "");
1468                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1469                                return (status);
1470                        }
1471                        qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

1473                        /*
1474                         * Copy the "RNR Retry count" value from the primary
1475                         * path.  Just as we did above, we need to hardcode
1476                         * the optional flag here (see below).
1477                         */
1478                        qpc_path->rnr_retry = rc->rc_rnr_retry_cnt;

1480                        /*
1481                         * Check for valid alternate path port number and fill
1482                         * it in
1483                         */
1484                        portnum = rc->rc_alt_path.cep_hca_port_num;
1485                        if (tavor_portnum_is_valid(state, portnum)) {
1486                                qpc->alt_addr_path.portnum = portnum;
1487                        } else {
1488                                TNF_PROBE_1(tavor_qp_init2rtr_inv_altport_fail,
1489                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
1490                                    portnum);
1491                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1492                                return (IBT_HCA_PORT_INVALID);
1493                        }

1495                        /*
1496                         * Check for valid alternate path PKey index and fill
1497                         * it in
1498                         */
1499                        pkeyindx = rc->rc_alt_path.cep_pkey_ix;
1500                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1501                                qpc->alt_addr_path.pkey_indx = pkeyindx;
1502                        } else {
1503                                TNF_PROBE_1(tavor_qp_init2rtr_inv_altpkey_fail,
1504                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
1505                                    pkeyindx);
1506                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1507                                return (IBT_PKEY_IX_ILLEGAL);
1508                        }
1509                        opmask |= (TAVOR_CMD_OP_ALT_PATH |
1510                            TAVOR_CMD_OP_ALT_RNRRETRY);
1511                }

1513        } else if (qp->qp_serv_type == TAVOR_QP_UC) {
```



```
1514                uc = &info_p->qp_transport.uc;
1515                qpc_path = &qpc->pri_addr_path;
1516                adds_vect = &uc->uc_path.cep_adds_vect;

1518                /*
1519                 * Set the common primary address path fields
1520                 */
1521                status = tavor_set_addr_path(state, adds_vect, qpc_path,
1522                    TAVOR_ADDRPATH_QP, qp);
1523                if (status != DDI_SUCCESS) {
1524                        TNF_PROBE_0(tavor_qp_init2rtr_setaddrpath_fail,
1525                            TAVOR_TNF_ERROR, "");
1526                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1527                        return (status);
1528                }

1530                /*
1531                 * Setup the destination QP, recv PSN, MTU, max msg size,etc.
1532                 * Note max message size is defined to be the maximum IB
1533                 * allowed message size (which is 2^31 bytes).  Also max
1534                 * MTU is defined by HCA port properties.
1535                 */
1536                qpc->rem_qpn      = uc->uc_dst_qpn;
1537                qpc->next_rcv_psn = uc->uc_rq_psn;
1538                qpc->msg_max      = TAVOR_QP_LOG_MAX_MSGSZ;
1539                mtu = uc->uc_path_mtu;
1540                if (tavor_qp_validate_mtu(state, mtu) != DDI_SUCCESS) {
1541                        TNF_PROBE_1(tavor_qp_init2rtr_inv_mtu_fail,
1542                            TAVOR_TNF_ERROR, "", tnf_uint, mtu, mtu);
1543                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1544                        return (IBT_HCA_PORT_MTU_EXCEEDED);
1545                }
1546                qpc->mtu = mtu;

1548                /*
1549                 * Save away the MTU value.  This is used in future sqd2sqd
1550                 * transitions, as the MTU must remain the same in future
1551                 * changes.
1552                 */
1553                qp->qp_save_mtu = qpc->mtu;

1555                /*
1556                 * If we are attempting to modify the PKey index for this QP,
1557                 * then check for valid PKey index and fill it in.  Also set
1558                 * the appropriate flag in the "opmask" parameter.
1559                 */
1560                if (flags & IBT_CEP_SET_PKEY_IX) {
1561                        pkeyindx = uc->uc_path.cep_pkey_ix;
1562                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1563                                qpc->pri_addr_path.pkey_indx = pkeyindx;
1564                                opmask |= TAVOR_CMD_OP_PKEYINDX;
1565                        } else {
1566                                TNF_PROBE_1(tavor_qp_init2rtr_inv_pkey_fail,
1567                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
1568                                    pkeyindx);
1569                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1570                                return (IBT_PKEY_IX_ILLEGAL);
1571                        }
1572                }

1574                /*
1575                 * Check if any of the flags indicate a change in the RDMA
1576                 * Write (recv) enable/disable and set the appropriate flag
1577                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
1578                 * not valid for UC transport.
1579                 */
```

```
1580                if (flags & IBT_CEP_SET_RDMA_W) {
1581                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1582                        opmask |= TAVOR_CMD_OP_RWE;
1583                }
1585                /*
1586                 * Check for optional alternate path and fill in the
1587                 * appropriate QPC fields if one is specified
1588                 */
1589                if (flags & IBT_CEP_SET_ALT_PATH) {
1590                        qpc_path = &qpc->alt_addr_path;
1591                        adds_vect = &uc->uc_alt_path.cep_adds_vect;
1593                        /* Set the common alternate address path fields */
1594                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
1595                            TAVOR_ADDRPATH_QP, qp);
1596                        if (status != DDI_SUCCESS) {
1597                                TNF_PROBE_0(tavor_qp_init2rtr_setaddrpath_fail,
1598                                    TAVOR_TNF_ERROR, "");
1599                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1600                                return (status);
1601                        }
1603                        /*
1604                         * Check for valid alternate path port number and fill
1605                         * it in
1606                         */
1607                        portnum = uc->uc_alt_path.cep_hca_port_num;
1608                        if (tavor_portnum_is_valid(state, portnum)) {
1609                                qpc->alt_addr_path.portnum = portnum;
1610                        } else {
1611                                TNF_PROBE_1(tavor_qp_init2rtr_inv_altport_fail,
1612                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
1613                                    portnum);
1614                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1615                                return (IBT_HCA_PORT_INVALID);
1616                        }
1618                        /*
1619                         * Check for valid alternate path PKey index and fill
1620                         * it in
1621                         */
1622                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
1623                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1624                                qpc->alt_addr_path.pkey_indx = pkeyindx;
1625                        } else {
1626                                TNF_PROBE_1(tavor_qp_init2rtr_inv_altpkey_fail,
1627                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
1628                                    pkeyindx);
1629                                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1630                                return (IBT_PKEY_IX_ILLEGAL);
1631                        }
1632                        opmask |= TAVOR_CMD_OP_ALT_PATH;
1633                }
1634        } else {
1635                /*
1636                 * Invalid QP transport type. If we got here then it's a
1637                 * warning of a probably serious problem.  So print a message
1638                 * and return failure
1639                 */
1640                TAVOR_WARNING(state, "unknown QP transport type in init2rtr");
1641                TNF_PROBE_0(tavor_qp_init2rtr_inv_transtype_fail,
1642                    TAVOR_TNF_ERROR, "");
1643                TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1644                return (ibc_get_ci_failure(0));
1645        }
```



```
1647        /*
1648         * Post the INIT2RTR_QP command to the Tavor firmware
1649         *
1650         * We do a TAVOR_NOSLEEP here because we are still holding the
1651         * "qp_lock".  If we got raised to interrupt level by priority
1652         * inversion, we do not want to block in this routine waiting for
1653         * success.
1654         */
1655        status = tavor_cmn_qp_cmd_post(state, INIT2RTR_QP, qpc, qp->qp_qpnum,
1656            opmask, TAVOR_CMD_NOSLEEP_SPIN);
1657        if (status != TAVOR_CMD_SUCCESS) {
1658                if (status != TAVOR_CMD_BAD_QP_STATE) {
1659                        cmn_err(CE_CONT, "Tavor: INIT2RTR_QP command failed: "
1660                            "%08x\n", status);
1661                        TNF_PROBE_1(tavor_qp_init2rtr_cmd_fail,
1662                            TAVOR_TNF_ERROR, "", tnf_uint, status, status);
1663                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1664                        return (ibc_get_ci_failure(0));
1665                } else {
1666                        TNF_PROBE_0(tavor_qp_init2rtr_inv_qpstate_fail,
1667                            TAVOR_TNF_ERROR, "");
1668                        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1669                        return (IBT_QP_STATE_INVALID);
1670                }
1671        }
1673        TAVOR_TNF_EXIT(tavor_qp_init2rtr);
1674        return (DDI_SUCCESS);
1675 }


1678 /*
1679  * tavor_qp_rtr2rts()
1680  *    Context: Can be called from interrupt or base context.
1681  */
1682 static int
1683 tavor_qp_rtr2rts(tavor_state_t *state, tavor_qphdl_t qp,
1684     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
1685 {
1686        tavor_hw_qpc_t          *qpc;
1687        ibt_qp_rc_attr_t        *rc;
1688        ibt_qp_ud_attr_t        *ud;
1689        ibt_qp_uc_attr_t        *uc;
1690        tavor_hw_addr_path_t    *qpc_path;
1691        ibt_adds_vect_t         *adds_vect;
1692        uint_t                  portnum, pkeyindx, rdma_ra_out, sra_max;
1693        uint32_t                opmask = 0;
1694        int                     status;
1696        TAVOR_TNF_ENTER(tavor_qp_rtr2rts);
1698        ASSERT(MUTEX_HELD(&qp->qp_lock));
1700        /*
1701         * Grab the temporary QPC entry from QP software state
1702         */
1703        qpc = &qp->qpc;
1705        /*
1706         * Fill in the common and/or Tavor-specific fields in the QPC
1707         */
1708        qpc->flight_lim = TAVOR_QP_FLIGHT_LIM_UNLIMITED;
1710        /*
1711         * Now fill in the QPC fields which are specific to transport type
```

```
1712          */
1713         if (qp->qp_serv_type == TAVOR_QP_UD) {
1714                 ud = &info_p->qp_transport.ud;

1716                 /* Set the send PSN */
1717                 qpc->next_snd_psn = ud->ud_sq_psn;

1719                 /*
1720                  * If we are attempting to modify the QKey for this QP, then
1721                  * fill it in and set the appropriate flag in the "opmask"
1722                  * parameter.
1723                  */
1724                 if (flags & IBT_CEP_SET_QKEY) {
1725                         qpc->qkey = ud->ud_qkey;
1726                         opmask |= TAVOR_CMD_OP_QKEY;
1727                 }

1729         } else if (qp->qp_serv_type == TAVOR_QP_RC) {
1730                 rc = &info_p->qp_transport.rc;
1731                 qpc_path = &qpc->pri_addr_path;

1733                 /*
1734                  * Setup the send PSN, ACK timeout, and retry counts
1735                  */
1736                 qpc->next_snd_psn       = rc->rc_sq_psn;
1737                 qpc_path->ack_timeout   = rc->rc_path.cep_timeout;
1738                 qpc_path->rnr_retry     = rc->rc_rnr_retry_cnt;
1739                 qpc->retry_cnt          = rc->rc_retry_cnt;

1741                 /*
1742                  * Set "ack_req_freq" based on the configuration variable
1743                  */
1744                 qpc->ack_req_freq = state->ts_cfg_profile->cp_ackreq_freq;

1746                 /*
1747                  * Check that the number of specified "outgoing RDMA resources"
1748                  * is valid.  And if it is, then setup the "sra_max"
1749                  * appropriately
1750                  */
1751                 rdma_ra_out = rc->rc_rdma_ra_out;
1752                 if (tavor_qp_validate_init_depth(state, rc, &sra_max) !=
1753                     DDI_SUCCESS) {
1754                         TNF_PROBE_1(tavor_qp_rtr2rts_inv_rdma_out_fail,
1755                             TAVOR_TNF_ERROR, "", tnf_uint, rdma_ra_out,
1756                             rdma_ra_out);
1757                         TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1758                         return (IBT_INVALID_PARAM);
1759                 }
1760                 qpc->sra_max = sra_max;

1762                 /*
1763                  * Configure the QP to allow (sending of) all types of RC
1764                  * traffic.  Tavor hardware allows these bits to be set to
1765                  * zero (thereby disabling certain outgoing RDMA types), but
1766                  * we do not desire to do this.
1767                  */
1768                 qpc->sre = qpc->swe = qpc->sae = 1;
1769                 qpc->sic = 0;

1771                 /*
1772                  * Check if any of the flags indicate a change in the RDMA
1773                  * (recv) enable/disable flags and set the appropriate flag in
1774                  * the "opmask" parameter
1775                  */
1776                 opmask |= tavor_check_rdma_enable_flags(flags, info_p, qpc);
```



```
1778                 /*
1779                  * If we are attempting to modify the path migration state for
1780                  * this QP, then check for valid state and fill it in.  Also
1781                  * set the appropriate flag in the "opmask" parameter.
1782                  */
1783                 if (flags & IBT_CEP_SET_MIG) {
1784                         if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
1785                                 qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
1786                         } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
1787                                 qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
1788                         } else {
1789                                 TNF_PROBE_1(tavor_qp_rtr2rts_inv_mig_state_fail,
1790                                     TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
1791                                     rc->rc_mig_state);
1792                                 TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1793                                 return (IBT_QP_APM_STATE_INVALID);
1794                         }
1795                         opmask |= TAVOR_CMD_OP_PM_STATE;
1796                 }

1798                 /*
1799                  * If we are attempting to modify the "Minimum RNR NAK" value
1800                  * for this QP, then fill it in and set the appropriate flag
1801                  * in the "opmask" parameter.
1802                  */
1803                 if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
1804                         qpc->min_rnr_nak = rc->rc_min_rnr_nak;
1805                         opmask |= TAVOR_CMD_OP_MINRNRNAK;
1806                 }

1808                 /*
1809                  * Check for optional alternate path and fill in the
1810                  * appropriate QPC fields if one is specified
1811                  */
1812                 if (flags & IBT_CEP_SET_ALT_PATH) {
1813                         qpc_path = &qpc->alt_addr_path;
1814                         adds_vect = &rc->rc_alt_path.cep_adds_vect;

1816                         /* Set the common alternate address path fields */
1817                         status = tavor_set_addr_path(state, adds_vect, qpc_path,
1818                             TAVOR_ADDRPATH_QP, qp);
1819                         if (status != DDI_SUCCESS) {
1820                                 TNF_PROBE_0(tavor_qp_rtr2rts_setaddrpath_fail,
1821                                     TAVOR_TNF_ERROR, "");
1822                                 TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1823                                 return (status);
1824                         }

1826                         qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

1828                         /*
1829                          * Copy the "RNR Retry count" value from the primary
1830                          * path.  Just as we did above, we need to hardcode
1831                          * the optional flag here (see below).
1832                          */
1833                         qpc_path->rnr_retry = rc->rc_rnr_retry_cnt;

1835                         /*
1836                          * Check for valid alternate path port number and fill
1837                          * it in
1838                          */
1839                         portnum = rc->rc_alt_path.cep_hca_port_num;
1840                         if (tavor_portnum_is_valid(state, portnum)) {
1841                                 qpc->alt_addr_path.portnum = portnum;
1842                         } else {
1843                                 TNF_PROBE_1(tavor_qp_rtr2rts_inv_altport_fail,
```

```
1844                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
1845                                    portnum);
1846                            TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1847                            return (IBT_HCA_PORT_INVALID);
1848                    }
1849
1850                    /*
1851                     * Check for valid alternate path PKey index and fill
1852                     * it in
1853                     */
1854                    pkeyindx = rc->rc_alt_path.cep_pkey_ix;
1855                    if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1856                            qpc->alt_addr_path.pkey_indx = pkeyindx;
1857                    } else {
1858                            TNF_PROBE_1(tavor_qp_rtr2rts_inv_altpkey_fail,
1859                                TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
1860                                pkeyindx);
1861                            TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1862                            return (IBT_PKEY_IX_ILLEGAL);
1863                    }
1864                    opmask |= (TAVOR_CMD_OP_ALT_PATH |
1865                        TAVOR_CMD_OP_ALT_RNRRETRY);
1866            }
1867
1868        } else if (qp->qp_serv_type == TAVOR_QP_UC) {
1869            uc = &info_p->qp_transport.uc;
1870
1871            /* Set the send PSN */
1872            qpc->next_snd_psn = uc->uc_sq_psn;
1873
1874            /*
1875             * Configure the QP to allow (sending of) all types of allowable
1876             * UC traffic (i.e. RDMA Write).
1877             */
1878            qpc->swe = 1;
1879
1880            /*
1881             * Check if any of the flags indicate a change in the RDMA
1882             * Write (recv) enable/disable and set the appropriate flag
1883             * in the "opmask" parameter. Note: RDMA Read and Atomic are
1884             * not valid for UC transport.
1885             */
1886            if (flags & IBT_CEP_SET_RDMA_W) {
1887                    qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
1888                    opmask |= TAVOR_CMD_OP_RWE;
1889            }
1890
1891            /*
1892             * If we are attempting to modify the path migration state for
1893             * this QP, then check for valid state and fill it in.  Also
1894             * set the appropriate flag in the "opmask" parameter.
1895             */
1896            if (flags & IBT_CEP_SET_MIG) {
1897                    if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
1898                            qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
1899                    } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
1900                            qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
1901                    } else {
1902                            TNF_PROBE_1(tavor_qp_rtr2rts_inv_mig_state_fail,
1903                                TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
1904                                uc->uc_mig_state);
1905                            TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1906                            return (IBT_QP_APM_STATE_INVALID);
1907                    }
1908                    opmask |= TAVOR_CMD_OP_PM_STATE;
1909            }
```



```
1911            /*
1912             * Check for optional alternate path and fill in the
1913             * appropriate QPC fields if one is specified
1914             */
1915            if (flags & IBT_CEP_SET_ALT_PATH) {
1916                    qpc_path = &qpc->alt_addr_path;
1917                    adds_vect = &uc->uc_alt_path.cep_adds_vect;
1918
1919                    /* Set the common alternate address path fields */
1920                    status = tavor_set_addr_path(state, adds_vect, qpc_path,
1921                        TAVOR_ADDRPATH_QP, qp);
1922                    if (status != DDI_SUCCESS) {
1923                            TNF_PROBE_0(tavor_qp_rtr2rts_setaddrpath_fail,
1924                                TAVOR_TNF_ERROR, "");
1925                            TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1926                            return (status);
1927                    }
1928
1929                    /*
1930                     * Check for valid alternate path port number and fill
1931                     * it in
1932                     */
1933                    portnum = uc->uc_alt_path.cep_hca_port_num;
1934                    if (tavor_portnum_is_valid(state, portnum)) {
1935                            qpc->alt_addr_path.portnum = portnum;
1936                    } else {
1937                            TNF_PROBE_1(tavor_qp_rtr2rts_inv_altport_fail,
1938                                TAVOR_TNF_ERROR, "", tnf_uint, altport,
1939                                portnum);
1940                            TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1941                            return (IBT_HCA_PORT_INVALID);
1942                    }
1943
1944                    /*
1945                     * Check for valid alternate path PKey index and fill
1946                     * it in
1947                     */
1948                    pkeyindx = uc->uc_alt_path.cep_pkey_ix;
1949                    if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
1950                            qpc->alt_addr_path.pkey_indx = pkeyindx;
1951                    } else {
1952                            TNF_PROBE_1(tavor_qp_rtr2rts_inv_altpkey_fail,
1953                                TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
1954                                pkeyindx);
1955                            TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1956                            return (IBT_PKEY_IX_ILLEGAL);
1957                    }
1958                    opmask |= TAVOR_CMD_OP_ALT_PATH;
1959            }
1960        } else {
1961                /*
1962                 * Invalid QP transport type. If we got here then it's a
1963                 * warning of a probably serious problem.  So print a message
1964                 * and return failure
1965                 */
1966                TAVOR_WARNING(state, "unknown QP transport type in rtr2rts");
1967                TNF_PROBE_0(tavor_qp_rtr2rts_inv_transtype_fail,
1968                    TAVOR_TNF_ERROR, "");
1969                TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1970                return (ibc_get_ci_failure(0));
1971        }
1972
1973        /*
1974         * Post the RTR2RTS_QP command to the Tavor firmware
1975         *
```

```
1976          * We do a TAVOR_NOSLEEP here because we are still holding the
1977          * "qp_lock".  If we got raised to interrupt level by priority
1978          * inversion, we do not want to block in this routine waiting for
1979          * success.
1980          */
1981         status = tavor_cmn_qp_cmd_post(state, RTR2RTS_QP, qpc, qp->qp_qpnum,
1982             opmask, TAVOR_CMD_NOSLEEP_SPIN);
1983         if (status != TAVOR_CMD_SUCCESS) {
1984                 if (status != TAVOR_CMD_BAD_QP_STATE) {
1985                         cmn_err(CE_CONT, "Tavor: RTR2RTS_QP command failed: "
1986                             "%08x\n", status);
1987                         TNF_PROBE_1(tavor_qp_rtr2rts_cmd_fail,
1988                             TAVOR_TNF_ERROR, "", tnf_uint, status, status);
1989                         TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1990                         return (ibc_get_ci_failure(0));
1991                 } else {
1992                         TNF_PROBE_0(tavor_qp_rtr2rts_inv_qpstate_fail,
1993                             TAVOR_TNF_ERROR, "");
1994                         TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
1995                         return (IBT_QP_STATE_INVALID);
1996                 }
1997         }

1999         TAVOR_TNF_EXIT(tavor_qp_rtr2rts);
2000         return (DDI_SUCCESS);
2001 }



2004 /*
2005  * tavor_qp_rts2rts()
2006  *    Context: Can be called from interrupt or base context.
2007  */
2008 static int
2009 tavor_qp_rts2rts(tavor_state_t *state, tavor_qphdl_t qp,
2010     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
2011 {
2012         tavor_hw_qpc_t          *qpc;
2013         ibt_qp_rc_attr_t        *rc;
2014         ibt_qp_ud_attr_t        *ud;
2015         ibt_qp_uc_attr_t        *uc;
2016         tavor_hw_addr_path_t    *qpc_path;
2017         ibt_adds_vect_t         *adds_vect;
2018         uint_t                  portnum, pkeyindx;
2019         uint32_t                opmask = 0;
2020         int                     status;

2022         TAVOR_TNF_ENTER(tavor_qp_rts2rts);

2024         ASSERT(MUTEX_HELD(&qp->qp_lock));

2026         /*
2027          * Grab the temporary QPC entry from QP software state
2028          */
2029         qpc = &qp->qpc;

2031         /*
2032          * Since there are no common and/or Tavor-specific fields to be filled
2033          * in for this command, we begin with the QPC fields which are
2034          * specific to transport type.
2035          */
2036         if (qp->qp_serv_type == TAVOR_QP_UD) {
2037                 ud = &info_p->qp_transport.ud;

2039                 /*
2040                  * If we are attempting to modify the QKey for this QP, then
2041                  * fill it in and set the appropriate flag in the "opmask"
```



```
2042                  * parameter.
2043                  */
2044                 if (flags & IBT_CEP_SET_QKEY) {
2045                         qpc->qkey = ud->ud_qkey;
2046                         opmask |= TAVOR_CMD_OP_QKEY;
2047                 }

2049         } else if (qp->qp_serv_type == TAVOR_QP_RC) {
2050                 rc = &info_p->qp_transport.rc;

2052                 /*
2053                  * Check if any of the flags indicate a change in the RDMA
2054                  * (recv) enable/disable flags and set the appropriate flag in
2055                  * the "opmask" parameter
2056                  */
2057                 opmask |= tavor_check_rdma_enable_flags(flags, info_p, qpc);

2059                 /*
2060                  * If we are attempting to modify the path migration state for
2061                  * this QP, then check for valid state and fill it in.  Also
2062                  * set the appropriate flag in the "opmask" parameter.
2063                  */
2064                 if (flags & IBT_CEP_SET_MIG) {
2065                         if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
2066                                 qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
2067                         } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
2068                                 qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
2069                         } else {
2070                                 TNF_PROBE_1(tavor_qp_rts2rts_inv_mig_state_fail,
2071                                     TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
2072                                     rc->rc_mig_state);
2073                                 TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2074                                 return (IBT_QP_APM_STATE_INVALID);
2075                         }
2076                         opmask |= TAVOR_CMD_OP_PM_STATE;
2077                 }

2079                 /*
2080                  * If we are attempting to modify the "Minimum RNR NAK" value
2081                  * for this QP, then fill it in and set the appropriate flag
2082                  * in the "opmask" parameter.
2083                  */
2084                 if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
2085                         qpc->min_rnr_nak = rc->rc_min_rnr_nak;
2086                         opmask |= TAVOR_CMD_OP_MINRNRNAK;
2087                 }

2089                 /*
2090                  * Check for optional alternate path and fill in the
2091                  * appropriate QPC fields if one is specified
2092                  */
2093                 if (flags & IBT_CEP_SET_ALT_PATH) {
2094                         qpc_path = &qpc->alt_addr_path;
2095                         adds_vect = &rc->rc_alt_path.cep_adds_vect;

2097                         /* Set the common alternate address path fields */
2098                         status = tavor_set_addr_path(state, adds_vect, qpc_path,
2099                             TAVOR_ADDRPATH_QP, qp);
2100                         if (status != DDI_SUCCESS) {
2101                                 TNF_PROBE_0(tavor_qp_rts2rts_setaddrpath_fail,
2102                                     TAVOR_TNF_ERROR, "");
2103                                 TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2104                                 return (status);
2105                         }
2106                         qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;
```

```
2108                        /*
2109                         * Check for valid alternate path port number and fill
2110                         * it in
2111                         */
2112                        portnum = rc->rc_alt_path.cep_hca_port_num;
2113                        if (tavor_portnum_is_valid(state, portnum)) {
2114                                qpc->alt_addr_path.portnum = portnum;
2115                        } else {
2116                                TNF_PROBE_1(tavor_qp_rts2rts_inv_altport_fail,
2117                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
2118                                    portnum);
2119                                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2120                                return (IBT_HCA_PORT_INVALID);
2121                        }
2122
2123                        /*
2124                         * Check for valid alternate path PKey index and fill
2125                         * it in
2126                         */
2127                        pkeyindx = rc->rc_alt_path.cep_pkey_ix;
2128                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2129                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2130                        } else {
2131                                TNF_PROBE_1(tavor_qp_rts2rts_inv_altpkey_fail,
2132                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
2133                                    pkeyindx);
2134                                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2135                                return (IBT_PKEY_IX_ILLEGAL);
2136                        }
2137                        opmask |= TAVOR_CMD_OP_ALT_PATH;
2138                }
2139
2140        } else if (qp->qp_serv_type == TAVOR_QP_UC) {
2141                uc = &info_p->qp_transport.uc;
2142
2143                /*
2144                 * Check if any of the flags indicate a change in the RDMA
2145                 * Write (recv) enable/disable and set the appropriate flag
2146                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
2147                 * not valid for UC transport.
2148                 */
2149                if (flags & IBT_CEP_SET_RDMA_W) {
2150                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
2151                        opmask |= TAVOR_CMD_OP_RWE;
2152                }
2153
2154                /*
2155                 * If we are attempting to modify the path migration state for
2156                 * this QP, then check for valid state and fill it in.  Also
2157                 * set the appropriate flag in the "opmask" parameter.
2158                 */
2159                if (flags & IBT_CEP_SET_MIG) {
2160                        if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
2161                                qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
2162                        } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
2163                                qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
2164                        } else {
2165                                TNF_PROBE_1(tavor_qp_rts2rts_inv_mig_state_fail,
2166                                    TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
2167                                    uc->uc_mig_state);
2168                                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2169                                return (IBT_QP_APM_STATE_INVALID);
2170                        }
2171                        opmask |= TAVOR_CMD_OP_PM_STATE;
2172                }
```



```
2174                /*
2175                 * Check for optional alternate path and fill in the
2176                 * appropriate QPC fields if one is specified
2177                 */
2178                if (flags & IBT_CEP_SET_ALT_PATH) {
2179                        qpc_path = &qpc->alt_addr_path;
2180                        adds_vect = &uc->uc_alt_path.cep_adds_vect;
2181
2182                        /* Set the common alternate address path fields */
2183                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
2184                            TAVOR_ADDRPATH_QP, qp);
2185                        if (status != DDI_SUCCESS) {
2186                                TNF_PROBE_0(tavor_qp_rts2rts_setaddrpath_fail,
2187                                    TAVOR_TNF_ERROR, "");
2188                                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2189                                return (status);
2190                        }
2191
2192                        /*
2193                         * Check for valid alternate path port number and fill
2194                         * it in
2195                         */
2196                        portnum = uc->uc_alt_path.cep_hca_port_num;
2197                        if (tavor_portnum_is_valid(state, portnum)) {
2198                                qpc->alt_addr_path.portnum = portnum;
2199                        } else {
2200                                TNF_PROBE_1(tavor_qp_rts2rts_inv_altport_fail,
2201                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
2202                                    portnum);
2203                                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2204                                return (IBT_HCA_PORT_INVALID);
2205                        }
2206
2207                        /*
2208                         * Check for valid alternate path PKey index and fill
2209                         * it in
2210                         */
2211                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
2212                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2213                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2214                        } else {
2215                                TNF_PROBE_1(tavor_qp_rts2rts_inv_altpkey_fail,
2216                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
2217                                    pkeyindx);
2218                                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2219                                return (IBT_PKEY_IX_ILLEGAL);
2220                        }
2221                        opmask |= TAVOR_CMD_OP_ALT_PATH;
2222                }
2223        } else {
2224                /*
2225                 * Invalid QP transport type. If we got here then it's a
2226                 * warning of a probably serious problem.  So print a message
2227                 * and return failure
2228                 */
2229                TAVOR_WARNING(state, "unknown QP transport type in rts2rts");
2230                TNF_PROBE_0(tavor_qp_rts2rts_inv_transtype_fail,
2231                    TAVOR_TNF_ERROR, "");
2232                TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2233                return (ibc_get_ci_failure(0));
2234        }
2235
2236        /*
2237         * Post the RTS2RTS_QP command to the Tavor firmware
2238         *
2239         * We do a TAVOR_NOSLEEP here because we are still holding the
```

```
2240          * "qp_lock".  If we got raised to interrupt level by priority
2241          * inversion, we do not want to block in this routine waiting for
2242          * success.
2243          */
2244         status = tavor_cmn_qp_cmd_post(state, RTS2RTS_QP, qpc, qp->qp_qpnum,
2245             opmask, TAVOR_CMD_NOSLEEP_SPIN);
2246         if (status != TAVOR_CMD_SUCCESS) {
2247                 if (status != TAVOR_CMD_BAD_QP_STATE) {
2248                         cmn_err(CE_CONT, "Tavor: RTS2RTS_QP command failed: "
2249                             "%08x\n", status);
2250                         TNF_PROBE_1(tavor_qp_rts2rts_cmd_fail,
2251                             TAVOR_TNF_ERROR, "", tnf_uint, status, status);
2252                         TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2253                         return (ibc_get_ci_failure(0));
2254                 } else {
2255                         TNF_PROBE_0(tavor_qp_rts2rts_inv_qpstate_fail,
2256                             TAVOR_TNF_ERROR, "");
2257                         TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2258                         return (IBT_QP_STATE_INVALID);
2259                 }
2260         }

2262         TAVOR_TNF_EXIT(tavor_qp_rts2rts);
2263         return (DDI_SUCCESS);
2264 }


2267 /*
2268  * tavor_qp_rts2sqd()
2269  *    Context: Can be called from interrupt or base context.
2270  */
2271 static int
2272 tavor_qp_rts2sqd(tavor_state_t *state, tavor_qphdl_t qp,
2273     ibt_cep_modify_flags_t flags)
2274 {
2275         int                     status;

2277         TAVOR_TNF_ENTER(tavor_qp_rts2sqd);

2279         ASSERT(MUTEX_HELD(&qp->qp_lock));

2281         /*
2282          * Set a flag to indicate whether or not the consumer is interested
2283          * in receiving the SQ drained event.  Since we are going to always
2284          * request hardware generation of the SQD event, we use the value in
2285          * "qp_forward_sqd_event" to determine whether or not to pass the event
2286          * to the IBTF or to silently consume it.
2287          */
2288         qp->qp_forward_sqd_event = (flags & IBT_CEP_SET_SQD_EVENT) ? 1 : 0;

2290         /*
2291          * Post the RTS2SQD_QP command to the Tavor firmware
2292          *
2293          * We do a TAVOR_NOSLEEP here because we are still holding the
2294          * "qp_lock".  If we got raised to interrupt level by priority
2295          * inversion, we do not want to block in this routine waiting for
2296          * success.
2297          */
2298         status = tavor_cmn_qp_cmd_post(state, RTS2SQD_QP, NULL, qp->qp_qpnum,
2299             0, TAVOR_CMD_NOSLEEP_SPIN);
2300         if (status != TAVOR_CMD_SUCCESS) {
2301                 if (status != TAVOR_CMD_BAD_QP_STATE) {
2302                         cmn_err(CE_CONT, "Tavor: RTS2SQD_QP command failed: "
2303                             "%08x\n", status);
2304                         TNF_PROBE_1(tavor_qp_rts2sqd_cmd_fail,
2305                             TAVOR_TNF_ERROR, "", tnf_uint, status, status);
```



```
2306                         TAVOR_TNF_EXIT(tavor_qp_rts2sqd);
2307                         return (ibc_get_ci_failure(0));
2308                 } else {
2309                         TNF_PROBE_0(tavor_qp_rts2sqd_inv_qpstate_fail,
2310                             TAVOR_TNF_ERROR, "");
2311                         TAVOR_TNF_EXIT(tavor_qp_rts2sqd);
2312                         return (IBT_QP_STATE_INVALID);
2313                 }
2314         }

2316         /*
2317          * Mark the current QP state as "SQ Draining".  This allows us to
2318          * distinguish between the two underlying states in SQD. (see QueryQP()
2319          * code in tavor_qp.c)
2320          */
2321         qp->qp_sqd_still_draining = 1;

2323         TAVOR_TNF_EXIT(tavor_qp_rts2sqd);
2324         return (DDI_SUCCESS);
2325 }


2328 /*
2329  * tavor_qp_sqd2rts()
2330  *    Context: Can be called from interrupt or base context.
2331  */
2332 static int
2333 tavor_qp_sqd2rts(tavor_state_t *state, tavor_qphdl_t qp,
2334     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
2335 {
2336         tavor_hw_qpc_t          *qpc;
2337         ibt_qp_rc_attr_t        *rc;
2338         ibt_qp_ud_attr_t        *ud;
2339         ibt_qp_uc_attr_t        *uc;
2340         tavor_hw_addr_path_t    *qpc_path;
2341         ibt_adds_vect_t         *adds_vect;
2342         uint_t                  portnum, pkeyindx;
2343         uint32_t                opmask = 0;
2344         int                     status;

2346         TAVOR_TNF_ENTER(tavor_qp_sqd2rts);

2348         ASSERT(MUTEX_HELD(&qp->qp_lock));

2350         /*
2351          * Grab the temporary QPC entry from QP software state
2352          */
2353         qpc = &qp->qpc;

2355         /*
2356          * Since there are no common and/or Tavor-specific fields to be filled
2357          * in for this command, we begin with the QPC fields which are
2358          * specific to transport type.
2359          */
2360         if (qp->qp_serv_type == TAVOR_QP_UD) {
2361                 ud = &info_p->qp_transport.ud;

2363                 /*
2364                  * If we are attempting to modify the QKey for this QP, then
2365                  * fill it in and set the appropriate flag in the "opmask"
2366                  * parameter.
2367                  */
2368                 if (flags & IBT_CEP_SET_QKEY) {
2369                         qpc->qkey = ud->ud_qkey;
2370                         opmask |= TAVOR_CMD_OP_QKEY;
2371                 }
```

```
2373        } else if (qp->qp_serv_type == TAVOR_QP_RC) {
2374                rc = &info_p->qp_transport.rc;

2376                /*
2377                 * Check if any of the flags indicate a change in the RDMA
2378                 * (recv) enable/disable flags and set the appropriate flag in
2379                 * the "opmask" parameter
2380                 */
2381                opmask |= tavor_check_rdma_enable_flags(flags, info_p, qpc);

2383                /*
2384                 * If we are attempting to modify the path migration state for
2385                 * this QP, then check for valid state and fill it in.  Also
2386                 * set the appropriate flag in the "opmask" parameter.
2387                 */
2388                if (flags & IBT_CEP_SET_MIG) {
2389                        if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
2390                                qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
2391                        } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
2392                                qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
2393                        } else {
2394                                TNF_PROBE_1(tavor_qp_sqd2rts_inv_mig_state_fail,
2395                                    TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
2396                                    rc->rc_mig_state);
2397                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2398                                return (IBT_QP_APM_STATE_INVALID);
2399                        }
2400                        opmask |= TAVOR_CMD_OP_PM_STATE;
2401                }

2403                /*
2404                 * Check for optional alternate path and fill in the
2405                 * appropriate QPC fields if one is specified
2406                 */
2407                if (flags & IBT_CEP_SET_ALT_PATH) {
2408                        qpc_path = &qpc->alt_addr_path;
2409                        adds_vect = &rc->rc_alt_path.cep_adds_vect;

2411                        /* Set the common alternate address path fields */
2412                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
2413                            TAVOR_ADDRPATH_QP, qp);
2414                        if (status != DDI_SUCCESS) {
2415                                TNF_PROBE_0(tavor_qp_sqd2rts_setaddrpath_fail,
2416                                    TAVOR_TNF_ERROR, "");
2417                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2418                                return (status);
2419                        }
2420                        qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

2422                        /*
2423                         * Check for valid alternate path port number and fill
2424                         * it in
2425                         */
2426                        portnum = rc->rc_alt_path.cep_hca_port_num;
2427                        if (tavor_portnum_is_valid(state, portnum)) {
2428                                qpc->alt_addr_path.portnum = portnum;
2429                        } else {
2430                                TNF_PROBE_1(tavor_qp_sqd2rts_inv_altport_fail,
2431                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
2432                                    portnum);
2433                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2434                                return (IBT_HCA_PORT_INVALID);
2435                        }

2437                        /*
```



```
2438                         * Check for valid alternate path PKey index and fill
2439                         * it in
2440                         */
2441                        pkeyindx = rc->rc_alt_path.cep_pkey_ix;
2442                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2443                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2444                        } else {
2445                                TNF_PROBE_1(tavor_qp_sqd2rts_inv_altpkey_fail,
2446                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
2447                                    pkeyindx);
2448                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2449                                return (IBT_PKEY_IX_ILLEGAL);
2450                        }
2451                        opmask |= TAVOR_CMD_OP_ALT_PATH;
2452                }

2454                /*
2455                 * If we are attempting to modify the "Minimum RNR NAK" value
2456                 * for this QP, then fill it in and set the appropriate flag
2457                 * in the "opmask" parameter.
2458                 */
2459                if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
2460                        qpc->min_rnr_nak = rc->rc_min_rnr_nak;
2461                        opmask |= TAVOR_CMD_OP_MINRNRNAK;
2462                }

2464        } else if (qp->qp_serv_type == TAVOR_QP_UC) {
2465                uc = &info_p->qp_transport.uc;

2467                /*
2468                 * Check if any of the flags indicate a change in the RDMA
2469                 * Write (recv) enable/disable and set the appropriate flag
2470                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
2471                 * not valid for UC transport.
2472                 */
2473                if (flags & IBT_CEP_SET_RDMA_W) {
2474                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
2475                        opmask |= TAVOR_CMD_OP_RWE;
2476                }

2478                /*
2479                 * If we are attempting to modify the path migration state for
2480                 * this QP, then check for valid state and fill it in.  Also
2481                 * set the appropriate flag in the "opmask" parameter.
2482                 */
2483                if (flags & IBT_CEP_SET_MIG) {
2484                        if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
2485                                qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
2486                        } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
2487                                qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
2488                        } else {
2489                                TNF_PROBE_1(tavor_qp_sqd2rts_inv_mig_state_fail,
2490                                    TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
2491                                    uc->uc_mig_state);
2492                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2493                                return (IBT_QP_APM_STATE_INVALID);
2494                        }
2495                        opmask |= TAVOR_CMD_OP_PM_STATE;
2496                }

2498                /*
2499                 * Check for optional alternate path and fill in the
2500                 * appropriate QPC fields if one is specified
2501                 */
2502                if (flags & IBT_CEP_SET_ALT_PATH) {
2503                        qpc_path = &qpc->alt_addr_path;
```

```
2504                        adds_vect = &uc->uc_alt_path.cep_adds_vect;

2506                        /* Set the common alternate address path fields */
2507                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
2508                            TAVOR_ADDRPATH_QP, qp);
2509                        if (status != DDI_SUCCESS) {
2510                                TNF_PROBE_0(tavor_qp_sqd2rts_setaddrpath_fail,
2511                                    TAVOR_TNF_ERROR, "");
2512                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2513                                return (status);
2514                        }

2516                        /*
2517                         * Check for valid alternate path port number and fill
2518                         * it in
2519                         */
2520                        portnum = uc->uc_alt_path.cep_hca_port_num;
2521                        if (tavor_portnum_is_valid(state, portnum)) {
2522                                qpc->alt_addr_path.portnum = portnum;
2523                        } else {
2524                                TNF_PROBE_1(tavor_qp_sqd2rts_inv_altport_fail,
2525                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
2526                                    portnum);
2527                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2528                                return (IBT_HCA_PORT_INVALID);
2529                        }

2531                        /*
2532                         * Check for valid alternate path PKey index and fill
2533                         * it in
2534                         */
2535                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
2536                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2537                                qpc->alt_addr_path.pkey_indx = pkeyindx;
2538                        } else {
2539                                TNF_PROBE_1(tavor_qp_sqd2rts_inv_altpkey_fail,
2540                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
2541                                    pkeyindx);
2542                                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2543                                return (IBT_PKEY_IX_ILLEGAL);
2544                        }
2545                        opmask |= TAVOR_CMD_OP_ALT_PATH;
2546                }
2547        } else {
2548                /*
2549                 * Invalid QP transport type. If we got here then it's a
2550                 * warning of a probably serious problem.  So print a message
2551                 * and return failure
2552                 */
2553                TAVOR_WARNING(state, "unknown QP transport type in sqd2rts");
2554                TNF_PROBE_0(tavor_qp_sqd2rts_inv_transtype_fail,
2555                    TAVOR_TNF_ERROR, "");
2556                TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2557                return (ibc_get_ci_failure(0));
2558        }

2560        /*
2561         * Post the SQD2RTS_QP command to the Tavor firmware
2562         *
2563         * We do a TAVOR_NOSLEEP here because we are still holding the
2564         * "qp_lock".  If we got raised to interrupt level by priority
2565         * inversion, we do not want to block in this routine waiting for
2566         * success.
2567         */
2568        status = tavor_cmn_qp_cmd_post(state, SQD2RTS_QP, qpc, qp->qp_qpnum,
2569            opmask, TAVOR_CMD_NOSLEEP_SPIN);
```



```
2570        if (status != TAVOR_CMD_SUCCESS) {
2571                if (status != TAVOR_CMD_BAD_QP_STATE) {
2572                        cmn_err(CE_CONT, "Tavor: SQD2RTS_QP command failed: "
2573                            "%08x\n", status);
2574                        TNF_PROBE_1(tavor_qp_sqd2rts_cmd_fail,
2575                            TAVOR_TNF_ERROR, "", tnf_uint, status, status);
2576                        TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2577                        return (ibc_get_ci_failure(0));
2578                } else {
2579                        TNF_PROBE_0(tavor_qp_sqd2rts_inv_qpstate_fail,
2580                            TAVOR_TNF_ERROR, "");
2581                        TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2582                        return (IBT_QP_STATE_INVALID);
2583                }
2584        }

2586        TAVOR_TNF_EXIT(tavor_qp_sqd2rts);
2587        return (DDI_SUCCESS);
2588 }


2591 /*
2592  * tavor_qp_sqd2sqd()
2593  *    Context: Can be called from interrupt or base context.
2594  */
2595 static int
2596 tavor_qp_sqd2sqd(tavor_state_t *state, tavor_qphdl_t qp,
2597     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
2598 {
2599        tavor_hw_qpc_t          *qpc;
2600        ibt_qp_rc_attr_t        *rc;
2601        ibt_qp_ud_attr_t        *ud;
2602        ibt_qp_uc_attr_t        *uc;
2603        tavor_hw_addr_path_t    *qpc_path;
2604        ibt_adds_vect_t         *adds_vect;
2605        uint_t                  portnum, pkeyindx, rdma_ra_out, rdma_ra_in;
2606        uint_t                  rra_max, sra_max;
2607        uint32_t                opmask = 0;
2608        int                     status;

2610        TAVOR_TNF_ENTER(tavor_qp_sqd2sqd);

2612        ASSERT(MUTEX_HELD(&qp->qp_lock));

2614        /*
2615         * Grab the temporary QPC entry from QP software state
2616         */
2617        qpc = &qp->qpc;

2619        /*
2620         * Since there are no common and/or Tavor-specific fields to be filled
2621         * in for this command, we begin with the QPC fields which are
2622         * specific to transport type.
2623         */
2624        if (qp->qp_serv_type == TAVOR_QP_UD) {
2625                ud = &info_p->qp_transport.ud;

2627                /*
2628                 * If we are attempting to modify the PKey index for this QP,
2629                 * then check for valid PKey index and fill it in.  Also set
2630                 * the appropriate flag in the "opmask" parameter.
2631                 */
2632                if (flags & IBT_CEP_SET_PKEY_IX) {
2633                        pkeyindx = ud->ud_pkey_ix;
2634                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2635                                qpc->pri_addr_path.pkey_indx = pkeyindx;
```

```
2636                                opmask |= TAVOR_CMD_OP_PKEYINDX;
2637                                qp->qp_pkeyindx = pkeyindx;
2638                        } else {
2639                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_pkey_fail,
2640                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
2641                                    pkeyindx);
2642                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2643                                return (IBT_PKEY_IX_ILLEGAL);
2644                        }
2645                }

2647                /*
2648                 * If we are attempting to modify the QKey for this QP, then
2649                 * fill it in and set the appropriate flag in the "opmask"
2650                 * parameter.
2651                 */
2652                if (flags & IBT_CEP_SET_QKEY) {
2653                        qpc->qkey = ud->ud_qkey;
2654                        opmask |= TAVOR_CMD_OP_QKEY;
2655                }

2657        } else if (qp->qp_serv_type == TAVOR_QP_RC) {
2658                rc = &info_p->qp_transport.rc;

2660                /*
2661                 * Check if any of the flags indicate a change in the RDMA
2662                 * (recv) enable/disable flags and set the appropriate flag in
2663                 * the "opmask" parameter
2664                 */
2665                opmask |= tavor_check_rdma_enable_flags(flags, info_p, qpc);

2667                /*
2668                 * Check for optional primary path and fill in the
2669                 * appropriate QPC fields if one is specified
2670                 */
2671                if (flags & IBT_CEP_SET_ADDS_VECT) {
2672                        qpc_path = &qpc->pri_addr_path;
2673                        adds_vect = &rc->rc_path.cep_adds_vect;

2675                        /* Set the common primary address path fields */
2676                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
2677                            TAVOR_ADDRPATH_QP, qp);
2678                        if (status != DDI_SUCCESS) {
2679                                TNF_PROBE_0(tavor_qp_sqd2sqd_setaddrpath_fail,
2680                                    TAVOR_TNF_ERROR, "");
2681                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2682                                return (status);
2683                        }
2684                        qpc_path->rnr_retry = rc->rc_rnr_retry_cnt;
2685                        qpc_path->ack_timeout = rc->rc_path.cep_timeout;
2686                        qpc->retry_cnt = rc->rc_retry_cnt;

2688                        /*
2689                         * MTU changes as part of sqd2sqd are not allowed.
2690                         * Simply keep the same MTU value here, stored in the
2691                         * qphdl from init2rtr time.
2692                         */
2693                        qpc->mtu = qp->qp_save_mtu;

2695                        opmask |= (TAVOR_CMD_OP_PRIM_PATH |
2696                            TAVOR_CMD_OP_RETRYCNT | TAVOR_CMD_OP_ACKTIMEOUT |
2697                            TAVOR_CMD_OP_PRIM_RNRRETRY);
2698                }

2700                /*
2701                 * If we are attempting to modify the path migration state for
```



```
2702                 * this QP, then check for valid state and fill it in.  Also
2703                 * set the appropriate flag in the "opmask" parameter.
2704                 */
2705                if (flags & IBT_CEP_SET_MIG) {
2706                        if (rc->rc_mig_state == IBT_STATE_MIGRATED) {
2707                                qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
2708                        } else if (rc->rc_mig_state == IBT_STATE_REARMED) {
2709                                qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
2710                        } else {
2711                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_mig_state_fail,
2712                                    TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
2713                                    rc->rc_mig_state);
2714                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2715                                return (IBT_QP_APM_STATE_INVALID);
2716                        }
2717                        opmask |= TAVOR_CMD_OP_PM_STATE;
2718                }

2720                /*
2721                 * If we are attempting to modify the PKey index for this QP,
2722                 * then check for valid PKey index and fill it in.  Also set
2723                 * the appropriate flag in the "opmask" parameter.
2724                 */
2725                if (flags & IBT_CEP_SET_PKEY_IX) {
2726                        pkeyindx = rc->rc_path.cep_pkey_ix;
2727                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2728                                qpc->pri_addr_path.pkey_indx = pkeyindx;
2729                                opmask |= TAVOR_CMD_OP_PKEYINDX;
2730                        } else {
2731                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_pkey_fail,
2732                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
2733                                    pkeyindx);
2734                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2735                                return (IBT_PKEY_IX_ILLEGAL);
2736                        }
2737                }

2739                /*
2740                 * If we are attempting to modify the port for this QP, then
2741                 * check for valid port number and fill it in.  Also set the
2742                 * appropriate flag in the "opmask" parameter.
2743                 */
2744                if (flags & IBT_CEP_SET_PORT) {
2745                        portnum = rc->rc_path.cep_hca_port_num;
2746                        if (tavor_portnum_is_valid(state, portnum)) {
2747                                qpc->pri_addr_path.portnum = portnum;
2748                        } else {
2749                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_port_fail,
2750                                    TAVOR_TNF_ERROR, "", tnf_uint, port,
2751                                    portnum);
2752                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2753                                return (IBT_HCA_PORT_INVALID);
2754                        }
2755                        opmask |= TAVOR_CMD_OP_PRIM_PORT;
2756                }

2758                /*
2759                 * Check for optional alternate path and fill in the
2760                 * appropriate QPC fields if one is specified
2761                 */
2762                if (flags & IBT_CEP_SET_ALT_PATH) {
2763                        qpc_path = &qpc->alt_addr_path;
2764                        adds_vect = &rc->rc_alt_path.cep_adds_vect;

2766                        /* Set the common alternate address path fields */
2767                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
```

```
2768                            TAVOR_ADDRPATH_QP, qp);
2769                    if (status != DDI_SUCCESS) {
2770                            TNF_PROBE_0(tavor_qp_sqd2sqd_setaddrpath_fail,
2771                                TAVOR_TNF_ERROR, "");
2772                            TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2773                            return (status);
2774                    }
2775                    qpc_path->ack_timeout = rc->rc_alt_path.cep_timeout;

2777                    /*
2778                     * Check for valid alternate path port number and fill
2779                     * it in
2780                     */
2781                    portnum = rc->rc_alt_path.cep_hca_port_num;
2782                    if (tavor_portnum_is_valid(state, portnum)) {
2783                            qpc->alt_addr_path.portnum = portnum;
2784                    } else {
2785                            TNF_PROBE_1(tavor_qp_sqd2sqd_inv_altport_fail,
2786                                TAVOR_TNF_ERROR, "", tnf_uint, altport,
2787                                portnum);
2788                            TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2789                            return (IBT_HCA_PORT_INVALID);
2790                    }

2792                    /*
2793                     * Check for valid alternate path PKey index and fill
2794                     * it in
2795                     */
2796                    pkeyindx = rc->rc_alt_path.cep_pkey_ix;
2797                    if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2798                            qpc->alt_addr_path.pkey_indx = pkeyindx;
2799                    } else {
2800                            TNF_PROBE_1(tavor_qp_sqd2sqd_inv_altpkey_fail,
2801                                TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
2802                                pkeyindx);
2803                            TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2804                            return (IBT_PKEY_IX_ILLEGAL);
2805                    }
2806                    opmask |= TAVOR_CMD_OP_ALT_PATH;
2807            }

2809            /*
2810             * If we are attempting to modify the number of "outgoing
2811             * RDMA resources" for this QP, then check for valid value and
2812             * fill it in.  Also set the appropriate flag in the "opmask"
2813             * parameter.
2814             */
2815            if (flags & IBT_CEP_SET_RDMARA_OUT) {
2816                    rdma_ra_out = rc->rc_rdma_ra_out;
2817                    if (tavor_qp_validate_init_depth(state, rc,
2818                        &sra_max) != DDI_SUCCESS) {
2819                            TNF_PROBE_1(tavor_qp_sqd2sqd_inv_rdma_out_fail,
2820                                TAVOR_TNF_ERROR, "", tnf_uint, rdma_ra_out,
2821                                rdma_ra_out);
2822                            TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2823                            return (IBT_INVALID_PARAM);
2824                    }
2825                    qpc->sra_max = sra_max;
2826                    opmask |= TAVOR_CMD_OP_SRA_SET;
2827            }

2829            /*
2830             * If we are attempting to modify the number of "incoming
2831             * RDMA resources" for this QP, then check for valid value and
2832             * update the "rra_max" and "ra_buf_index" fields in the QPC to
2833             * point to the pre-allocated RDB resources (in DDR).  Also set
```



```
2834             * the appropriate flag in the "opmask" parameter.
2835             */
2836            if (flags & IBT_CEP_SET_RDMARA_IN) {
2837                    rdma_ra_in = rc->rc_rdma_ra_in;
2838                    if (tavor_qp_validate_resp_rsrc(state, rc,
2839                        &rra_max) != DDI_SUCCESS) {
2840                            TNF_PROBE_1(tavor_qp_sqd2sqd_inv_rdma_in_fail,
2841                                TAVOR_TNF_ERROR, "", tnf_uint, rdma_ra_in,
2842                                rdma_ra_in);
2843                            TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2844                            return (IBT_INVALID_PARAM);
2845                    }
2846                    qpc->rra_max = rra_max;
2847                    qpc->ra_buff_indx = qp->qp_rdb_ddraddr >>
2848                        TAVOR_RDB_SIZE_SHIFT;
2849                    opmask |= TAVOR_CMD_OP_RRA_SET;
2850            }

2852            /*
2853             * If we are attempting to modify the "Local Ack Timeout" value
2854             * for this QP, then fill it in and set the appropriate flag in
2855             * the "opmask" parameter.
2856             */
2857            if (flags & IBT_CEP_SET_TIMEOUT) {
2858                    qpc_path = &qpc->pri_addr_path;
2859                    qpc_path->ack_timeout = rc->rc_path.cep_timeout;
2860                    opmask |= TAVOR_CMD_OP_ACKTIMEOUT;
2861            }

2863            /*
2864             * If we are attempting to modify the "Retry Count" for this QP,
2865             * then fill it in and set the appropriate flag in the "opmask"
2866             * parameter.
2867             */
2868            if (flags & IBT_CEP_SET_RETRY) {
2869                    qpc->retry_cnt = rc->rc_retry_cnt;
2870                    opmask |= TAVOR_CMD_OP_PRIM_RNRRETRY;
2871            }

2873            /*
2874             * If we are attempting to modify the "RNR Retry Count" for this
2875             * QP, then fill it in and set the appropriate flag in the
2876             * "opmask" parameter.
2877             */
2878            if (flags & IBT_CEP_SET_RNR_NAK_RETRY) {
2879                    qpc_path = &qpc->pri_addr_path;
2880                    qpc_path->rnr_retry = rc->rc_rnr_retry_cnt;
2881                    opmask |= TAVOR_CMD_OP_RETRYCNT;
2882            }

2884            /*
2885             * If we are attempting to modify the "Minimum RNR NAK" value
2886             * for this QP, then fill it in and set the appropriate flag
2887             * in the "opmask" parameter.
2888             */
2889            if (flags & IBT_CEP_SET_MIN_RNR_NAK) {
2890                    qpc->min_rnr_nak = rc->rc_min_rnr_nak;
2891                    opmask |= TAVOR_CMD_OP_MINRNRNAK;
2892            }

2894    } else if (qp->qp_serv_type == TAVOR_QP_UC) {
2895            uc = &info_p->qp_transport.uc;

2897            /*
2898             * Check if any of the flags indicate a change in the RDMA
2899             * Write (recv) enable/disable and set the appropriate flag
```

```
2900                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
2901                 * not valid for UC transport.
2902                 */
2903                if (flags & IBT_CEP_SET_RDMA_W) {
2904                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
2905                        opmask |= TAVOR_CMD_OP_RWE;
2906                }

2908                /*
2909                 * Check for optional primary path and fill in the
2910                 * appropriate QPC fields if one is specified
2911                 */
2912                if (flags & IBT_CEP_SET_ADDS_VECT) {
2913                        qpc_path = &qpc->pri_addr_path;
2914                        adds_vect = &uc->uc_path.cep_adds_vect;

2916                        /* Set the common primary address path fields */
2917                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
2918                            TAVOR_ADDRPATH_QP, qp);
2919                        if (status != DDI_SUCCESS) {
2920                                TNF_PROBE_0(tavor_qp_sqd2sqd_setaddrpath_fail,
2921                                    TAVOR_TNF_ERROR, "");
2922                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2923                                return (status);
2924                        }

2926                        /*
2927                         * MTU changes as part of sqd2sqd are not allowed.
2928                         * Simply keep the same MTU value here, stored in the
2929                         * qphdl from init2rtr time.
2930                         */
2931                        qpc->mtu = qp->qp_save_mtu;

2933                        opmask |= TAVOR_CMD_OP_PRIM_PATH;
2934                }

2936                /*
2937                 * If we are attempting to modify the path migration state for
2938                 * this QP, then check for valid state and fill it in.  Also
2939                 * set the appropriate flag in the "opmask" parameter.
2940                 */
2941                if (flags & IBT_CEP_SET_MIG) {
2942                        if (uc->uc_mig_state == IBT_STATE_MIGRATED) {
2943                                qpc->pm_state = TAVOR_QP_PMSTATE_MIGRATED;
2944                        } else if (uc->uc_mig_state == IBT_STATE_REARMED) {
2945                                qpc->pm_state = TAVOR_QP_PMSTATE_REARM;
2946                        } else {
2947                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_mig_state_fail,
2948                                    TAVOR_TNF_ERROR, "", tnf_uint, mig_state,
2949                                    uc->uc_mig_state);
2950                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2951                                return (IBT_QP_APM_STATE_INVALID);
2952                        }
2953                        opmask |= TAVOR_CMD_OP_PM_STATE;
2954                }

2956                /*
2957                 * If we are attempting to modify the PKey index for this QP,
2958                 * then check for valid PKey index and fill it in.  Also set
2959                 * the appropriate flag in the "opmask" parameter.
2960                 */
2961                if (flags & IBT_CEP_SET_PKEY_IX) {
2962                        pkeyindx = uc->uc_path.cep_pkey_ix;
2963                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
2964                                qpc->pri_addr_path.pkey_indx = pkeyindx;
2965                                opmask |= TAVOR_CMD_OP_PKEYINDX;
```



```
2966                        } else {
2967                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_pkey,
2968                                    TAVOR_TNF_ERROR, "", tnf_uint, pkeyindx,
2969                                    pkeyindx);
2970                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2971                                return (IBT_PKEY_IX_ILLEGAL);
2972                        }
2973                }

2975                /*
2976                 * Check for optional alternate path and fill in the
2977                 * appropriate QPC fields if one is specified
2978                 */
2979                if (flags & IBT_CEP_SET_ALT_PATH) {
2980                        qpc_path = &qpc->alt_addr_path;
2981                        adds_vect = &uc->uc_alt_path.cep_adds_vect;

2983                        /* Set the common alternate address path fields */
2984                        status = tavor_set_addr_path(state, adds_vect, qpc_path,
2985                            TAVOR_ADDRPATH_QP, qp);
2986                        if (status != DDI_SUCCESS) {
2987                                TNF_PROBE_0(tavor_qp_sqd2sqd_setaddrpath_fail,
2988                                    TAVOR_TNF_ERROR, "");
2989                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
2990                                return (status);
2991                        }

2993                        /*
2994                         * Check for valid alternate path port number and fill
2995                         * it in
2996                         */
2997                        portnum = uc->uc_alt_path.cep_hca_port_num;
2998                        if (tavor_portnum_is_valid(state, portnum)) {
2999                                qpc->alt_addr_path.portnum = portnum;
3000                        } else {
3001                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_altport_fail,
3002                                    TAVOR_TNF_ERROR, "", tnf_uint, altport,
3003                                    portnum);
3004                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
3005                                return (IBT_HCA_PORT_INVALID);
3006                        }

3008                        /*
3009                         * Check for valid alternate path PKey index and fill
3010                         * it in
3011                         */
3012                        pkeyindx = uc->uc_alt_path.cep_pkey_ix;
3013                        if (tavor_pkeyindex_is_valid(state, pkeyindx)) {
3014                                qpc->alt_addr_path.pkey_indx = pkeyindx;
3015                        } else {
3016                                TNF_PROBE_1(tavor_qp_sqd2sqd_inv_altpkey_fail,
3017                                    TAVOR_TNF_ERROR, "", tnf_uint, altpkeyindx,
3018                                    pkeyindx);
3019                                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
3020                                return (IBT_PKEY_IX_ILLEGAL);
3021                        }
3022                        opmask |= TAVOR_CMD_OP_ALT_PATH;
3023                }
3024        } else {
3025                /*
3026                 * Invalid QP transport type. If we got here then it's a
3027                 * warning of a probably serious problem.  So print a message
3028                 * and return failure
3029                 */
3030                TAVOR_WARNING(state, "unknown QP transport type in sqd2sqd");
3031                TNF_PROBE_0(tavor_qp_sqd2sqd_inv_transtype_fail,
```

```
3032                    TAVOR_TNF_ERROR, "");
3033                TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
3034                return (ibc_get_ci_failure(0));
3035        }

3037        /*
3038         * Post the SQD2SQD_QP command to the Tavor firmware
3039         *
3040         * We do a TAVOR_NOSLEEP here because we are still holding the
3041         * "qp_lock".  If we got raised to interrupt level by priority
3042         * inversion, we do not want to block in this routine waiting for
3043         * success.
3044         */
3045        status = tavor_cmn_qp_cmd_post(state, SQD2SQD_QP, qpc, qp->qp_qpnum,
3046            opmask, TAVOR_CMD_NOSLEEP_SPIN);
3047        if (status != TAVOR_CMD_SUCCESS) {
3048                if (status != TAVOR_CMD_BAD_QP_STATE) {
3049                        cmn_err(CE_CONT, "Tavor: SQD2SQD_QP command failed: "
3050                            "%08x\n", status);
3051                        TNF_PROBE_1(tavor_qp_sqd2sqd_cmd_fail,
3052                            TAVOR_TNF_ERROR, "", tnf_uint, status, status);
3053                        TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
3054                        return (ibc_get_ci_failure(0));
3055                } else {
3056                        TNF_PROBE_0(tavor_qp_sqd2sqd_inv_qpstate_fail,
3057                            TAVOR_TNF_ERROR, "");
3058                        TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
3059                        return (IBT_QP_STATE_INVALID);
3060                }
3061        }

3063        TAVOR_TNF_EXIT(tavor_qp_sqd2sqd);
3064        return (DDI_SUCCESS);
3065 }


3068 /*
3069  * tavor_qp_sqerr2rts()
3070  *    Context: Can be called from interrupt or base context.
3071  */
3072 static int
3073 tavor_qp_sqerr2rts(tavor_state_t *state, tavor_qphdl_t qp,
3074     ibt_cep_modify_flags_t flags, ibt_qp_info_t *info_p)
3075 {
3076        tavor_hw_qpc_t          *qpc;
3077        ibt_qp_ud_attr_t        *ud;
3078        uint32_t                opmask = 0;
3079        int                     status;

3081        TAVOR_TNF_ENTER(tavor_qp_sqerr2rts);

3083        ASSERT(MUTEX_HELD(&qp->qp_lock));

3085        /*
3086         * Grab the temporary QPC entry from QP software state
3087         */
3088        qpc = &qp->qpc;

3090        /*
3091         * Since there are no common and/or Tavor-specific fields to be filled
3092         * in for this command, we begin with the QPC fields which are
3093         * specific to transport type.
3094         */
3095        if (qp->qp_serv_type == TAVOR_QP_UD) {
3096                ud = &info_p->qp_transport.ud;
```



```
3098                /*
3099                 * If we are attempting to modify the QKey for this QP, then
3100                 * fill it in and set the appropriate flag in the "opmask"
3101                 * parameter.
3102                 */
3103                if (flags & IBT_CEP_SET_QKEY) {
3104                        qpc->qkey = ud->ud_qkey;
3105                        opmask |= TAVOR_CMD_OP_QKEY;
3106                }

3108        } else if (qp->qp_serv_type == TAVOR_QP_UC) {

3110                /*
3111                 * Check if any of the flags indicate a change in the RDMA
3112                 * Write (recv) enable/disable and set the appropriate flag
3113                 * in the "opmask" parameter. Note: RDMA Read and Atomic are
3114                 * not valid for UC transport.
3115                 */
3116                if (flags & IBT_CEP_SET_RDMA_W) {
3117                        qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
3118                        opmask |= TAVOR_CMD_OP_RWE;
3119                }
3120        } else {
3121                /*
3122                 * Invalid QP transport type. If we got here then it's a
3123                 * warning of a probably serious problem.  So print a message
3124                 * and return failure
3125                 */
3126                TAVOR_WARNING(state, "unknown QP transport type in sqerr2rts");
3127                TNF_PROBE_0(tavor_qp_sqerr2rts_inv_transtype_fail,
3128                    TAVOR_TNF_ERROR, "");
3129                TAVOR_TNF_EXIT(tavor_qp_sqerr2rts);
3130                return (ibc_get_ci_failure(0));
3131        }

3133        /*
3134         * Post the SQERR2RTS_QP command to the Tavor firmware
3135         *
3136         * We do a TAVOR_NOSLEEP here because we are still holding the
3137         * "qp_lock".  If we got raised to interrupt level by priority
3138         * inversion, we do not want to block in this routine waiting for
3139         * success.
3140         */
3141        status = tavor_cmn_qp_cmd_post(state, SQERR2RTS_QP, qpc, qp->qp_qpnum,
3142            opmask, TAVOR_CMD_NOSLEEP_SPIN);
3143        if (status != TAVOR_CMD_SUCCESS) {
3144                if (status != TAVOR_CMD_BAD_QP_STATE) {
3145                        cmn_err(CE_CONT, "Tavor: SQERR2RTS_QP command failed: "
3146                            "%08x\n", status);
3147                        TNF_PROBE_1(tavor_qp_sqerr2rts_cmd_fail,
3148                            TAVOR_TNF_ERROR, "", tnf_uint, status, status);
3149                        TAVOR_TNF_EXIT(tavor_qp_sqerr2rts);
3150                        return (ibc_get_ci_failure(0));
3151                } else {
3152                        TNF_PROBE_0(tavor_qp_sqerr2rts_inv_qpstate_fail,
3153                            TAVOR_TNF_ERROR, "");
3154                        TAVOR_TNF_EXIT(tavor_qp_sqerr2rts);
3155                        return (IBT_QP_STATE_INVALID);
3156                }
3157        }

3159        TAVOR_TNF_EXIT(tavor_qp_sqerr2rts);
3160        return (DDI_SUCCESS);
3161 }
```

```
3164 /*
3165  * tavor_qp_to_error()
3166  *    Context: Can be called from interrupt or base context.
3167  */
3168 static int
3169 tavor_qp_to_error(tavor_state_t *state, tavor_qphdl_t qp)
3170 {
3171         int     status;

3173         TAVOR_TNF_ENTER(tavor_qp_to_error);

3175         ASSERT(MUTEX_HELD(&qp->qp_lock));

3177         /*
3178          * Post the TOERR_QP command to the Tavor firmware
3179          *
3180          * We do a TAVOR_NOSLEEP here because we are still holding the
3181          * "qp_lock".  If we got raised to interrupt level by priority
3182          * inversion, we do not want to block in this routine waiting for
3183          * success.
3184          */
3185         status = tavor_cmn_qp_cmd_post(state, TOERR_QP, NULL, qp->qp_qpnum,
3186             0, TAVOR_CMD_NOSLEEP_SPIN);
3187         if (status != TAVOR_CMD_SUCCESS) {
3188                 cmn_err(CE_CONT, "Tavor: TOERR_QP command failed: %08x\n",
3189                     status);
3190                 TNF_PROBE_1(tavor_qp_to_error_cmd_fail,
3191                     TAVOR_TNF_ERROR, "", tnf_uint, status, status);
3192                 TAVOR_TNF_EXIT(tavor_qp_to_error);
3193                 return (ibc_get_ci_failure(0));
3194         }

3196         TAVOR_TNF_EXIT(tavor_qp_to_error);
3197         return (DDI_SUCCESS);
3198 }


3201 /*
3202  * tavor_qp_to_reset()
3203  *    Context: Can be called from interrupt or base context.
3204  */
3205 int
3206 tavor_qp_to_reset(tavor_state_t *state, tavor_qphdl_t qp)
3207 {
3208         tavor_hw_qpc_t  *qpc;
3209         int             status;

3211         TAVOR_TNF_ENTER(tavor_qp_to_reset);

3213         ASSERT(MUTEX_HELD(&qp->qp_lock));

3215         /*
3216          * Grab the temporary QPC entry from QP software state
3217          */
3218         qpc = &qp->qpc;

3220         /*
3221          * Post the TORST_QP command to the Tavor firmware
3222          *
3223          * We do a TAVOR_NOSLEEP here because we are still holding the
3224          * "qp_lock".  If we got raised to interrupt level by priority
3225          * inversion, we do not want to block in this routine waiting for
3226          * success.
3227          */
3228         status = tavor_cmn_qp_cmd_post(state, TORST_QP, qpc, qp->qp_qpnum,
3229             0, TAVOR_CMD_NOSLEEP_SPIN);
```



```
3230         if (status != TAVOR_CMD_SUCCESS) {
3231                 cmn_err(CE_CONT, "Tavor: TORST_QP command failed: %08x\n",
3232                     status);
3233                 TNF_PROBE_1(tavor_qp_to_reset_cmd_fail,
3234                     TAVOR_TNF_ERROR, "", tnf_uint, status, status);
3235                 TAVOR_TNF_EXIT(tavor_qp_to_reset);
3236                 return (ibc_get_ci_failure(0));
3237         }

3239         TAVOR_TNF_EXIT(tavor_qp_to_reset);
3240         return (DDI_SUCCESS);
3241 }


3244 /*
3245  * tavor_qp_reset2err()
3246  *    Context: Can be called from interrupt or base context.
3247  */
3248 static int
3249 tavor_qp_reset2err(tavor_state_t *state, tavor_qphdl_t qp)
3250 {
3251         tavor_hw_qpc_t  *qpc;
3252         int             status;

3254         TAVOR_TNF_ENTER(tavor_qp_reset2err);

3256         ASSERT(MUTEX_HELD(&qp->qp_lock));

3258         /*
3259          * In order to implement the transition from "Reset" directly to the
3260          * "Error" state, it is necessary to first give ownership of the QP
3261          * context to the Tavor hardware.  This is accomplished by transitioning
3262          * the QP to "Init" as an intermediate step and then, immediately
3263          * transitioning to "Error".
3264          *
3265          * When this function returns success, the QP context will be owned by
3266          * the Tavor hardware and will be in the "Error" state.
3267          */

3269         /*
3270          * Grab the temporary QPC entry from QP software state
3271          */
3272         qpc = &qp->qpc;

3274         /*
3275          * Fill in the common and/or Tavor-specific fields in the QPC
3276          */
3277         if (qp->qp_is_special) {
3278                 qpc->serv_type  = TAVOR_QP_MLX;
3279         } else {
3280                 qpc->serv_type  = qp->qp_serv_type;
3281         }
3282         qpc->pm_state           = TAVOR_QP_PMSTATE_MIGRATED;
3283         qpc->de                 = TAVOR_QP_DESC_EVT_ENABLED;
3284         qpc->sched_q            = TAVOR_QP_SCHEDQ_GET(qp->qp_qpnum);
3285         if (qp->qp_is_umap) {
3286                 qpc->usr_page = qp->qp_uarpg;
3287         } else {
3288                 qpc->usr_page = 0;
3289         }
3290         qpc->pd                 = qp->qp_pdhdl->pd_pdnum;
3291         qpc->wqe_baseaddr       = 0;
3292         qpc->wqe_lkey           = qp->qp_mrhdl->mr_lkey;
3293         qpc->ssc                = qp->qp_sq_sigtype;
3294         qpc->cqn_snd            = qp->qp_sq_cqhdl->cq_cqnum;
3295         qpc->rsc                = TAVOR_QP_RQ_ALL_SIGNALED;
```

```
3296         qpc->cqn_rcv            = qp->qp_rq_cqhdl->cq_cqnum;
3297         qpc->srq_en             = qp->qp_srq_en;
3298
3299         if (qp->qp_srq_en == TAVOR_QP_SRQ_ENABLED) {
3300                 qpc->srq_number = qp->qp_srqhdl->srq_srqnum;
3301         } else {
3302                 qpc->srq_number = 0;
3303         }
3304
3305         /*
3306          * Now fill in the QPC fields which are specific to transport type
3307          */
3308         if (qp->qp_serv_type == TAVOR_QP_UD) {
3309                 /* Set the UD parameters to an invalid default */
3310                 qpc->qkey = 0;
3311                 qpc->pri_addr_path.portnum = 1;
3312                 qpc->pri_addr_path.pkey_indx = 0;
3313
3314         } else if (qp->qp_serv_type == TAVOR_QP_RC) {
3315                 /* Set the RC parameters to invalid default */
3316                 qpc->rre = 0;
3317                 qpc->rwe = 0;
3318                 qpc->rae = 0;
3319                 qpc->pri_addr_path.portnum = 1;
3320                 qpc->pri_addr_path.pkey_indx = 0;
3321
3322         } else if (qp->qp_serv_type == TAVOR_QP_UC) {
3323                 /* Set the UC parameters to invalid default */
3324                 qpc->rwe = 0;
3325                 qpc->pri_addr_path.portnum = 1;
3326                 qpc->pri_addr_path.pkey_indx = 0;
3327
3328         } else {
3329                 /*
3330                  * Invalid QP transport type. If we got here then it's a
3331                  * warning of a probably serious problem.  So print a message
3332                  * and return failure
3333                  */
3334                 TAVOR_WARNING(state, "unknown QP transport type in rst2err");
3335                 TNF_PROBE_0(tavor_qp_reset2err_inv_transtype_fail,
3336                     TAVOR_TNF_ERROR, "");
3337                 TAVOR_TNF_EXIT(tavor_qp_reset2err);
3338                 return (ibc_get_ci_failure(0));
3339         }
3340
3341         /*
3342          * Post the RST2INIT_QP command to the Tavor firmware
3343          *
3344          * We do a TAVOR_NOSLEEP here because we are still holding the
3345          * "qp_lock".  If we got raised to interrupt level by priority
3346          * inversion, we do not want to block in this routine waiting for
3347          * success.
3348          */
3349         status = tavor_cmn_qp_cmd_post(state, RST2INIT_QP, qpc, qp->qp_qpnum,
3350             0, TAVOR_CMD_NOSLEEP_SPIN);
3351         if (status != TAVOR_CMD_SUCCESS) {
3352                 cmn_err(CE_CONT, "Tavor: RST2INIT_QP command failed: %08x\n",
3353                     status);
3354                 TNF_PROBE_1(tavor_qp_reset2err_rst2init_cmd_fail,
3355                     TAVOR_TNF_ERROR, "", tnf_uint, status, status);
3356                 TAVOR_TNF_EXIT(tavor_qp_reset2err);
3357                 return (ibc_get_ci_failure(0));
3358         }
3359
3360         /*
3361          * Now post the TOERR_QP command to the Tavor firmware
```



```
3362          *
3363          * We still do a TAVOR_NOSLEEP here because we are still holding the
3364          * "qp_lock".  Note:  If this fails (which it really never should),
3365          * it indicates a serious problem in the HW or SW.  We try to move
3366          * the QP back to the "Reset" state if possible and print a warning
3367          * message if not.  In any case, we return an error here.
3368          */
3369         status = tavor_cmn_qp_cmd_post(state, TOERR_QP, NULL, qp->qp_qpnum,
3370             0, TAVOR_CMD_NOSLEEP_SPIN);
3371         if (status != TAVOR_CMD_SUCCESS) {
3372                 cmn_err(CE_CONT, "Tavor: TOERR_QP command failed: %08x\n",
3373                     status);
3374                 if (tavor_qp_to_reset(state, qp) != DDI_SUCCESS) {
3375                         TAVOR_WARNING(state, "failed to reset QP context");
3376                 }
3377                 TNF_PROBE_1(tavor_qp_reset2err_toerr_cmd_fail,
3378                     TAVOR_TNF_ERROR, "", tnf_uint, status, status);
3379                 TAVOR_TNF_EXIT(tavor_qp_reset2err);
3380                 return (ibc_get_ci_failure(0));
3381         }
3382
3383         TAVOR_TNF_EXIT(tavor_qp_reset2err);
3384         return (DDI_SUCCESS);
3385 }
3386
3387
3388 /*
3389  * tavor_check_rdma_enable_flags()
3390  *    Context: Can be called from interrupt or base context.
3391  */
3392 static uint_t
3393 tavor_check_rdma_enable_flags(ibt_cep_modify_flags_t flags,
3394     ibt_qp_info_t *info_p, tavor_hw_qpc_t *qpc)
3395 {
3396         uint_t  opmask = 0;
3397
3398         if (flags & IBT_CEP_SET_RDMA_R) {
3399                 qpc->rre = (info_p->qp_flags & IBT_CEP_RDMA_RD) ? 1 : 0;
3400                 opmask |= TAVOR_CMD_OP_RRE;
3401         }
3402
3403         if (flags & IBT_CEP_SET_RDMA_W) {
3404                 qpc->rwe = (info_p->qp_flags & IBT_CEP_RDMA_WR) ? 1 : 0;
3405                 opmask |= TAVOR_CMD_OP_RWE;
3406         }
3407
3408         if (flags & IBT_CEP_SET_ATOMIC) {
3409                 qpc->rae = (info_p->qp_flags & IBT_CEP_ATOMIC) ? 1 : 0;
3410                 opmask |= TAVOR_CMD_OP_RAE;
3411         }
3412
3413         return (opmask);
3414 }
3415
3416 /*
3417  * tavor_qp_validate_resp_rsrc()
3418  *    Context: Can be called from interrupt or base context.
3419  */
3420 static int
3421 tavor_qp_validate_resp_rsrc(tavor_state_t *state, ibt_qp_rc_attr_t *rc,
3422     uint_t *rra_max)
3423 {
3424         uint_t  rdma_ra_in;
3425
3426         rdma_ra_in = rc->rc_rdma_ra_in;
```

```
3428         /*
3429          * Check if number of responder resources is too large.  Return an
3430          * error if it is
3431          */
3432         if (rdma_ra_in > state->ts_cfg_profile->cp_hca_max_rdma_in_qp) {
3433                 return (IBT_INVALID_PARAM);
3434         }

3436         /*
3437          * If the number of responder resources is too small, round it up.
3438          * Then find the next highest power-of-2
3439          */
3440         if (rdma_ra_in == 0) {
3441                 rdma_ra_in = 1;
3442         }
3443         if (ISP2(rdma_ra_in)) {
  36         if ((rdma_ra_in & (rdma_ra_in - 1)) == 0) {
3444                 *rra_max = highbit(rdma_ra_in) - 1;
3445         } else {
3446                 *rra_max = highbit(rdma_ra_in);
3447         }
3448         return (DDI_SUCCESS);
3449 }


3452 /*
3453  * tavor_qp_validate_init_depth()
3454  *    Context: Can be called from interrupt or base context.
3455  */
3456 static int
3457 tavor_qp_validate_init_depth(tavor_state_t *state, ibt_qp_rc_attr_t *rc,
3458     uint_t *sra_max)
3459 {
3460         uint_t  rdma_ra_out;

3462         rdma_ra_out = rc->rc_rdma_ra_out;

3464         /*
3465          * Check if requested initiator depth is too large.  Return an error
3466          * if it is
3467          */
3468         if (rdma_ra_out > state->ts_cfg_profile->cp_hca_max_rdma_out_qp) {
3469                 return (IBT_INVALID_PARAM);
3470         }

3472         /*
3473          * If the requested initiator depth is too small, round it up.
3474          * Then find the next highest power-of-2
3475          */
3476         if (rdma_ra_out == 0) {
3477                 rdma_ra_out = 1;
3478         }
3479         if (ISP2(rdma_ra_out)) {
  72         if ((rdma_ra_out & (rdma_ra_out - 1)) == 0) {
3480                 *sra_max = highbit(rdma_ra_out) - 1;
3481         } else {
3482                 *sra_max = highbit(rdma_ra_out);
3483         }
3484         return (DDI_SUCCESS);
3485 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   95569 Thu Oct 23 10:42:14 2014
new/usr/src/uts/common/io/ib/adapters/tavor/tavor_rsrc.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * tavor_rsrc.c
 *    Tavor Resource Management Routines
 *
 *    Implements all the routines necessary for setup, teardown, and
 *    alloc/free of all Tavor resources, including those that are managed
 *    by Tavor hardware or which live in Tavor's direct attached DDR memory.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/vmem.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/tavor/tavor.h>

/*
 * The following routines are used for initializing and destroying
 * the resource pools used by the Tavor resource allocation routines.
 * They consist of four classes of object:
 *
 * Mailboxes:  The "In" and "Out" mailbox types are used by the Tavor
 *    command interface routines.  Mailboxes are used to pass information
 *    back and forth to the Tavor firmware.  Either type of mailbox may
 *    be allocated from Tavor's direct attached DDR memory or from system
 *    memory (although currently all "In" mailboxes are in DDR and all "out"
 *    mailboxes come from system memory.
 *
 * HW entry objects:  These objects represent resources required by the Tavor
 *    hardware.  These objects include things like Queue Pair contexts (QPC),
```



```
 *    Completion Queue contexts (CQC), Event Queue contexts (EQC), RDB (for
 *    supporting RDMA Read/Atomic), Multicast Group entries (MCG), Memory
 *    Protection Table entries (MPT), Memory Translation Table entries (MTT).
 *
 *    What these objects all have in common is that they are each required
 *    to come from DDR memory, they are always allocated from tables, and
 *    they are not to be directly accessed (read or written) by driver
 *    software.
 *    One notable exceptions to this rule are the Extended QP contexts (EQPC),
 *    and the UAR scratch area (UAR_SCR), both of which are not directly
 *    accessible through the Tavor resource allocation routines, but both
 *    of which are also required to reside in DDR memory and are not to be
 *    manipulated by driver software (they are privately managed by Tavor
 *    hardware).
 *    The other notable exceptions are the UAR pages (UAR_PG) which are
 *    allocated from the UAR address space rather than DDR, and the UD
 *    address vectors (UDAV) which are similar to the common object types
 *    with the major difference being that UDAVs _are_ directly read and
 *    written by driver software.
 *
 * SW handle objects: These objects represent resources required by Tavor
 *    driver software.  They are primarily software tracking structures,
 *    which are allocated from system memory (using kmem_cache).  Several of
 *    the objects have both a "constructor" and "destructor" method
 *    associated with them (see below).
 *
 * Protection Domain (PD) handle objects:  These objects are very much like
 *    a SW handle object with the notable difference that all PD handle
 *    objects have an actual Protection Domain number (PD) associated with
 *    them (and the PD number is allocated/managed through a separate
 *    vmem_arena specifically set aside for this purpose.
 */

static int tavor_rsrc_mbox_init(tavor_state_t *state,
    tavor_rsrc_mbox_info_t *info);
static void tavor_rsrc_mbox_fini(tavor_state_t *state,
    tavor_rsrc_mbox_info_t *info);

static int tavor_rsrc_hw_entries_init(tavor_state_t *state,
    tavor_rsrc_hw_entry_info_t *info);
static void tavor_rsrc_hw_entries_fini(tavor_state_t *state,
    tavor_rsrc_hw_entry_info_t *info);

static int tavor_rsrc_sw_handles_init(tavor_state_t *state,
    tavor_rsrc_sw_hdl_info_t *info);
static void tavor_rsrc_sw_handles_fini(tavor_state_t *state,
    tavor_rsrc_sw_hdl_info_t *info);

static int tavor_rsrc_pd_handles_init(tavor_state_t *state,
    tavor_rsrc_sw_hdl_info_t *info);
static void tavor_rsrc_pd_handles_fini(tavor_state_t *state,
    tavor_rsrc_sw_hdl_info_t *info);

/*
 * The following routines are used for allocating and freeing the specific
 * types of objects described above from their associated resource pools.
 */
static int tavor_rsrc_mbox_alloc(tavor_rsrc_pool_info_t *pool_info,
    uint_t num, tavor_rsrc_t *hdl);
static void tavor_rsrc_mbox_free(tavor_rsrc_pool_info_t *pool_info,
    tavor_rsrc_t *hdl);

static int tavor_rsrc_hw_entry_alloc(tavor_rsrc_pool_info_t *pool_info,
    uint_t num, uint_t num_align, ddi_acc_handle_t acc_handle,
    uint_t sleepflag, tavor_rsrc_t *hdl);
static void tavor_rsrc_hw_entry_free(tavor_rsrc_pool_info_t *pool_info,
```

```
128     tavor_rsrc_t *hdl);

130 static int tavor_rsrc_swhdl_alloc(tavor_rsrc_pool_info_t *pool_info,
131     uint_t sleepflag, tavor_rsrc_t *hdl);
132 static void tavor_rsrc_swhdl_free(tavor_rsrc_pool_info_t *pool_info,
133     tavor_rsrc_t *hdl);

135 static int tavor_rsrc_pdhdl_alloc(tavor_rsrc_pool_info_t *pool_info,
136     uint_t sleepflag, tavor_rsrc_t *hdl);
137 static void tavor_rsrc_pdhdl_free(tavor_rsrc_pool_info_t *pool_info,
138     tavor_rsrc_t *hdl);

140 /*
141  * The following routines are the constructors and destructors for several
142  * of the SW handle type objects.  For certain types of SW handles objects
143  * (all of which are implemented using kmem_cache), we need to do some
144  * special field initialization (specifically, mutex_init/destroy).  These
145  * routines enable that init and teardown.
146  */
147 static int tavor_rsrc_pdhdl_constructor(void *pd, void *priv, int flags);
148 static void tavor_rsrc_pdhdl_destructor(void *pd, void *state);
149 static int tavor_rsrc_cqhdl_constructor(void *cq, void *priv, int flags);
150 static void tavor_rsrc_cqhdl_destructor(void *cq, void *state);
151 static int tavor_rsrc_qphdl_constructor(void *cq, void *priv, int flags);
152 static void tavor_rsrc_qphdl_destructor(void *cq, void *state);
153 static int tavor_rsrc_srqhdl_constructor(void *srq, void *priv, int flags);
154 static void tavor_rsrc_srqhdl_destructor(void *srq, void *state);
155 static int tavor_rsrc_refcnt_constructor(void *rc, void *priv, int flags);
156 static void tavor_rsrc_refcnt_destructor(void *rc, void *state);
157 static int tavor_rsrc_ahhdl_constructor(void *ah, void *priv, int flags);
158 static void tavor_rsrc_ahhdl_destructor(void *ah, void *state);
159 static int tavor_rsrc_mrhdl_constructor(void *mr, void *priv, int flags);
160 static void tavor_rsrc_mrhdl_destructor(void *mr, void *state);

162 /*
163  * Special routine to calculate and return the size of a MCG object based
164  * on current driver configuration (specifically, the number of QP per MCG
165  * that has been configured.
166  */
167 static int tavor_rsrc_mcg_entry_get_size(tavor_state_t *state,
168     uint_t *mcg_size_shift);


171 /*
172  * tavor_rsrc_alloc()
173  *
174  *    Context: Can be called from interrupt or base context.
175  *    The "sleepflag" parameter is used by all object allocators to
176  *    determine whether to SLEEP for resources or not.
177  */
178 int
179 tavor_rsrc_alloc(tavor_state_t *state, tavor_rsrc_type_t rsrc, uint_t num,
180     uint_t sleepflag, tavor_rsrc_t **hdl)
181 {
182         tavor_rsrc_pool_info_t  *rsrc_pool;
183         tavor_rsrc_t            *tmp_rsrc_hdl;
184         int                     flag, status = DDI_FAILURE;

186         TAVOR_TNF_ENTER(tavor_rsrc_alloc);

188         ASSERT(state != NULL);
189         ASSERT(hdl != NULL);

191         rsrc_pool = &state->ts_rsrc_hdl[rsrc];
192         ASSERT(rsrc_pool != NULL);
```



```
194         /*
195          * Allocate space for the object used to track the resource handle
196          */
197         flag = (sleepflag == TAVOR_SLEEP) ? KM_SLEEP : KM_NOSLEEP;
198         tmp_rsrc_hdl = (tavor_rsrc_t *)kmem_cache_alloc(state->ts_rsrc_cache,
199             flag);
200         if (tmp_rsrc_hdl == NULL) {
201                 TNF_PROBE_0(tavor_rsrc_alloc_kmca_fail, TAVOR_TNF_ERROR, "");
202                 TAVOR_TNF_EXIT(tavor_rsrc_alloc);
203                 return (DDI_FAILURE);
204         }
205         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*tmp_rsrc_hdl))

207         /*
208          * Set rsrc_hdl type.  This is later used by the tavor_rsrc_free call
209          * to know what type of resource is being freed.
210          */
211         tmp_rsrc_hdl->rsrc_type = rsrc;

213         /*
214          * Depending on resource type, call the appropriate alloc routine
215          */
216         switch (rsrc_pool->rsrc_type) {
217         case TAVOR_IN_MBOX:
218         case TAVOR_OUT_MBOX:
219         case TAVOR_INTR_IN_MBOX:
220         case TAVOR_INTR_OUT_MBOX:
221                 status = tavor_rsrc_mbox_alloc(rsrc_pool, num, tmp_rsrc_hdl);
222                 break;

224         case TAVOR_QPC:
225         case TAVOR_CQC:
226         case TAVOR_SRQC:
227         case TAVOR_EQC:
228         case TAVOR_RDB:
229                 /*
230                  * Because these objects are NOT accessed by Tavor driver
231                  * software, we set the acc_handle parameter to zero.  But
232                  * if they are allocated in multiples, we specify here that
233                  * they must be aligned on a more restrictive boundary.
234                  */
235                 status = tavor_rsrc_hw_entry_alloc(rsrc_pool, num, num, 0,
236                     sleepflag, tmp_rsrc_hdl);
237                 break;

239         case TAVOR_MPT:
240                 /*
241                  * Because these MPT objects are sometimes accessed by Tavor
242                  * driver software (FMR), we set the acc_handle parameter.  But
243                  * if they are allocated in multiples, we specify here that
244                  * they must be aligned on a more restrictive boundary.
245                  */
246                 status = tavor_rsrc_hw_entry_alloc(rsrc_pool, num, num,
247                     state->ts_reg_ddrhdl, sleepflag, tmp_rsrc_hdl);
248                 break;

250         case TAVOR_MCG:
251                 /*
252                  * Tavor MCG entries are also NOT accessed by Tavor driver
253                  * software, but because MCG entries do not have the same
254                  * alignnment restrictions we loosen the constraint here.
255                  */
256                 status = tavor_rsrc_hw_entry_alloc(rsrc_pool, num, 1, 0,
257                     sleepflag, tmp_rsrc_hdl);
258                 break;
```

```
260         case TAVOR_MTT:
261         case TAVOR_UDAV:
262                 /*
263                  * Because MTT segments are among the few HW resources that
264                  * may be allocated in odd numbers, we specify a less
265                  * restrictive alignment than for the above resources.
266                  *
267                  * Also because both UDAV and MTT segment objects are read
268                  * and/or written by Tavor driver software, we set the
269                  * acc_handle parameter to point to the ddi_acc_handle_t for
270                  * the Tavor DDR memory.
271                  */
272                 status = tavor_rsrc_hw_entry_alloc(rsrc_pool, num, 1,
273                     state->ts_reg_ddrhdl, sleepflag, tmp_rsrc_hdl);
274                 break;
275
276         case TAVOR_UARPG:
277                 /*
278                  * Because UAR pages are written by Tavor driver software (for
279                  * doorbells), we set the acc_handle parameter to point to
280                  * the ddi_acc_handle_t for the Tavor UAR memory.
281                  */
282                 status = tavor_rsrc_hw_entry_alloc(rsrc_pool, num, 1,
283                     state->ts_reg_uarhdl, sleepflag, tmp_rsrc_hdl);
284                 break;
285
286         case TAVOR_MRHDL:
287         case TAVOR_EQHDL:
288         case TAVOR_CQHDL:
289         case TAVOR_SRQHDL:
290         case TAVOR_AHHDL:
291         case TAVOR_QPHDL:
292         case TAVOR_REFCNT:
293                 status = tavor_rsrc_swhdl_alloc(rsrc_pool, sleepflag,
294                     tmp_rsrc_hdl);
295                 break;
296
297         case TAVOR_PDHDL:
298                 status = tavor_rsrc_pdhdl_alloc(rsrc_pool, sleepflag,
299                     tmp_rsrc_hdl);
300                 break;
301
302         default:
303                 TAVOR_WARNING(state, "unexpected resource type in alloc");
304                 TNF_PROBE_0(tavor_rsrc_alloc_inv_rsrctype_fail,
305                     TAVOR_TNF_ERROR, "");
306                 break;
307         }
308
309         /*
310          * If the resource allocation failed, then free the special resource
311          * tracking structure and return failure.  Otherwise return the
312          * handle for the resource tracking structure.
313          */
314         if (status != DDI_SUCCESS) {
315                 kmem_cache_free(state->ts_rsrc_cache, tmp_rsrc_hdl);
316                 tmp_rsrc_hdl = NULL;
317                 TNF_PROBE_1(tavor_rsrc_alloc_fail, TAVOR_TNF_ERROR, "",
318                     tnf_uint, rsrc_type, rsrc_pool->rsrc_type);
319                 TAVOR_TNF_EXIT(tavor_rsrc_alloc);
320                 return (DDI_FAILURE);
321         } else {
322                 *hdl = tmp_rsrc_hdl;
323                 TAVOR_TNF_EXIT(tavor_rsrc_alloc);
324                 return (DDI_SUCCESS);
325         }
```



```
326 }
327
328
329 /*
330  * tavor_rsrc_free()
331  *    Context: Can be called from interrupt or base context.
332  */
333 void
334 tavor_rsrc_free(tavor_state_t *state, tavor_rsrc_t **hdl)
335 {
336         tavor_rsrc_pool_info_t  *rsrc_pool;
337
338         TAVOR_TNF_ENTER(tavor_rsrc_free);
339
340         ASSERT(state != NULL);
341         ASSERT(hdl != NULL);
342
343         rsrc_pool = &state->ts_rsrc_hdl[(*hdl)->rsrc_type];
344         ASSERT(rsrc_pool != NULL);
345
346         /*
347          * Depending on resource type, call the appropriate free routine
348          */
349         switch (rsrc_pool->rsrc_type) {
350         case TAVOR_IN_MBOX:
351         case TAVOR_OUT_MBOX:
352         case TAVOR_INTR_IN_MBOX:
353         case TAVOR_INTR_OUT_MBOX:
354                 tavor_rsrc_mbox_free(rsrc_pool, *hdl);
355                 break;
356
357         case TAVOR_QPC:
358         case TAVOR_CQC:
359         case TAVOR_SRQC:
360         case TAVOR_EQC:
361         case TAVOR_RDB:
362         case TAVOR_MCG:
363         case TAVOR_MPT:
364         case TAVOR_MTT:
365         case TAVOR_UDAV:
366         case TAVOR_UARPG:
367                 tavor_rsrc_hw_entry_free(rsrc_pool, *hdl);
368                 break;
369
370         case TAVOR_MRHDL:
371         case TAVOR_EQHDL:
372         case TAVOR_CQHDL:
373         case TAVOR_SRQHDL:
374         case TAVOR_AHHDL:
375         case TAVOR_QPHDL:
376         case TAVOR_REFCNT:
377                 tavor_rsrc_swhdl_free(rsrc_pool, *hdl);
378                 break;
379
380         case TAVOR_PDHDL:
381                 tavor_rsrc_pdhdl_free(rsrc_pool, *hdl);
382                 break;
383
384         default:
385                 TAVOR_WARNING(state, "unexpected resource type in free");
386                 TNF_PROBE_0(tavor_rsrc_free_inv_rsrctype_fail,
387                     TAVOR_TNF_ERROR, "");
388                 break;
389         }
390
391         /*
```

```
392          * Free the special resource tracking structure, set the handle to
393          * NULL, and return.
394          */
395         kmem_cache_free(state->ts_rsrc_cache, *hdl);
396         *hdl = NULL;
397
398         TAVOR_TNF_EXIT(tavor_rsrc_free);
399 }
400
401
402 /*
403  * tavor_rsrc_init_phase1()
404  *
405  *    Completes the first phase of Tavor resource/configuration init.
406  *    This involves creating the kmem_cache for the "tavor_rsrc_t"
407  *    structs, allocating the space for the resource pool handles,
408  *    and setting up the "Out" mailboxes.
409  *
410  *    When this function completes, the Tavor driver is ready to
411  *    post the following commands which return information only in the
412  *    "Out" mailbox: QUERY_DDR, QUERY_FW, QUERY_DEV_LIM, and QUERY_ADAPTER
413  *    If any of these commands are to be posted at this time, they must be
414  *    done so only when "spinning" (as the outstanding command list and
415  *    EQ setup code has not yet run)
416  *
417  *    Context: Only called from attach() path context
418  */
419 int
420 tavor_rsrc_init_phase1(tavor_state_t *state)
421 {
422         tavor_rsrc_pool_info_t          *rsrc_pool;
423         tavor_rsrc_mbox_info_t          mbox_info;
424         tavor_rsrc_cleanup_level_t      cleanup;
425         tavor_cfg_profile_t             *cfgprof;
426         uint64_t                        num, size;
427         int                             status;
428         char                            *errormsg, *rsrc_name;
429
430         TAVOR_TNF_ENTER(tavor_rsrc_init_phase1);
431
432         ASSERT(state != NULL);
433
434         /* This is where Phase 1 of resource initialization begins */
435         cleanup = TAVOR_RSRC_CLEANUP_LEVEL0;
436
437         /* Build kmem cache name from Tavor instance */
438         rsrc_name = (char *)kmem_zalloc(TAVOR_RSRC_NAME_MAXLEN, KM_SLEEP);
439         TAVOR_RSRC_NAME(rsrc_name, TAVOR_RSRC_CACHE);
440
441         /*
442          * Create the kmem_cache for "tavor_rsrc_t" structures
443          * (kmem_cache_create will SLEEP until successful)
444          */
445         state->ts_rsrc_cache = kmem_cache_create(rsrc_name,
446             sizeof (tavor_rsrc_t), 0, NULL, NULL, NULL, NULL, NULL, 0);
447
448         /*
449          * Allocate an array of tavor_rsrc_pool_info_t's (used in all
450          * subsequent resource allocations)
451          */
452         state->ts_rsrc_hdl = kmem_zalloc(TAVOR_NUM_RESOURCES *
453             sizeof (tavor_rsrc_pool_info_t), KM_SLEEP);
454
455         cfgprof = state->ts_cfg_profile;
456
457         /*
```



```
458          * Initialize the resource pool for "Out" mailboxes.  Notice that
459          * the number of "Out" mailboxes, their size, and their location
460          * (DDR or system memory) is configurable.  By default, however,
461          * all "Out" mailboxes are located in system memory only (because
462          * they are primarily read from and never written to)
463          */
464         num  =  ((uint64_t)1 << cfgprof->cp_log_num_outmbox);
465         size =  ((uint64_t)1 << cfgprof->cp_log_outmbox_size);
466         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_OUT_MBOX];
467         rsrc_pool->rsrc_type      = TAVOR_OUT_MBOX;
468         rsrc_pool->rsrc_loc       = TAVOR_IN_SYSMEM;
469         rsrc_pool->rsrc_pool_size = (size * num);
470         rsrc_pool->rsrc_shift     = cfgprof->cp_log_outmbox_size;
471         rsrc_pool->rsrc_quantum   = size;
472         rsrc_pool->rsrc_align     = TAVOR_MBOX_ALIGN;
473         rsrc_pool->rsrc_state     = state;
474         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_OUTMBOX_VMEM);
475         mbox_info.mbi_num         = num;
476         mbox_info.mbi_size        = size;
477         mbox_info.mbi_rsrcpool    = rsrc_pool;
478         mbox_info.mbi_rsrcname    = rsrc_name;
479         status = tavor_rsrc_mbox_init(state, &mbox_info);
480         if (status != DDI_SUCCESS) {
481                 tavor_rsrc_fini(state, cleanup);
482                 /* Set "status" and "errormsg" and goto failure */
483                 TAVOR_TNF_FAIL(DDI_FAILURE, "out mailboxes");
484                 goto rsrcinitp1_fail;
485         }
486         cleanup = TAVOR_RSRC_CLEANUP_LEVEL1;
487
488         /*
489          * Initialize the Tavor "Out" mailbox list.  This step actually uses
490          * the tavor_rsrc_alloc() for TAVOR_OUT_MBOX to preallocate the
491          * "Out" mailboxes, bind them for DMA access, and arrange them into
492          * an easily accessed fast-allocation mechanism (see tavor_cmd.c
493          * for more details)
494          */
495         status = tavor_outmbox_list_init(state);
496         if (status != DDI_SUCCESS) {
497                 tavor_rsrc_fini(state, cleanup);
498                 /* Set "status" and "errormsg" and goto failure */
499                 TAVOR_TNF_FAIL(DDI_FAILURE, "out mailbox list");
500                 goto rsrcinitp1_fail;
501         }
502         cleanup = TAVOR_RSRC_CLEANUP_LEVEL2;
503
504         /*
505          * Initialize the resource pool for interrupt "Out" mailboxes.  Notice
506          * that the number of interrupt "Out" mailboxes, their size, and their
507          * location (DDR or system memory) is configurable.  By default,
508          * however, all interrupt "Out" mailboxes are located in system memory
509          * only (because they are primarily read from and never written to)
510          */
511         num  =  ((uint64_t)1 << cfgprof->cp_log_num_intr_outmbox);
512         size =  ((uint64_t)1 << cfgprof->cp_log_outmbox_size);
513         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_INTR_OUT_MBOX];
514         rsrc_pool->rsrc_type      = TAVOR_INTR_OUT_MBOX;
515         rsrc_pool->rsrc_loc       = TAVOR_IN_SYSMEM;
516         rsrc_pool->rsrc_pool_size = (size * num);
517         rsrc_pool->rsrc_shift     = cfgprof->cp_log_outmbox_size;
518         rsrc_pool->rsrc_quantum   = size;
519         rsrc_pool->rsrc_align     = TAVOR_MBOX_ALIGN;
520         rsrc_pool->rsrc_state     = state;
521         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_INTR_OUTMBOX_VMEM);
522         mbox_info.mbi_num         = num;
523         mbox_info.mbi_size        = size;
```

```
524         mbox_info.mbi_rsrcpool    = rsrc_pool;
525         mbox_info.mbi_rsrcname    = rsrc_name;
526         status = tavor_rsrc_mbox_init(state, &mbox_info);
527         if (status != DDI_SUCCESS) {
528                 tavor_rsrc_fini(state, cleanup);
529                 /* Set "status" and "errormsg" and goto failure */
530                 TAVOR_TNF_FAIL(DDI_FAILURE, "out intr mailboxes");
531                 goto rsrcinitp1_fail;
532         }
533         cleanup = TAVOR_RSRC_CLEANUP_LEVEL3;

535         /*
536          * Initialize the Tavor "Out" mailbox list.  This step actually uses
537          * the tavor_rsrc_alloc() for TAVOR_OUT_MBOX to preallocate the
538          * "Out" mailboxes, bind them for DMA access, and arrange them into
539          * an easily accessed fast-allocation mechanism (see tavor_cmd.c
540          * for more details)
541          */
542         status = tavor_intr_outmbox_list_init(state);
543         if (status != DDI_SUCCESS) {
544                 tavor_rsrc_fini(state, cleanup);
545                 /* Set "status" and "errormsg" and goto failure */
546                 TAVOR_TNF_FAIL(DDI_FAILURE, "out intr mailbox list");
547                 goto rsrcinitp1_fail;
548         }
549         cleanup = TAVOR_RSRC_CLEANUP_PHASE1_COMPLETE;

551         kmem_free(rsrc_name, TAVOR_RSRC_NAME_MAXLEN);
552         TAVOR_TNF_EXIT(tavor_rsrc_init_phase1);
553         return (DDI_SUCCESS);

555 rsrcinitp1_fail:
556         kmem_free(rsrc_name, TAVOR_RSRC_NAME_MAXLEN);
557         TNF_PROBE_1(tavor_rsrc_init_phase1_fail, TAVOR_TNF_ERROR, "",
558             tnf_string, msg, errormsg);
559         TAVOR_TNF_EXIT(tavor_rsrc_init_phase1);
560         return (status);
561 }


564 /*
565  * tavor_rsrc_init_phase2()
566  *    Context: Only called from attach() path context
567  */
568 int
569 tavor_rsrc_init_phase2(tavor_state_t *state)
570 {
571         tavor_rsrc_sw_hdl_info_t        hdl_info;
572         tavor_rsrc_hw_entry_info_t      entry_info;
573         tavor_rsrc_mbox_info_t          mbox_info;
574         tavor_rsrc_pool_info_t          *rsrc_pool;
575         tavor_rsrc_cleanup_level_t      cleanup;
576         tavor_cfg_profile_t             *cfgprof;
577         uint64_t                        num, max, size, num_prealloc;
578         uint64_t                        ddr_size, fw_size;
579         uint_t                          mcg_size, mcg_size_shift;
580         uint_t                          uarscr_size, mttsegment_sz;
581         int                             status;
582         char                            *errormsg, *rsrc_name;

584         TAVOR_TNF_ENTER(tavor_rsrc_init_phase2);

586         ASSERT(state != NULL);

588         /* Phase 2 initialization begins where Phase 1 left off */
589         cleanup = TAVOR_RSRC_CLEANUP_PHASE1_COMPLETE;
```



```
591         /*
592          * Calculate the extent of the DDR size and portion of which that
593          * is already reserved for Tavor firmware.  (Note: this information
594          * is available because the QUERY_DDR and QUERY_FW commands have
595          * been posted to Tavor firmware prior to calling this routine)
596          */
597         ddr_size = state->ts_ddr.ddr_endaddr - state->ts_ddr.ddr_baseaddr + 1;
598         fw_size  = state->ts_fw.fw_endaddr - state->ts_fw.fw_baseaddr + 1;

600         /* Build the DDR vmem arena name from Tavor instance */
601         rsrc_name = (char *)kmem_zalloc(TAVOR_RSRC_NAME_MAXLEN, KM_SLEEP);
602         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_VMEM);

604         /*
605          * Do a vmem_create for the entire DDR range (not including the
606          * portion consumed by Tavor firmware).  This creates the vmem arena
607          * from which all other DDR objects (specifically, tables of HW
608          * entries) will be allocated.
609          */
610         state->ts_ddrvmem = vmem_create(rsrc_name,
611             (void *)(uintptr_t)state->ts_ddr.ddr_baseaddr, (ddr_size - fw_size),
612             sizeof (uint64_t), NULL, NULL, NULL, 0, VM_SLEEP);
613         if (state->ts_ddrvmem == NULL) {
614                 tavor_rsrc_fini(state, cleanup);
615                 /* Set "status" and "errormsg" and goto failure */
616                 TAVOR_TNF_FAIL(DDI_FAILURE, "DDR vmem");
617                 goto rsrcinitp2_fail;
618         }
619         cleanup = TAVOR_RSRC_CLEANUP_LEVEL5;

621         /*
622          * Initialize the resource pools for all objects that exist in
623          * Tavor DDR memory.  This includes ("In") mailboxes, context tables
624          * (QPC, CQC, EQC, etc...), and other miscellaneous HW objects.
625          */
626         cfgprof = state->ts_cfg_profile;

628         /*
629          * Initialize the resource pool for the MPT table entries.  Notice
630          * that the number of MPTs is configurable.  The configured value must
631          * be less that the maximum value (obtained from the QUERY_DEV_LIM
632          * command) or the initialization will fail.  Note also that a certain
633          * number of MPTs must be set aside for Tavor firmware use.
634          */
635         num = ((uint64_t)1 << cfgprof->cp_log_num_mpt);
636         max = ((uint64_t)1 << state->ts_devlim.log_max_mpt);
637         num_prealloc = ((uint64_t)1 << state->ts_devlim.log_rsvd_mpt);
638         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_MPT];
639         rsrc_pool->rsrc_type      = TAVOR_MPT;
640         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
641         rsrc_pool->rsrc_pool_size = (TAVOR_MPT_SIZE * num);
642         rsrc_pool->rsrc_shift     = TAVOR_MPT_SIZE_SHIFT;
643         rsrc_pool->rsrc_quantum   = TAVOR_MPT_SIZE;
644         rsrc_pool->rsrc_align     = (TAVOR_MPT_SIZE * num);
645         rsrc_pool->rsrc_state     = state;
646         rsrc_pool->rsrc_start     = NULL;
647         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_MPT_VMEM);
648         entry_info.hwi_num        = num;
649         entry_info.hwi_max        = max;
650         entry_info.hwi_prealloc   = num_prealloc;
651         entry_info.hwi_rsrcpool   = rsrc_pool;
652         entry_info.hwi_rsrcname   = rsrc_name;
653         status = tavor_rsrc_hw_entries_init(state, &entry_info);
654         if (status != DDI_SUCCESS) {
655                 tavor_rsrc_fini(state, cleanup);
```

```
656                 /* Set "status" and "errormsg" and goto failure */
657                 TAVOR_TNF_FAIL(DDI_FAILURE, "MPT table");
658                 goto rsrcinitp2_fail;
659         }
660         cleanup = TAVOR_RSRC_CLEANUP_LEVEL6;

662         /*
663          * Initialize the resource pool for the MTT table entries.  Notice
664          * that the number of MTTs is configurable.  The configured value must
665          * be less that the maximum value (obtained from the QUERY_DEV_LIM
666          * command) or the initialization will fail.  Note also that a certain
667          * number of MTT segments must be set aside for Tavor firmware use.
668          */
669         mttsegment_sz = (TAVOR_MTTSEG_SIZE << TAVOR_MTT_SIZE_SHIFT);
670         num = ((uint64_t)1 << cfgprof->cp_log_num_mttseg);
671         max = ((uint64_t)1 << state->ts_devlim.log_max_mttseg);
672         num_prealloc = ((uint64_t)1 << state->ts_devlim.log_rsvd_mttseg);
673         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_MTT];
674         rsrc_pool->rsrc_type      = TAVOR_MTT;
675         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
676         rsrc_pool->rsrc_pool_size = (TAVOR_MTT_SIZE * num);
677         rsrc_pool->rsrc_shift     = TAVOR_MTT_SIZE_SHIFT;
678         rsrc_pool->rsrc_quantum   = mttsegment_sz;
679         rsrc_pool->rsrc_align     = (TAVOR_MTT_SIZE * num);
680         rsrc_pool->rsrc_state     = state;
681         rsrc_pool->rsrc_start     = NULL;
682         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_MTT_VMEM);
683         entry_info.hwi_num        = num;
684         entry_info.hwi_max        = max;
685         entry_info.hwi_prealloc   = num_prealloc;
686         entry_info.hwi_rsrcpool   = rsrc_pool;
687         entry_info.hwi_rsrcname   = rsrc_name;
688         status = tavor_rsrc_hw_entries_init(state, &entry_info);
689         if (status != DDI_SUCCESS) {
690                 tavor_rsrc_fini(state, cleanup);
691                 /* Set "status" and "errormsg" and goto failure */
692                 TAVOR_TNF_FAIL(DDI_FAILURE, "MTT table");
693                 goto rsrcinitp2_fail;
694         }
695         cleanup = TAVOR_RSRC_CLEANUP_LEVEL7;

697         /*
698          * Initialize the resource pool for the QPC table entries.  Notice
699          * that the number of QPs is configurable.  The configured value must
700          * be less that the maximum value (obtained from the QUERY_DEV_LIM
701          * command) or the initialization will fail.  Note also that a certain
702          * number of QP contexts must be set aside for Tavor firmware use.
703          */
704         num = ((uint64_t)1 << cfgprof->cp_log_num_qp);
705         max = ((uint64_t)1 << state->ts_devlim.log_max_qp);
706         num_prealloc = ((uint64_t)1 << state->ts_devlim.log_rsvd_qp);
707         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_QPC];
708         rsrc_pool->rsrc_type      = TAVOR_QPC;
709         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
710         rsrc_pool->rsrc_pool_size = (TAVOR_QPC_SIZE * num);
711         rsrc_pool->rsrc_shift     = TAVOR_QPC_SIZE_SHIFT;
712         rsrc_pool->rsrc_quantum   = TAVOR_QPC_SIZE;
713         rsrc_pool->rsrc_align     = (TAVOR_QPC_SIZE * num);
714         rsrc_pool->rsrc_state     = state;
715         rsrc_pool->rsrc_start     = NULL;
716         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_QPC_VMEM);
717         entry_info.hwi_num        = num;
718         entry_info.hwi_max        = max;
719         entry_info.hwi_prealloc   = num_prealloc;
720         entry_info.hwi_rsrcpool   = rsrc_pool;
721         entry_info.hwi_rsrcname   = rsrc_name;
```



```
722         status = tavor_rsrc_hw_entries_init(state, &entry_info);
723         if (status != DDI_SUCCESS) {
724                 tavor_rsrc_fini(state, cleanup);
725                 /* Set "status" and "errormsg" and goto failure */
726                 TAVOR_TNF_FAIL(DDI_FAILURE, "QPC table");
727                 goto rsrcinitp2_fail;
728         }
729         cleanup = TAVOR_RSRC_CLEANUP_LEVEL8;

731         /*
732          * Initialize the resource pool for the RDB table entries.  Notice
733          * that the number of RDBs is configurable.  The configured value must
734          * be less that the maximum value (obtained from the QUERY_DEV_LIM
735          * command) or the initialization will fail.
736          */
737         num = ((uint64_t)1 << cfgprof->cp_log_num_rdb);
738         max = ((uint64_t)1 << state->ts_devlim.log_max_ra_glob);
739         num_prealloc = 0;
740         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_RDB];
741         rsrc_pool->rsrc_type      = TAVOR_RDB;
742         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
743         rsrc_pool->rsrc_pool_size = (TAVOR_RDB_SIZE * num);
744         rsrc_pool->rsrc_shift     = TAVOR_RDB_SIZE_SHIFT;
745         rsrc_pool->rsrc_quantum   = TAVOR_RDB_SIZE;
746         rsrc_pool->rsrc_align     = (TAVOR_RDB_SIZE * num);
747         rsrc_pool->rsrc_state     = state;
748         rsrc_pool->rsrc_start     = NULL;
749         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_RDB_VMEM);
750         entry_info.hwi_num        = num;
751         entry_info.hwi_max        = max;
752         entry_info.hwi_prealloc   = num_prealloc;
753         entry_info.hwi_rsrcpool   = rsrc_pool;
754         entry_info.hwi_rsrcname   = rsrc_name;
755         status = tavor_rsrc_hw_entries_init(state, &entry_info);
756         if (status != DDI_SUCCESS) {
757                 tavor_rsrc_fini(state, cleanup);
758                 /* Set "status" and "errormsg" and goto failure */
759                 TAVOR_TNF_FAIL(DDI_FAILURE, "RDB table");
760                 goto rsrcinitp2_fail;
761         }
762         cleanup = TAVOR_RSRC_CLEANUP_LEVEL9;

764         /*
765          * Initialize the resource pool for the CQC table entries.  Notice
766          * that the number of CQs is configurable.  The configured value must
767          * be less that the maximum value (obtained from the QUERY_DEV_LIM
768          * command) or the initialization will fail.  Note also that a certain
769          * number of CQ contexts must be set aside for Tavor firmware use.
770          */
771         num = ((uint64_t)1 << cfgprof->cp_log_num_cq);
772         max = ((uint64_t)1 << state->ts_devlim.log_max_cq);
773         num_prealloc = ((uint64_t)1 << state->ts_devlim.log_rsvd_cq);
774         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_CQC];
775         rsrc_pool->rsrc_type      = TAVOR_CQC;
776         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
777         rsrc_pool->rsrc_pool_size = (TAVOR_CQC_SIZE * num);
778         rsrc_pool->rsrc_shift     = TAVOR_CQC_SIZE_SHIFT;
779         rsrc_pool->rsrc_quantum   = TAVOR_CQC_SIZE;
780         rsrc_pool->rsrc_align     = (TAVOR_CQC_SIZE * num);
781         rsrc_pool->rsrc_state     = state;
782         rsrc_pool->rsrc_start     = NULL;
783         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_CQC_VMEM);
784         entry_info.hwi_num        = num;
785         entry_info.hwi_max        = max;
786         entry_info.hwi_prealloc   = num_prealloc;
787         entry_info.hwi_rsrcpool   = rsrc_pool;
```

```
788         entry_info.hwi_rsrcname   = rsrc_name;
789         status = tavor_rsrc_hw_entries_init(state, &entry_info);
790         if (status != DDI_SUCCESS) {
791                 tavor_rsrc_fini(state, cleanup);
792                 /* Set "status" and "errormsg" and goto failure */
793                 TAVOR_TNF_FAIL(DDI_FAILURE, "CQC table");
794                 goto rsrcinitp2_fail;
795         }
796         cleanup = TAVOR_RSRC_CLEANUP_LEVEL10;

798         /*
799          * Initialize the resource pool for the Extended QPC table entries.
800          * Notice that the number of EQPCs must be the same as the number
801          * of QP contexts.  So the initialization is constructed in a
802          * similar way as above (for TAVOR_QPC).  One notable difference
803          * here, however, is that by setting the rsrc_quantum field to
804          * zero (indicating a zero-sized object) we indicate that the
805          * object is not allocatable.  The EQPC table is, in fact, managed
806          * internally by the hardware and it is, therefore, unnecessary to
807          * initialize an additional vmem_arena for this type of object.
808          */
809         num = ((uint64_t)1 << cfgprof->cp_log_num_qp);
810         max = ((uint64_t)1 << state->ts_devlim.log_max_qp);
811         num_prealloc = 0;
812         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_EQPC];
813         rsrc_pool->rsrc_type      = TAVOR_EQPC;
814         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
815         rsrc_pool->rsrc_pool_size = (TAVOR_EQPC_SIZE * num);
816         rsrc_pool->rsrc_shift     = 0;
817         rsrc_pool->rsrc_quantum   = 0;
818         rsrc_pool->rsrc_align     = TAVOR_EQPC_SIZE;
819         rsrc_pool->rsrc_state     = state;
820         rsrc_pool->rsrc_start     = NULL;
821         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_EQPC_VMEM);
822         entry_info.hwi_num        = num;
823         entry_info.hwi_max        = max;
824         entry_info.hwi_prealloc   = num_prealloc;
825         entry_info.hwi_rsrcpool   = rsrc_pool;
826         entry_info.hwi_rsrcname   = rsrc_name;
827         status = tavor_rsrc_hw_entries_init(state, &entry_info);
828         if (status != DDI_SUCCESS) {
829                 tavor_rsrc_fini(state, cleanup);
830                 /* Set "status" and "errormsg" and goto failure */
831                 TAVOR_TNF_FAIL(DDI_FAILURE, "Extended QPC table");
832                 goto rsrcinitp2_fail;
833         }
834         cleanup = TAVOR_RSRC_CLEANUP_LEVEL11;

836         /*
837          * Initialize the resource pool for the UD address vector table
838          * entries.  Notice that the number of UDAVs is configurable.  The
839          * configured value must be less that the maximum value (obtained
840          * from the QUERY_DEV_LIM command) or the initialization will fail.
841          */
842         num = ((uint64_t)1 << cfgprof->cp_log_num_ah);
843         max = ((uint64_t)1 << state->ts_devlim.log_max_av);
844         num_prealloc = 0;
845         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_UDAV];
846         rsrc_pool->rsrc_type      = TAVOR_UDAV;
847         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
848         rsrc_pool->rsrc_pool_size = (TAVOR_UDAV_SIZE * num);
849         rsrc_pool->rsrc_shift     = TAVOR_UDAV_SIZE_SHIFT;
850         rsrc_pool->rsrc_quantum   = TAVOR_UDAV_SIZE;
851         rsrc_pool->rsrc_align     = TAVOR_UDAV_SIZE;
852         rsrc_pool->rsrc_state     = state;
853         rsrc_pool->rsrc_start     = NULL;
```



```
854         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_UDAV_VMEM);
855         entry_info.hwi_num        = num;
856         entry_info.hwi_max        = max;
857         entry_info.hwi_prealloc   = num_prealloc;
858         entry_info.hwi_rsrcpool   = rsrc_pool;
859         entry_info.hwi_rsrcname   = rsrc_name;
860         status = tavor_rsrc_hw_entries_init(state, &entry_info);
861         if (status != DDI_SUCCESS) {
862                 tavor_rsrc_fini(state, cleanup);
863                 /* Set "status" and "errormsg" and goto failure */
864                 TAVOR_TNF_FAIL(DDI_FAILURE, "UDAV table");
865                 goto rsrcinitp2_fail;
866         }
867         cleanup = TAVOR_RSRC_CLEANUP_LEVEL12;

869         /*
870          * Initialize the resource pool for the UAR scratch table entries.
871          * Notice that the number of UARSCRs is configurable.  The configured
872          * value must be less that the maximum value (obtained from the
873          * QUERY_DEV_LIM command) or the initialization will fail.
874          * Like the EQPCs above, UARSCR objects are not allocatable.  The
875          * UARSCR table is also managed internally by the hardware and it
876          * is, therefore, unnecessary to initialize an additional vmem_arena
877          * for this type of object.  We indicate this by setting the
878          * rsrc_quantum field to zero (indicating a zero-sized object).
879          */
880         uarscr_size = state->ts_devlim.uarscr_entry_sz;
881         num = ((uint64_t)1 << cfgprof->cp_log_num_uar);
882         max = ((uint64_t)1 << (state->ts_devlim.log_max_uar_sz + 20 -
883             PAGESHIFT));
884         num_prealloc = 0;
885         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_UAR_SCR];
886         rsrc_pool->rsrc_type      = TAVOR_UAR_SCR;
887         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
888         rsrc_pool->rsrc_pool_size = (uarscr_size * num);
889         rsrc_pool->rsrc_shift     = 0;
890         rsrc_pool->rsrc_quantum   = 0;
891         rsrc_pool->rsrc_align     = uarscr_size;
892         rsrc_pool->rsrc_state     = state;
893         rsrc_pool->rsrc_start     = NULL;
894         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_UARSCR_VMEM);
895         entry_info.hwi_num        = num;
896         entry_info.hwi_max        = max;
897         entry_info.hwi_prealloc   = num_prealloc;
898         entry_info.hwi_rsrcpool   = rsrc_pool;
899         entry_info.hwi_rsrcname   = rsrc_name;
900         status = tavor_rsrc_hw_entries_init(state, &entry_info);
901         if (status != DDI_SUCCESS) {
902                 tavor_rsrc_fini(state, cleanup);
903                 /* Set "status" and "errormsg" and goto failure */
904                 TAVOR_TNF_FAIL(DDI_FAILURE, "UAR scratch table");
905                 goto rsrcinitp2_fail;
906         }
907         cleanup = TAVOR_RSRC_CLEANUP_LEVEL13;

909         /*
910          * Initialize the resource pool for the SRQC table entries.  Notice
911          * that the number of SRQs is configurable.  The configured value must
912          * be less that the maximum value (obtained from the QUERY_DEV_LIM
913          * command) or the initialization will fail.  Note also that a certain
914          * number of SRQ contexts must be set aside for Tavor firmware use.
915          *
916          * Note: We only allocate these resources if SRQ is enabled in the
917          * config profile; see below.
918          */
919         num = ((uint64_t)1 << cfgprof->cp_log_num_srq);
```

```
920         max = ((uint64_t)1 << state->ts_devlim.log_max_srq);
921         num_prealloc = ((uint64_t)1 << state->ts_devlim.log_rsvd_srq);

923         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_SRQC];
924         rsrc_pool->rsrc_type      = TAVOR_SRQC;
925         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
926         rsrc_pool->rsrc_pool_size = (TAVOR_SRQC_SIZE * num);
927         rsrc_pool->rsrc_shift     = TAVOR_SRQC_SIZE_SHIFT;
928         rsrc_pool->rsrc_quantum   = TAVOR_SRQC_SIZE;
929         rsrc_pool->rsrc_align     = (TAVOR_SRQC_SIZE * num);
930         rsrc_pool->rsrc_state     = state;
931         rsrc_pool->rsrc_start     = NULL;
932         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_SRQC_VMEM);
933         entry_info.hwi_num        = num;
934         entry_info.hwi_max        = max;
935         entry_info.hwi_prealloc   = num_prealloc;
936         entry_info.hwi_rsrcpool   = rsrc_pool;
937         entry_info.hwi_rsrcname   = rsrc_name;

939         /*
940          * SRQ support is configurable.  Only if SRQ is enabled (the default)
941          * do we actually try to configure these resources.  Otherwise, we
942          * simply set the cleanup level and continue on to the next resource
943          */
944         if (state->ts_cfg_profile->cp_srq_enable != 0) {
945                 status = tavor_rsrc_hw_entries_init(state, &entry_info);
946                 if (status != DDI_SUCCESS) {
947                         tavor_rsrc_fini(state, cleanup);
948                         /* Set "status" and "errormsg" and goto failure */
949                         TAVOR_TNF_FAIL(DDI_FAILURE, "SRQC table");
950                         goto rsrcinitp2_fail;
951                 }
952         }
953         cleanup = TAVOR_RSRC_CLEANUP_LEVEL14;

955         /*
956          * Initialize the resource pool for "In" mailboxes.  Notice that
957          * the number of "In" mailboxes, their size, and their location
958          * (DDR or system memory) is configurable.  By default, however,
959          * all "In" mailboxes are located in system memory only (because
960          * they are primarily written to and rarely read from)
961          */
962         num  =  ((uint64_t)1 << cfgprof->cp_log_num_inmbox);
963         size =  ((uint64_t)1 << cfgprof->cp_log_inmbox_size);
964         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_IN_MBOX];
965         rsrc_pool->rsrc_type      = TAVOR_IN_MBOX;
966         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
967         rsrc_pool->rsrc_pool_size = (size * num);
968         rsrc_pool->rsrc_shift     = cfgprof->cp_log_inmbox_size;
969         rsrc_pool->rsrc_quantum   = size;
970         rsrc_pool->rsrc_align     = TAVOR_MBOX_ALIGN;
971         rsrc_pool->rsrc_state     = state;
972         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_INMBOX_VMEM);
973         mbox_info.mbi_num         = num;
974         mbox_info.mbi_size        = size;
975         mbox_info.mbi_rsrcpool    = rsrc_pool;
976         mbox_info.mbi_rsrcname    = rsrc_name;
977         status = tavor_rsrc_mbox_init(state, &mbox_info);
978         if (status != DDI_SUCCESS) {
979                 tavor_rsrc_fini(state, cleanup);
980                 /* Set "status" and "errormsg" and goto failure */
981                 TAVOR_TNF_FAIL(DDI_FAILURE, "in mailboxes");
982                 goto rsrcinitp2_fail;
983         }
984         cleanup = TAVOR_RSRC_CLEANUP_LEVEL15;
```



```
986         /*
987          * Initialize the Tavor "In" mailbox list.  This step actually uses
988          * the tavor_rsrc_alloc() for TAVOR_IN_MBOX to preallocate the
989          * "In" mailboxes, bind them for DMA access, and arrange them into
990          * an easily accessed fast-allocation mechanism (see tavor_cmd.c
991          * for more details)
992          */
993         status = tavor_inmbox_list_init(state);
994         if (status != DDI_SUCCESS) {
995                 tavor_rsrc_fini(state, cleanup);
996                 /* Set "status" and "errormsg" and goto failure */
997                 TAVOR_TNF_FAIL(DDI_FAILURE, "in mailbox list");
998                 goto rsrcinitp2_fail;
999         }
1000        cleanup = TAVOR_RSRC_CLEANUP_LEVEL16;

1002        /*
1003         * Initialize the resource pool for interrupt "In" mailboxes.  Notice
1004         * that the number of interrupt "In" mailboxes, their size, and their
1005         * location (DDR or system memory) is configurable.  By default,
1006         * however, all interrupt "In" mailboxes are located in system memory
1007         * only (because they are primarily written to and rarely read from)
1008         */
1009        num  =  ((uint64_t)1 << cfgprof->cp_log_num_intr_inmbox);
1010        size =  ((uint64_t)1 << cfgprof->cp_log_inmbox_size);
1011        rsrc_pool = &state->ts_rsrc_hdl[TAVOR_INTR_IN_MBOX];
1012        rsrc_pool->rsrc_type      = TAVOR_INTR_IN_MBOX;
1013        rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
1014        rsrc_pool->rsrc_pool_size = (size * num);
1015        rsrc_pool->rsrc_shift     = cfgprof->cp_log_inmbox_size;
1016        rsrc_pool->rsrc_quantum   = size;
1017        rsrc_pool->rsrc_align     = TAVOR_MBOX_ALIGN;
1018        rsrc_pool->rsrc_state     = state;
1019        TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_INTR_INMBOX_VMEM);
1020        mbox_info.mbi_num         = num;
1021        mbox_info.mbi_size        = size;
1022        mbox_info.mbi_rsrcpool    = rsrc_pool;
1023        mbox_info.mbi_rsrcname    = rsrc_name;
1024        status = tavor_rsrc_mbox_init(state, &mbox_info);
1025        if (status != DDI_SUCCESS) {
1026                tavor_rsrc_fini(state, cleanup);
1027                /* Set "status" and "errormsg" and goto failure */
1028                TAVOR_TNF_FAIL(DDI_FAILURE, "in intr mailboxes");
1029                goto rsrcinitp2_fail;
1030        }
1031        cleanup = TAVOR_RSRC_CLEANUP_LEVEL17;

1033        /*
1034         * Initialize the Tavor interrupt "In" mailbox list.  This step
1035         * actually uses the tavor_rsrc_alloc() for TAVOR_IN_MBOX to
1036         * preallocate the interrupt "In" mailboxes, bind them for DMA access,
1037         * and arrange them into an easily accessed fast-allocation mechanism
1038         * (see tavor_cmd.c for more details)
1039         */
1040        status = tavor_intr_inmbox_list_init(state);
1041        if (status != DDI_SUCCESS) {
1042                tavor_rsrc_fini(state, cleanup);
1043                /* Set "status" and "errormsg" and goto failure */
1044                TAVOR_TNF_FAIL(DDI_FAILURE, "in intr mailbox list");
1045                goto rsrcinitp2_fail;
1046        }
1047        cleanup = TAVOR_RSRC_CLEANUP_LEVEL18;

1049        /*
1050         * Initialize the Tavor command handling interfaces.  This step
1051         * sets up the outstanding command tracking mechanism for easy access
```

```
1052          * and fast allocation (see tavor_cmd.c for more details).
1053          */
1054         status = tavor_outstanding_cmdlist_init(state);
1055         if (status != DDI_SUCCESS) {
1056                 tavor_rsrc_fini(state, cleanup);
1057                 /* Set "status" and "errormsg" and goto failure */
1058                 TAVOR_TNF_FAIL(DDI_FAILURE, "outstanding cmd list");
1059                 goto rsrcinitp2_fail;
1060         }
1061         cleanup = TAVOR_RSRC_CLEANUP_LEVEL19;
1062
1063         /*
1064          * Calculate (and validate) the size of Multicast Group (MCG) entries
1065          */
1066         status = tavor_rsrc_mcg_entry_get_size(state, &mcg_size_shift);
1067         if (status != DDI_SUCCESS) {
1068                 tavor_rsrc_fini(state, cleanup);
1069                 /* Set "status" and "errormsg" and goto failure */
1070                 TAVOR_TNF_FAIL(DDI_FAILURE, "failed get MCG size");
1071                 goto rsrcinitp2_fail;
1072         }
1073         mcg_size = TAVOR_MCGMEM_SZ(state);
1074
1075         /*
1076          * Initialize the resource pool for the MCG table entries.  Notice
1077          * that the number of MCGs is configurable.  The configured value must
1078          * be less that the maximum value (obtained from the QUERY_DEV_LIM
1079          * command) or the initialization will fail.  Note also that a certain
1080          * number of MCGs must be set aside for Tavor firmware use (they
1081          * correspond to the number of MCGs used by the internal hash
1082          * function.
1083          */
1084         num = ((uint64_t)1 << cfgprof->cp_log_num_mcg);
1085         max = ((uint64_t)1 << state->ts_devlim.log_max_mcg);
1086         num_prealloc = ((uint64_t)1 << cfgprof->cp_log_num_mcg_hash);
1087         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_MCG];
1088         rsrc_pool->rsrc_type      = TAVOR_MCG;
1089         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
1090         rsrc_pool->rsrc_pool_size = (mcg_size * num);
1091         rsrc_pool->rsrc_shift     = mcg_size_shift;
1092         rsrc_pool->rsrc_quantum   = mcg_size;
1093         rsrc_pool->rsrc_align     = mcg_size;
1094         rsrc_pool->rsrc_state     = state;
1095         rsrc_pool->rsrc_start     = NULL;
1096         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_MCG_VMEM);
1097         entry_info.hwi_num        = num;
1098         entry_info.hwi_max        = max;
1099         entry_info.hwi_prealloc   = num_prealloc;
1100         entry_info.hwi_rsrcpool   = rsrc_pool;
1101         entry_info.hwi_rsrcname   = rsrc_name;
1102         status = tavor_rsrc_hw_entries_init(state, &entry_info);
1103         if (status != DDI_SUCCESS) {
1104                 tavor_rsrc_fini(state, cleanup);
1105                 /* Set "status" and "errormsg" and goto failure */
1106                 TAVOR_TNF_FAIL(DDI_FAILURE, "MCG table");
1107                 goto rsrcinitp2_fail;
1108         }
1109         cleanup = TAVOR_RSRC_CLEANUP_LEVEL20;
1110
1111         /*
1112          * Initialize the resource pool for the EQC table entries.  Notice
1113          * that the number of EQs is hardcoded.  The hardcoded value should
1114          * be less that the maximum value (obtained from the QUERY_DEV_LIM
1115          * command) or the initialization will fail.
1116          */
1117         num = TAVOR_NUM_EQ;
```



```
1118         max = ((uint64_t)1 << state->ts_devlim.log_max_eq);
1119         num_prealloc = 0;
1120         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_EQC];
1121         rsrc_pool->rsrc_type      = TAVOR_EQC;
1122         rsrc_pool->rsrc_loc       = TAVOR_IN_DDR;
1123         rsrc_pool->rsrc_pool_size = (TAVOR_EQC_SIZE * num);
1124         rsrc_pool->rsrc_shift     = TAVOR_EQC_SIZE_SHIFT;
1125         rsrc_pool->rsrc_quantum   = TAVOR_EQC_SIZE;
1126         rsrc_pool->rsrc_align     = (TAVOR_EQC_SIZE * num);
1127         rsrc_pool->rsrc_state     = state;
1128         rsrc_pool->rsrc_start     = NULL;
1129         TAVOR_RSRC_NAME(rsrc_name, TAVOR_DDR_EQC_VMEM);
1130         entry_info.hwi_num        = num;
1131         entry_info.hwi_max        = max;
1132         entry_info.hwi_prealloc   = num_prealloc;
1133         entry_info.hwi_rsrcpool   = rsrc_pool;
1134         entry_info.hwi_rsrcname   = rsrc_name;
1135         status = tavor_rsrc_hw_entries_init(state, &entry_info);
1136         if (status != DDI_SUCCESS) {
1137                 tavor_rsrc_fini(state, cleanup);
1138                 /* Set "status" and "errormsg" and goto failure */
1139                 TAVOR_TNF_FAIL(DDI_FAILURE, "EQC table");
1140                 goto rsrcinitp2_fail;
1141         }
1142         cleanup = TAVOR_RSRC_CLEANUP_LEVEL21;
1143
1144         /*
1145          * Initialize the resource pools for all objects that exist in
1146          * system memory.  This includes PD handles, MR handle, EQ handles,
1147          * QP handles, etc.  These objects are almost entirely managed using
1148          * kmem_cache routines.  (See comment above for more detail)
1149          */
1150
1151         /*
1152          * Initialize the resource pool for the PD handles.  Notice
1153          * that the number of PDHDLs is configurable.  The configured value
1154          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
1155          * command) or the initialization will fail.  Note also that the PD
1156          * handle has constructor and destructor methods associated with it.
1157          */
1158         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_PDHDL];
1159         rsrc_pool->rsrc_type      = TAVOR_PDHDL;
1160         rsrc_pool->rsrc_loc       = TAVOR_IN_SYSMEM;
1161         rsrc_pool->rsrc_quantum   = sizeof (struct tavor_sw_pd_s);
1162         rsrc_pool->rsrc_state     = state;
1163         TAVOR_RSRC_NAME(rsrc_name, TAVOR_PDHDL_CACHE);
1164         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_pd);
1165         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_pd);
1166         hdl_info.swi_rsrcpool     = rsrc_pool;
1167         hdl_info.swi_constructor = tavor_rsrc_pdhdl_constructor;
1168         hdl_info.swi_destructor   = tavor_rsrc_pdhdl_destructor;
1169         hdl_info.swi_rsrcname     = rsrc_name;
1170         hdl_info.swi_flags        = TAVOR_SWHDL_KMEMCACHE_INIT;
1171         status = tavor_rsrc_pd_handles_init(state, &hdl_info);
1172         if (status != DDI_SUCCESS) {
1173                 tavor_rsrc_fini(state, cleanup);
1174                 /* Set "status" and "errormsg" and goto failure */
1175                 TAVOR_TNF_FAIL(DDI_FAILURE, "PD handle");
1176                 goto rsrcinitp2_fail;
1177         }
1178         cleanup = TAVOR_RSRC_CLEANUP_LEVEL22;
1179
1180         /*
1181          * Initialize the resource pool for the MR handles.  Notice
1182          * that the number of MRHDLs is configurable.  The configured value
1183          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
```

```
1184          * command) or the initialization will fail.
1185          */
1186         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_MRHDL];
1187         rsrc_pool->rsrc_type     = TAVOR_MRHDL;
1188         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1189         rsrc_pool->rsrc_quantum  = sizeof (struct tavor_sw_mr_s);
1190         rsrc_pool->rsrc_state    = state;
1191         TAVOR_RSRC_NAME(rsrc_name, TAVOR_MRHDL_CACHE);
1192         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_mpt);
1193         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_mpt);
1194         hdl_info.swi_rsrcpool    = rsrc_pool;
1195         hdl_info.swi_constructor = tavor_rsrc_mrhdl_constructor;
1196         hdl_info.swi_destructor  = tavor_rsrc_mrhdl_destructor;
1197         hdl_info.swi_rsrcname    = rsrc_name;
1198         hdl_info.swi_flags       = TAVOR_SWHDL_KMEMCACHE_INIT;
1199         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1200         if (status != DDI_SUCCESS) {
1201                 tavor_rsrc_fini(state, cleanup);
1202                 /* Set "status" and "errormsg" and goto failure */
1203                 TAVOR_TNF_FAIL(DDI_FAILURE, "MR handle");
1204                 goto rsrcinitp2_fail;
1205         }
1206         cleanup = TAVOR_RSRC_CLEANUP_LEVEL23;

1208         /*
1209          * Initialize the resource pool for the EQ handles.  Notice
1210          * that the number of EQHDLs is hardcoded.  The hardcoded value
1211          * should be less that the maximum value (obtained from the
1212          * QUERY_DEV_LIM command) or the initialization will fail.
1213          */
1214         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_EQHDL];
1215         rsrc_pool->rsrc_type     = TAVOR_EQHDL;
1216         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1217         rsrc_pool->rsrc_quantum  = sizeof (struct tavor_sw_eq_s);
1218         rsrc_pool->rsrc_state    = state;
1219         TAVOR_RSRC_NAME(rsrc_name, TAVOR_EQHDL_CACHE);
1220         hdl_info.swi_num = TAVOR_NUM_EQ;
1221         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_eq);
1222         hdl_info.swi_rsrcpool    = rsrc_pool;
1223         hdl_info.swi_constructor = NULL;
1224         hdl_info.swi_destructor  = NULL;
1225         hdl_info.swi_rsrcname    = rsrc_name;
1226         hdl_info.swi_flags       = TAVOR_SWHDL_KMEMCACHE_INIT;
1227         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1228         if (status != DDI_SUCCESS) {
1229                 tavor_rsrc_fini(state, cleanup);
1230                 /* Set "status" and "errormsg" and goto failure */
1231                 TAVOR_TNF_FAIL(DDI_FAILURE, "EQ handle");
1232                 goto rsrcinitp2_fail;
1233         }
1234         cleanup = TAVOR_RSRC_CLEANUP_LEVEL24;

1236         /*
1237          * Initialize the resource pool for the CQ handles.  Notice
1238          * that the number of CQHDLs is configurable.  The configured value
1239          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
1240          * command) or the initialization will fail.  Note also that the CQ
1241          * handle has constructor and destructor methods associated with it.
1242          */
1243         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_CQHDL];
1244         rsrc_pool->rsrc_type     = TAVOR_CQHDL;
1245         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1246         rsrc_pool->rsrc_quantum  = sizeof (struct tavor_sw_cq_s);
1247         rsrc_pool->rsrc_state    = state;
1248         TAVOR_RSRC_NAME(rsrc_name, TAVOR_CQHDL_CACHE);
1249         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_cq);
```



```
1250         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_cq);
1251         hdl_info.swi_rsrcpool    = rsrc_pool;
1252         hdl_info.swi_constructor = tavor_rsrc_cqhdl_constructor;
1253         hdl_info.swi_destructor  = tavor_rsrc_cqhdl_destructor;
1254         hdl_info.swi_rsrcname    = rsrc_name;
1255         hdl_info.swi_flags       = (TAVOR_SWHDL_KMEMCACHE_INIT |
1256             TAVOR_SWHDL_TABLE_INIT);
1257         hdl_info.swi_prealloc_sz = sizeof (tavor_cqhdl_t);
1258         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1259         if (status != DDI_SUCCESS) {
1260                 tavor_rsrc_fini(state, cleanup);
1261                 /* Set "status" and "errormsg" and goto failure */
1262                 TAVOR_TNF_FAIL(DDI_FAILURE, "CQ handle");
1263                 goto rsrcinitp2_fail;
1264         }

1266         /*
1267          * Save away the pointer to the central list of CQ handle pointers
1268          * This this is used as a mechanism to enable fast CQnumber-to-CQhandle
1269          * lookup during EQ event processing.  The table is a list of
1270          * tavor_cqhdl_t allocated by the above routine because of the
1271          * TAVOR_SWHDL_TABLE_INIT flag.  The table has as many tavor_cqhdl_t
1272          * as the number of CQs.
1273          */
1274         state->ts_cqhdl = hdl_info.swi_table_ptr;
1275         cleanup = TAVOR_RSRC_CLEANUP_LEVEL25;

1277         /*
1278          * Initialize the resource pool for the SRQ handles.  Notice
1279          * that the number of SRQHDLs is configurable.  The configured value
1280          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
1281          * command) or the initialization will fail.  Note also that the SRQ
1282          * handle has constructor and destructor methods associated with it.
1283          *
1284          * Note: We only allocate these resources if SRQ is enabled in the
1285          * config profile; see below.
1286          */
1287         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_SRQHDL];
1288         rsrc_pool->rsrc_type     = TAVOR_SRQHDL;
1289         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1290         rsrc_pool->rsrc_quantum  = sizeof (struct tavor_sw_srq_s);
1291         rsrc_pool->rsrc_state    = state;
1292         TAVOR_RSRC_NAME(rsrc_name, TAVOR_SRQHDL_CACHE);
1293         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_srq);
1294         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_srq);
1295         hdl_info.swi_rsrcpool    = rsrc_pool;
1296         hdl_info.swi_constructor = tavor_rsrc_srqhdl_constructor;
1297         hdl_info.swi_destructor  = tavor_rsrc_srqhdl_destructor;
1298         hdl_info.swi_rsrcname    = rsrc_name;
1299         hdl_info.swi_flags       = (TAVOR_SWHDL_KMEMCACHE_INIT |
1300             TAVOR_SWHDL_TABLE_INIT);
1301         hdl_info.swi_prealloc_sz = sizeof (tavor_srqhdl_t);

1303         /*
1304          * SRQ support is configurable.  Only if SRQ is enabled (the default)
1305          * do we actually try to configure these resources.  Otherwise, we
1306          * simply set the cleanup level and continue on to the next resource
1307          */
1308         if (state->ts_cfg_profile->cp_srq_enable != 0) {
1309                 status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1310                 if (status != DDI_SUCCESS) {
1311                         tavor_rsrc_fini(state, cleanup);
1312                         /* Set "status" and "errormsg" and goto failure */
1313                         TAVOR_TNF_FAIL(DDI_FAILURE, "SRQ handle");
1314                         goto rsrcinitp2_fail;
1315                 }
```

```
1317                 /*
1318                  * Save away the pointer to the central list of SRQ handle
1319                  * pointers This this is used as a mechanism to enable fast
1320                  * SRQnumber-to-SRQhandle lookup.  The table is a list of
1321                  * tavor_srqhdl_t allocated by the above routine because of the
1322                  * TAVOR_SWHDL_TABLE_INIT flag.  The table has as many
1323                  * tavor_srqhdl_t as the number of SRQs.
1324                  */
1325                 state->ts_srqhdl = hdl_info.swi_table_ptr;
1326         }
1327         cleanup = TAVOR_RSRC_CLEANUP_LEVEL26;
1328
1329         /*
1330          * Initialize the resource pool for the address handles.  Notice
1331          * that the number of AHHDLs is configurable.  The configured value
1332          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
1333          * command) or the initialization will fail.
1334          */
1335         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_AHHDL];
1336         rsrc_pool->rsrc_type     = TAVOR_AHHDL;
1337         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1338         rsrc_pool->rsrc_quantum  = sizeof (struct tavor_sw_ah_s);
1339         rsrc_pool->rsrc_state    = state;
1340         TAVOR_RSRC_NAME(rsrc_name, TAVOR_AHHDL_CACHE);
1341         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_ah);
1342         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_av);
1343         hdl_info.swi_rsrcpool    = rsrc_pool;
1344         hdl_info.swi_constructor = tavor_rsrc_ahhdl_constructor;
1345         hdl_info.swi_destructor  = tavor_rsrc_ahhdl_destructor;
1346         hdl_info.swi_rsrcname    = rsrc_name;
1347         hdl_info.swi_flags       = TAVOR_SWHDL_KMEMCACHE_INIT;
1348         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1349         if (status != DDI_SUCCESS) {
1350                 tavor_rsrc_fini(state, cleanup);
1351                 /* Set "status" and "errormsg" and goto failure */
1352                 TAVOR_TNF_FAIL(DDI_FAILURE, "AH handle");
1353                 goto rsrcinitp2_fail;
1354         }
1355         cleanup = TAVOR_RSRC_CLEANUP_LEVEL27;
1356
1357         /*
1358          * Initialize the resource pool for the QP handles.  Notice
1359          * that the number of QPHDLs is configurable.  The configured value
1360          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
1361          * command) or the initialization will fail.  Note also that the QP
1362          * handle has constructor and destructor methods associated with it.
1363          */
1364         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_QPHDL];
1365         rsrc_pool->rsrc_type     = TAVOR_QPHDL;
1366         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1367         rsrc_pool->rsrc_quantum  = sizeof (struct tavor_sw_qp_s);
1368         rsrc_pool->rsrc_state    = state;
1369         TAVOR_RSRC_NAME(rsrc_name, TAVOR_QPHDL_CACHE);
1370         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_qp);
1371         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_qp);
1372         hdl_info.swi_rsrcpool    = rsrc_pool;
1373         hdl_info.swi_constructor = tavor_rsrc_qphdl_constructor;
1374         hdl_info.swi_destructor  = tavor_rsrc_qphdl_destructor;
1375         hdl_info.swi_rsrcname    = rsrc_name;
1376         hdl_info.swi_flags       = (TAVOR_SWHDL_KMEMCACHE_INIT |
1377             TAVOR_SWHDL_TABLE_INIT);
1378         hdl_info.swi_prealloc_sz = sizeof (tavor_qphdl_t);
1379         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1380         if (status != DDI_SUCCESS) {
1381                 tavor_rsrc_fini(state, cleanup);
```



```
1382                 /* Set "status" and "errormsg" and goto failure */
1383                 TAVOR_TNF_FAIL(DDI_FAILURE, "QP handle");
1384                 goto rsrcinitp2_fail;
1385         }
1386
1387         /*
1388          * Save away the pointer to the central list of QP handle pointers
1389          * This this is used as a mechanism to enable fast QPnumber-to-QPhandle
1390          * lookup during CQ event processing.  The table is a list of
1391          * tavor_qphdl_t allocated by the above routine because of the
1392          * TAVOR_SWHDL_TABLE_INIT flag.  The table has as many tavor_qphdl_t
1393          * as the number of QPs.
1394          */
1395         state->ts_qphdl = hdl_info.swi_table_ptr;
1396         cleanup = TAVOR_RSRC_CLEANUP_LEVEL28;
1397
1398         /*
1399          * Initialize the resource pool for the reference count handles.
1400          * Notice that the number of REFCNTs is configurable, but it's value
1401          * is set to the number of MPTs.  Since REFCNTs are used to support
1402          * shared memory regions, it is possible that we might require as
1403          * one REFCNT for every MPT.
1404          */
1405         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_REFCNT];
1406         rsrc_pool->rsrc_type     = TAVOR_REFCNT;
1407         rsrc_pool->rsrc_loc      = TAVOR_IN_SYSMEM;
1408         rsrc_pool->rsrc_quantum  = sizeof (tavor_sw_refcnt_t);
1409         rsrc_pool->rsrc_state    = state;
1410         TAVOR_RSRC_NAME(rsrc_name, TAVOR_REFCNT_CACHE);
1411         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_mpt);
1412         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_mpt);
1413         hdl_info.swi_rsrcpool    = rsrc_pool;
1414         hdl_info.swi_constructor = tavor_rsrc_refcnt_constructor;
1415         hdl_info.swi_destructor  = tavor_rsrc_refcnt_destructor;
1416         hdl_info.swi_rsrcname    = rsrc_name;
1417         hdl_info.swi_flags       = TAVOR_SWHDL_KMEMCACHE_INIT;
1418         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1419         if (status != DDI_SUCCESS) {
1420                 tavor_rsrc_fini(state, cleanup);
1421                 /* Set "status" and "errormsg" and goto failure */
1422                 TAVOR_TNF_FAIL(DDI_FAILURE, "reference count handle");
1423                 goto rsrcinitp2_fail;
1424         }
1425         cleanup = TAVOR_RSRC_CLEANUP_LEVEL29;
1426
1427         /*
1428          * Initialize the resource pool for the MCG handles.  Notice that for
1429          * these MCG handles, we are allocating a table of structures (used to
1430          * keep track of the MCG entries that are being written to hardware
1431          * and to speed up multicast attach/detach operations).
1432          */
1433         hdl_info.swi_num = ((uint64_t)1 << cfgprof->cp_log_num_mcg);
1434         hdl_info.swi_max = ((uint64_t)1 << state->ts_devlim.log_max_mcg);
1435         hdl_info.swi_flags = TAVOR_SWHDL_TABLE_INIT;
1436         hdl_info.swi_prealloc_sz = sizeof (struct tavor_sw_mcg_list_s);
1437         status = tavor_rsrc_sw_handles_init(state, &hdl_info);
1438         if (status != DDI_SUCCESS) {
1439                 tavor_rsrc_fini(state, cleanup);
1440                 /* Set "status" and "errormsg" and goto failure */
1441                 TAVOR_TNF_FAIL(DDI_FAILURE, "MCG handle");
1442                 goto rsrcinitp2_fail;
1443         }
1444         state->ts_mcghdl = hdl_info.swi_table_ptr;
1445         cleanup = TAVOR_RSRC_CLEANUP_LEVEL30;
1446
1447         /*
```

```
1448          * Initialize the resource pools for all objects that exist in
1449          * UAR memory.  The only objects that are allocated from UAR memory
1450          * are the UAR pages which are used for holding Tavor hardware's
1451          * doorbell registers.
1452          */
1454         /*
1455          * Initialize the resource pool for the UAR pages.  Notice
1456          * that the number of UARPGs is configurable.  The configured value
1457          * must be less that the maximum value (obtained from the QUERY_DEV_LIM
1458          * command) or the initialization will fail.  Note also that by
1459          * specifying the rsrc_start parameter in advance, we direct the
1460          * initialization routine not to attempt to allocated space from the
1461          * Tavor DDR vmem_arena.
1462          */
1463         num  = ((uint64_t)1 << cfgprof->cp_log_num_uar);
1464         max  = ((uint64_t)1 << (state->ts_devlim.log_max_uar_sz + 20 -
1465             PAGESHIFT));
1466         num_prealloc = 0;
1467         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_UARPG];
1468         rsrc_pool->rsrc_type      = TAVOR_UARPG;
1469         rsrc_pool->rsrc_loc       = TAVOR_IN_UAR;
1470         rsrc_pool->rsrc_pool_size = (num << PAGESHIFT);
1471         rsrc_pool->rsrc_shift     = PAGESHIFT;
1472         rsrc_pool->rsrc_quantum   = PAGESIZE;
1473         rsrc_pool->rsrc_align     = PAGESIZE;
1474         rsrc_pool->rsrc_state     = state;
1475         rsrc_pool->rsrc_start     = (void *)state->ts_reg_uar_baseaddr;
1476         TAVOR_RSRC_NAME(rsrc_name, TAVOR_UAR_VMEM);
1477         entry_info.hwi_num        = num;
1478         entry_info.hwi_max        = max;
1479         entry_info.hwi_prealloc   = num_prealloc;
1480         entry_info.hwi_rsrcpool   = rsrc_pool;
1481         entry_info.hwi_rsrcname   = rsrc_name;
1482         status = tavor_rsrc_hw_entries_init(state, &entry_info);
1483         if (status != DDI_SUCCESS) {
1484                 tavor_rsrc_fini(state, cleanup);
1485                 /* Set "status" and "errormsg" and goto failure */
1486                 TAVOR_TNF_FAIL(DDI_FAILURE, "UAR page table");
1487                 goto rsrcinitp2_fail;
1488         }
1489         cleanup = TAVOR_RSRC_CLEANUP_ALL;
1491         kmem_free(rsrc_name, TAVOR_RSRC_NAME_MAXLEN);
1492         TAVOR_TNF_EXIT(tavor_rsrc_init_phase2);
1493         return (DDI_SUCCESS);
1495 rsrcinitp2_fail:
1496         kmem_free(rsrc_name, TAVOR_RSRC_NAME_MAXLEN);
1497         TNF_PROBE_1(tavor_rsrc_init_phase2_fail, TAVOR_TNF_ERROR, "",
1498             tnf_string, msg, errormsg);
1499         TAVOR_TNF_EXIT(tavor_rsrc_init_phase2);
1500         return (status);
1501 }
1504 /*
1505  * tavor_rsrc_fini()
1506  *    Context: Only called from attach() and/or detach() path contexts
1507  */
1508 void
1509 tavor_rsrc_fini(tavor_state_t *state, tavor_rsrc_cleanup_level_t clean)
1510 {
1511         tavor_rsrc_sw_hdl_info_t        hdl_info;
1512         tavor_rsrc_hw_entry_info_t      entry_info;
1513         tavor_rsrc_mbox_info_t          mbox_info;
```



```
1514         tavor_cfg_profile_t             *cfgprof;
1516         TAVOR_TNF_ENTER(tavor_rsrc_fini);
1518         ASSERT(state != NULL);
1520         cfgprof = state->ts_cfg_profile;
1522         switch (clean) {
1523         /*
1524          * If we add more resources that need to be cleaned up here, we should
1525          * ensure that TAVOR_RSRC_CLEANUP_ALL is still the first entry (i.e.
1526          * corresponds to the last resource allocated).
1527          */
1528         case TAVOR_RSRC_CLEANUP_ALL:
1529                 /* Cleanup the UAR page resource pool */
1530                 entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_UARPG];
1531                 tavor_rsrc_hw_entries_fini(state, &entry_info);
1532                 /* FALLTHROUGH */
1534         case TAVOR_RSRC_CLEANUP_LEVEL30:
1535                 /* Cleanup the central MCG handle pointers list */
1536                 hdl_info.swi_rsrcpool  = NULL;
1537                 hdl_info.swi_table_ptr = state->ts_mcghdl;
1538                 hdl_info.swi_num =
1539                     ((uint64_t)1 << cfgprof->cp_log_num_mcg);
1540                 hdl_info.swi_prealloc_sz = sizeof (struct tavor_sw_mcg_list_s);
1541                 tavor_rsrc_sw_handles_fini(state, &hdl_info);
1542                 /* FALLTHROUGH */
1544         case TAVOR_RSRC_CLEANUP_LEVEL29:
1545                 /* Cleanup the reference count resource pool */
1546                 hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_REFCNT];
1547                 hdl_info.swi_table_ptr = NULL;
1548                 tavor_rsrc_sw_handles_fini(state, &hdl_info);
1549                 /* FALLTHROUGH */
1551         case TAVOR_RSRC_CLEANUP_LEVEL28:
1552                 /* Cleanup the QP handle resource pool */
1553                 hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_QPHDL];
1554                 hdl_info.swi_table_ptr = state->ts_qphdl;
1555                 hdl_info.swi_num =
1556                     ((uint64_t)1 << cfgprof->cp_log_num_qp);
1557                 hdl_info.swi_prealloc_sz = sizeof (tavor_qphdl_t);
1558                 tavor_rsrc_sw_handles_fini(state, &hdl_info);
1559                 /* FALLTHROUGH */
1561         case TAVOR_RSRC_CLEANUP_LEVEL27:
1562                 /* Cleanup the address handle resource pool */
1563                 hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_AHHDL];
1564                 hdl_info.swi_table_ptr = NULL;
1565                 tavor_rsrc_sw_handles_fini(state, &hdl_info);
1566                 /* FALLTHROUGH */
1568         case TAVOR_RSRC_CLEANUP_LEVEL26:
1569                 /*
1570                  * Cleanup the SRQ handle resource pool.
1571                  *
1572                  * Note: We only clean up if SRQ is enabled.  Otherwise we
1573                  * simply fallthrough to the next resource cleanup.
1574                  */
1575                 if (state->ts_cfg_profile->cp_srq_enable != 0) {
1576                         hdl_info.swi_rsrcpool  =
1577                             &state->ts_rsrc_hdl[TAVOR_SRQHDL];
1578                         hdl_info.swi_table_ptr = state->ts_srqhdl;
1579                         hdl_info.swi_num =
```

```
1580                            ((uint64_t)1 << cfgprof->cp_log_num_srq);
1581                    hdl_info.swi_prealloc_sz = sizeof (tavor_srqhdl_t);
1582                    tavor_rsrc_sw_handles_fini(state, &hdl_info);
1583            }
1584            /* FALLTHROUGH */

1586        case TAVOR_RSRC_CLEANUP_LEVEL25:
1587            /* Cleanup the CQ handle resource pool */
1588            hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_CQHDL];
1589            hdl_info.swi_table_ptr = state->ts_cqhdl;
1590            hdl_info.swi_num =
1591                ((uint64_t)1 << cfgprof->cp_log_num_cq);
1592            hdl_info.swi_prealloc_sz = sizeof (tavor_cqhdl_t);
1593            tavor_rsrc_sw_handles_fini(state, &hdl_info);
1594            /* FALLTHROUGH */

1596        case TAVOR_RSRC_CLEANUP_LEVEL24:
1597            /* Cleanup the EQ handle resource pool */
1598            hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_EQHDL];
1599            hdl_info.swi_table_ptr = NULL;
1600            tavor_rsrc_sw_handles_fini(state, &hdl_info);
1601            /* FALLTHROUGH */

1603        case TAVOR_RSRC_CLEANUP_LEVEL23:
1604            /* Cleanup the MR handle resource pool */
1605            hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_MRHDL];
1606            hdl_info.swi_table_ptr = NULL;
1607            tavor_rsrc_sw_handles_fini(state, &hdl_info);
1608            /* FALLTHROUGH */

1610        case TAVOR_RSRC_CLEANUP_LEVEL22:
1611            /* Cleanup the PD handle resource pool */
1612            hdl_info.swi_rsrcpool  = &state->ts_rsrc_hdl[TAVOR_PDHDL];
1613            hdl_info.swi_table_ptr = NULL;
1614            tavor_rsrc_pd_handles_fini(state, &hdl_info);
1615            /* FALLTHROUGH */

1617        case TAVOR_RSRC_CLEANUP_LEVEL21:
1618            /* Cleanup the EQC table resource pool */
1619            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_EQC];
1620            tavor_rsrc_hw_entries_fini(state, &entry_info);
1621            /* FALLTHROUGH */

1623        case TAVOR_RSRC_CLEANUP_LEVEL20:
1624            /* Cleanup the MCG table resource pool */
1625            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_MCG];
1626            tavor_rsrc_hw_entries_fini(state, &entry_info);
1627            /* FALLTHROUGH */

1629        case TAVOR_RSRC_CLEANUP_LEVEL19:
1630            /* Cleanup the outstanding command list  */
1631            tavor_outstanding_cmdlist_fini(state);
1632            /* FALLTHROUGH */

1634        case TAVOR_RSRC_CLEANUP_LEVEL18:
1635            /* Cleanup the "In" mailbox list  */
1636            tavor_intr_inmbox_list_fini(state);
1637            /* FALLTHROUGH */

1639        case TAVOR_RSRC_CLEANUP_LEVEL17:
1640            /* Cleanup the interrupt "In" mailbox resource pool */
1641            mbox_info.mbi_rsrcpool = &state->ts_rsrc_hdl[
1642                TAVOR_INTR_IN_MBOX];
1643            tavor_rsrc_mbox_fini(state, &mbox_info);
1644            /* FALLTHROUGH */
```



```
1646        case TAVOR_RSRC_CLEANUP_LEVEL16:
1647            /* Cleanup the "In" mailbox list  */
1648            tavor_inmbox_list_fini(state);
1649            /* FALLTHROUGH */

1651        case TAVOR_RSRC_CLEANUP_LEVEL15:
1652            /* Cleanup the "In" mailbox resource pool */
1653            mbox_info.mbi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_IN_MBOX];
1654            tavor_rsrc_mbox_fini(state, &mbox_info);
1655            /* FALLTHROUGH */

1657        case TAVOR_RSRC_CLEANUP_LEVEL14:
1658            /*
1659             * Cleanup the SRQC table resource pool.
1660             *
1661             * Note: We only clean up if SRQ is enabled.  Otherwise we
1662             * simply fallthrough to the next resource cleanup.
1663             */
1664            if (state->ts_cfg_profile->cp_srq_enable != 0) {
1665                entry_info.hwi_rsrcpool =
1666                    &state->ts_rsrc_hdl[TAVOR_SRQC];
1667                tavor_rsrc_hw_entries_fini(state, &entry_info);
1668            }
1669            /* FALLTHROUGH */

1671        case TAVOR_RSRC_CLEANUP_LEVEL13:
1672            /* Cleanup the UAR scratch table resource pool */
1673            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_UAR_SCR];
1674            tavor_rsrc_hw_entries_fini(state, &entry_info);
1675            /* FALLTHROUGH */

1677        case TAVOR_RSRC_CLEANUP_LEVEL12:
1678            /* Cleanup the UDAV table resource pool */
1679            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_UDAV];
1680            tavor_rsrc_hw_entries_fini(state, &entry_info);
1681            /* FALLTHROUGH */

1683        case TAVOR_RSRC_CLEANUP_LEVEL11:
1684            /* Cleanup the EQPC table resource pool */
1685            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_EQPC];
1686            tavor_rsrc_hw_entries_fini(state, &entry_info);
1687            /* FALLTHROUGH */

1689        case TAVOR_RSRC_CLEANUP_LEVEL10:
1690            /* Cleanup the CQC table resource pool */
1691            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_CQC];
1692            tavor_rsrc_hw_entries_fini(state, &entry_info);
1693            /* FALLTHROUGH */

1695        case TAVOR_RSRC_CLEANUP_LEVEL9:
1696            /* Cleanup the RDB table resource pool */
1697            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_RDB];
1698            tavor_rsrc_hw_entries_fini(state, &entry_info);
1699            /* FALLTHROUGH */

1701        case TAVOR_RSRC_CLEANUP_LEVEL8:
1702            /* Cleanup the QPC table resource pool */
1703            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_QPC];
1704            tavor_rsrc_hw_entries_fini(state, &entry_info);
1705            /* FALLTHROUGH */

1707        case TAVOR_RSRC_CLEANUP_LEVEL7:
1708            /* Cleanup the MTT table resource pool */
1709            entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_MTT];
1710            tavor_rsrc_hw_entries_fini(state, &entry_info);
1711            /* FALLTHROUGH */
```

```
1713         case TAVOR_RSRC_CLEANUP_LEVEL6:
1714                 /* Cleanup the MPT table resource pool */
1715                 entry_info.hwi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_MPT];
1716                 tavor_rsrc_hw_entries_fini(state, &entry_info);
1717                 /* FALLTHROUGH */

1719         case TAVOR_RSRC_CLEANUP_LEVEL5:
1720                 /* Destroy the vmem arena for DDR memory */
1721                 vmem_destroy(state->ts_ddrvmem);
1722                 break;

1724         /*
1725          * The cleanup below comes from the "Phase 1" initialization step.
1726          * (see tavor_rsrc_init_phase1() above)
1727          */
1728         case TAVOR_RSRC_CLEANUP_PHASE1_COMPLETE:
1729                 /* Cleanup the interrupt "Out" mailbox list  */
1730                 tavor_intr_outmbox_list_fini(state);
1731                 /* FALLTHROUGH */

1733         case TAVOR_RSRC_CLEANUP_LEVEL3:
1734                 /* Cleanup the "Out" mailbox resource pool */
1735                 mbox_info.mbi_rsrcpool = &state->ts_rsrc_hdl[
1736                     TAVOR_INTR_OUT_MBOX];
1737                 tavor_rsrc_mbox_fini(state, &mbox_info);
1738                 /* FALLTHROUGH */

1740         case TAVOR_RSRC_CLEANUP_LEVEL2:
1741                 /* Cleanup the "Out" mailbox list  */
1742                 tavor_outmbox_list_fini(state);
1743                 /* FALLTHROUGH */

1745         case TAVOR_RSRC_CLEANUP_LEVEL1:
1746                 /* Cleanup the "Out" mailbox resource pool */
1747                 mbox_info.mbi_rsrcpool = &state->ts_rsrc_hdl[TAVOR_OUT_MBOX];
1748                 tavor_rsrc_mbox_fini(state, &mbox_info);
1749                 /* FALLTHROUGH */

1751         case TAVOR_RSRC_CLEANUP_LEVEL0:
1752                 /* Free the array of tavor_rsrc_pool_info_t's */
1753                 kmem_free(state->ts_rsrc_hdl, TAVOR_NUM_RESOURCES *
1754                     sizeof (tavor_rsrc_pool_info_t));
1755                 kmem_cache_destroy(state->ts_rsrc_cache);
1756                 break;

1758         default:
1759                 TAVOR_WARNING(state, "unexpected resource cleanup level");
1760                 TNF_PROBE_0(tavor_rsrc_fini_default_fail, TAVOR_TNF_ERROR, "");
1761                 TAVOR_TNF_EXIT(tavor_rsrc_fini);
1762                 return;
1763         }

1765         TAVOR_TNF_EXIT(tavor_rsrc_fini);
1766 }


1769 /*
1770  * tavor_rsrc_mbox_init()
1771  *    Context: Only called from attach() path context
1772  */
1773 static int
1774 tavor_rsrc_mbox_init(tavor_state_t *state, tavor_rsrc_mbox_info_t *info)
1775 {
1776         tavor_rsrc_pool_info_t  *rsrc_pool;
1777         tavor_rsrc_priv_mbox_t  *priv;
```



```
1778         vmem_t                  *vmp;
1779         uint64_t                offset;
1780         uint_t                  dma_xfer_mode;

1782         TAVOR_TNF_ENTER(tavor_rsrc_mbox_init);

1784         ASSERT(state != NULL);
1785         ASSERT(info != NULL);

1787         rsrc_pool = info->mbi_rsrcpool;
1788         ASSERT(rsrc_pool != NULL);

1790         dma_xfer_mode = state->ts_cfg_profile->cp_streaming_consistent;

1792         /* Allocate and initialize mailbox private structure */
1793         priv = kmem_zalloc(sizeof (tavor_rsrc_priv_mbox_t), KM_SLEEP);
1794         priv->pmb_dip           = state->ts_dip;
1795         priv->pmb_acchdl        = state->ts_reg_ddrhdl;
1796         priv->pmb_devaccattr    = state->ts_reg_accattr;
1797         priv->pmb_xfer_mode     = dma_xfer_mode;

1799         /*
1800          * Initialize many of the default DMA attributes.  Then set alignment
1801          * and scatter-gather restrictions specific for mailbox memory.
1802          */
1803         tavor_dma_attr_init(&priv->pmb_dmaattr);
1804         priv->pmb_dmaattr.dma_attr_align  = TAVOR_MBOX_ALIGN;
1805         priv->pmb_dmaattr.dma_attr_sgllen = 1;

1807         rsrc_pool->rsrc_private = priv;

1809         /* Is object in DDR memory or system memory? */
1810         if (rsrc_pool->rsrc_loc == TAVOR_IN_DDR) {
1811                 rsrc_pool->rsrc_ddr_offset = vmem_xalloc(state->ts_ddrvmem,
1812                     rsrc_pool->rsrc_pool_size, rsrc_pool->rsrc_align,
1813                     0, 0, NULL, NULL, VM_SLEEP);
1814                 if (rsrc_pool->rsrc_ddr_offset == NULL) {
1815                         /* Unable to alloc space for mailboxes */
1816                         kmem_free(priv, sizeof (tavor_rsrc_priv_mbox_t));
1817                         TNF_PROBE_0(tavor_rsrc_mbox_init_vma_fail,
1818                             TAVOR_TNF_ERROR, "");
1819                         TAVOR_TNF_EXIT(tavor_rsrc_mbox_init);
1820                         return (DDI_FAILURE);
1821                 }

1823                 /* Calculate offset and starting point (in DDR) */
1824                 offset = ((uintptr_t)rsrc_pool->rsrc_ddr_offset -
1825                     state->ts_ddr.ddr_baseaddr);
1826                 rsrc_pool->rsrc_start =
1827                     (void *)(uintptr_t)((uintptr_t)state->ts_reg_ddr_baseaddr +
1828                     offset);

1830                 /* Create new vmem arena for the mailboxes */
1831                 vmp = vmem_create(info->mbi_rsrcname,
1832                     rsrc_pool->rsrc_ddr_offset, rsrc_pool->rsrc_pool_size,
1833                     rsrc_pool->rsrc_quantum, NULL, NULL, NULL, 0, VM_SLEEP);
1834                 if (vmp == NULL) {
1835                         /* Unable to create vmem arena */
1836                         vmem_xfree(state->ts_ddrvmem,
1837                             rsrc_pool->rsrc_ddr_offset,
1838                             rsrc_pool->rsrc_pool_size);
1839                         kmem_free(priv, sizeof (tavor_rsrc_priv_mbox_t));
1840                         TNF_PROBE_0(tavor_rsrc_mbox_init_vmem_create_fail,
1841                             TAVOR_TNF_ERROR, "");
1842                         TAVOR_TNF_EXIT(tavor_rsrc_mbox_init);
1843                         return (DDI_FAILURE);
```

```
1844                 }
1845                 rsrc_pool->rsrc_vmp = vmp;
1846         } else {
1847                 rsrc_pool->rsrc_ddr_offset = NULL;
1848                 rsrc_pool->rsrc_start = NULL;
1849                 rsrc_pool->rsrc_vmp = NULL;
1850         }

1852         TAVOR_TNF_EXIT(tavor_rsrc_mbox_init);
1853         return (DDI_SUCCESS);
1854 }


1857 /*
1858  * tavor_rsrc_mbox_fini()
1859  *    Context: Only called from attach() and/or detach() path contexts
1860  */
1861 static void
1862 tavor_rsrc_mbox_fini(tavor_state_t *state, tavor_rsrc_mbox_info_t *info)
1863 {
1864         tavor_rsrc_pool_info_t  *rsrc_pool;

1866         TAVOR_TNF_ENTER(tavor_rsrc_mbox_fini);

1868         ASSERT(state != NULL);
1869         ASSERT(info != NULL);

1871         rsrc_pool = info->mbi_rsrcpool;
1872         ASSERT(rsrc_pool != NULL);

1874         /* If mailboxes are DDR memory, then destroy and free up vmem */
1875         if (rsrc_pool->rsrc_loc == TAVOR_IN_DDR) {

1877                 /* Destroy the specially created mbox vmem arena */
1878                 vmem_destroy(rsrc_pool->rsrc_vmp);

1880                 /* Free up the region from the ddr_vmem arena */
1881                 vmem_xfree(state->ts_ddrvmem, rsrc_pool->rsrc_ddr_offset,
1882                     rsrc_pool->rsrc_pool_size);
1883         }

1885         /* Free up the private struct */
1886         kmem_free(rsrc_pool->rsrc_private, sizeof (tavor_rsrc_priv_mbox_t));

1888         TAVOR_TNF_EXIT(tavor_rsrc_mbox_fini);
1889 }


1892 /*
1893  * tavor_rsrc_hw_entries_init()
1894  *    Context: Only called from attach() path context
1895  */
1896 static int
1897 tavor_rsrc_hw_entries_init(tavor_state_t *state,
1898     tavor_rsrc_hw_entry_info_t *info)
1899 {
1900         tavor_rsrc_pool_info_t  *rsrc_pool;
1901         tavor_rsrc_t            *rsvd_rsrc = NULL;
1902         vmem_t                  *vmp;
1903         uint64_t                num_hwentry, max_hwentry, num_prealloc;
1904         uint64_t                offset;
1905         int                     status;

1907         TAVOR_TNF_ENTER(tavor_rsrc_hw_entries_init);

1909         ASSERT(state != NULL);
```



```
1910         ASSERT(info != NULL);

1912         rsrc_pool       = info->hwi_rsrcpool;
1913         ASSERT(rsrc_pool != NULL);
1914         num_hwentry     = info->hwi_num;
1915         max_hwentry     = info->hwi_max;
1916         num_prealloc    = info->hwi_prealloc;

1918         /* Make sure number of HW entries makes sense */
1919         if (num_hwentry > max_hwentry) {
1920                 TNF_PROBE_2(tavor_rsrc_hw_entries_init_toomany_fail,
1921                     TAVOR_TNF_ERROR, "", tnf_string, errmsg, "number of HW "
1922                     "entries exceeds device maximum", tnf_uint, maxhw,
1923                     max_hwentry);
1924                 TAVOR_TNF_EXIT(tavor_rsrc_hw_entries_init);
1925                 return (DDI_FAILURE);
1926         }

1928         /*
1929          * Determine if we need to allocate DDR space to set up the
1930          * "rsrc_start" pointer.  Not necessary if "rsrc_start" has already
1931          * been initialized (as is the case for the UAR page init).
1932          */
1933         if (rsrc_pool->rsrc_start == NULL) {
1934                 /* Make sure HW entries table is aligned as specified */
1935                 rsrc_pool->rsrc_ddr_offset = vmem_xalloc(state->ts_ddrvmem,
1936                     rsrc_pool->rsrc_pool_size, rsrc_pool->rsrc_align,
1937                     0, 0, NULL, NULL, VM_NOSLEEP | VM_FIRSTFIT);
1938                 if (rsrc_pool->rsrc_ddr_offset == NULL) {
1939                         /* Unable to alloc space for aligned HW table */
1940                         TNF_PROBE_0(tavor_rsrc_hw_entry_table_vmxalloc_fail,
1941                             TAVOR_TNF_ERROR, "");
1942                         TAVOR_TNF_EXIT(tavor_rsrc_hw_entries_init);
1943                         return (DDI_FAILURE);
1944                 }

1946                 /* Calculate offset and starting point (in DDR) */
1947                 offset = ((uintptr_t)rsrc_pool->rsrc_ddr_offset -
1948                     state->ts_ddr.ddr_baseaddr);
1949                 rsrc_pool->rsrc_start =
1950                     (void *)(uintptr_t)((uintptr_t)state->ts_reg_ddr_baseaddr +
1951                     offset);
1952         } else {
1953                 rsrc_pool->rsrc_ddr_offset = rsrc_pool->rsrc_start;
1954         }

1956         /*
1957          * Create new vmem arena for the HW entries table (if rsrc_quantum
1958          * is non-zero).  Otherwise if rsrc_quantum is zero, then these HW
1959          * entries are not going to be dynamically allocatable (i.e. they
1960          * won't be allocated/freed through tavor_rsrc_alloc/free).  This
1961          * latter option is used for EQPC and UARSCR resource which are, in
1962          * fact, managed by the Tavor hardware.
1963          */
1964         if (rsrc_pool->rsrc_quantum != 0) {
1965                 vmp = vmem_create(info->hwi_rsrcname,
1966                     rsrc_pool->rsrc_ddr_offset, rsrc_pool->rsrc_pool_size,
1967                     rsrc_pool->rsrc_quantum, NULL, NULL, NULL, 0, VM_SLEEP);
1968                 if (vmp == NULL) {
1969                         /* Unable to create vmem arena */
1970                         if (rsrc_pool->rsrc_ddr_offset !=
1971                             rsrc_pool->rsrc_start) {
1972                                 vmem_xfree(state->ts_ddrvmem,
1973                                     rsrc_pool->rsrc_ddr_offset,
1974                                     rsrc_pool->rsrc_pool_size);
1975                         }
```

```
1976                        TNF_PROBE_0(tavor_rsrc_hw_entries_init_vmemcreate_fail,
1977                            TAVOR_TNF_ERROR, "");
1978                        TAVOR_TNF_EXIT(tavor_rsrc_hw_entries_init);
1979                        return (DDI_FAILURE);
1980                }
1981                rsrc_pool->rsrc_vmp = vmp;
1982        } else {
1983                rsrc_pool->rsrc_vmp = NULL;
1984        }

1986        /* The first HW entries may be reserved by Tavor firmware */
1987        if (num_prealloc != 0) {
1988                status = tavor_rsrc_alloc(state, rsrc_pool->rsrc_type,
1989                    num_prealloc, TAVOR_SLEEP, &rsvd_rsrc);
1990                if (status != DDI_SUCCESS) {
1991                        /* Unable to preallocate the reserved HW entries */
1992                        if (rsrc_pool->rsrc_vmp != NULL) {
1993                                vmem_destroy(rsrc_pool->rsrc_vmp);
1994                        }
1995                        if (rsrc_pool->rsrc_ddr_offset !=
1996                            rsrc_pool->rsrc_start) {
1997                                vmem_xfree(state->ts_ddrvmem,
1998                                    rsrc_pool->rsrc_ddr_offset,
1999                                    rsrc_pool->rsrc_pool_size);
2000                        }
2001                        TNF_PROBE_0(tavor_rsrc_hw_entries_init_pre_fail,
2002                            TAVOR_TNF_ERROR, "");
2003                        TAVOR_TNF_EXIT(tavor_rsrc_hw_entries_init);
2004                        return (DDI_FAILURE);
2005                }
2006        }
2007        rsrc_pool->rsrc_private = rsvd_rsrc;

2009        TAVOR_TNF_EXIT(tavor_rsrc_hw_entries_init);
2010        return (DDI_SUCCESS);
2011 }


2014 /*
2015  * tavor_rsrc_hw_entries_fini()
2016  *    Context: Only called from attach() and/or detach() path contexts
2017  */
2018 static void
2019 tavor_rsrc_hw_entries_fini(tavor_state_t *state,
2020     tavor_rsrc_hw_entry_info_t *info)
2021 {
2022        tavor_rsrc_pool_info_t  *rsrc_pool;
2023        tavor_rsrc_t            *rsvd_rsrc;

2025        TAVOR_TNF_ENTER(tavor_rsrc_hw_entries_fini);

2027        ASSERT(state != NULL);
2028        ASSERT(info != NULL);

2030        rsrc_pool = info->hwi_rsrcpool;
2031        ASSERT(rsrc_pool != NULL);

2033        /* Free up any "reserved" (i.e. preallocated) HW entries */
2034        rsvd_rsrc = (tavor_rsrc_t *)rsrc_pool->rsrc_private;
2035        if (rsvd_rsrc != NULL) {
2036                tavor_rsrc_free(state, &rsvd_rsrc);
2037        }

2039        /*
2040         * If we've actually setup a vmem arena for the HW entries, then
2041         * destroy it now
```



```
2042         */
2043        if (rsrc_pool->rsrc_vmp != NULL) {
2044                vmem_destroy(rsrc_pool->rsrc_vmp);
2045        }

2047        /*
2048         * Determine if a region was allocated from the tavor_ddr_vmem
2049         * arena (and free it up if necessary)
2050         */
2051        if (rsrc_pool->rsrc_ddr_offset != rsrc_pool->rsrc_start) {
2052                vmem_xfree(state->ts_ddrvmem, rsrc_pool->rsrc_ddr_offset,
2053                    rsrc_pool->rsrc_pool_size);
2054        }

2056        TAVOR_TNF_EXIT(tavor_rsrc_hw_entries_fini);
2057 }


2060 /*
2061  * tavor_rsrc_sw_handles_init()
2062  *    Context: Only called from attach() path context
2063  */
2064 /* ARGSUSED */
2065 static int
2066 tavor_rsrc_sw_handles_init(tavor_state_t *state, tavor_rsrc_sw_hdl_info_t *info)
2067 {
2068        tavor_rsrc_pool_info_t  *rsrc_pool;
2069        uint64_t                num_swhdl, max_swhdl, prealloc_sz;

2071        TAVOR_TNF_ENTER(tavor_rsrc_sw_handles_init);

2073        ASSERT(state != NULL);
2074        ASSERT(info != NULL);

2076        rsrc_pool       = info->swi_rsrcpool;
2077        ASSERT(rsrc_pool != NULL);
2078        num_swhdl       = info->swi_num;
2079        max_swhdl       = info->swi_max;
2080        prealloc_sz     = info->swi_prealloc_sz;

2082        /* Make sure number of SW handles makes sense */
2083        if (num_swhdl > max_swhdl) {
2084                TNF_PROBE_2(tavor_rsrc_sw_handles_init_toomany_fail,
2085                    TAVOR_TNF_ERROR, "", tnf_string, errmsg, "number of SW "
2086                    "handles exceeds maximum", tnf_uint, maxsw, max_swhdl);
2087                TAVOR_TNF_EXIT(tavor_rsrc_sw_handles_init);
2088                return (DDI_FAILURE);
2089        }

2091        /*
2092         * Depending on the flags parameter, create a kmem_cache for some
2093         * number of software handle structures.  Note: kmem_cache_create()
2094         * will SLEEP until successful.
2095         */
2096        if (info->swi_flags & TAVOR_SWHDL_KMEMCACHE_INIT) {
2097                rsrc_pool->rsrc_private = kmem_cache_create(
2098                    info->swi_rsrcname, rsrc_pool->rsrc_quantum, 0,
2099                    info->swi_constructor, info->swi_destructor, NULL,
2100                    rsrc_pool->rsrc_state, NULL, 0);
2101        }

2103        /* Allocate the central list of SW handle pointers */
2104        if (info->swi_flags & TAVOR_SWHDL_TABLE_INIT) {
2105                info->swi_table_ptr = kmem_zalloc(num_swhdl * prealloc_sz,
2106                    KM_SLEEP);
2107        }
```

```
2109         TAVOR_TNF_EXIT(tavor_rsrc_sw_handles_init);
2110         return (DDI_SUCCESS);
2111 }


2114 /*
2115  * tavor_rsrc_sw_handles_fini()
2116  *    Context: Only called from attach() and/or detach() path contexts
2117  */
2118 /* ARGSUSED */
2119 static void
2120 tavor_rsrc_sw_handles_fini(tavor_state_t *state, tavor_rsrc_sw_hdl_info_t *info)
2121 {
2122         tavor_rsrc_pool_info_t  *rsrc_pool;
2123         uint64_t                num_swhdl, prealloc_sz;

2125         TAVOR_TNF_ENTER(tavor_rsrc_sw_handles_fini);

2127         ASSERT(state != NULL);
2128         ASSERT(info != NULL);

2130         rsrc_pool       = info->swi_rsrcpool;
2131         num_swhdl       = info->swi_num;
2132         prealloc_sz     = info->swi_prealloc_sz;

2134         /*
2135          * If a "software handle" kmem_cache exists for this resource, then
2136          * destroy it now
2137          */
2138         if (rsrc_pool != NULL) {
2139                 kmem_cache_destroy(rsrc_pool->rsrc_private);
2140         }

2142         /* Free up this central list of SW handle pointers */
2143         if (info->swi_table_ptr != NULL) {
2144                 kmem_free(info->swi_table_ptr, num_swhdl * prealloc_sz);
2145         }

2147         TAVOR_TNF_EXIT(tavor_rsrc_sw_handles_fini);
2148 }


2151 /*
2152  * tavor_rsrc_pd_handles_init()
2153  *    Context: Only called from attach() path context
2154  */
2155 static int
2156 tavor_rsrc_pd_handles_init(tavor_state_t *state, tavor_rsrc_sw_hdl_info_t *info)
2157 {
2158         tavor_rsrc_pool_info_t  *rsrc_pool;
2159         vmem_t                  *vmp;
2160         char                    vmem_name[TAVOR_RSRC_NAME_MAXLEN];
2161         int                     status;

2163         TAVOR_TNF_ENTER(tavor_rsrc_pd_handles_init);

2165         ASSERT(state != NULL);
2166         ASSERT(info != NULL);

2168         rsrc_pool = info->swi_rsrcpool;
2169         ASSERT(rsrc_pool != NULL);

2171         /* Initialize the resource pool for software handle table */
2172         status = tavor_rsrc_sw_handles_init(state, info);
2173         if (status != DDI_SUCCESS) {
```



```
2174                 TNF_PROBE_0(tavor_rsrc_pdhdl_alloc_fail, TAVOR_TNF_ERROR, "");
2175                 TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_alloc);
2176                 return (DDI_FAILURE);
2177         }

2179         /* Build vmem arena name from Tavor instance */
2180         TAVOR_RSRC_NAME(vmem_name, TAVOR_PDHDL_VMEM);

2182         /* Create new vmem arena for PD numbers */
2183         vmp = vmem_create(vmem_name, (caddr_t)1, info->swi_num, 1, NULL,
2184             NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
2185         if (vmp == NULL) {
2186                 /* Unable to create vmem arena */
2187                 info->swi_table_ptr = NULL;
2188                 tavor_rsrc_sw_handles_fini(state, info);
2189                 TNF_PROBE_0(tavor_rsrc_pd_handles_init_vmem_create_fail,
2190                     TAVOR_TNF_ERROR, "");
2191                 TAVOR_TNF_EXIT(tavor_rsrc_pd_handles_init);
2192                 return (DDI_FAILURE);
2193         }
2194         rsrc_pool->rsrc_vmp = vmp;

2196         TAVOR_TNF_EXIT(tavor_rsrc_pd_handles_init);
2197         return (DDI_SUCCESS);
2198 }


2201 /*
2202  * tavor_rsrc_pd_handles_fini()
2203  *    Context: Only called from attach() and/or detach() path contexts
2204  */
2205 static void
2206 tavor_rsrc_pd_handles_fini(tavor_state_t *state, tavor_rsrc_sw_hdl_info_t *info)
2207 {
2208         tavor_rsrc_pool_info_t  *rsrc_pool;

2210         TAVOR_TNF_ENTER(tavor_rsrc_pd_handles_fini);

2212         ASSERT(state != NULL);
2213         ASSERT(info != NULL);

2215         rsrc_pool = info->swi_rsrcpool;

2217         /* Destroy the specially created UAR scratch table vmem arena */
2218         vmem_destroy(rsrc_pool->rsrc_vmp);

2220         /* Destroy the "tavor_sw_pd_t" kmem_cache */
2221         tavor_rsrc_sw_handles_fini(state, info);

2223         TAVOR_TNF_EXIT(tavor_rsrc_pd_handles_fini);
2224 }


2227 /*
2228  * tavor_rsrc_mbox_alloc()
2229  *    Context: Only called from attach() path context
2230  */
2231 static int
2232 tavor_rsrc_mbox_alloc(tavor_rsrc_pool_info_t *pool_info, uint_t num,
2233     tavor_rsrc_t *hdl)
2234 {
2235         tavor_rsrc_priv_mbox_t  *priv;
2236         void                    *addr;
2237         caddr_t                 kaddr;
2238         uint64_t                offset;
2239         size_t                  real_len, temp_len;
```

```
2240         int                     status;

2242         TAVOR_TNF_ENTER(tavor_rsrc_mbox_alloc);

2244         ASSERT(pool_info != NULL);
2245         ASSERT(hdl != NULL);

2247         /* Get the private pointer for the mailboxes */
2248         priv = pool_info->rsrc_private;
2249         ASSERT(priv != NULL);

2251         /*
2252          * Allocate a DMA handle for the mailbox.  This will be used for
2253          * two purposes (potentially).  First, it could be used below in
2254          * the call to ddi_dma_mem_alloc() - if the mailbox is to come from
2255          * system memory.  Second, it is definitely used later to bind
2256          * the mailbox for DMA access from/by the hardware.
2257          */
2258         status = ddi_dma_alloc_handle(priv->pmb_dip, &priv->pmb_dmaattr,
2259             DDI_DMA_SLEEP, NULL, &hdl->tr_dmahdl);
2260         if (status != DDI_SUCCESS) {
2261                 TNF_PROBE_1(tavor_rsrc_mbox_alloc_dmahdl_fail, TAVOR_TNF_ERROR,
2262                     "", tnf_uint, status, status);
2263                 TAVOR_TNF_EXIT(tavor_rsrc_mbox_alloc);
2264                 return (DDI_FAILURE);
2265         }

2267         /* Is mailbox in DDR memory or system memory? */
2268         if (pool_info->rsrc_loc == TAVOR_IN_DDR) {
2269                 /* Use vmem_alloc() to get DDR address of mbox */
2270                 hdl->tr_len = (num * pool_info->rsrc_quantum);
2271                 addr = vmem_alloc(pool_info->rsrc_vmp, hdl->tr_len,
2272                     VM_SLEEP);
2273                 if (addr == NULL) {
2274                         /* No more DDR available for mailbox entries */
2275                         ddi_dma_free_handle(&hdl->tr_dmahdl);
2276                         TNF_PROBE_0(tavor_rsrc_mbox_alloc_vma_fail,
2277                             TAVOR_TNF_ERROR, "");
2278                         TAVOR_TNF_EXIT(tavor_rsrc_mbox_alloc);
2279                         return (DDI_FAILURE);
2280                 }
2281                 hdl->tr_acchdl = priv->pmb_acchdl;

2283                 /* Calculate kernel virtual address (from the DDR offset) */
2284                 offset = ((uintptr_t)addr -
2285                     (uintptr_t)pool_info->rsrc_ddr_offset);
2286                 hdl->tr_addr = (void *)(uintptr_t)(offset +
2287                     (uintptr_t)pool_info->rsrc_start);

2289         } else { /* TAVOR_IN_SYSMEM */

2291                 /* Use ddi_dma_mem_alloc() to get memory for mailbox */
2292                 temp_len = (num * pool_info->rsrc_quantum);
2293                 status = ddi_dma_mem_alloc(hdl->tr_dmahdl, temp_len,
2294                     &priv->pmb_devaccattr, priv->pmb_xfer_mode, DDI_DMA_SLEEP,
2295                     NULL, &kaddr, &real_len, &hdl->tr_acchdl);
2296                 if (status != DDI_SUCCESS) {
2297                         /* No more sys memory available for mailbox entries */
2298                         ddi_dma_free_handle(&hdl->tr_dmahdl);
2299                         TNF_PROBE_0(tavor_rsrc_mbox_alloc_dma_memalloc_fail,
2300                             TAVOR_TNF_ERROR, "");
2301                         TAVOR_TNF_EXIT(tavor_rsrc_mbox_alloc);
2302                         return (DDI_FAILURE);
2303                 }
2304                 hdl->tr_addr = (void *)kaddr;
2305                 hdl->tr_len  = real_len;
```



```
2306         }

2308         TAVOR_TNF_EXIT(tavor_rsrc_mbox_alloc);
2309         return (DDI_SUCCESS);
2310 }


2313 /*
2314  * tavor_rsrc_mbox_free()
2315  *    Context: Can be called from interrupt or base context.
2316  */
2317 static void
2318 tavor_rsrc_mbox_free(tavor_rsrc_pool_info_t *pool_info, tavor_rsrc_t *hdl)
2319 {
2320         void            *addr;
2321         uint64_t        offset;

2323         TAVOR_TNF_ENTER(tavor_rsrc_mbox_free);

2325         ASSERT(pool_info != NULL);
2326         ASSERT(hdl != NULL);

2328         /* Is mailbox in DDR memory or system memory? */
2329         if (pool_info->rsrc_loc == TAVOR_IN_DDR) {

2331                 /* Calculate the allocated address (the mbox's DDR offset) */
2332                 offset = ((uintptr_t)hdl->tr_addr -
2333                     (uintptr_t)pool_info->rsrc_start);
2334                 addr = (void *)(uintptr_t)(offset +
2335                     (uintptr_t)pool_info->rsrc_ddr_offset);

2337                 /* Use vmem_free() to free up DDR memory for mailbox */
2338                 vmem_free(pool_info->rsrc_vmp, addr, hdl->tr_len);

2340         } else { /* TAVOR_IN_SYSMEM */

2342                 /* Use ddi_dma_mem_free() to free up sys memory for mailbox */
2343                 ddi_dma_mem_free(&hdl->tr_acchdl);
2344         }

2346         /* Free the DMA handle for the mailbox */
2347         ddi_dma_free_handle(&hdl->tr_dmahdl);

2349         TAVOR_TNF_EXIT(tavor_rsrc_mbox_free);
2350 }


2353 /*
2354  * tavor_rsrc_hw_entry_alloc()
2355  *    Context: Can be called from interrupt or base context.
2356  */
2357 static int
2358 tavor_rsrc_hw_entry_alloc(tavor_rsrc_pool_info_t *pool_info, uint_t num,
2359     uint_t num_align, ddi_acc_handle_t acc_handle, uint_t sleepflag,
2360     tavor_rsrc_t *hdl)
2361 {
2362         void            *addr;
2363         uint64_t        offset;
2364         uint32_t        align;
2365         int             flag;

2367         TAVOR_TNF_ENTER(tavor_rsrc_hw_entry_alloc);

2369         ASSERT(pool_info != NULL);
2370         ASSERT(hdl != NULL);
```

```
2372         /*
2373          * Tavor hardware entries (QPC, CQC, EQC, MPT, MTT, etc.) do not
2374          * use dma_handle (because they are in Tavor locally attached DDR
2375          * memory) and, generally, don't use the acc_handle (because the
2376          * entries are not directly accessed by software).  The exceptions
2377          * to this rule are the UARPG and UDAV entries.
2378          */

2380         /*
2381          * Use vmem_xalloc() to get a properly aligned pointer (based on
2382          * the number requested) to the HW entry(ies).  This handles the
2383          * cases (for special QPCs and for RDB entries) where we need more
2384          * than one and need to ensure that they are properly aligned.
2385          */
2386         flag = (sleepflag == TAVOR_SLEEP) ? VM_SLEEP : VM_NOSLEEP;
2387         hdl->tr_len = (num * pool_info->rsrc_quantum);
2388         align       = (num_align * pool_info->rsrc_quantum);
2389         addr = vmem_xalloc(pool_info->rsrc_vmp, hdl->tr_len,
2390             align, 0, 0, NULL, NULL, flag | VM_FIRSTFIT);
2391         if (addr == NULL) {
2392                 /* No more HW entries available */
2393                 TNF_PROBE_0(tavor_rsrc_hw_entry_alloc_vmxa_fail,
2394                     TAVOR_TNF_ERROR, "");
2395                 TAVOR_TNF_EXIT(tavor_rsrc_hw_entry_alloc);
2396                 return (DDI_FAILURE);
2397         }

2399         /* If an access handle was provided, fill it in */
2400         if (acc_handle != 0) {
2401                 hdl->tr_acchdl = acc_handle;
2402         }

2404         /* Calculate vaddr and HW table index (from the DDR offset) */
2405         offset = ((uintptr_t)addr - (uintptr_t)pool_info->rsrc_ddr_offset);
2406         hdl->tr_addr = (void *)(uintptr_t)(offset +
2407             (uintptr_t)pool_info->rsrc_start);
2408         hdl->tr_indx = (offset >> pool_info->rsrc_shift);

2410         TAVOR_TNF_EXIT(tavor_rsrc_hw_entry_alloc);
2411         return (DDI_SUCCESS);
2412 }


2415 /*
2416  * tavor_rsrc_hw_entry_free()
2417  *    Context: Can be called from interrupt or base context.
2418  */
2419 static void
2420 tavor_rsrc_hw_entry_free(tavor_rsrc_pool_info_t *pool_info, tavor_rsrc_t *hdl)
2421 {
2422         void            *addr;
2423         uint64_t        offset;

2425         TAVOR_TNF_ENTER(tavor_rsrc_hw_entry_free);

2427         ASSERT(pool_info != NULL);
2428         ASSERT(hdl != NULL);

2430         /* Calculate the allocated address (the entry's DDR offset) */
2431         offset = ((uintptr_t)hdl->tr_addr - (uintptr_t)pool_info->rsrc_start);
2432         addr   = (void *)(uintptr_t)(offset +
2433             (uintptr_t)pool_info->rsrc_ddr_offset);

2435         /* Use vmem_xfree() to free up the HW table entry */
2436         vmem_xfree(pool_info->rsrc_vmp, addr, hdl->tr_len);
```



```
2438         TAVOR_TNF_EXIT(tavor_rsrc_hw_entry_free);
2439 }


2442 /*
2443  * tavor_rsrc_swhdl_alloc()
2444  *    Context: Can be called from interrupt or base context.
2445  */
2446 static int
2447 tavor_rsrc_swhdl_alloc(tavor_rsrc_pool_info_t *pool_info, uint_t sleepflag,
2448     tavor_rsrc_t *hdl)
2449 {
2450         void    *addr;
2451         int     flag;

2453         TAVOR_TNF_ENTER(tavor_rsrc_swhdl_alloc);

2455         ASSERT(pool_info != NULL);
2456         ASSERT(hdl != NULL);

2458         /* Allocate the software handle structure */
2459         flag = (sleepflag == TAVOR_SLEEP) ? KM_SLEEP : KM_NOSLEEP;
2460         addr = kmem_cache_alloc(pool_info->rsrc_private, flag);
2461         if (addr == NULL) {
2462                 TNF_PROBE_0(tavor_rsrc_swhdl_alloc_kmca_fail, TAVOR_TNF_ERROR,
2463                     "");
2464                 TAVOR_TNF_EXIT(tavor_rsrc_swhdl_alloc);
2465                 return (DDI_FAILURE);
2466         }
2467         hdl->tr_len  = pool_info->rsrc_quantum;
2468         hdl->tr_addr = addr;

2470         TAVOR_TNF_EXIT(tavor_rsrc_swhdl_alloc);
2471         return (DDI_SUCCESS);
2472 }


2475 /*
2476  * tavor_rsrc_swhdl_free()
2477  *    Context: Can be called from interrupt or base context.
2478  */
2479 static void
2480 tavor_rsrc_swhdl_free(tavor_rsrc_pool_info_t *pool_info, tavor_rsrc_t *hdl)
2481 {
2482         TAVOR_TNF_ENTER(tavor_rsrc_swhdl_free);

2484         ASSERT(pool_info != NULL);
2485         ASSERT(hdl != NULL);

2487         /* Free the software handle structure */
2488         kmem_cache_free(pool_info->rsrc_private, hdl->tr_addr);

2490         TAVOR_TNF_EXIT(tavor_rsrc_swhdl_free);
2491 }


2494 /*
2495  * tavor_rsrc_pdhdl_alloc()
2496  *    Context: Can be called from interrupt or base context.
2497  */
2498 static int
2499 tavor_rsrc_pdhdl_alloc(tavor_rsrc_pool_info_t *pool_info, uint_t sleepflag,
2500     tavor_rsrc_t *hdl)
2501 {
2502         tavor_pdhdl_t   addr;
2503         void            *tmpaddr;
```

```
2504         int             flag, status;

2506         TAVOR_TNF_ENTER(tavor_rsrc_pdhdl_alloc);

2508         ASSERT(pool_info != NULL);
2509         ASSERT(hdl != NULL);

2511         /* Allocate the software handle */
2512         status = tavor_rsrc_swhdl_alloc(pool_info, sleepflag, hdl);
2513         if (status != DDI_SUCCESS) {
2514                 TNF_PROBE_0(tavor_rsrc_pdhdl_alloc_fail, TAVOR_TNF_ERROR, "");
2515                 TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_alloc);
2516                 return (DDI_FAILURE);
2517         }
2518         addr = (tavor_pdhdl_t)hdl->tr_addr;
2519         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*addr))

2521         /* Allocate a PD number for the handle */
2522         flag = (sleepflag == TAVOR_SLEEP) ? VM_SLEEP : VM_NOSLEEP;
2523         tmpaddr = vmem_alloc(pool_info->rsrc_vmp, 1, flag);
2524         if (tmpaddr == NULL) {
2525                 /* No more PD number entries available */
2526                 tavor_rsrc_swhdl_free(pool_info, hdl);
2527                 TNF_PROBE_0(tavor_rsrc_pdhdl_alloc_vma_fail,
2528                     TAVOR_TNF_ERROR, "");
2529                 TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_alloc);
2530                 return (DDI_FAILURE);
2531         }
2532         addr->pd_pdnum = (uint32_t)(uintptr_t)tmpaddr;
2533         addr->pd_rsrcp = hdl;
2534         hdl->tr_indx   = addr->pd_pdnum;

2536         TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_alloc);
2537         return (DDI_SUCCESS);
2538 }


2541 /*
2542  * tavor_rsrc_pdhdl_free()
2543  *    Context: Can be called from interrupt or base context.
2544  */
2545 static void
2546 tavor_rsrc_pdhdl_free(tavor_rsrc_pool_info_t *pool_info, tavor_rsrc_t *hdl)
2547 {
2548         TAVOR_TNF_ENTER(tavor_rsrc_pdhdl_free);

2550         ASSERT(pool_info != NULL);
2551         ASSERT(hdl != NULL);

2553         /* Use vmem_free() to free up the PD number */
2554         vmem_free(pool_info->rsrc_vmp, (void *)(uintptr_t)hdl->tr_indx, 1);

2556         /* Free the software handle structure */
2557         tavor_rsrc_swhdl_free(pool_info, hdl);

2559         TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_free);
2560 }


2563 /*
2564  * tavor_rsrc_pdhdl_constructor()
2565  *    Context: Can be called from interrupt or base context.
2566  */
2567 /* ARGSUSED */
2568 static int
2569 tavor_rsrc_pdhdl_constructor(void *pd, void *priv, int flags)
```



```
2570 {
2571         tavor_pdhdl_t   pdhdl;
2572         tavor_state_t   *state;

2574         TAVOR_TNF_ENTER(tavor_rsrc_pdhdl_constructor);

2576         pdhdl = (tavor_pdhdl_t)pd;
2577         state = (tavor_state_t *)priv;

2579         mutex_init(&pdhdl->pd_lock, NULL, MUTEX_DRIVER,
2580             DDI_INTR_PRI(state->ts_intrmsi_pri));

2582         TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_constructor);
2583         return (DDI_SUCCESS);
2584 }


2587 /*
2588  * tavor_rsrc_pdhdl_destructor()
2589  *    Context: Can be called from interrupt or base context.
2590  */
2591 /* ARGSUSED */
2592 static void
2593 tavor_rsrc_pdhdl_destructor(void *pd, void *priv)
2594 {
2595         tavor_pdhdl_t   pdhdl;

2597         TAVOR_TNF_ENTER(tavor_rsrc_pdhdl_destructor);

2599         pdhdl = (tavor_pdhdl_t)pd;

2601         mutex_destroy(&pdhdl->pd_lock);

2603         TAVOR_TNF_EXIT(tavor_rsrc_pdhdl_destructor);
2604 }


2607 /*
2608  * tavor_rsrc_cqhdl_constructor()
2609  *    Context: Can be called from interrupt or base context.
2610  */
2611 /* ARGSUSED */
2612 static int
2613 tavor_rsrc_cqhdl_constructor(void *cq, void *priv, int flags)
2614 {
2615         tavor_cqhdl_t   cqhdl;
2616         tavor_state_t   *state;

2618         TAVOR_TNF_ENTER(tavor_rsrc_cqhdl_constructor);

2620         cqhdl = (tavor_cqhdl_t)cq;
2621         state = (tavor_state_t *)priv;

2623         mutex_init(&cqhdl->cq_lock, NULL, MUTEX_DRIVER,
2624             DDI_INTR_PRI(state->ts_intrmsi_pri));
2625         mutex_init(&cqhdl->cq_wrid_wqhdr_lock, NULL, MUTEX_DRIVER,
2626             DDI_INTR_PRI(state->ts_intrmsi_pri));

2628         TAVOR_TNF_EXIT(tavor_rsrc_cqhdl_constructor);
2629         return (DDI_SUCCESS);
2630 }


2633 /*
2634  * tavor_rsrc_cqhdl_destructor()
2635  *    Context: Can be called from interrupt or base context.
```

```
 */
/* ARGSUSED */
static void
tavor_rsrc_cqhdl_destructor(void *cq, void *priv)
{
        tavor_cqhdl_t   cqhdl;

        TAVOR_TNF_ENTER(tavor_rsrc_cqhdl_destructor);

        cqhdl = (tavor_cqhdl_t)cq;

        mutex_destroy(&cqhdl->cq_wrid_wqhdr_lock);
        mutex_destroy(&cqhdl->cq_lock);

        TAVOR_TNF_EXIT(tavor_rsrc_cqhdl_destructor);
}


/*
 * tavor_rsrc_qphdl_constructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static int
tavor_rsrc_qphdl_constructor(void *qp, void *priv, int flags)
{
        tavor_qphdl_t   qphdl;
        tavor_state_t   *state;

        TAVOR_TNF_ENTER(tavor_rsrc_qphdl_constructor);

        qphdl = (tavor_qphdl_t)qp;
        state = (tavor_state_t *)priv;

        mutex_init(&qphdl->qp_lock, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(state->ts_intrmsi_pri));

        TAVOR_TNF_EXIT(tavor_rsrc_qphdl_constructor);
        return (DDI_SUCCESS);
}


/*
 * tavor_rsrc_qphdl_destructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static void
tavor_rsrc_qphdl_destructor(void *qp, void *priv)
{
        tavor_qphdl_t   qphdl;

        TAVOR_TNF_ENTER(tavor_rsrc_qphdl_destructor);

        qphdl = (tavor_qphdl_t)qp;

        mutex_destroy(&qphdl->qp_lock);

        TAVOR_TNF_ENTER(tavor_rsrc_qphdl_destructor);
}


/*
 * tavor_rsrc_srqhdl_constructor()
 *    Context: Can be called from interrupt or base context.
 */
```



```
/* ARGSUSED */
static int
tavor_rsrc_srqhdl_constructor(void *srq, void *priv, int flags)
{
        tavor_srqhdl_t  srqhdl;
        tavor_state_t   *state;

        TAVOR_TNF_ENTER(tavor_rsrc_srqhdl_constructor);

        srqhdl = (tavor_srqhdl_t)srq;
        state = (tavor_state_t *)priv;

        mutex_init(&srqhdl->srq_lock, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(state->ts_intrmsi_pri));

        TAVOR_TNF_EXIT(tavor_rsrc_srqhdl_constructor);
        return (DDI_SUCCESS);
}


/*
 * tavor_rsrc_srqhdl_destructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static void
tavor_rsrc_srqhdl_destructor(void *srq, void *priv)
{
        tavor_srqhdl_t  srqhdl;

        TAVOR_TNF_ENTER(tavor_rsrc_srqhdl_destructor);

        srqhdl = (tavor_srqhdl_t)srq;

        mutex_destroy(&srqhdl->srq_lock);

        TAVOR_TNF_EXIT(tavor_rsrc_srqhdl_destructor);
}


/*
 * tavor_rsrc_refcnt_constructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static int
tavor_rsrc_refcnt_constructor(void *rc, void *priv, int flags)
{
        tavor_sw_refcnt_t       *refcnt;
        tavor_state_t           *state;

        TAVOR_TNF_ENTER(tavor_rsrc_refcnt_constructor);

        refcnt = (tavor_sw_refcnt_t *)rc;
        state  = (tavor_state_t *)priv;

        mutex_init(&refcnt->swrc_lock, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(state->ts_intrmsi_pri));

        TAVOR_TNF_EXIT(tavor_rsrc_refcnt_constructor);
        return (DDI_SUCCESS);
}


/*
 * tavor_rsrc_refcnt_destructor()
```

```
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static void
tavor_rsrc_refcnt_destructor(void *rc, void *priv)
{
        tavor_sw_refcnt_t       *refcnt;

        TAVOR_TNF_ENTER(tavor_rsrc_refcnt_destructor);

        refcnt = (tavor_sw_refcnt_t *)rc;

        mutex_destroy(&refcnt->swrc_lock);

        TAVOR_TNF_ENTER(tavor_rsrc_refcnt_destructor);
}


/*
 * tavor_rsrc_ahhdl_constructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static int
tavor_rsrc_ahhdl_constructor(void *ah, void *priv, int flags)
{
        tavor_ahhdl_t   ahhdl;
        tavor_state_t   *state;

        TAVOR_TNF_ENTER(tavor_rsrc_ahhdl_constructor);

        ahhdl = (tavor_ahhdl_t)ah;
        state = (tavor_state_t *)priv;

        mutex_init(&ahhdl->ah_lock, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(state->ts_intrmsi_pri));

        TAVOR_TNF_EXIT(tavor_rsrc_ahhdl_constructor);
        return (DDI_SUCCESS);
}


/*
 * tavor_rsrc_ahhdl_destructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static void
tavor_rsrc_ahhdl_destructor(void *ah, void *priv)
{
        tavor_ahhdl_t   ahhdl;

        TAVOR_TNF_ENTER(tavor_rsrc_ahhdl_destructor);

        ahhdl = (tavor_ahhdl_t)ah;

        mutex_destroy(&ahhdl->ah_lock);

        TAVOR_TNF_ENTER(tavor_rsrc_ahhdl_destructor);
}


/*
 * tavor_rsrc_mrhdl_constructor()
 *    Context: Can be called from interrupt or base context.
 */
```



```
/* ARGSUSED */
static int
tavor_rsrc_mrhdl_constructor(void *mr, void *priv, int flags)
{
        tavor_mrhdl_t   mrhdl;
        tavor_state_t   *state;

        TAVOR_TNF_ENTER(tavor_rsrc_mrhdl_constructor);

        mrhdl = (tavor_mrhdl_t)mr;
        state = (tavor_state_t *)priv;

        mutex_init(&mrhdl->mr_lock, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(state->ts_intrmsi_pri));

        TAVOR_TNF_EXIT(tavor_rsrc_mrhdl_constructor);
        return (DDI_SUCCESS);
}


/*
 * tavor_rsrc_mrhdl_destructor()
 *    Context: Can be called from interrupt or base context.
 */
/* ARGSUSED */
static void
tavor_rsrc_mrhdl_destructor(void *mr, void *priv)
{
        tavor_mrhdl_t   mrhdl;

        TAVOR_TNF_ENTER(tavor_rsrc_mrhdl_destructor);

        mrhdl = (tavor_mrhdl_t)mr;

        mutex_destroy(&mrhdl->mr_lock);

        TAVOR_TNF_ENTER(tavor_rsrc_mrhdl_destructor);
}


/*
 * tavor_rsrc_mcg_entry_get_size()
 */
static int
tavor_rsrc_mcg_entry_get_size(tavor_state_t *state, uint_t *mcg_size_shift)
{
        uint_t  num_qp_per_mcg, max_qp_per_mcg, log2;

        TAVOR_TNF_ENTER(tavor_rsrc_mcg_entry_get_size);

        /*
         * Round the configured number of QP per MCG to next larger
         * power-of-2 size and update.
         */
        num_qp_per_mcg = state->ts_cfg_profile->cp_num_qp_per_mcg + 8;
        log2 = highbit(num_qp_per_mcg);
        if (ISP2(num_qp_per_mcg)) {
  36    if ((num_qp_per_mcg & (num_qp_per_mcg - 1)) == 0) {
                log2 = log2 - 1;
        }
        state->ts_cfg_profile->cp_num_qp_per_mcg = (1 << log2) - 8;

        /* Now make sure number of QP per MCG makes sense */
        num_qp_per_mcg = state->ts_cfg_profile->cp_num_qp_per_mcg;
        max_qp_per_mcg = (1 << state->ts_devlim.log_max_qp_mcg);
        if (num_qp_per_mcg > max_qp_per_mcg) {
```

```
2899                 TNF_PROBE_1(tavor_rsrc_mcg_getsz_toomany_qppermcg_fail,
2900                     TAVOR_TNF_ERROR, "", tnf_uint, maxqpmcg, max_qp_per_mcg);
2901                 TAVOR_TNF_EXIT(tavor_rsrc_mcg_entry_get_size);
2902                 return (DDI_FAILURE);
2903         }

2905         /* Return the (shift) size of an individual MCG HW entry */
2906         *mcg_size_shift = log2 + 2;

2908         TAVOR_TNF_EXIT(tavor_rsrc_mcg_entry_get_size);
2909         return (DDI_SUCCESS);
2910 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   37154 Thu Oct 23 10:42:14 2014
new/usr/src/uts/common/io/ib/adapters/tavor/tavor_srq.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * tavor_srq.c
 *    Tavor Shared Receive Queue Processing Routines
 *
 *    Implements all the routines necessary for allocating, freeing, querying,
 *    modifying and posting shared receive queues.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/bitmap.h>

#include <sys/ib/adapters/tavor/tavor.h>

static void tavor_srq_sgl_to_logwqesz(tavor_state_t *state, uint_t num_sgl,
    tavor_qp_wq_type_t wq_type, uint_t *logwqesz, uint_t *max_sgl);

/*
 * tavor_srq_alloc()
 *    Context: Can be called only from user or kernel context.
 */
int
tavor_srq_alloc(tavor_state_t *state, tavor_srq_info_t *srqinfo,
    uint_t sleepflag, tavor_srq_options_t *op)
{
        ibt_srq_hdl_t           ibt_srqhdl;
        tavor_pdhdl_t           pd;
        ibt_srq_sizes_t         *sizes;
        ibt_srq_sizes_t         *real_sizes;
        tavor_srqhdl_t          *srqhdl;
```



```
        ibt_srq_flags_t         flags;
        tavor_rsrc_t            *srqc, *rsrc;
        tavor_hw_srqc_t         srqc_entry;
        uint32_t                *buf;
        tavor_srqhdl_t          srq;
        tavor_umap_db_entry_t   *umapdb;
        ibt_mr_attr_t           mr_attr;
        tavor_mr_options_t      mr_op;
        tavor_mrhdl_t           mr;
        uint64_t                addr;
        uint64_t                value, srq_desc_off;
        uint32_t                lkey;
        uint32_t                log_srq_size;
        uint32_t                uarpg;
        uint_t                  wq_location, dma_xfer_mode, srq_is_umap;
        int                     flag, status;
        char                    *errormsg;
        uint_t                  max_sgl;
        uint_t                  wqesz;

        _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*sizes))

        TAVOR_TNF_ENTER(tavor_srq_alloc);

        /*
         * Check the "options" flag.  Currently this flag tells the driver
         * whether or not the SRQ's work queues should be come from normal
         * system memory or whether they should be allocated from DDR memory.
         */
        if (op == NULL) {
                wq_location = TAVOR_QUEUE_LOCATION_NORMAL;
        } else {
                wq_location = op->srqo_wq_loc;
        }

        /*
         * Extract the necessary info from the tavor_srq_info_t structure
         */
        real_sizes = srqinfo->srqi_real_sizes;
        sizes      = srqinfo->srqi_sizes;
        pd         = srqinfo->srqi_pd;
        ibt_srqhdl = srqinfo->srqi_ibt_srqhdl;
        flags      = srqinfo->srqi_flags;
        srqhdl     = srqinfo->srqi_srqhdl;

        /*
         * Determine whether SRQ is being allocated for userland access or
         * whether it is being allocated for kernel access.  If the SRQ is
         * being allocated for userland access, then lookup the UAR doorbell
         * page number for the current process.  Note:  If this is not found
         * (e.g. if the process has not previously open()'d the Tavor driver),
         * then an error is returned.
         */
        srq_is_umap = (flags & IBT_SRQ_USER_MAP) ? 1 : 0;
        if (srq_is_umap) {
                status = tavor_umap_db_find(state->ts_instance, ddi_get_pid(),
                    MLNX_UMAP_UARPG_RSRC, &value, 0, NULL);
                if (status != DDI_SUCCESS) {
                        /* Set "status" and "errormsg" and goto failure */
                        TAVOR_TNF_FAIL(IBT_INVALID_PARAM, "failed UAR page");
                        goto srqalloc_fail3;
                }
                uarpg = ((tavor_rsrc_t *)(uintptr_t)value)->tr_indx;
        }

        /* Increase PD refcnt */
```

```
128         tavor_pd_refcnt_inc(pd);
130         /* Allocate an SRQ context entry */
131         status = tavor_rsrc_alloc(state, TAVOR_SRQC, 1, sleepflag, &srqc);
132         if (status != DDI_SUCCESS) {
133                 /* Set "status" and "errormsg" and goto failure */
134                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed SRQ context");
135                 goto srqalloc_fail1;
136         }
138         /* Allocate the SRQ Handle entry */
139         status = tavor_rsrc_alloc(state, TAVOR_SRQHDL, 1, sleepflag, &rsrc);
140         if (status != DDI_SUCCESS) {
141                 /* Set "status" and "errormsg" and goto failure */
142                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed SRQ handle");
143                 goto srqalloc_fail2;
144         }
146         srq = (tavor_srqhdl_t)rsrc->tr_addr;
147         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*srq))
149         srq->srq_srqnum = srqc->tr_indx;        /* just use index */
151         /*
152          * If this will be a user-mappable SRQ, then allocate an entry for
153          * the "userland resources database".  This will later be added to
154          * the database (after all further SRQ operations are successful).
155          * If we fail here, we must undo the reference counts and the
156          * previous resource allocation.
157          */
158         if (srq_is_umap) {
159                 umapdb = tavor_umap_db_alloc(state->ts_instance,
160                     srq->srq_srqnum, MLNX_UMAP_SRQMEM_RSRC,
161                     (uint64_t)(uintptr_t)rsrc);
162                 if (umapdb == NULL) {
163                         /* Set "status" and "errormsg" and goto failure */
164                         TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed umap add");
165                         goto srqalloc_fail3;
166                 }
167         }
169         /*
170          * Calculate the appropriate size for the SRQ.
171          * Note:  All Tavor SRQs must be a power-of-2 in size.  Also
172          * they may not be any smaller than TAVOR_SRQ_MIN_SIZE.  This step
173          * is to round the requested size up to the next highest power-of-2
174          */
175         sizes->srq_wr_sz = max(sizes->srq_wr_sz, TAVOR_SRQ_MIN_SIZE);
176         log_srq_size = highbit(sizes->srq_wr_sz);
177         if (ISP2(sizes->srq_wr_sz)) {
 35         if ((sizes->srq_wr_sz & (sizes->srq_wr_sz - 1)) == 0) {
178                 log_srq_size = log_srq_size - 1;
179         }
181         /*
182          * Next we verify that the rounded-up size is valid (i.e. consistent
183          * with the device limits and/or software-configured limits).  If not,
184          * then obviously we have a lot of cleanup to do before returning.
185          */
186         if (log_srq_size > state->ts_cfg_profile->cp_log_max_srq_sz) {
187                 /* Set "status" and "errormsg" and goto failure */
188                 TAVOR_TNF_FAIL(IBT_HCA_WR_EXCEEDED, "max SRQ size");
189                 goto srqalloc_fail4;
190         }
192         /*
```



```
193          * Next we verify that the requested number of SGL is valid (i.e.
194          * consistent with the device limits and/or software-configured
195          * limits).  If not, then obviously the same cleanup needs to be done.
196          */
197         max_sgl = state->ts_cfg_profile->cp_srq_max_sgl;
198         if (sizes->srq_sgl_sz > max_sgl) {
199                 /* Set "status" and "errormsg" and goto failure */
200                 TAVOR_TNF_FAIL(IBT_HCA_SGL_EXCEEDED, "max SRQ SGL");
201                 goto srqalloc_fail4;
202         }
204         /*
205          * Determine the SRQ's WQE sizes.  This depends on the requested
206          * number of SGLs.  Note: This also has the side-effect of
207          * calculating the real number of SGLs (for the calculated WQE size)
208          */
209         tavor_srq_sgl_to_logwqesz(state, sizes->srq_sgl_sz,
210             TAVOR_QP_WQ_TYPE_RECVQ, &srq->srq_wq_log_wqesz,
211             &srq->srq_wq_sgl);
213         /*
214          * Allocate the memory for SRQ work queues.  Note:  The location from
215          * which we will allocate these work queues has been passed in through
216          * the tavor_qp_options_t structure.  Since Tavor work queues are not
217          * allowed to cross a 32-bit (4GB) boundary, the alignment of the work
218          * queue memory is very important.  We used to allocate work queues
219          * (the combined receive and send queues) so that they would be aligned
220          * on their combined size.  That alignment guaranteed that they would
221          * never cross the 4GB boundary (Tavor work queues are on the order of
222          * MBs at maximum).  Now we are able to relax this alignment constraint
223          * by ensuring that the IB address assigned to the queue memory (as a
224          * result of the tavor_mr_register() call) is offset from zero.
225          * Previously, we had wanted to use the ddi_dma_mem_alloc() routine to
226          * guarantee the alignment, but when attempting to use IOMMU bypass
227          * mode we found that we were not allowed to specify any alignment that
228          * was more restrictive than the system page size.  So we avoided this
229          * constraint by passing two alignment values, one for the memory
230          * allocation itself and the other for the DMA handle (for later bind).
231          * This used to cause more memory than necessary to be allocated (in
232          * order to guarantee the more restrictive alignment contraint).  But
233          * be guaranteeing the zero-based IB virtual address for the queue, we
234          * are able to conserve this memory.
235          *
236          * Note: If SRQ is not user-mappable, then it may come from either
237          * kernel system memory or from HCA-attached local DDR memory.
238          *
239          * Note2: We align this queue on a pagesize boundary.  This is required
240          * to make sure that all the resulting IB addresses will start at 0, for
241          * a zero-based queue.  By making sure we are aligned on at least a
242          * page, any offset we use into our queue will be the same as when we
243          * perform tavor_srq_modify() operations later.
244          */
245         wqesz = (1 << srq->srq_wq_log_wqesz);
246         srq->srq_wqinfo.qa_size = (1 << log_srq_size) * wqesz;
247         srq->srq_wqinfo.qa_alloc_align = PAGESIZE;
248         srq->srq_wqinfo.qa_bind_align = PAGESIZE;
249         if (srq_is_umap) {
250                 srq->srq_wqinfo.qa_location = TAVOR_QUEUE_LOCATION_USERLAND;
251         } else {
252                 srq->srq_wqinfo.qa_location = wq_location;
253         }
254         status = tavor_queue_alloc(state, &srq->srq_wqinfo, sleepflag);
255         if (status != DDI_SUCCESS) {
256                 /* Set "status" and "errormsg" and goto failure */
257                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed srq");
258                 goto srqalloc_fail4;
```

```
259         }
260         buf = (uint32_t *)srq->srq_wqinfo.qa_buf_aligned;
261         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*buf))

263         /*
264          * Register the memory for the SRQ work queues.  The memory for the SRQ
265          * must be registered in the Tavor TPT tables.  This gives us the LKey
266          * to specify in the SRQ context later.  Note: If the work queue is to
267          * be allocated from DDR memory, then only a "bypass" mapping is
268          * appropriate.  And if the SRQ memory is user-mappable, then we force
269          * DDI_DMA_CONSISTENT mapping.  Also, in order to meet the alignment
270          * restriction, we pass the "mro_bind_override_addr" flag in the call
271          * to tavor_mr_register().  This guarantees that the resulting IB vaddr
272          * will be zero-based (modulo the offset into the first page).  If we
273          * fail here, we still have the bunch of resource and reference count
274          * cleanup to do.
275          */
276         flag = (sleepflag == TAVOR_SLEEP) ? IBT_MR_SLEEP :
277             IBT_MR_NOSLEEP;
278         mr_attr.mr_vaddr = (uint64_t)(uintptr_t)buf;
279         mr_attr.mr_len   = srq->srq_wqinfo.qa_size;
280         mr_attr.mr_as    = NULL;
281         mr_attr.mr_flags = flag | IBT_MR_ENABLE_LOCAL_WRITE;
282         if (srq_is_umap) {
283                 mr_op.mro_bind_type   = state->ts_cfg_profile->cp_iommu_bypass;
284         } else {
285                 if (wq_location == TAVOR_QUEUE_LOCATION_NORMAL) {
286                         mr_op.mro_bind_type =
287                             state->ts_cfg_profile->cp_iommu_bypass;
288                         dma_xfer_mode =
289                             state->ts_cfg_profile->cp_streaming_consistent;
290                         if (dma_xfer_mode == DDI_DMA_STREAMING) {
291                                 mr_attr.mr_flags |= IBT_MR_NONCOHERENT;
292                         }
293                 } else {
294                         mr_op.mro_bind_type = TAVOR_BINDMEM_BYPASS;
295                 }
296         }
297         mr_op.mro_bind_dmahdl = srq->srq_wqinfo.qa_dmahdl;
298         mr_op.mro_bind_override_addr = 1;
299         status = tavor_mr_register(state, pd, &mr_attr, &mr, &mr_op);
300         if (status != DDI_SUCCESS) {
301                 /* Set "status" and "errormsg" and goto failure */
302                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed register mr");
303                 goto srqalloc_fail5;
304         }
305         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*mr))
306         addr = mr->mr_bindinfo.bi_addr;
307         lkey = mr->mr_lkey;

309         /*
310          * Calculate the offset between the kernel virtual address space
311          * and the IB virtual address space.  This will be used when
312          * posting work requests to properly initialize each WQE.
313          */
314         srq_desc_off = (uint64_t)(uintptr_t)srq->srq_wqinfo.qa_buf_aligned -
315             (uint64_t)mr->mr_bindinfo.bi_addr;

317         /*
318          * Create WQL and Wridlist for use by this SRQ
319          */
320         srq->srq_wrid_wql = tavor_wrid_wql_create(state);
321         if (srq->srq_wrid_wql == NULL) {
322                 /* Set "status" and "errormsg" and goto failure */
323                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed wql create");
324                 goto srqalloc_fail6;
```



```
325         }
326         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*(srq->srq_wrid_wql)))

328         srq->srq_wridlist = tavor_wrid_get_list(1 << log_srq_size);
329         if (srq->srq_wridlist == NULL) {
330                 /* Set "status" and "errormsg" and goto failure */
331                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed wridlist create");
332                 goto srqalloc_fail7;
333         }
334         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*(srq->srq_wridlist)))

336         srq->srq_wridlist->wl_srq_en = 1;
337         srq->srq_wridlist->wl_free_list_indx = -1;

339         /*
340          * Fill in all the return arguments (if necessary).  This includes
341          * real queue size and real SGLs.
342          */
343         if (real_sizes != NULL) {
344                 real_sizes->srq_wr_sz = (1 << log_srq_size);
345                 real_sizes->srq_sgl_sz = srq->srq_wq_sgl;
346         }

348         /*
349          * Fill in the SRQC entry.  This is the final step before passing
350          * ownership of the SRQC entry to the Tavor hardware.  We use all of
351          * the information collected/calculated above to fill in the
352          * requisite portions of the SRQC.  Note: If this SRQ is going to be
353          * used for userland access, then we need to set the UAR page number
354          * appropriately (otherwise it's a "don't care")
355          */
356         bzero(&srqc_entry, sizeof (tavor_hw_srqc_t));
357         srqc_entry.wqe_addr_h      = (addr >> 32);
358         srqc_entry.next_wqe_addr_l = 0;
359         srqc_entry.ds              = (wqesz >> 4);
360         srqc_entry.state           = TAVOR_SRQ_STATE_HW_OWNER;
361         srqc_entry.pd              = pd->pd_pdnum;
362         srqc_entry.lkey            = lkey;
363         srqc_entry.wqe_cnt         = 0;
364         if (srq_is_umap) {
365                 srqc_entry.uar     = uarpg;
366         } else {
367                 srqc_entry.uar     = 0;
368         }

370         /*
371          * Write the SRQC entry to hardware.  Lastly, we pass ownership of
372          * the entry to the hardware (using the Tavor SW2HW_SRQ firmware
373          * command).  Note: In general, this operation shouldn't fail.  But
374          * if it does, we have to undo everything we've done above before
375          * returning error.
376          */
377         status = tavor_cmn_ownership_cmd_post(state, SW2HW_SRQ, &srqc_entry,
378             sizeof (tavor_hw_srqc_t), srq->srq_srqnum,
379             sleepflag);
380         if (status != TAVOR_CMD_SUCCESS) {
381                 cmn_err(CE_CONT, "Tavor: SW2HW_SRQ command failed: %08x\n",
382                     status);
383                 TNF_PROBE_1(tavor_srq_alloc_sw2hw_srq_cmd_fail,
384                     TAVOR_TNF_ERROR, "", tnf_uint, status, status);
385                 /* Set "status" and "errormsg" and goto failure */
386                 TAVOR_TNF_FAIL(IBT_FAILURE, "tavor SW2HW_SRQ command");
387                 goto srqalloc_fail8;
388         }

390         /*
```

```
391          * Fill in the rest of the Tavor SRQ handle.  We can update
392          * the following fields for use in further operations on the SRQ.
393          */
394         srq->srq_srqcrsrcp = srqc;
395         srq->srq_rsrcp     = rsrc;
396         srq->srq_mrhdl     = mr;
397         srq->srq_refcnt    = 0;
398         srq->srq_is_umap   = srq_is_umap;
399         srq->srq_uarpg     = (srq->srq_is_umap) ? uarpg : 0;
400         srq->srq_umap_dhp  = (devmap_cookie_t)NULL;
401         srq->srq_pdhdl     = pd;
402         srq->srq_wq_lastwqeindx = -1;
403         srq->srq_wq_bufsz  = (1 << log_srq_size);
404         srq->srq_wq_buf    = buf;
405         srq->srq_desc_off  = srq_desc_off;
406         srq->srq_hdlrarg   = (void *)ibt_srqhdl;
407         srq->srq_state     = 0;
408         srq->srq_real_sizes.srq_wr_sz = (1 << log_srq_size);
409         srq->srq_real_sizes.srq_sgl_sz = srq->srq_wq_sgl;

411         /* Determine if later ddi_dma_sync will be necessary */
412         srq->srq_sync = TAVOR_SRQ_IS_SYNC_REQ(state, srq->srq_wqinfo);

414         /*
415          * Put SRQ handle in Tavor SRQNum-to-SRQhdl list.  Then fill in the
416          * "srqhdl" and return success
417          */
418         ASSERT(state->ts_srqhdl[srqc->tr_indx] == NULL);
419         state->ts_srqhdl[srqc->tr_indx] = srq;

421         /*
422          * If this is a user-mappable SRQ, then we need to insert the
423          * previously allocated entry into the "userland resources database".
424          * This will allow for later lookup during devmap() (i.e. mmap())
425          * calls.
426          */
427         if (srq->srq_is_umap) {
428                 tavor_umap_db_add(umapdb);
429         } else {
430                 mutex_enter(&srq->srq_wrid_wql->wql_lock);
431                 tavor_wrid_list_srq_init(srq->srq_wridlist, srq, 0);
432                 mutex_exit(&srq->srq_wrid_wql->wql_lock);
433         }

435         *srqhdl = srq;

437         TAVOR_TNF_EXIT(tavor_srq_alloc);
438         return (status);

440 /*
441  * The following is cleanup for all possible failure cases in this routine
442  */
443 srqalloc_fail8:
444         kmem_free(srq->srq_wridlist->wl_wre, srq->srq_wridlist->wl_size *
445             sizeof (tavor_wrid_entry_t));
446         kmem_free(srq->srq_wridlist, sizeof (tavor_wrid_list_hdr_t));
447 srqalloc_fail7:
448         tavor_wql_refcnt_dec(srq->srq_wrid_wql);
449 srqalloc_fail6:
450         if (tavor_mr_deregister(state, &mr, TAVOR_MR_DEREG_ALL,
451             TAVOR_SLEEPFLAG_FOR_CONTEXT()) != DDI_SUCCESS) {
452                 TAVOR_WARNING(state, "failed to deregister SRQ memory");
453         }
454 srqalloc_fail5:
455         tavor_queue_free(state, &srq->srq_wqinfo);
456 srqalloc_fail4:
```



```
457         if (srq_is_umap) {
458                 tavor_umap_db_free(umapdb);
459         }
460 srqalloc_fail3:
461         tavor_rsrc_free(state, &rsrc);
462 srqalloc_fail2:
463         tavor_rsrc_free(state, &srqc);
464 srqalloc_fail1:
465         tavor_pd_refcnt_dec(pd);
466 srqalloc_fail:
467         TNF_PROBE_1(tavor_srq_alloc_fail, TAVOR_TNF_ERROR, "",
468             tnf_string, msg, errormsg);
469         TAVOR_TNF_EXIT(tavor_srq_alloc);
470         return (status);
471 }
```

*unchanged_portion_omitted_*

```
635 /*
636  * tavor_srq_modify()
637  *    Context: Can be called only from user or kernel context.
638  */
639 int
640 tavor_srq_modify(tavor_state_t *state, tavor_srqhdl_t srq, uint_t size,
641     uint_t *real_size, uint_t sleepflag)
642 {
643         tavor_qalloc_info_t     new_srqinfo, old_srqinfo;
644         tavor_rsrc_t            *mtt, *mpt, *old_mtt;
645         tavor_bind_info_t       bind;
646         tavor_bind_info_t       old_bind;
647         tavor_rsrc_pool_info_t  *rsrc_pool;
648         tavor_mrhdl_t           mr;
649         tavor_hw_mpt_t          mpt_entry;
650         tavor_wrid_entry_t      *wre_new, *wre_old;
651         uint64_t                mtt_ddrbaseaddr, mtt_addr;
652         uint64_t                srq_desc_off;
653         uint32_t                *buf, srq_old_bufsz;
654         uint32_t                wqesz;
655         uint_t                  max_srq_size;
656         uint_t                  dma_xfer_mode, mtt_pgsize_bits;
657         uint_t                  srq_sync, log_srq_size, maxprot;
658         uint_t                  wq_location;
659         int                     status;
660         char                    *errormsg;

662         TAVOR_TNF_ENTER(tavor_srq_modify);

664         /*
665          * Check the "inddr" flag.  This flag tells the driver whether or not
666          * the SRQ's work queues should be come from normal system memory or
667          * whether they should be allocated from DDR memory.
668          */
669         wq_location = state->ts_cfg_profile->cp_srq_wq_inddr;

671         /*
672          * If size requested is larger than device capability, return
673          * Insufficient Resources
674          */
675         max_srq_size = (1 << state->ts_cfg_profile->cp_log_max_srq_sz);
676         if (size > max_srq_size) {
677                 TNF_PROBE_0(tavor_srq_modify_size_larger_than_maxsize,
678                     TAVOR_TNF_ERROR, "");
679                 TAVOR_TNF_EXIT(tavor_srq_modify);
680                 return (IBT_HCA_WR_EXCEEDED);
681         }
```

```
683         /*
684          * Calculate the appropriate size for the SRQ.
685          * Note:  All Tavor SRQs must be a power-of-2 in size.  Also
686          * they may not be any smaller than TAVOR_SRQ_MIN_SIZE.  This step
687          * is to round the requested size up to the next highest power-of-2
688          */
689         size = max(size, TAVOR_SRQ_MIN_SIZE);
690         log_srq_size = highbit(size);
691         if (ISP2(size)) {
549         if ((size & (size - 1)) == 0) {
692                 log_srq_size = log_srq_size - 1;
693         }

695         /*
696          * Next we verify that the rounded-up size is valid (i.e. consistent
697          * with the device limits and/or software-configured limits).
698          */
699         if (log_srq_size > state->ts_cfg_profile->cp_log_max_srq_sz) {
700                 /* Set "status" and "errormsg" and goto failure */
701                 TAVOR_TNF_FAIL(IBT_HCA_WR_EXCEEDED, "max SRQ size");
702                 goto srqmodify_fail;
703         }

705         /*
706          * Allocate the memory for newly resized Shared Receive Queue.
707          *
708          * Note: If SRQ is not user-mappable, then it may come from either
709          * kernel system memory or from HCA-attached local DDR memory.
710          *
711          * Note2: We align this queue on a pagesize boundary.  This is required
712          * to make sure that all the resulting IB addresses will start at 0,
713          * for a zero-based queue.  By making sure we are aligned on at least a
714          * page, any offset we use into our queue will be the same as it was
715          * when we allocated it at tavor_srq_alloc() time.
716          */
717         wqesz = (1 << srq->srq_wq_log_wqesz);
718         new_srqinfo.qa_size = (1 << log_srq_size) * wqesz;
719         new_srqinfo.qa_alloc_align = PAGESIZE;
720         new_srqinfo.qa_bind_align  = PAGESIZE;
721         if (srq->srq_is_umap) {
722                 new_srqinfo.qa_location = TAVOR_QUEUE_LOCATION_USERLAND;
723         } else {
724                 new_srqinfo.qa_location = wq_location;
725         }
726         status = tavor_queue_alloc(state, &new_srqinfo, sleepflag);
727         if (status != DDI_SUCCESS) {
728                 /* Set "status" and "errormsg" and goto failure */
729                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE, "failed srq");
730                 goto srqmodify_fail;
731         }
732         buf = (uint32_t *)new_srqinfo.qa_buf_aligned;
733         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*buf))

735         /*
736          * Allocate the memory for the new WRE list.  This will be used later
737          * when we resize the wridlist based on the new SRQ size.
738          */
739         wre_new = (tavor_wrid_entry_t *)kmem_zalloc((1 << log_srq_size) *
740             sizeof (tavor_wrid_entry_t), sleepflag);
741         if (wre_new == NULL) {
742                 /* Set "status" and "errormsg" and goto failure */
743                 TAVOR_TNF_FAIL(IBT_INSUFF_RESOURCE,
744                     "failed wre_new alloc");
745                 goto srqmodify_fail;
746         }
```



```
748         /*
749          * Fill in the "bind" struct.  This struct provides the majority
750          * of the information that will be used to distinguish between an
751          * "addr" binding (as is the case here) and a "buf" binding (see
752          * below).  The "bind" struct is later passed to tavor_mr_mem_bind()
753          * which does most of the "heavy lifting" for the Tavor memory
754          * registration routines.
755          */
756         _NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(bind))
757         bzero(&bind, sizeof (tavor_bind_info_t));
758         bind.bi_type  = TAVOR_BINDHDL_VADDR;
759         bind.bi_addr  = (uint64_t)(uintptr_t)buf;
760         bind.bi_len   = new_srqinfo.qa_size;
761         bind.bi_as    = NULL;
762         bind.bi_flags = sleepflag == TAVOR_SLEEP ? IBT_MR_SLEEP :
763             IBT_MR_NOSLEEP | IBT_MR_ENABLE_LOCAL_WRITE;
764         if (srq->srq_is_umap) {
765                 bind.bi_bypass = state->ts_cfg_profile->cp_iommu_bypass;
766         } else {
767                 if (wq_location == TAVOR_QUEUE_LOCATION_NORMAL) {
768                         bind.bi_bypass =
769                             state->ts_cfg_profile->cp_iommu_bypass;
770                         dma_xfer_mode =
771                             state->ts_cfg_profile->cp_streaming_consistent;
772                         if (dma_xfer_mode == DDI_DMA_STREAMING) {
773                                 bind.bi_flags |= IBT_MR_NONCOHERENT;
774                         }
775                 } else {
776                         bind.bi_bypass = TAVOR_BINDMEM_BYPASS;
777                 }
778         }
779         status = tavor_mr_mtt_bind(state, &bind, new_srqinfo.qa_dmahdl, &mtt,
780             &mtt_pgsize_bits);
781         if (status != DDI_SUCCESS) {
782                 /* Set "status" and "errormsg" and goto failure */
783                 TAVOR_TNF_FAIL(status, "failed mtt bind");
784                 kmem_free(wre_new, srq->srq_wq_bufsz *
785                     sizeof (tavor_wrid_entry_t));
786                 tavor_queue_free(state, &new_srqinfo);
787                 goto srqmodify_fail;
788         }

790         /*
791          * Calculate the offset between the kernel virtual address space
792          * and the IB virtual address space.  This will be used when
793          * posting work requests to properly initialize each WQE.
794          *
795          * Note: bind addr is zero-based (from alloc) so we calculate the
796          * correct new offset here.
797          */
798         bind.bi_addr = bind.bi_addr & ((1 << mtt_pgsize_bits) - 1);
799         srq_desc_off = (uint64_t)(uintptr_t)new_srqinfo.qa_buf_aligned -
800             (uint64_t)bind.bi_addr;

802         /*
803          * Get the base address for the MTT table.  This will be necessary
804          * below when we are modifying the MPT entry.
805          */
806         rsrc_pool = &state->ts_rsrc_hdl[TAVOR_MTT];
807         mtt_ddrbaseaddr = (uint64_t)(uintptr_t)rsrc_pool->rsrc_ddr_offset;

809         /*
810          * Fill in the MPT entry.  This is the final step before passing
811          * ownership of the MPT entry to the Tavor hardware.  We use all of
812          * the information collected/calculated above to fill in the
813          * requisite portions of the MPT.
```

```
814          */
815         bzero(&mpt_entry, sizeof (tavor_hw_mpt_t));
816         mpt_entry.reg_win_len   = bind.bi_len;
817         mtt_addr = mtt_ddrbaseaddr + (mtt->tr_indx << TAVOR_MTT_SIZE_SHIFT);
818         mpt_entry.mttseg_addr_h = mtt_addr >> 32;
819         mpt_entry.mttseg_addr_l = mtt_addr >> 6;

821         /*
822          * Now we grab the SRQ lock.  Since we will be updating the actual
823          * SRQ location and the producer/consumer indexes, we should hold
824          * the lock.
825          *
826          * We do a TAVOR_NOSLEEP here (and below), though, because we are
827          * holding the "srq_lock" and if we got raised to interrupt level
828          * by priority inversion, we would not want to block in this routine
829          * waiting for success.
830          */
831         mutex_enter(&srq->srq_lock);

833         /*
834          * Copy old entries to new buffer
835          */
836         srq_old_bufsz = srq->srq_wq_bufsz;
837         bcopy(srq->srq_wq_buf, buf, srq_old_bufsz * wqesz);

839         /* Determine if later ddi_dma_sync will be necessary */
840         srq_sync = TAVOR_SRQ_IS_SYNC_REQ(state, srq->srq_wqinfo);

842         /* Sync entire "new" SRQ for use by hardware (if necessary) */
843         if (srq_sync) {
844                 (void) ddi_dma_sync(bind.bi_dmahdl, 0,
845                     new_srqinfo.qa_size, DDI_DMA_SYNC_FORDEV);
846         }

848         /*
849          * Setup MPT information for use in the MODIFY_MPT command
850          */
851         mr = srq->srq_mrhdl;
852         mutex_enter(&mr->mr_lock);
853         mpt = srq->srq_mrhdl->mr_mptrsrcp;

855         /*
856          * MODIFY_MPT
857          *
858          * If this fails for any reason, then it is an indication that
859          * something (either in HW or SW) has gone seriously wrong.  So we
860          * print a warning message and return.
861          */
862         status = tavor_modify_mpt_cmd_post(state, &mpt_entry, mpt->tr_indx,
863             TAVOR_CMD_MODIFY_MPT_RESIZESRQ, sleepflag);
864         if (status != TAVOR_CMD_SUCCESS) {
865                 cmn_err(CE_CONT, "Tavor: MODIFY_MPT command failed: %08x\n",
866                     status);
867                 TNF_PROBE_1(tavor_mr_common_reg_sw2hw_mpt_cmd_fail,
868                     TAVOR_TNF_ERROR, "", tnf_uint, status, status);
869                 TAVOR_TNF_FAIL(status, "MODIFY_MPT command failed");
870                 (void) tavor_mr_mtt_unbind(state, &srq->srq_mrhdl->mr_bindinfo,
871                     srq->srq_mrhdl->mr_mttrsrcp);
872                 kmem_free(wre_new, srq->srq_wq_bufsz *
873                     sizeof (tavor_wrid_entry_t));
874                 tavor_queue_free(state, &new_srqinfo);
875                 mutex_exit(&mr->mr_lock);
876                 mutex_exit(&srq->srq_lock);
877                 return (ibc_get_ci_failure(0));
878         }
```



```
880         /*
881          * Update the Tavor Shared Receive Queue handle with all the new
882          * information.  At the same time, save away all the necessary
883          * information for freeing up the old resources
884          */
885         old_srqinfo        = srq->srq_wqinfo;
886         old_mtt            = srq->srq_mrhdl->mr_mttrsrcp;
887         bcopy(&srq->srq_mrhdl->mr_bindinfo, &old_bind,
888             sizeof (tavor_bind_info_t));

890         /* Now set the new info */
891         srq->srq_wqinfo    = new_srqinfo;
892         srq->srq_wq_buf    = buf;
893         srq->srq_wq_bufsz  = (1 << log_srq_size);
894         bcopy(&bind, &srq->srq_mrhdl->mr_bindinfo, sizeof (tavor_bind_info_t));
895         srq->srq_mrhdl->mr_mttrsrcp = mtt;
896         srq->srq_desc_off  = srq_desc_off;
897         srq->srq_real_sizes.srq_wr_sz = (1 << log_srq_size);

899         /* Update MR mtt pagesize */
900         mr->mr_logmttpgsz = mtt_pgsize_bits;
901         mutex_exit(&mr->mr_lock);

903 #ifdef __lock_lint
904         mutex_enter(&srq->srq_wrid_wql->wql_lock);
905 #else
906         if (srq->srq_wrid_wql != NULL) {
907                 mutex_enter(&srq->srq_wrid_wql->wql_lock);
908         }
909 #endif

911         /*
912          * Initialize new wridlist, if needed.
913          *
914          * If a wridlist already is setup on an SRQ (the QP associated with an
915          * SRQ has moved "from_reset") then we must update this wridlist based
916          * on the new SRQ size.  We allocate the new size of Work Request ID
917          * Entries, copy over the old entries to the new list, and
918          * re-initialize the srq wridlist in non-umap case
919          */
920         wre_old = NULL;
921         if (srq->srq_wridlist != NULL) {
922                 wre_old = srq->srq_wridlist->wl_wre;

924                 bcopy(wre_old, wre_new, srq_old_bufsz *
925                     sizeof (tavor_wrid_entry_t));

927                 /* Setup new sizes in wre */
928                 srq->srq_wridlist->wl_wre = wre_new;
929                 srq->srq_wridlist->wl_size = srq->srq_wq_bufsz;

931                 if (!srq->srq_is_umap) {
932                         tavor_wrid_list_srq_init(srq->srq_wridlist, srq,
933                             srq_old_bufsz);
934                 }
935         }

937 #ifdef __lock_lint
938         mutex_exit(&srq->srq_wrid_wql->wql_lock);
939 #else
940         if (srq->srq_wrid_wql != NULL) {
941                 mutex_exit(&srq->srq_wrid_wql->wql_lock);
942         }
943 #endif

945         /*
```

```
 946          * If "old" SRQ was a user-mappable SRQ that is currently mmap()'d out
 947          * to a user process, then we need to call devmap_devmem_remap() to
 948          * invalidate the mapping to the SRQ memory.  We also need to
 949          * invalidate the SRQ tracking information for the user mapping.
 950          *
 951          * Note: On failure, the remap really shouldn't ever happen.  So, if it
 952          * does, it is an indication that something has gone seriously wrong.
 953          * So we print a warning message and return error (knowing, of course,
 954          * that the "old" SRQ memory will be leaked)
 955          */
 956         if ((srq->srq_is_umap) && (srq->srq_umap_dhp != NULL)) {
 957                 maxprot = (PROT_READ | PROT_WRITE | PROT_USER);
 958                 status = devmap_devmem_remap(srq->srq_umap_dhp,
 959                     state->ts_dip, 0, 0, srq->srq_wqinfo.qa_size, maxprot,
 960                     DEVMAP_MAPPING_INVALID, NULL);
 961                 if (status != DDI_SUCCESS) {
 962                         mutex_exit(&srq->srq_lock);
 963                         TAVOR_WARNING(state, "failed in SRQ memory "
 964                             "devmap_devmem_remap()");
 965                         /* We can, however, free the memory for old wre */
 966                         if (wre_old != NULL) {
 967                                 kmem_free(wre_old, srq_old_bufsz *
 968                                     sizeof (tavor_wrid_entry_t));
 969                         }
 970                         TAVOR_TNF_EXIT(tavor_srq_modify);
 971                         return (ibc_get_ci_failure(0));
 972                 }
 973                 srq->srq_umap_dhp = (devmap_cookie_t)NULL;
 974         }

 976         /*
 977          * Drop the SRQ lock now.  The only thing left to do is to free up
 978          * the old resources.
 979          */
 980         mutex_exit(&srq->srq_lock);

 982         /*
 983          * Unbind the MTT entries.
 984          */
 985         status = tavor_mr_mtt_unbind(state, &old_bind, old_mtt);
 986         if (status != DDI_SUCCESS) {
 987                 TAVOR_WARNING(state, "failed to unbind old SRQ memory");
 988                 /* Set "status" and "errormsg" and goto failure */
 989                 TAVOR_TNF_FAIL(ibc_get_ci_failure(0),
 990                     "failed to unbind (old)");
 991                 goto srqmodify_fail;
 992         }

 994         /* Free the memory for old wre */
 995         if (wre_old != NULL) {
 996                 kmem_free(wre_old, srq_old_bufsz *
 997                     sizeof (tavor_wrid_entry_t));
 998         }

1000         /* Free the memory for the old SRQ */
1001         tavor_queue_free(state, &old_srqinfo);

1003         /*
1004          * Fill in the return arguments (if necessary).  This includes the
1005          * real new completion queue size.
1006          */
1007         if (real_size != NULL) {
1008                 *real_size = (1 << log_srq_size);
1009         }

1011         TAVOR_TNF_EXIT(tavor_srq_modify);
```



```
1012         return (DDI_SUCCESS);

1014 srqmodify_fail:
1015         TNF_PROBE_1(tavor_srq_modify_fail, TAVOR_TNF_ERROR, "",
1016             tnf_string, msg, errormsg);
1017         TAVOR_TNF_EXIT(tavor_srq_modify);
1018         return (status);
1019 }
```

_____unchanged_portion_omitted_

```
1083 /*
1084  * tavor_srq_sgl_to_logwqesz()
1085  *    Context: Can be called from interrupt or base context.
1086  */
1087 static void
1088 tavor_srq_sgl_to_logwqesz(tavor_state_t *state, uint_t num_sgl,
1089     tavor_qp_wq_type_t wq_type, uint_t *logwqesz, uint_t *max_sgl)
1090 {
1091         uint_t  max_size, log2, actual_sgl;

1093         TAVOR_TNF_ENTER(tavor_srq_sgl_to_logwqesz);

1095         switch (wq_type) {
1096         case TAVOR_QP_WQ_TYPE_RECVQ:
1097                 /*
1098                  * Use requested maximum SGL to calculate max descriptor size
1099                  * (while guaranteeing that the descriptor size is a
1100                  * power-of-2 cachelines).
1101                  */
1102                 max_size = (TAVOR_QP_WQE_MLX_RCV_HDRS + (num_sgl << 4));
1103                 log2 = highbit(max_size);
1104                 if (ISP2(max_size)) {
 962                 if ((max_size & (max_size - 1)) == 0) {
1105                         log2 = log2 - 1;
1106                 }

1108                 /* Make sure descriptor is at least the minimum size */
1109                 log2 = max(log2, TAVOR_QP_WQE_LOG_MINIMUM);

1111                 /* Calculate actual number of SGL (given WQE size) */
1112                 actual_sgl = ((1 << log2) - TAVOR_QP_WQE_MLX_RCV_HDRS) >> 4;
1113                 break;

1115         default:
1116                 TAVOR_WARNING(state, "unexpected work queue type");
1117                 TNF_PROBE_0(tavor_srq_sgl_to_logwqesz_inv_wqtype_fail,
1118                     TAVOR_TNF_ERROR, "");
1119                 break;
1120         }

1122         /* Fill in the return values */
1123         *logwqesz = log2;
1124         *max_sgl  = min(state->ts_cfg_profile->cp_srq_max_sgl, actual_sgl);

1126         TAVOR_TNF_EXIT(tavor_qp_sgl_to_logwqesz);
1127 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    7164 Thu Oct 23 10:42:14 2014
new/usr/src/uts/common/io/ib/clients/rds/rdssubr.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/ib/clients/rds/rds.h>
#include <sys/ib/clients/rds/rds_kstat.h>

#include <inet/ipclassifier.h>

struct rds_kstat_s rds_kstat = {
        {"rds_nports",                  KSTAT_DATA_ULONG},
        {"rds_nsessions",               KSTAT_DATA_ULONG},
        {"rds_tx_bytes",                KSTAT_DATA_ULONG},
        {"rds_tx_pkts",                 KSTAT_DATA_ULONG},
        {"rds_tx_errors",               KSTAT_DATA_ULONG},
        {"rds_rx_bytes",                KSTAT_DATA_ULONG},
        {"rds_rx_pkts",                 KSTAT_DATA_ULONG},
        {"rds_rx_pkts_pending",         KSTAT_DATA_ULONG},
        {"rds_rx_errors",               KSTAT_DATA_ULONG},
        {"rds_tx_acks",                 KSTAT_DATA_ULONG},
        {"rds_post_recv_buf_called",    KSTAT_DATA_ULONG},
        {"rds_stalls_triggered",        KSTAT_DATA_ULONG},
        {"rds_stalls_sent",             KSTAT_DATA_ULONG},
        {"rds_unstalls_triggered",      KSTAT_DATA_ULONG},
        {"rds_unstalls_sent",           KSTAT_DATA_ULONG},
        {"rds_stalls_recvd",            KSTAT_DATA_ULONG},
        {"rds_unstalls_recvd",          KSTAT_DATA_ULONG},
        {"rds_stalls_ignored",          KSTAT_DATA_ULONG},
        {"rds_enobufs",                 KSTAT_DATA_ULONG},
        {"rds_ewouldblocks",            KSTAT_DATA_ULONG},
        {"rds_failovers",               KSTAT_DATA_ULONG},
        {"rds_port_quota",              KSTAT_DATA_ULONG},
        {"rds_port_quota_adjusted",     KSTAT_DATA_ULONG},
};

kstat_t *rds_kstatsp;
static kmutex_t rds_kstat_mutex;
```



```
struct  kmem_cache      *rds_alloc_cache;

uint_t  rds_bind_fanout_size = RDS_BIND_FANOUT_SIZE;
rds_bf_t *rds_bind_fanout;

void
rds_increment_kstat(kstat_named_t *ksnp, boolean_t lock, uint_t num)
{
        if (lock)
                mutex_enter(&rds_kstat_mutex);
        ksnp->value.ul += num;
        if (lock)
                mutex_exit(&rds_kstat_mutex);
}

void
rds_decrement_kstat(kstat_named_t *ksnp, boolean_t lock, uint_t num)
{
        if (lock)
                mutex_enter(&rds_kstat_mutex);
        ksnp->value.ul -= num;
        if (lock)
                mutex_exit(&rds_kstat_mutex);
}

void
rds_set_kstat(kstat_named_t *ksnp, boolean_t lock, ulong_t num)
{
        if (lock)
                mutex_enter(&rds_kstat_mutex);
        ksnp->value.ul = num;
        if (lock)
                mutex_exit(&rds_kstat_mutex);
}

ulong_t
rds_get_kstat(kstat_named_t *ksnp, boolean_t lock)
{
        ulong_t value;

        if (lock)
                mutex_enter(&rds_kstat_mutex);
        value = ksnp->value.ul;
        if (lock)
                mutex_exit(&rds_kstat_mutex);

        return (value);
}


void
rds_fini()
{
        int     i;

        for (i = 0; i < rds_bind_fanout_size; i++) {
                mutex_destroy(&rds_bind_fanout[i].rds_bf_lock);
        }
        kmem_free(rds_bind_fanout, rds_bind_fanout_size * sizeof (rds_bf_t));

        kmem_cache_destroy(rds_alloc_cache);
        kstat_delete(rds_kstatsp);
}
```

```
128 void
129 rds_init()
130 {
131         rds_alloc_cache = kmem_cache_create("rds_alloc_cache",
132             sizeof (rds_t), 0, NULL, NULL, NULL, NULL, NULL, 0);
133         rds_hash_init();
134         /*
135          * kstats
136          */
137         rds_kstatsp = kstat_create("rds", 0,
138             "rds_kstat", "misc", KSTAT_TYPE_NAMED,
139             sizeof (rds_kstat) / sizeof (kstat_named_t),
140             KSTAT_FLAG_VIRTUAL | KSTAT_FLAG_WRITABLE);
141         if (rds_kstatsp != NULL) {
142                 rds_kstatsp->ks_lock = &rds_kstat_mutex;
143                 rds_kstatsp->ks_data = (void *)&rds_kstat;
144                 kstat_install(rds_kstatsp);
145         }
146 }

148 #define UINT_32_BITS 31
149 void
150 rds_hash_init()
151 {
152         int i;

154         if (!ISP2(rds_bind_fanout_size)) {
 26         if (rds_bind_fanout_size & (rds_bind_fanout_size - 1)) {
155                 /* Not a power of two. Round up to nearest power of two */
156                 for (i = 0; i < UINT_32_BITS; i++) {
157                         if (rds_bind_fanout_size < (1 << i))
158                                 break;
159                 }
160                 rds_bind_fanout_size = 1 << i;
161         }
162         rds_bind_fanout = kmem_zalloc(rds_bind_fanout_size *
163             sizeof (rds_bf_t), KM_SLEEP);
164         for (i = 0; i < rds_bind_fanout_size; i++) {
165                 mutex_init(&rds_bind_fanout[i].rds_bf_lock, NULL, MUTEX_DEFAULT,
166                     NULL);
167         }
168 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  159699 Thu Oct 23 10:42:15 2014
new/usr/src/uts/common/io/myri10ge/drv/myri10ge.c
5255 uts shouldn't open-code ISP2
**********************************************************
_____unchanged_portion_omitted_
2063 static int
2064 myri10ge_start_locked(struct myri10ge_priv *mgp)
2065 {
2066         myri10ge_cmd_t cmd;
2067         int status, big_pow2, i;
2068         volatile uint8_t *itable;

2070         status = DDI_SUCCESS;
2071         /* Allocate DMA resources and receive buffers */

2073         status = myri10ge_reset(mgp);
2074         if (status != 0) {
2075                 cmn_err(CE_WARN, "%s: failed reset\n", mgp->name);
2076                 return (DDI_FAILURE);
2077         }

2079         if (mgp->num_slices > 1) {
2080                 cmd.data0 = mgp->num_slices;
2081                 cmd.data1 = 1; /* use MSI-X */
2082                 status = myri10ge_send_cmd(mgp, MXGEFW_CMD_ENABLE_RSS_QUEUES,
2083                     &cmd);
2084                 if (status != 0) {
2085                         cmn_err(CE_WARN,
2086                             "%s: failed to set number of slices\n",
2087                             mgp->name);
2088                         goto abort_with_nothing;
2089                 }
2090                 /* setup the indirection table */
2091                 cmd.data0 = mgp->num_slices;
2092                 status = myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_RSS_TABLE_SIZE,
2093                     &cmd);

2095                 status |= myri10ge_send_cmd(mgp,
2096                     MXGEFW_CMD_GET_RSS_TABLE_OFFSET, &cmd);
2097                 if (status != 0) {
2098                         cmn_err(CE_WARN,
2099                             "%s: failed to setup rss tables\n", mgp->name);
2100                 }

2102                 /* just enable an identity mapping */
2103                 itable = mgp->sram + cmd.data0;
2104                 for (i = 0; i < mgp->num_slices; i++)
2105                         itable[i] = (uint8_t)i;

2107                 if (myri10ge_rss_hash & MYRI10GE_TOEPLITZ_HASH) {
2108                         status = myri10ge_init_toeplitz(mgp);
2109                         if (status != 0) {
2110                                 cmn_err(CE_WARN, "%s: failed to setup "
2111                                     "toeplitz tx hash table", mgp->name);
2112                                 goto abort_with_nothing;
2113                         }
2114                 }
2115                 cmd.data0 = 1;
2116                 cmd.data1 = myri10ge_rss_hash;
2117                 status = myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_RSS_ENABLE,
2118                     &cmd);
2119                 if (status != 0) {
2120                         cmn_err(CE_WARN,
2121                             "%s: failed to enable slices\n", mgp->name);
2122                         goto abort_with_toeplitz;
```



```
2123                 }
2124         }

2126         for (i = 0; i < mgp->num_slices; i++) {
2127                 status = myri10ge_setup_slice(&mgp->ss[i]);
2128                 if (status != 0)
2129                         goto abort_with_slices;
2130         }

2132         /*
2133          * Tell the MCP how many buffers he has, and to
2134          *  bring the ethernet interface up
2135          *
2136          * Firmware needs the big buff size as a power of 2.  Lie and
2137          * tell him the buffer is larger, because we only use 1
2138          * buffer/pkt, and the mtu will prevent overruns
2139          */
2140         big_pow2 = myri10ge_mtu + MXGEFW_PAD;
2141         while (!ISP2(big_pow2))
2141         while ((big_pow2 & (big_pow2 - 1)) != 0)
2142                 big_pow2++;

2144         /* now give firmware buffers sizes, and MTU */
2145         cmd.data0 = myri10ge_mtu;
2146         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_MTU, &cmd);
2147         cmd.data0 = myri10ge_small_bytes;
2148         status |=
2149             myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_SMALL_BUFFER_SIZE, &cmd);
2150         cmd.data0 = big_pow2;
2151         status |= myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_BIG_BUFFER_SIZE, &cmd);
2152         if (status) {
2153                 cmn_err(CE_WARN, "%s: Couldn't set buffer sizes\n", mgp->name);
2154                 goto abort_with_slices;
2155         }


2158         cmd.data0 = 1;
2159         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_TSO_MODE, &cmd);
2160         if (status) {
2161                 cmn_err(CE_WARN, "%s: unable to setup TSO (%d)\n",
2162                     mgp->name, status);
2163         } else {
2164                 mgp->features |= MYRI10GE_TSO;
2165         }

2167         mgp->link_state = -1;
2168         mgp->rdma_tags_available = 15;
2169         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_ETHERNET_UP, &cmd);
2170         if (status) {
2171                 cmn_err(CE_WARN, "%s: unable to start ethernet\n", mgp->name);
2172                 goto abort_with_slices;
2173         }
2174         mgp->running = MYRI10GE_ETH_RUNNING;
2175         return (DDI_SUCCESS);

2177 abort_with_slices:
2178         for (i = 0; i < mgp->num_slices; i++)
2179                 myri10ge_teardown_slice(&mgp->ss[i]);

2181         mgp->running = MYRI10GE_ETH_STOPPED;

2183 abort_with_toeplitz:
2184         if (mgp->toeplitz_hash_table != NULL) {
2185                 kmem_free(mgp->toeplitz_hash_table,
2186                     sizeof (uint32_t) * 12 * 256);
2187                 mgp->toeplitz_hash_table = NULL;
```

```
2188         }

2190 abort_with_nothing:
2191         return (DDI_FAILURE);
2192 }
______unchanged_portion_omitted_


5385 static int
5386 myri10ge_probe_slices(struct myri10ge_priv *mgp)
5387 {
5388         myri10ge_cmd_t cmd;
5389         int status;

5391         mgp->num_slices = 1;

5393         /* hit the board with a reset to ensure it is alive */
5394         (void) memset(&cmd, 0, sizeof (cmd));
5395         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_RESET, &cmd);
5396         if (status != 0) {
5397                 cmn_err(CE_WARN, "%s: failed reset\n", mgp->name);
5398                 return (ENXIO);
5399         }

5401         if (myri10ge_use_msix == 0)
5402                 return (0);

5404         /* tell it the size of the interrupt queues */
5405         cmd.data0 = mgp->max_intr_slots * sizeof (struct mcp_slot);
5406         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_SET_INTRQ_SIZE, &cmd);
5407         if (status != 0) {
5408                 cmn_err(CE_WARN, "%s: failed MXGEFW_CMD_SET_INTRQ_SIZE\n",
5409                     mgp->name);
5410                 return (ENXIO);
5411         }

5413         /* ask the maximum number of slices it supports */
5414         status = myri10ge_send_cmd(mgp, MXGEFW_CMD_GET_MAX_RSS_QUEUES,
5415             &cmd);
5416         if (status != 0)
5417                 return (0);

5419         mgp->num_slices = cmd.data0;

5421         /*
5422          * if the admin did not specify a limit to how many
5423          * slices we should use, cap it automatically to the
5424          * number of CPUs currently online
5425          */
5426         if (myri10ge_max_slices == -1)
5427                 myri10ge_max_slices = ncpus;

5429         if (mgp->num_slices > myri10ge_max_slices)
5430                 mgp->num_slices = myri10ge_max_slices;


5433         /*
5434          * Now try to allocate as many MSI-X vectors as we have
5435          * slices. We give up on MSI-X if we can only get a single
5436          * vector.
5437          */
5438         while (mgp->num_slices > 1) {
5439                 /* make sure it is a power of two */
5440                 while (!ISP2(mgp->num_slices))
5440                 while (mgp->num_slices & (mgp->num_slices - 1))
5441                         mgp->num_slices--;
```



```
5442                 if (mgp->num_slices == 1)
5443                         return (0);

5445                 status = myri10ge_add_intrs(mgp, 0);
5446                 if (status == 0) {
5447                         myri10ge_rem_intrs(mgp, 0);
5448                         if (mgp->intr_cnt == mgp->num_slices) {
5449                                 if (myri10ge_verbose)
5450                                         printf("Got %d slices!\n",
5451                                             mgp->num_slices);
5452                                 return (0);
5453                         }
5454                         mgp->num_slices = mgp->intr_cnt;
5455                 } else {
5456                         mgp->num_slices = mgp->num_slices / 2;
5457                 }
5458         }

5460         if (myri10ge_verbose)
5461                 printf("Got %d slices\n", mgp->num_slices);
5462         return (0);
5463 }
______unchanged_portion_omitted_
```

```
**********************************************************
   41650 Thu Oct 23 10:42:15 2014
new/usr/src/uts/common/io/ntxn/unm_gem.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_
```

```
 699 static void
 700 unm_check_options(unm_adapter *adapter)
 701 {
 702         int                     i, ring, tx_desc, rx_desc, rx_jdesc, maxrx;
 703         unm_recv_context_t      *recv_ctx;
 704         unm_rcv_desc_ctx_t      *rcv_desc;
 705         uint8_t                 revid = adapter->ahw.revision_id;
 706         dev_info_t              *dip = adapter->dip;

 708         /*
 709          * Reduce number of regular rcv desc to half on x86.
 710          */
 711         maxrx = MAX_RCV_DESCRIPTORS;
 712 #if !defined(_LP64)
 713         maxrx /= 2;
 714 #endif /* !_LP64 */

 716         verbmsg = ddi_prop_get_int(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
 717             dmesg_propname, 0);

 719         adapter->tx_bcopy_threshold = ddi_prop_get_int(DDI_DEV_T_ANY,
 720             dip, DDI_PROP_DONTPASS, txbcopythreshold_propname,
 721             UNM_TX_BCOPY_THRESHOLD);
 722         adapter->rx_bcopy_threshold = ddi_prop_get_int(DDI_DEV_T_ANY,
 723             dip, DDI_PROP_DONTPASS, rxbcopythreshold_propname,
 724             UNM_RX_BCOPY_THRESHOLD);

 726         tx_desc = ddi_prop_get_int(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
 727             txringsize_propname, MAX_CMD_DESCRIPTORS_HOST);
 728         if (tx_desc >= 256 && tx_desc <= MAX_CMD_DESCRIPTORS && ISP2(tx_desc)) {
 728         if (tx_desc >= 256 && tx_desc <= MAX_CMD_DESCRIPTORS &&
 729             !(tx_desc & (tx_desc - 1))) {
 729                 adapter->MaxTxDescCount = tx_desc;
 730         } else {
 731                 cmn_err(CE_WARN, "%s%d: TxRingSize defaulting to %d, since "
 732                     ".conf value is not 2 power aligned in range 256 - %d\n",
 733                     adapter->name, adapter->instance, MAX_CMD_DESCRIPTORS_HOST,
 734                     MAX_CMD_DESCRIPTORS);
 735                 adapter->MaxTxDescCount = MAX_CMD_DESCRIPTORS_HOST;
 736         }

 738         rx_desc = ddi_prop_get_int(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
 739             rxringsize_propname, maxrx);
 740         if (rx_desc >= NX_MIN_DRIVER_RDS_SIZE &&
 741             rx_desc <= NX_MAX_SUPPORTED_RDS_SIZE && ISP2(rx_desc)) {
 742             rx_desc <= NX_MAX_SUPPORTED_RDS_SIZE &&
 743             !(rx_desc & (rx_desc - 1))) {
 742                 adapter->MaxRxDescCount = rx_desc;
 743         } else {
 744                 cmn_err(CE_WARN, "%s%d: RxRingSize defaulting to %d, since "
 745                     ".conf value is not 2 power aligned in range %d - %d\n",
 746                     adapter->name, adapter->instance, MAX_RCV_DESCRIPTORS,
 747                     NX_MIN_DRIVER_RDS_SIZE, NX_MAX_SUPPORTED_RDS_SIZE);
 748                 adapter->MaxRxDescCount = MAX_RCV_DESCRIPTORS;
 749         }

 751         rx_jdesc = ddi_prop_get_int(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
 752             jumborxringsize_propname, MAX_JUMBO_RCV_DESCRIPTORS);
 753         if (rx_jdesc >= NX_MIN_DRIVER_RDS_SIZE &&
```



```
 754             rx_jdesc <= NX_MAX_SUPPORTED_JUMBO_RDS_SIZE && ISP2(rx_jdesc)) {
 756             rx_jdesc <= NX_MAX_SUPPORTED_JUMBO_RDS_SIZE &&
 757             !(rx_jdesc & (rx_jdesc - 1))) {
 755                 adapter->MaxJumboRxDescCount = rx_jdesc;
 756         } else {
 757                 cmn_err(CE_WARN, "%s%d: JumboRingSize defaulting to %d, since "
 758                     ".conf value is not 2 power aligned in range %d - %d\n",
 759                     adapter->name, adapter->instance, MAX_JUMBO_RCV_DESCRIPTORS,
 760                     NX_MIN_DRIVER_RDS_SIZE, NX_MAX_SUPPORTED_JUMBO_RDS_SIZE);
 761                 adapter->MaxJumboRxDescCount = MAX_JUMBO_RCV_DESCRIPTORS;
 762         }

 764         /*
 765          * Solaris does not use LRO, but older firmware needs to have a
 766          * couple of descriptors for initialization.
 767          */
 768         adapter->MaxLroRxDescCount = (adapter->fw_major < 4) ? 2 : 0;

 770         adapter->mtu = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
 771             DDI_PROP_DONTPASS, defaultmtu_propname, MTU_SIZE);

 773         if (adapter->mtu < MTU_SIZE) {
 774                 cmn_err(CE_WARN, "Raising mtu to %d\n", MTU_SIZE);
 775                 adapter->mtu = MTU_SIZE;
 776         }
 777         adapter->maxmtu = NX_IS_REVISION_P2(revid) ? P2_MAX_MTU : P3_MAX_MTU;
 778         if (adapter->mtu > adapter->maxmtu) {
 779                 cmn_err(CE_WARN, "Lowering mtu to %d\n", adapter->maxmtu);
 780                 adapter->mtu = adapter->maxmtu;
 781         }

 783         adapter->maxmtu = adapter->mtu + NX_MAX_ETHERHDR;

 785         /*
 786          * If we are not expecting to receive jumbo frames, save memory and
 787          * do not allocate.
 788          */
 789         if (adapter->mtu <= MTU_SIZE)
 790                 adapter->MaxJumboRxDescCount = NX_MIN_DRIVER_RDS_SIZE;

 792         for (i = 0; i < MAX_RCV_CTX; ++i) {
 793                 recv_ctx = &adapter->recv_ctx[i];

 795                 for (ring = 0; ring < adapter->max_rds_rings; ring++) {
 796                         rcv_desc = &recv_ctx->rcv_desc[ring];

 798                         switch (RCV_DESC_TYPE(ring)) {
 799                         case RCV_DESC_NORMAL:
 800                                 rcv_desc->MaxRxDescCount =
 801                                     adapter->MaxRxDescCount;
 802                                 if (adapter->ahw.cut_through) {
 803                                         rcv_desc->dma_size =
 804                                             NX_CT_DEFAULT_RX_BUF_LEN;
 805                                         rcv_desc->buf_size = rcv_desc->dma_size;
 806                                 } else {
 807                                         rcv_desc->dma_size =
 808                                             NX_RX_NORMAL_BUF_MAX_LEN;
 809                                         rcv_desc->buf_size =
 810                                             rcv_desc->dma_size +
 811                                             IP_ALIGNMENT_BYTES;
 812                                 }
 813                                 break;

 815                         case RCV_DESC_JUMBO:
 816                                 rcv_desc->MaxRxDescCount =
 817                                     adapter->MaxJumboRxDescCount;
```

```
818                                 if (adapter->ahw.cut_through) {
819                                         rcv_desc->dma_size =
820                                             rcv_desc->buf_size =
821                                             NX_P3_RX_JUMBO_BUF_MAX_LEN;
822                                 } else {
823                                         if (NX_IS_REVISION_P2(revid))
824                                                 rcv_desc->dma_size =
825                                                     NX_P2_RX_JUMBO_BUF_MAX_LEN;
826                                         else
827                                                 rcv_desc->dma_size =
828                                                     NX_P3_RX_JUMBO_BUF_MAX_LEN;
829                                         rcv_desc->buf_size =
830                                             rcv_desc->dma_size +
831                                             IP_ALIGNMENT_BYTES;
832                                 }
833                                 break;

835                         case RCV_RING_LRO:
836                                 rcv_desc->MaxRxDescCount =
837                                     adapter->MaxLroRxDescCount;
838                                 rcv_desc->buf_size = MAX_RX_LRO_BUFFER_LENGTH;
839                                 rcv_desc->dma_size = RX_LRO_DMA_MAP_LEN;
840                                 break;
841                         default:
842                                 break;
843                         }
844                 }
845         }
846 }
```

_____***unchanged_portion_omitted_***

****
111799 Thu Oct 23 10:42:15 2014
new/usr/src/uts/common/krtld/kobj.c
5255 uts shouldn't open-code ISP2
****
_____***unchanged_portion_omitted_***

```
3771 static int
3772 kobj_comp_setup(struct _buf *file, struct compinfo *cip)
3773 {
3774         struct comphdr *hdr;

3776         /*
3777          * read the compressed image into memory,
3778          * so we can deompress from there
3779          */
3780         file->_dsize = cip->fsize;
3781         file->_dbuf = kobj_alloc(cip->fsize, KM_WAIT|KM_TMP);
3782         if (kobj_read(file->_fd, file->_dbuf, cip->fsize, 0) != cip->fsize) {
3783                 kobj_free(file->_dbuf, cip->fsize);
3784                 return (-1);
3785         }

3787         hdr = kobj_comphdr(file);
3788         if (hdr->ch_magic != CH_MAGIC_ZLIB || hdr->ch_version != CH_VERSION ||
3789             hdr->ch_algorithm != CH_ALG_ZLIB || hdr->ch_fsize == 0 ||
3790             !ISP2(hdr->ch_blksize)) {
3790             (hdr->ch_blksize & (hdr->ch_blksize - 1)) != 0) {
3791                 kobj_free(file->_dbuf, cip->fsize);
3792                 return (-1);
3793         }
3794         file->_base = kobj_alloc(hdr->ch_blksize, KM_WAIT|KM_TMP);
3795         file->_bsize = hdr->ch_blksize;
3796         return (0);
3797 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    8951 Thu Oct 23 10:42:15 2014
new/usr/src/uts/common/os/group.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/systm.h>
#include <sys/param.h>
#include <sys/debug.h>
#include <sys/kmem.h>
#include <sys/group.h>
#include <sys/cmn_err.h>


#define GRP_SET_SIZE_DEFAULT 2

static void group_grow_set(group_t *);
static void group_shrink_set(group_t *);
static void group_pack_set(void **, uint_t);

/*
 * Initialize a group_t
 */
void
group_create(group_t *g)
{
        bzero(g, sizeof (group_t));
}

/*
 * Destroy a group_t
 * The group must already be empty
 */
void
group_destroy(group_t *g)
{
        ASSERT(g->grp_size == 0);

        if (g->grp_capacity > 0) {
                kmem_free(g->grp_set, g->grp_capacity * sizeof (void *));
```



```
                g->grp_capacity = 0;
        }
        g->grp_set = NULL;
}

/*
 * Empty a group_t
 * Capacity is preserved.
 */
void
group_empty(group_t *g)
{
        int     i;
        int     sz = g->grp_size;

        g->grp_size = 0;
        for (i = 0; i < sz; i++)
                g->grp_set[i] = NULL;
}

/*
 * Add element "e" to group "g"
 *
 * Returns -1 if addition would result in overcapacity, and
 * resize operations aren't allowed, and 0 otherwise
 */
int
group_add(group_t *g, void *e, int gflag)
{
        int     entry;

        if ((gflag & GRP_NORESIZE) &&
            g->grp_size == g->grp_capacity)
                return (-1);

        ASSERT(g->grp_size != g->grp_capacity || (gflag & GRP_RESIZE));

        entry = g->grp_size++;
        if (g->grp_size > g->grp_capacity)
                group_grow_set(g);

        ASSERT(g->grp_set[entry] == NULL);
        g->grp_set[entry] = e;

        return (0);
}

/*
 * Remove element "e" from group "g"
 *
 * Returns -1 if "e" was not present in "g" and 0 otherwise
 */
int
group_remove(group_t *g, void *e, int gflag)
{
        int     i;

        if (g->grp_size == 0)
                return (-1);

        /*
         * Find the element in the group's set
         */
        for (i = 0; i < g->grp_size; i++)
                if (g->grp_set[i] == e)
                        break;
```

```
128         if (g->grp_set[i] != e)
129                 return (-1);

131         g->grp_set[i] = NULL;
132         group_pack_set(g->grp_set, g->grp_size);
133         g->grp_size--;

135         if ((gflag & GRP_RESIZE) &&
136             g->grp_size > GRP_SET_SIZE_DEFAULT && ISP2(g->grp_size))
 26             g->grp_size > GRP_SET_SIZE_DEFAULT &&
 27             ((g->grp_size - 1) & g->grp_size) == 0)
137                 group_shrink_set(g);

139         return (0);
140 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  181686 Thu Oct 23 10:42:16 2014
new/usr/src/uts/common/os/kmem.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

3702 /*
3703  * It must be valid to call the destructor (if any) on a newly created object.
3704  * That is, the constructor (if any) must leave the object in a valid state for
3705  * the destructor.
3706  */
3707 kmem_cache_t *
3708 kmem_cache_create(
3709         char *name,             /* descriptive name for this cache */
3710         size_t bufsize,         /* size of the objects it manages */
3711         size_t align,           /* required object alignment */
3712         int (*constructor)(void *, void *, int), /* object constructor */
3713         void (*destructor)(void *, void *),     /* object destructor */
3714         void (*reclaim)(void *), /* memory reclaim callback */
3715         void *private,          /* pass-thru arg for constr/destr/reclaim */
3716         vmem_t *vmp,            /* vmem source for slab allocation */
3717         int cflags)             /* cache creation flags */
3718 {
3719         int cpu_seqid;
3720         size_t chunksize;
3721         kmem_cache_t *cp;
3722         kmem_magtype_t *mtp;
3723         size_t csize = KMEM_CACHE_SIZE(max_ncpus);

3725 #ifdef  DEBUG
3726         /*
3727          * Cache names should conform to the rules for valid C identifiers
3728          */
3729         if (!strident_valid(name)) {
3730                 cmn_err(CE_CONT,
3731                     "kmem_cache_create: '%s' is an invalid cache name\n"
3732                     "cache names must conform to the rules for "
3733                     "C identifiers\n", name);
3734         }
3735 #endif  /* DEBUG */

3737         if (vmp == NULL)
3738                 vmp = kmem_default_arena;

3740         /*
3741          * If this kmem cache has an identifier vmem arena as its source, mark
3742          * it such to allow kmem_reap_idspace().
3743          */
3744         ASSERT(!(cflags & KMC_IDENTIFIER));   /* consumer should not set this */
3745         if (vmp->vm_cflags & VMC_IDENTIFIER)
3746                 cflags |= KMC_IDENTIFIER;

3748         /*
3749          * Get a kmem_cache structure.  We arrange that cp->cache_cpu[]
3750          * is aligned on a KMEM_CPU_CACHE_SIZE boundary to prevent
3751          * false sharing of per-CPU data.
3752          */
3753         cp = vmem_xalloc(kmem_cache_arena, csize, KMEM_CPU_CACHE_SIZE,
3754             P2NPHASE(csize, KMEM_CPU_CACHE_SIZE), 0, NULL, NULL, VM_SLEEP);
3755         bzero(cp, csize);
3756         list_link_init(&cp->cache_link);

3758         if (align == 0)
3759                 align = KMEM_ALIGN;
```



```
3761         /*
3762          * If we're not at least KMEM_ALIGN aligned, we can't use free
3763          * memory to hold bufctl information (because we can't safely
3764          * perform word loads and stores on it).
3765          */
3766         if (align < KMEM_ALIGN)
3767                 cflags |= KMC_NOTOUCH;

3769         if (!ISP2(align) || align > vmp->vm_quantum)
3769         if ((align & (align - 1)) != 0 || align > vmp->vm_quantum)
3770                 panic("kmem_cache_create: bad alignment %lu", align);

3772         mutex_enter(&kmem_flags_lock);
3773         if (kmem_flags & KMF_RANDOMIZE)
3774                 kmem_flags = (((kmem_flags | ~KMF_RANDOM) + 1) & KMF_RANDOM) |
3775                     KMF_RANDOMIZE;
3776         cp->cache_flags = (kmem_flags | cflags) & KMF_DEBUG;
3777         mutex_exit(&kmem_flags_lock);

3779         /*
3780          * Make sure all the various flags are reasonable.
3781          */
3782         ASSERT(!(cflags & KMC_NOHASH) || !(cflags & KMC_NOTOUCH));

3784         if (cp->cache_flags & KMF_LITE) {
3785                 if (bufsize >= kmem_lite_minsize &&
3786                     align <= kmem_lite_maxalign &&
3787                     P2PHASE(bufsize, kmem_lite_maxalign) != 0) {
3788                         cp->cache_flags |= KMF_BUFTAG;
3789                         cp->cache_flags &= ~(KMF_AUDIT | KMF_FIREWALL);
3790                 } else {
3791                         cp->cache_flags &= ~KMF_DEBUG;
3792                 }
3793         }

3795         if (cp->cache_flags & KMF_DEADBEEF)
3796                 cp->cache_flags |= KMF_REDZONE;

3798         if ((cflags & KMC_QCACHE) && (cp->cache_flags & KMF_AUDIT))
3799                 cp->cache_flags |= KMF_NOMAGAZINE;

3801         if (cflags & KMC_NODEBUG)
3802                 cp->cache_flags &= ~KMF_DEBUG;

3804         if (cflags & KMC_NOTOUCH)
3805                 cp->cache_flags &= ~KMF_TOUCH;

3807         if (cflags & KMC_PREFILL)
3808                 cp->cache_flags |= KMF_PREFILL;

3810         if (cflags & KMC_NOHASH)
3811                 cp->cache_flags &= ~(KMF_AUDIT | KMF_FIREWALL);

3813         if (cflags & KMC_NOMAGAZINE)
3814                 cp->cache_flags |= KMF_NOMAGAZINE;

3816         if ((cp->cache_flags & KMF_AUDIT) && !(cflags & KMC_NOTOUCH))
3817                 cp->cache_flags |= KMF_REDZONE;

3819         if (!(cp->cache_flags & KMF_AUDIT))
3820                 cp->cache_flags &= ~KMF_CONTENTS;

3822         if ((cp->cache_flags & KMF_BUFTAG) && bufsize >= kmem_minfirewall &&
3823             !(cp->cache_flags & KMF_LITE) && !(cflags & KMC_NOHASH))
3824                 cp->cache_flags |= KMF_FIREWALL;
```

```
3826         if (vmp != kmem_default_arena || kmem_firewall_arena == NULL)
3827                 cp->cache_flags &= ~KMF_FIREWALL;

3829         if (cp->cache_flags & KMF_FIREWALL) {
3830                 cp->cache_flags &= ~KMF_BUFTAG;
3831                 cp->cache_flags |= KMF_NOMAGAZINE;
3832                 ASSERT(vmp == kmem_default_arena);
3833                 vmp = kmem_firewall_arena;
3834         }

3836         /*
3837          * Set cache properties.
3838          */
3839         (void) strncpy(cp->cache_name, name, KMEM_CACHE_NAMELEN);
3840         strident_canon(cp->cache_name, KMEM_CACHE_NAMELEN + 1);
3841         cp->cache_bufsize = bufsize;
3842         cp->cache_align = align;
3843         cp->cache_constructor = constructor;
3844         cp->cache_destructor = destructor;
3845         cp->cache_reclaim = reclaim;
3846         cp->cache_private = private;
3847         cp->cache_arena = vmp;
3848         cp->cache_cflags = cflags;

3850         /*
3851          * Determine the chunk size.
3852          */
3853         chunksize = bufsize;

3855         if (align >= KMEM_ALIGN) {
3856                 chunksize = P2ROUNDUP(chunksize, KMEM_ALIGN);
3857                 cp->cache_bufctl = chunksize - KMEM_ALIGN;
3858         }

3860         if (cp->cache_flags & KMF_BUFTAG) {
3861                 cp->cache_bufctl = chunksize;
3862                 cp->cache_buftag = chunksize;
3863                 if (cp->cache_flags & KMF_LITE)
3864                         chunksize += KMEM_BUFTAG_LITE_SIZE(kmem_lite_count);
3865                 else
3866                         chunksize += sizeof (kmem_buftag_t);
3867         }

3869         if (cp->cache_flags & KMF_DEADBEEF) {
3870                 cp->cache_verify = MIN(cp->cache_buftag, kmem_maxverify);
3871                 if (cp->cache_flags & KMF_LITE)
3872                         cp->cache_verify = sizeof (uint64_t);
3873         }

3875         cp->cache_contents = MIN(cp->cache_bufctl, kmem_content_maxsave);

3877         cp->cache_chunksize = chunksize = P2ROUNDUP(chunksize, align);

3879         /*
3880          * Now that we know the chunk size, determine the optimal slab size.
3881          */
3882         if (vmp == kmem_firewall_arena) {
3883                 cp->cache_slabsize = P2ROUNDUP(chunksize, vmp->vm_quantum);
3884                 cp->cache_mincolor = cp->cache_slabsize - chunksize;
3885                 cp->cache_maxcolor = cp->cache_mincolor;
3886                 cp->cache_flags |= KMF_HASH;
3887                 ASSERT(!(cp->cache_flags & KMF_BUFTAG));
3888         } else if ((cflags & KMC_NOHASH) || (!(cflags & KMC_NOTOUCH) &&
3889             !(cp->cache_flags & KMF_AUDIT) &&
3890             chunksize < vmp->vm_quantum / KMEM_VOID_FRACTION)) {
3891                 cp->cache_slabsize = vmp->vm_quantum;
```



```
3892                 cp->cache_mincolor = 0;
3893                 cp->cache_maxcolor =
3894                     (cp->cache_slabsize - sizeof (kmem_slab_t)) % chunksize;
3895                 ASSERT(chunksize + sizeof (kmem_slab_t) <= cp->cache_slabsize);
3896                 ASSERT(!(cp->cache_flags & KMF_AUDIT));
3897         } else {
3898                 size_t chunks, bestfit, waste, slabsize;
3899                 size_t minwaste = LONG_MAX;

3901                 for (chunks = 1; chunks <= KMEM_VOID_FRACTION; chunks++) {
3902                         slabsize = P2ROUNDUP(chunksize * chunks,
3903                             vmp->vm_quantum);
3904                         chunks = slabsize / chunksize;
3905                         waste = (slabsize % chunksize) / chunks;
3906                         if (waste < minwaste) {
3907                                 minwaste = waste;
3908                                 bestfit = slabsize;
3909                         }
3910                 }
3911                 if (cflags & KMC_QCACHE)
3912                         bestfit = VMEM_QCACHE_SLABSIZE(vmp->vm_qcache_max);
3913                 cp->cache_slabsize = bestfit;
3914                 cp->cache_mincolor = 0;
3915                 cp->cache_maxcolor = bestfit % chunksize;
3916                 cp->cache_flags |= KMF_HASH;
3917         }

3919         cp->cache_maxchunks = (cp->cache_slabsize / cp->cache_chunksize);
3920         cp->cache_partial_binshift = highbit(cp->cache_maxchunks / 16) + 1;

3922         /*
3923          * Disallowing prefill when either the DEBUG or HASH flag is set or when
3924          * there is a constructor avoids some tricky issues with debug setup
3925          * that may be revisited later. We cannot allow prefill in a
3926          * metadata cache because of potential recursion.
3927          */
3928         if (vmp == kmem_msb_arena ||
3929             cp->cache_flags & (KMF_HASH | KMF_BUFTAG) ||
3930             cp->cache_constructor != NULL)
3931                 cp->cache_flags &= ~KMF_PREFILL;

3933         if (cp->cache_flags & KMF_HASH) {
3934                 ASSERT(!(cflags & KMC_NOHASH));
3935                 cp->cache_bufctl_cache = (cp->cache_flags & KMF_AUDIT) ?
3936                     kmem_bufctl_audit_cache : kmem_bufctl_cache;
3937         }

3939         if (cp->cache_maxcolor >= vmp->vm_quantum)
3940                 cp->cache_maxcolor = vmp->vm_quantum - 1;

3942         cp->cache_color = cp->cache_mincolor;

3944         /*
3945          * Initialize the rest of the slab layer.
3946          */
3947         mutex_init(&cp->cache_lock, NULL, MUTEX_DEFAULT, NULL);

3949         avl_create(&cp->cache_partial_slabs, kmem_partial_slab_cmp,
3950             sizeof (kmem_slab_t), offsetof(kmem_slab_t, slab_link));
3951         /* LINTED: E_TRUE_LOGICAL_EXPR */
3952         ASSERT(sizeof (list_node_t) <= sizeof (avl_node_t));
3953         /* reuse partial slab AVL linkage for complete slab list linkage */
3954         list_create(&cp->cache_complete_slabs,
3955             sizeof (kmem_slab_t), offsetof(kmem_slab_t, slab_link));

3957         if (cp->cache_flags & KMF_HASH) {
```

```
3958                 cp->cache_hash_table = vmem_alloc(kmem_hash_arena,
3959                     KMEM_HASH_INITIAL * sizeof (void *), VM_SLEEP);
3960                 bzero(cp->cache_hash_table,
3961                     KMEM_HASH_INITIAL * sizeof (void *));
3962                 cp->cache_hash_mask = KMEM_HASH_INITIAL - 1;
3963                 cp->cache_hash_shift = highbit((ulong_t)chunksize) - 1;
3964         }

3966         /*
3967          * Initialize the depot.
3968          */
3969         mutex_init(&cp->cache_depot_lock, NULL, MUTEX_DEFAULT, NULL);

3971         for (mtp = kmem_magtype; chunksize <= mtp->mt_minbuf; mtp++)
3972                 continue;

3974         cp->cache_magtype = mtp;

3976         /*
3977          * Initialize the CPU layer.
3978          */
3979         for (cpu_seqid = 0; cpu_seqid < max_ncpus; cpu_seqid++) {
3980                 kmem_cpu_cache_t *ccp = &cp->cache_cpu[cpu_seqid];
3981                 mutex_init(&ccp->cc_lock, NULL, MUTEX_DEFAULT, NULL);
3982                 ccp->cc_flags = cp->cache_flags;
3983                 ccp->cc_rounds = -1;
3984                 ccp->cc_prounds = -1;
3985         }

3987         /*
3988          * Create the cache's kstats.
3989          */
3990         if ((cp->cache_kstat = kstat_create("unix", 0, cp->cache_name,
3991             "kmem_cache", KSTAT_TYPE_NAMED,
3992             sizeof (kmem_cache_kstat) / sizeof (kstat_named_t),
3993             KSTAT_FLAG_VIRTUAL)) != NULL) {
3994                 cp->cache_kstat->ks_data = &kmem_cache_kstat;
3995                 cp->cache_kstat->ks_update = kmem_cache_kstat_update;
3996                 cp->cache_kstat->ks_private = cp;
3997                 cp->cache_kstat->ks_lock = &kmem_cache_kstat_lock;
3998                 kstat_install(cp->cache_kstat);
3999         }

4001         /*
4002          * Add the cache to the global list.  This makes it visible
4003          * to kmem_update(), so the cache must be ready for business.
4004          */
4005         mutex_enter(&kmem_cache_lock);
4006         list_insert_tail(&kmem_caches, cp);
4007         mutex_exit(&kmem_cache_lock);

4009         if (kmem_ready)
4010                 kmem_cache_magazine_enable(cp);

4012         return (cp);
4013 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   54562 Thu Oct 23 10:42:16 2014
new/usr/src/uts/common/os/vmem.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 909 /*
 910  * Checks if vmp is guaranteed to have a size-byte buffer somewhere on its
 911  * freelist.  If size is not a power-of-2, it can return a false-negative.
 912  *
 913  * Used to decide if a newly imported span is superfluous after re-acquiring
 914  * the arena lock.
 915  */
 916 static int
 917 vmem_canalloc(vmem_t *vmp, size_t size)
 918 {
 919         int hb;
 920         int flist = 0;
 921         ASSERT(MUTEX_HELD(&vmp->vm_lock));

 923         if (ISP2(size))
 923         if ((size & (size - 1)) == 0)
 924                 flist = lowbit(P2ALIGN(vmp->vm_freemap, size));
 925         else if ((hb = highbit(size)) < VMEM_FREELISTS)
 926                 flist = lowbit(P2ALIGN(vmp->vm_freemap, 1UL << hb));

 928         return (flist);
 929 }

 931 /*
 932  * Allocate size bytes at offset phase from an align boundary such that the
 933  * resulting segment [addr, addr + size) is a subset of [minaddr, maxaddr)
 934  * that does not straddle a nocross-aligned boundary.
 935  */
 936 void *
 937 vmem_xalloc(vmem_t *vmp, size_t size, size_t align_arg, size_t phase,
 938         size_t nocross, void *minaddr, void *maxaddr, int vmflag)
 939 {
 940         vmem_seg_t *vsp;
 941         vmem_seg_t *vbest = NULL;
 942         uintptr_t addr, taddr, start, end;
 943         uintptr_t align = (align_arg != 0) ? align_arg : vmp->vm_quantum;
 944         void *vaddr, *xvaddr = NULL;
 945         size_t xsize;
 946         int hb, flist, resv;
 947         uint32_t mtbf;

 949         if ((align | phase | nocross) & (vmp->vm_quantum - 1))
 950                 panic("vmem_xalloc(%p, %lu, %lu, %lu, %lu, %p, %p, %x): "
 951                     "parameters not vm_quantum aligned",
 952                     (void *)vmp, size, align_arg, phase, nocross,
 953                     minaddr, maxaddr, vmflag);

 955         if (nocross != 0 &&
 956             (align > nocross || P2ROUNDUP(phase + size, align) > nocross))
 957                 panic("vmem_xalloc(%p, %lu, %lu, %lu, %lu, %p, %p, %x): "
 958                     "overconstrained allocation",
 959                     (void *)vmp, size, align_arg, phase, nocross,
 960                     minaddr, maxaddr, vmflag);

 962         if (phase >= align || !ISP2(align) || !ISP2(nocross))
 962         if (phase >= align || (align & (align - 1)) != 0 ||
 963             (nocross & (nocross - 1)) != 0)
 963                 panic("vmem_xalloc(%p, %lu, %lu, %lu, %lu, %p, %p, %x): "
 964                     "parameters inconsistent or invalid",
```



```
 965                     (void *)vmp, size, align_arg, phase, nocross,
 966                     minaddr, maxaddr, vmflag);

 968         if ((mtbf = vmem_mtbf | vmp->vm_mtbf) != 0 && gethrtime() % mtbf == 0 &&
 969             (vmflag & (VM_NOSLEEP | VM_PANIC)) == VM_NOSLEEP)
 970                 return (NULL);

 972         mutex_enter(&vmp->vm_lock);
 973         for (;;) {
 974                 if (vmp->vm_nsegfree < VMEM_MINFREE &&
 975                     !vmem_populate(vmp, vmflag))
 976                         break;
 977 do_alloc:
 978                 /*
 979                  * highbit() returns the highest bit + 1, which is exactly
 980                  * what we want: we want to search the first freelist whose
 981                  * members are *definitely* large enough to satisfy our
 982                  * allocation.  However, there are certain cases in which we
 983                  * want to look at the next-smallest freelist (which *might*
 984                  * be able to satisfy the allocation):
 985                  *
 986                  * (1)  The size is exactly a power of 2, in which case
 987                  *      the smaller freelist is always big enough;
 988                  *
 989                  * (2)  All other freelists are empty;
 990                  *
 991                  * (3)  We're in the highest possible freelist, which is
 992                  *      always empty (e.g. the 4GB freelist on 32-bit systems);
 993                  *
 994                  * (4)  We're doing a best-fit or first-fit allocation.
 995                  */
 996                 if (ISP2(size)) {
 997                 if ((size & (size - 1)) == 0) {
 997                         flist = lowbit(P2ALIGN(vmp->vm_freemap, size));
 998                 } else {
 999                         hb = highbit(size);
1000                         if ((vmp->vm_freemap >> hb) == 0 ||
1001                             hb == VMEM_FREELISTS ||
1002                             (vmflag & (VM_BESTFIT | VM_FIRSTFIT)))
1003                                 hb--;
1004                         flist = lowbit(P2ALIGN(vmp->vm_freemap, 1UL << hb));
1005                 }

1007                 for (vbest = NULL, vsp = (flist == 0) ? NULL :
1008                     vmp->vm_freelist[flist - 1].vs_knext;
1009                     vsp != NULL; vsp = vsp->vs_knext) {
1010                         vmp->vm_kstat.vk_search.value.ui64++;
1011                         if (vsp->vs_start == 0) {
1012                                 /*
1013                                  * We're moving up to a larger freelist,
1014                                  * so if we've already found a candidate,
1015                                  * the fit can't possibly get any better.
1016                                  */
1017                                 if (vbest != NULL)
1018                                         break;
1019                                 /*
1020                                  * Find the next non-empty freelist.
1021                                  */
1022                                 flist = lowbit(P2ALIGN(vmp->vm_freemap,
1023                                     VS_SIZE(vsp)));
1024                                 if (flist-- == 0)
1025                                         break;
1026                                 vsp = (vmem_seg_t *)&vmp->vm_freelist[flist];
1027                                 ASSERT(vsp->vs_knext->vs_type == VMEM_FREE);
1028                                 continue;
1029                         }
```

```
1030                        if (vsp->vs_end - 1 < (uintptr_t)minaddr)
1031                                continue;
1032                        if (vsp->vs_start > (uintptr_t)maxaddr - 1)
1033                                continue;
1034                        start = MAX(vsp->vs_start, (uintptr_t)minaddr);
1035                        end = MIN(vsp->vs_end - 1, (uintptr_t)maxaddr - 1) + 1;
1036                        taddr = P2PHASEUP(start, align, phase);
1037                        if (P2BOUNDARY(taddr, size, nocross))
1038                                taddr +=
1039                                    P2ROUNDUP(P2NPHASE(taddr, nocross), align);
1040                        if ((taddr - start) + size > end - start ||
1041                            (vbest != NULL && VS_SIZE(vsp) >= VS_SIZE(vbest)))
1042                                continue;
1043                        vbest = vsp;
1044                        addr = taddr;
1045                        if (!(vmflag & VM_BESTFIT) || VS_SIZE(vbest) == size)
1046                                break;
1047                }
1048                if (vbest != NULL)
1049                        break;
1050                ASSERT(xvaddr == NULL);
1051                if (size == 0)
1052                        panic("vmem_xalloc(): size == 0");
1053                if (vmp->vm_source_alloc != NULL && nocross == 0 &&
1054                    minaddr == NULL && maxaddr == NULL) {
1055                        size_t aneeded, asize;
1056                        size_t aquantum = MAX(vmp->vm_quantum,
1057                            vmp->vm_source->vm_quantum);
1058                        size_t aphase = phase;
1059                        if ((align > aquantum) &&
1060                            !(vmp->vm_cflags & VMC_XALIGN)) {
1061                                aphase = (P2PHASE(phase, aquantum) != 0) ?
1062                                    align - vmp->vm_quantum : align - aquantum;
1063                                ASSERT(aphase >= phase);
1064                        }
1065                        aneeded = MAX(size + aphase, vmp->vm_min_import);
1066                        asize = P2ROUNDUP(aneeded, aquantum);

1068                        if (asize < size) {
1069                                /*
1070                                 * The rounding induced overflow; return NULL
1071                                 * if we are permitted to fail the allocation
1072                                 * (and explicitly panic if we aren't).
1073                                 */
1074                                if ((vmflag & VM_NOSLEEP) &&
1075                                    !(vmflag & VM_PANIC)) {
1076                                        mutex_exit(&vmp->vm_lock);
1077                                        return (NULL);
1078                                }

1080                                panic("vmem_xalloc(): size overflow");
1081                        }

1083                        /*
1084                         * Determine how many segment structures we'll consume.
1085                         * The calculation must be precise because if we're
1086                         * here on behalf of vmem_populate(), we are taking
1087                         * segments from a very limited reserve.
1088                         */
1089                        if (size == asize && !(vmp->vm_cflags & VMC_XALLOC))
1090                                resv = VMEM_SEGS_PER_SPAN_CREATE +
1091                                    VMEM_SEGS_PER_EXACT_ALLOC;
1092                        else if (phase == 0 &&
1093                            align <= vmp->vm_source->vm_quantum)
1094                                resv = VMEM_SEGS_PER_SPAN_CREATE +
1095                                    VMEM_SEGS_PER_LEFT_ALLOC;
```



```
1096                        else
1097                                resv = VMEM_SEGS_PER_ALLOC_MAX;

1099                        ASSERT(vmp->vm_nsegfree >= resv);
1100                        vmp->vm_nsegfree -= resv;       /* reserve our segs */
1101                        mutex_exit(&vmp->vm_lock);
1102                        if (vmp->vm_cflags & VMC_XALLOC) {
1103                                size_t oasize = asize;
1104                                vaddr = ((vmem_ximport_t *)
1105                                    vmp->vm_source_alloc)(vmp->vm_source,
1106                                    &asize, align, vmflag & VM_KMFLAGS);
1107                                ASSERT(asize >= oasize);
1108                                ASSERT(P2PHASE(asize,
1109                                    vmp->vm_source->vm_quantum) == 0);
1110                                ASSERT(!(vmp->vm_cflags & VMC_XALIGN) ||
1111                                    IS_P2ALIGNED(vaddr, align));
1112                        } else {
1113                                vaddr = vmp->vm_source_alloc(vmp->vm_source,
1114                                    asize, vmflag & VM_KMFLAGS);
1115                        }
1116                        mutex_enter(&vmp->vm_lock);
1117                        vmp->vm_nsegfree += resv;       /* claim reservation */
1118                        aneeded = size + align - vmp->vm_quantum;
1119                        aneeded = P2ROUNDUP(aneeded, vmp->vm_quantum);
1120                        if (vaddr != NULL) {
1121                                /*
1122                                 * Since we dropped the vmem lock while
1123                                 * calling the import function, other
1124                                 * threads could have imported space
1125                                 * and made our import unnecessary.  In
1126                                 * order to save space, we return
1127                                 * excess imports immediately.
1128                                 */
1129                                if (asize > aneeded &&
1130                                    vmp->vm_source_free != NULL &&
1131                                    vmem_canalloc(vmp, aneeded)) {
1132                                        ASSERT(resv >=
1133                                            VMEM_SEGS_PER_MIDDLE_ALLOC);
1134                                        xvaddr = vaddr;
1135                                        xsize = asize;
1136                                        goto do_alloc;
1137                                }
1138                                vbest = vmem_span_create(vmp, vaddr, asize, 1);
1139                                addr = P2PHASEUP(vbest->vs_start, align, phase);
1140                                break;
1141                        } else if (vmem_canalloc(vmp, aneeded)) {
1142                                /*
1143                                 * Our import failed, but another thread
1144                                 * added sufficient free memory to the arena
1145                                 * to satisfy our request.  Go back and
1146                                 * grab it.
1147                                 */
1148                                ASSERT(resv >= VMEM_SEGS_PER_MIDDLE_ALLOC);
1149                                goto do_alloc;
1150                        }
1151                }

1153                /*
1154                 * If the requestor chooses to fail the allocation attempt
1155                 * rather than reap wait and retry - get out of the loop.
1156                 */
1157                if (vmflag & VM_ABORT)
1158                        break;
1159                mutex_exit(&vmp->vm_lock);
1160                if (vmp->vm_cflags & VMC_IDENTIFIER)
1161                        kmem_reap_idspace();
```

```
1162                 else
1163                         kmem_reap();
1164                 mutex_enter(&vmp->vm_lock);
1165                 if (vmflag & VM_NOSLEEP)
1166                         break;
1167                 vmp->vm_kstat.vk_wait.value.ui64++;
1168                 cv_wait(&vmp->vm_cv, &vmp->vm_lock);
1169         }
1170         if (vbest != NULL) {
1171                 ASSERT(vbest->vs_type == VMEM_FREE);
1172                 ASSERT(vbest->vs_knext != vbest);
1173                 /* re-position to end of buffer */
1174                 if (vmflag & VM_ENDALLOC) {
1175                         addr += ((vbest->vs_end - (addr + size)) / align) *
1176                             align;
1177                 }
1178                 (void) vmem_seg_alloc(vmp, vbest, addr, size);
1179                 mutex_exit(&vmp->vm_lock);
1180                 if (xvaddr)
1181                         vmp->vm_source_free(vmp->vm_source, xvaddr, xsize);
1182                 ASSERT(P2PHASE(addr, align) == phase);
1183                 ASSERT(!P2BOUNDARY(addr, size, nocross));
1184                 ASSERT(addr >= (uintptr_t)minaddr);
1185                 ASSERT(addr + size - 1 <= (uintptr_t)maxaddr - 1);
1186                 return ((void *)addr);
1187         }
1188         vmp->vm_kstat.vk_fail.value.ui64++;
1189         mutex_exit(&vmp->vm_lock);
1190         if (vmflag & VM_PANIC)
1191                 panic("vmem_xalloc(%p, %lu, %lu, %lu, %lu, %p, %p, %x): "
1192                     "cannot satisfy mandatory allocation",
1193                     (void *)vmp, size, align_arg, phase, nocross,
1194                     minaddr, maxaddr, vmflag);
1195         ASSERT(xvaddr == NULL);
1196         return (NULL);
1197 }
```

*unchanged_portion_omitted_*

```
1255 /*
1256  * Allocate size bytes from arena vmp.  Returns the allocated address
1257  * on success, NULL on failure.  vmflag specifies VM_SLEEP or VM_NOSLEEP,
1258  * and may also specify best-fit, first-fit, or next-fit allocation policy
1259  * instead of the default instant-fit policy.  VM_SLEEP allocations are
1260  * guaranteed to succeed.
1261  */
1262 void *
1263 vmem_alloc(vmem_t *vmp, size_t size, int vmflag)
1264 {
1265         vmem_seg_t *vsp;
1266         uintptr_t addr;
1267         int hb;
1268         int flist = 0;
1269         uint32_t mtbf;

1271         if (size - 1 < vmp->vm_qcache_max)
1272                 return (kmem_cache_alloc(vmp->vm_qcache[(size - 1) >>
1273                     vmp->vm_qshift], vmflag & VM_KMFLAGS));

1275         if ((mtbf = vmem_mtbf | vmp->vm_mtbf) != 0 && gethrtime() % mtbf == 0 &&
1276             (vmflag & (VM_NOSLEEP | VM_PANIC)) == VM_NOSLEEP)
1277                 return (NULL);

1279         if (vmflag & VM_NEXTFIT)
1280                 return (vmem_nextfit_alloc(vmp, size, vmflag));

1282         if (vmflag & (VM_BESTFIT | VM_FIRSTFIT))
```



```
1283                 return (vmem_xalloc(vmp, size, vmp->vm_quantum, 0, 0,
1284                     NULL, NULL, vmflag));

1286         /*
1287          * Unconstrained instant-fit allocation from the segment list.
1288          */
1289         mutex_enter(&vmp->vm_lock);

1291         if (vmp->vm_nsegfree >= VMEM_MINFREE || vmem_populate(vmp, vmflag)) {
1292                 if (ISP2(size))
1293                 if ((size & (size - 1)) == 0)
1293                         flist = lowbit(P2ALIGN(vmp->vm_freemap, size));
1294                 else if ((hb = highbit(size)) < VMEM_FREELISTS)
1295                         flist = lowbit(P2ALIGN(vmp->vm_freemap, 1UL << hb));
1296         }

1298         if (flist-- == 0) {
1299                 mutex_exit(&vmp->vm_lock);
1300                 return (vmem_xalloc(vmp, size, vmp->vm_quantum,
1301                     0, 0, NULL, NULL, vmflag));
1302         }

1304         ASSERT(size <= (1UL << flist));
1305         vsp = vmp->vm_freelist[flist].vs_knext;
1306         addr = vsp->vs_start;
1307         if (vmflag & VM_ENDALLOC) {
1308                 addr += vsp->vs_end - (addr + size);
1309         }
1310         (void) vmem_seg_alloc(vmp, vsp, addr, size);
1311         mutex_exit(&vmp->vm_lock);
1312         return ((void *)addr);
1313 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   45402 Thu Oct 23 10:42:16 2014
new/usr/src/uts/common/vm/seg_kmem.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

1244 /*
1245  * This function is called to import new spans into the vmem arenas like
1246  * kmem_default_arena and kmem_oversize_arena. It first tries to import
1247  * spans from large page arena - kmem_lp_arena. In order to do this it might
1248  * have to "upgrade the requested size" to kmem_lp_arena quantum. If
1249  * it was not able to satisfy the upgraded request it then calls regular
1250  * segkmem_alloc() that satisfies the request by importing from "*vmp" arena
1251  */
1252 /*ARGSUSED*/
1253 void *
1254 segkmem_alloc_lp(vmem_t *vmp, size_t *sizep, size_t align, int vmflag)
1255 {
1256         size_t size;
1257         kthread_t *t = curthread;
1258         segkmem_lpcb_t *lpcb = &segkmem_lpcb;

1260         ASSERT(sizep != NULL);

1262         size = *sizep;

1264         if (lpcb->lp_uselp && !(t->t_flag & T_PANIC) &&
1265             !(vmflag & SEGKMEM_SHARELOCKED)) {

1267                 size_t kmemlp_qnt = segkmem_kmemlp_quantum;
1268                 size_t asize = P2ROUNDUP(size, kmemlp_qnt);
1269                 void  *addr = NULL;
1270                 ulong_t *lpthrtp = &lpcb->lp_throttle;
1271                 ulong_t lpthrt = *lpthrtp;
1272                 int     dowakeup = 0;
1273                 int     doalloc = 1;

1275                 ASSERT(kmem_lp_arena != NULL);
1276                 ASSERT(asize >= size);

1278                 if (lpthrt != 0) {
1279                         /* try to update the throttle value */
1280                         lpthrt = atomic_inc_ulong_nv(lpthrtp);
1281                         if (lpthrt >= segkmem_lpthrottle_max) {
1282                                 lpthrt = atomic_cas_ulong(lpthrtp, lpthrt,
1283                                     segkmem_lpthrottle_max / 4);
1284                         }

1286                         /*
1287                          * when we get above throttle start do an exponential
1288                          * backoff at trying large pages and reaping
1289                          */
1290                         if (lpthrt > segkmem_lpthrottle_start &&
1291                             !ISP2(lpthrt)) {
1291                             (lpthrt & (lpthrt - 1))) {
1292                                 lpcb->allocs_throttled++;
1293                                 lpthrt--;
1294                                 if (ISP2(lpthrt))
1294                                 if ((lpthrt & (lpthrt - 1)) == 0)
1295                                         kmem_reap();
1296                                 return (segkmem_alloc(vmp, size, vmflag));
1297                         }
1298                 }

1300                 if (!(vmflag & VM_NOSLEEP) &&
```



```
1301                     segkmem_heaplp_quantum >= (8 * kmemlp_qnt) &&
1302                     vmem_size(kmem_lp_arena, VMEM_FREE) <= kmemlp_qnt &&
1303                     asize < (segkmem_heaplp_quantum - kmemlp_qnt)) {

1305                         /*
1306                          * we are low on free memory in kmem_lp_arena
1307                          * we let only one guy to allocate heap_lp
1308                          * quantum size chunk that everybody is going to
1309                          * share
1310                          */
1311                         mutex_enter(&lpcb->lp_lock);

1313                         if (lpcb->lp_wait) {

1315                                 /* we are not the first one - wait */
1316                                 cv_wait(&lpcb->lp_cv, &lpcb->lp_lock);
1317                                 if (vmem_size(kmem_lp_arena, VMEM_FREE) <
1318                                     kmemlp_qnt)  {
1319                                         doalloc = 0;
1320                                 }
1321                         } else if (vmem_size(kmem_lp_arena, VMEM_FREE) <=
1322                             kmemlp_qnt) {

1324                                 /*
1325                                  * we are the first one, make sure we import
1326                                  * a large page
1327                                  */
1328                                 if (asize == kmemlp_qnt)
1329                                         asize += kmemlp_qnt;
1330                                 dowakeup = 1;
1331                                 lpcb->lp_wait = 1;
1332                         }

1334                         mutex_exit(&lpcb->lp_lock);
1335                 }

1337                 /*
1338                  * VM_ABORT flag prevents sleeps in vmem_xalloc when
1339                  * large pages are not available. In that case this allocation
1340                  * attempt will fail and we will retry allocation with small
1341                  * pages. We also do not want to panic if this allocation fails
1342                  * because we are going to retry.
1343                  */
1344                 if (doalloc) {
1345                         addr = vmem_alloc(kmem_lp_arena, asize,
1346                             (vmflag | VM_ABORT) & ~VM_PANIC);

1348                         if (dowakeup) {
1349                                 mutex_enter(&lpcb->lp_lock);
1350                                 ASSERT(lpcb->lp_wait != 0);
1351                                 lpcb->lp_wait = 0;
1352                                 cv_broadcast(&lpcb->lp_cv);
1353                                 mutex_exit(&lpcb->lp_lock);
1354                         }
1355                 }

1357                 if (addr != NULL) {
1358                         *sizep = asize;
1359                         *lpthrtp = 0;
1360                         return (addr);
1361                 }

1363                 if (vmflag & VM_NOSLEEP)
1364                         lpcb->nosleep_allocs_failed++;
1365                 else
1366                         lpcb->sleep_allocs_failed++;
```

```
1367                 lpcb->alloc_bytes_failed += size;

1369                 /* if large page throttling is not started yet do it */
1370                 if (segkmem_use_lpthrottle && lpthrt == 0) {
1371                         lpthrt = atomic_cas_ulong(lpthrtp, lpthrt, 1);
1372                 }
1373         }
1374         return (segkmem_alloc(vmp, size, vmflag));
1375 }
______unchanged_portion_omitted_

1440 /*
1441  * This function is called at system boot time by kmem_init right after
1442  * /etc/system file has been read. It checks based on hardware configuration
1443  * and /etc/system settings if system is going to use large pages. The
1444  * initialiazation necessary to actually start using large pages
1445  * happens later in the process after segkmem_heap_lp_init() is called.
1446  */
1447 int
1448 segkmem_lpsetup()
1449 {
1450         int use_large_pages = 0;

1452 #ifdef __sparc

1454         size_t memtotal = physmem * PAGESIZE;

1456         if (heap_lp_base == NULL) {
1457                 segkmem_lpsize = PAGESIZE;
1458                 return (0);
1459         }

1461         /* get a platform dependent value of large page size for kernel heap */
1462         segkmem_lpsize = get_segkmem_lpsize(segkmem_lpsize);

1464         if (segkmem_lpsize <= PAGESIZE) {
1465                 /*
1466                  * put virtual space reserved for the large page kernel
1467                  * back to the regular heap
1468                  */
1469                 vmem_xfree(heap_arena, heap_lp_base,
1470                     heap_lp_end - heap_lp_base);
1471                 heap_lp_base = NULL;
1472                 heap_lp_end = NULL;
1473                 segkmem_lpsize = PAGESIZE;
1474                 return (0);
1475         }

1477         /* set heap_lp quantum if necessary */
1478         if (segkmem_heaplp_quantum == 0 || !ISP2(segkmem_heaplp_quantum) ||
1478         if (segkmem_heaplp_quantum == 0 ||
1479             (segkmem_heaplp_quantum & (segkmem_heaplp_quantum - 1)) ||
1479             P2PHASE(segkmem_heaplp_quantum, segkmem_lpsize)) {
1480                 segkmem_heaplp_quantum = segkmem_lpsize;
1481         }

1483         /* set kmem_lp quantum if necessary */
1484         if (segkmem_kmemlp_quantum == 0 || !ISP2(segkmem_kmemlp_quantum) ||
1485         if (segkmem_kmemlp_quantum == 0 ||
1486             (segkmem_kmemlp_quantum & (segkmem_kmemlp_quantum - 1)) ||
1485             segkmem_kmemlp_quantum > segkmem_heaplp_quantum) {
1486                 segkmem_kmemlp_quantum = segkmem_heaplp_quantum;
1487         }

1489         /* set total amount of memory allowed for large page kernel heap */
1490         if (segkmem_kmemlp_max == 0) {
```



```
1491                 if (segkmem_kmemlp_pcnt == 0 || segkmem_kmemlp_pcnt > 100)
1492                         segkmem_kmemlp_pcnt = 12;
1493                 segkmem_kmemlp_max = (memtotal * segkmem_kmemlp_pcnt) / 100;
1494         }
1495         segkmem_kmemlp_max = P2ROUNDUP(segkmem_kmemlp_max,
1496             segkmem_heaplp_quantum);

1498         /* fix lp kmem preallocation request if necesssary */
1499         if (segkmem_kmemlp_min) {
1500                 segkmem_kmemlp_min = P2ROUNDUP(segkmem_kmemlp_min,
1501                     segkmem_heaplp_quantum);
1502                 if (segkmem_kmemlp_min > segkmem_kmemlp_max)
1503                         segkmem_kmemlp_min = segkmem_kmemlp_max;
1504         }

1506         use_large_pages = 1;
1507         segkmem_lpszc = page_szc(segkmem_lpsize);
1508         segkmem_lpshift = page_get_shift(segkmem_lpszc);

1510 #endif
1511         return (use_large_pages);
1512 }
______unchanged_portion_omitted_
```

```
**********************************************************
   58052 Thu Oct 23 10:42:16 2014
new/usr/src/uts/common/vm/seg_map.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 298 int
 299 segmap_create(struct seg *seg, void *argsp)
 300 {
 301         struct segmap_data *smd;
 302         struct smap *smp;
 303         struct smfree *sm;
 304         struct segmap_crargs *a = (struct segmap_crargs *)argsp;
 305         struct smaphash *shashp;
 306         union segmap_cpu *scpu;
 307         long i, npages;
 308         size_t hashsz;
 309         uint_t nfreelist;
 310         extern void prefetch_smap_w(void *);
 311         extern int max_ncpus;

 313         ASSERT(seg->s_as && RW_WRITE_HELD(&seg->s_as->a_lock));

 315         if (((uintptr_t)seg->s_base | seg->s_size) & MAXBOFFSET) {
 316                 panic("segkmap not MAXBSIZE aligned");
 317                 /*NOTREACHED*/
 318         }

 320         smd = kmem_zalloc(sizeof (struct segmap_data), KM_SLEEP);

 322         seg->s_data = (void *)smd;
 323         seg->s_ops = &segmap_ops;
 324         smd->smd_prot = a->prot;

 326         /*
 327          * Scale the number of smap freelists to be
 328          * proportional to max_ncpus * number of virtual colors.
 329          * The caller can over-ride this scaling by providing
 330          * a non-zero a->nfreelist argument.
 331          */
 332         nfreelist = a->nfreelist;
 333         if (nfreelist == 0)
 334                 nfreelist = max_ncpus;
 335         else if (nfreelist < 0 || nfreelist > 4 * max_ncpus) {
 336                 cmn_err(CE_WARN, "segmap_create: nfreelist out of range "
 337                 "%d, using %d", nfreelist, max_ncpus);
 338                 nfreelist = max_ncpus;
 339         }
 340         if (!ISP2(nfreelist)) {
 340         if (nfreelist & (nfreelist - 1)) {
 341                 /* round up nfreelist to the next power of two. */
 342                 nfreelist = 1 << (highbit(nfreelist));
 343         }

 345         /*
 346          * Get the number of virtual colors - must be a power of 2.
 347          */
 348         if (a->shmsize)
 349                 smd_ncolor = a->shmsize >> MAXBSHIFT;
 350         else
 351                 smd_ncolor = 1;
 352         ASSERT((smd_ncolor & (smd_ncolor - 1)) == 0);
 353         ASSERT(smd_ncolor <= SEGMAP_MAXCOLOR);
 354         smd_colormsk = smd_ncolor - 1;
 355         smd->smd_nfree = smd_nfree = smd_ncolor * nfreelist;
```



```
 356         smd_freemsk = smd_nfree - 1;

 358         /*
 359          * Allocate and initialize the freelist headers.
 360          * Note that sm_freeq[1] starts out as the release queue. This
 361          * is known when the smap structures are initialized below.
 362          */
 363         smd_free = smd->smd_free =
 364             kmem_zalloc(smd_nfree * sizeof (struct smfree), KM_SLEEP);
 365         for (i = 0; i < smd_nfree; i++) {
 366                 sm = &smd->smd_free[i];
 367                 mutex_init(&sm->sm_freeq[0].smq_mtx, NULL, MUTEX_DEFAULT, NULL);
 368                 mutex_init(&sm->sm_freeq[1].smq_mtx, NULL, MUTEX_DEFAULT, NULL);
 369                 sm->sm_allocq = &sm->sm_freeq[0];
 370                 sm->sm_releq = &sm->sm_freeq[1];
 371         }

 373         /*
 374          * Allocate and initialize the smap hash chain headers.
 375          * Compute hash size rounding down to the next power of two.
 376          */
 377         npages = MAP_PAGES(seg);
 378         smd->smd_npages = npages;
 379         hashsz = npages / SMAP_HASHAVELEN;
 380         hashsz = 1 << (highbit(hashsz)-1);
 381         smd_hashmsk = hashsz - 1;
 382         smd_hash = smd->smd_hash =
 383             kmem_alloc(hashsz * sizeof (struct smaphash), KM_SLEEP);
 384 #ifdef SEGMAP_HASHSTATS
 385         smd_hash_len =
 386             kmem_zalloc(hashsz * sizeof (unsigned int), KM_SLEEP);
 387 #endif
 388         for (i = 0, shashp = smd_hash; i < hashsz; i++, shashp++) {
 389                 shashp->sh_hash_list = NULL;
 390                 mutex_init(&shashp->sh_mtx, NULL, MUTEX_DEFAULT, NULL);
 391         }

 393         /*
 394          * Allocate and initialize the smap structures.
 395          * Link all slots onto the appropriate freelist.
 396          * The smap array is large enough to affect boot time
 397          * on large systems, so use memory prefetching and only
 398          * go through the array 1 time. Inline a optimized version
 399          * of segmap_smapadd to add structures to freelists with
 400          * knowledge that no locks are needed here.
 401          */
 402         smd_smap = smd->smd_sm =
 403             kmem_alloc(sizeof (struct smap) * npages, KM_SLEEP);

 405         for (smp = &smd->smd_sm[MAP_PAGES(seg) - 1];
 406             smp >= smd->smd_sm; smp--) {
 407                 struct smap *smpfreelist;
 408                 struct sm_freeq *releq;

 410                 prefetch_smap_w((char *)smp);

 412                 smp->sm_vp = NULL;
 413                 smp->sm_hash = NULL;
 414                 smp->sm_off = 0;
 415                 smp->sm_bitmap = 0;
 416                 smp->sm_refcnt = 0;
 417                 mutex_init(&smp->sm_mtx, NULL, MUTEX_DEFAULT, NULL);
 418                 smp->sm_free_ndx = SMP2SMF_NDX(smp);

 420                 sm = SMP2SMF(smp);
 421                 releq = sm->sm_releq;
```

```
423                 smpfreelist = releq->smq_free;
424                 if (smpfreelist == 0) {
425                         releq->smq_free = smp->sm_next = smp->sm_prev = smp;
426                 } else {
427                         smp->sm_next = smpfreelist;
428                         smp->sm_prev = smpfreelist->sm_prev;
429                         smpfreelist->sm_prev = smp;
430                         smp->sm_prev->sm_next = smp;
431                         releq->smq_free = smp->sm_next;
432                 }

434                 /*
435                  * sm_flag = 0 (no SM_QNDX_ZERO) implies smap on sm_freeq[1]
436                  */
437                 smp->sm_flags = 0;

439 #ifdef  SEGKPM_SUPPORT
440                 /*
441                  * Due to the fragile prefetch loop no
442                  * separate function is used here.
443                  */
444                 smp->sm_kpme_next = NULL;
445                 smp->sm_kpme_prev = NULL;
446                 smp->sm_kpme_page = NULL;
447 #endif
448         }

450         /*
451          * Allocate the per color indices that distribute allocation
452          * requests over the free lists. Each cpu will have a private
453          * rotor index to spread the allocations even across the available
454          * smap freelists. Init the scpu_last_smap field to the first
455          * smap element so there is no need to check for NULL.
456          */
457         smd_cpu =
458             kmem_zalloc(sizeof (union segmap_cpu) * max_ncpus, KM_SLEEP);
459         for (i = 0, scpu = smd_cpu; i < max_ncpus; i++, scpu++) {
460                 int j;
461                 for (j = 0; j < smd_ncolor; j++)
462                         scpu->scpu.scpu_free_ndx[j] = j;
463                 scpu->scpu.scpu_last_smap = smd_smap;
464         }

466         vpm_init();

468 #ifdef DEBUG
469         /*
470          * Keep track of which colors are used more often.
471          */
472         colors_used = kmem_zalloc(smd_nfree * sizeof (int), KM_SLEEP);
473 #endif /* DEBUG */

475         return (0);
476 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   27206 Thu Oct 23 10:42:17 2014
new/usr/src/uts/common/vm/vpm.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 126 #define VPM_DEBUG(x)    ((vpm_debug.x)++)

 128 int     steals;
 129 int     steals_mtbf = 7;
 130 int     contend;
 131 int     contend_mtbf = 127;

 133 #define VPM_MTBF(v, f)  (((++(v)) & (f)) != (f))

 135 #else   /* DEBUG */

 137 #define VPM_MTBF(v, f)  (1)
 138 #define VPM_DEBUG(x)    /* nothing */

 140 #endif

 142 /*
 143  * The vpm cache.
 144  *
 145  * The main purpose of having a cache here is to speed up page_lookup()
 146  * operations and also provide an LRU(default) behaviour of file pages. The
 147  * page_lookup() operation tends to be expensive if a page has to be
 148  * reclaimed from the system page cache("cachelist"). Once we speed up the
 149  * page_lookup()->page_reclaim() path then there there should be no need for
 150  * this cache. The system page cache(cachelist) should effectively serve the
 151  * purpose of caching file pages.
 152  *
 153  * This cache is very similar to segmap's smap cache. Each page in the
 154  * cache is tracked by the structure vpmap_t. But unlike segmap, there is no
 155  * hash table. The page_t has a reference to the vpmap_t when cached. For a
 156  * given vnode, offset the page is found by means of a page_lookup() operation.
 157  * Any page which has a mapping(i.e when cached) will not be in the
 158  * system 'cachelist'. Hence the page_lookup() will not have to do a
 159  * page_reclaim(). That is how the cache serves to speed up page_lookup()
 160  * operations.
 161  *
 162  * This cache can be disabled by setting vpm_cache_enable = 0 in /etc/system.
 163  */

 165 void
 166 vpm_init()
 167 {
 168         long  npages;
 169         struct vpmap *vpm;
 170         struct vpmfree *vpmflp;
 171         int i, ndx;
 172         extern void prefetch_smap_w(void *);

 174         if (!kpm_enable) {
 175                 vpm_enable = 0;
 176         }

 178         if (!vpm_enable || !vpm_cache_enable) {
 179                 return;
 180         }

 182         /*
 183          * Set the size of the cache.
 184          */
```



```
 185         vpm_cache_size = mmu_ptob((physmem * vpm_cache_percent)/100);
 186         if (vpm_cache_size < VPMAP_MINCACHE) {
 187                 vpm_cache_size = VPMAP_MINCACHE;
 188         }

 190         if (vpm_cache_size > VPMAP_MAXCACHE) {
 191                 vpm_cache_size = VPMAP_MAXCACHE;
 192         }

 194         /*
 195          * Number of freelists.
 196          */
 197         if (vpm_nfreelist == 0) {
 198                 vpm_nfreelist = max_ncpus;
 199         } else if (vpm_nfreelist < 0 || vpm_nfreelist > 2 * max_ncpus) {
 200                 cmn_err(CE_WARN, "vpmap create : number of freelist "
 201                 "vpm_nfreelist %d using %d", vpm_nfreelist, max_ncpus);
 202                 vpm_nfreelist = 2 * max_ncpus;
 203         }

 205         /*
 206          * Round it up to the next power of 2
 207          */
 208         if (!ISP2(vpm_nfreelist)) {
 208         if (vpm_nfreelist & (vpm_nfreelist - 1)) {
 209                 vpm_nfreelist = 1 << (highbit(vpm_nfreelist));
 210         }
 211         vpmd_freemsk = vpm_nfreelist - 1;

 213         /*
 214          * Use a per cpu rotor index to spread the allocations evenly
 215          * across the available vpm freelists.
 216          */
 217         vpmd_cpu = kmem_zalloc(sizeof (union vpm_cpu) * max_ncpus, KM_SLEEP);
 218         ndx = 0;
 219         for (i = 0; i < max_ncpus; i++) {

 221                 vpmd_cpu[i].vfree_ndx = ndx;
 222                 ndx = (ndx + 1) & vpmd_freemsk;
 223         }

 225         /*
 226          * Allocate and initialize the freelist.
 227          */
 228         vpmd_free = kmem_zalloc(vpm_nfreelist * sizeof (struct vpmfree),
 229             KM_SLEEP);
 230         for (i = 0; i < vpm_nfreelist; i++) {

 232                 vpmflp = &vpmd_free[i];
 233                 /*
 234                  * Set up initial queue pointers. They will get flipped
 235                  * back and forth.
 236                  */
 237                 vpmflp->vpm_allocq = &vpmflp->vpm_freeq[VPMALLOCQ];
 238                 vpmflp->vpm_releq = &vpmflp->vpm_freeq[VPMRELEQ];
 239         }

 241         npages = mmu_btop(vpm_cache_size);


 244         /*
 245          * Allocate and initialize the vpmap structs. We need to
 246          * walk the array backwards as the prefetch happens in reverse
 247          * order.
 248          */
 249         vpmd_vpmap = kmem_alloc(sizeof (struct vpmap) * npages, KM_SLEEP);
```

```
250         for (vpm = &vpmd_vpmap[npages - 1]; vpm >= vpmd_vpmap; vpm--) {
251                 struct vpmfree *vpmflp;
252                 union vpm_freeq *releq;
253                 struct vpmap *vpmapf;

255                 /*
256                  * Use prefetch as we have to walk thru a large number of
257                  * these data structures. We just use the smap's prefetch
258                  * routine as it does the same.
259                  */
260                 prefetch_smap_w((void *)vpm);

262                 vpm->vpm_vp = NULL;
263                 vpm->vpm_off = 0;
264                 vpm->vpm_pp = NULL;
265                 vpm->vpm_refcnt = 0;
266                 mutex_init(&vpm->vpm_mtx, NULL, MUTEX_DEFAULT, NULL);
267                 vpm->vpm_free_ndx = VPMAP2VMF_NDX(vpm);

269                 vpmflp = VPMAP2VMF(vpm);
270                 releq = vpmflp->vpm_releq;

272                 vpmapf = releq->vpmq_free;
273                 if (vpmapf == NULL) {
274                         releq->vpmq_free = vpm->vpm_next = vpm->vpm_prev = vpm;
275                 } else {
276                         vpm->vpm_next = vpmapf;
277                         vpm->vpm_prev = vpmapf->vpm_prev;
278                         vpmapf->vpm_prev = vpm;
279                         vpm->vpm_prev->vpm_next = vpm;
280                         releq->vpmq_free = vpm->vpm_next;
281                 }

283                 /*
284                  * Indicate that the vpmap is on the releq at start
285                  */
286                 vpm->vpm_ndxflg = VPMRELEQ;
287         }
288 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   66588 Thu Oct 23 10:42:17 2014
new/usr/src/uts/i86pc/io/acpi/drmach_acpi/drmach_acpi.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 882 static int
 883 drmach_init(void)
 884 {
 885         DRMACH_HANDLE   hdl;
 886         drmachid_t      id;
 887         uint_t          bnum;

 889         if (MAX_BOARDS > SHRT_MAX) {
 890                 cmn_err(CE_WARN, "!drmach_init: system has too many (%d) "
 891                     "hotplug capable boards.", MAX_BOARDS);
 892                 return (ENXIO);
 893         } else if (MAX_CMP_UNITS_PER_BOARD > 1) {
 894                 cmn_err(CE_WARN, "!drmach_init: DR doesn't support multiple "
 895                     "(%d) physical processors on one board.",
 896                     MAX_CMP_UNITS_PER_BOARD);
 897                 return (ENXIO);
 898         } else if (!ISP2(MAX_CORES_PER_CMP)) {
 898         } else if (MAX_CORES_PER_CMP & (MAX_CORES_PER_CMP - 1)) {
 899                 cmn_err(CE_WARN, "!drmach_init: number of logical CPUs (%d) in "
 900                     "physical processor is not power of 2.",
 901                     MAX_CORES_PER_CMP);
 902                 return (ENXIO);
 903         } else if (MAX_CPU_UNITS_PER_BOARD > DEVSET_CPU_NUMBER ||
 904             MAX_MEM_UNITS_PER_BOARD > DEVSET_MEM_NUMBER ||
 905             MAX_IO_UNITS_PER_BOARD > DEVSET_IO_NUMBER) {
 906                 cmn_err(CE_WARN, "!drmach_init: system has more CPU/memory/IO "
 907                     "units than the DR driver can handle.");
 908                 return (ENXIO);
 909         }

 911         rw_init(&drmach_cpr_rwlock, NULL, RW_DEFAULT, NULL);
 912         drmach_cpr_cid = callb_add(drmach_cpr_callb, NULL,
 913             CB_CL_CPR_PM, "drmach");

 915         rw_init(&drmach_boards_rwlock, NULL, RW_DEFAULT, NULL);
 916         drmach_boards = drmach_array_new(0, MAX_BOARDS - 1);
 917         drmach_domain.allow_dr = acpidev_dr_capable();

 919         for (bnum = 0; bnum < MAX_BOARDS; bnum++) {
 920                 hdl = NULL;
 921                 if (ACPI_FAILURE(acpidev_dr_get_board_handle(bnum, &hdl)) ||
 922                     hdl == NULL) {
 923                         cmn_err(CE_WARN, "!drmach_init: failed to lookup ACPI "
 924                             "handle for board %d.", bnum);
 925                         continue;
 926                 }
 927                 if (drmach_array_get(drmach_boards, bnum, &id) == -1) {
 928                         DRMACH_PR("!drmach_init: failed to get handle "
 929                             "for board %d.", bnum);
 930                         ASSERT(0);
 931                         goto error;
 932                 } else if (id == NULL) {
 933                         (void) drmach_board_new(bnum, 1);
 934                 }
 935         }

 937         /*
 938          * Walk descendants of the devinfo root node and hold
 939          * all devinfo branches of interest.
```



```
 940          */
 941         drmach_hold_devtree();

 943         return (0);

 945 error:
 946         drmach_array_dispose(drmach_boards, drmach_board_dispose);
 947         rw_destroy(&drmach_boards_rwlock);
 948         rw_destroy(&drmach_cpr_rwlock);
 949         return (ENXIO);
 950 }
______unchanged_portion_omitted_
```

```
**********************************************************
   29838 Thu Oct 23 10:42:17 2014
new/usr/src/uts/i86pc/io/pci/pci_tools.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/stat.h>
#include <sys/sunddi.h>
#include <vm/seg_kmem.h>
#include <sys/machparam.h>
#include <sys/sunndi.h>
#include <sys/ontrap.h>
#include <sys/psm.h>
#include <sys/pcie.h>
#include <sys/pci_cfgspace.h>
#include <sys/pci_tools.h>
#include <io/pci/pci_tools_ext.h>
#include <sys/apic.h>
#include <sys/apix.h>
#include <io/pci/pci_var.h>
#include <sys/pci_impl.h>
#include <sys/promif.h>
#include <sys/x86_archext.h>
#include <sys/cpuvar.h>
#include <sys/pci_cfgacc.h>

#ifdef __xpv
#include <sys/hypervisor.h>
#endif

#define PCIEX_BDF_OFFSET_DELTA  4
#define PCIEX_REG_FUNC_SHIFT    (PCI_REG_FUNC_SHIFT + PCIEX_BDF_OFFSET_DELTA)
#define PCIEX_REG_DEV_SHIFT     (PCI_REG_DEV_SHIFT + PCIEX_BDF_OFFSET_DELTA)
#define PCIEX_REG_BUS_SHIFT     (PCI_REG_BUS_SHIFT + PCIEX_BDF_OFFSET_DELTA)

#define SUCCESS 0

extern uint64_t mcfg_mem_base;
int pcitool_debug = 0;
```



```
/*
 * Offsets of BARS in config space.  First entry of 0 means config space.
 * Entries here correlate to pcitool_bars_t enumerated type.
 */
static uint8_t pci_bars[] = {
        0x0,
        PCI_CONF_BASE0,
        PCI_CONF_BASE1,
        PCI_CONF_BASE2,
        PCI_CONF_BASE3,
        PCI_CONF_BASE4,
        PCI_CONF_BASE5,
        PCI_CONF_ROM
};

/* Max offset allowed into config space for a particular device. */
static uint64_t max_cfg_size = PCI_CONF_HDR_SIZE;

static uint64_t pcitool_swap_endian(uint64_t data, int size);
static int pcitool_cfg_access(pcitool_reg_t *prg, boolean_t write_flag,
    boolean_t io_access);
static int pcitool_io_access(pcitool_reg_t *prg, boolean_t write_flag);
static int pcitool_mem_access(pcitool_reg_t *prg, uint64_t virt_addr,
    boolean_t write_flag);
static uint64_t pcitool_map(uint64_t phys_addr, size_t size, size_t *num_pages);
static void pcitool_unmap(uint64_t virt_addr, size_t num_pages);

/* Extern declarations */
extern int      (*psm_intr_ops)(dev_info_t *, ddi_intr_handle_impl_t *,
                    psm_intr_op_t, int *);

int
pcitool_init(dev_info_t *dip, boolean_t is_pciex)
{
        int instance = ddi_get_instance(dip);

        /* Create pcitool nodes for register access and interrupt routing. */

        if (ddi_create_minor_node(dip, PCI_MINOR_REG, S_IFCHR,
            PCI_MINOR_NUM(instance, PCI_TOOL_REG_MINOR_NUM),
            DDI_NT_REGACC, 0) != DDI_SUCCESS) {
                return (DDI_FAILURE);
        }

        if (ddi_create_minor_node(dip, PCI_MINOR_INTR, S_IFCHR,
            PCI_MINOR_NUM(instance, PCI_TOOL_INTR_MINOR_NUM),
            DDI_NT_INTRCTL, 0) != DDI_SUCCESS) {
                ddi_remove_minor_node(dip, PCI_MINOR_REG);
                return (DDI_FAILURE);
        }

        if (is_pciex)
                max_cfg_size = PCIE_CONF_HDR_SIZE;

        return (DDI_SUCCESS);
}

void
pcitool_uninit(dev_info_t *dip)
{
        ddi_remove_minor_node(dip, PCI_MINOR_INTR);
        ddi_remove_minor_node(dip, PCI_MINOR_REG);
}

/*ARGSUSED*/
```

```
128 static int
129 pcitool_set_intr(dev_info_t *dip, void *arg, int mode)
130 {
131         ddi_intr_handle_impl_t info_hdl;
132         pcitool_intr_set_t iset;
133         uint32_t old_cpu;
134         int ret, result;
135         size_t copyinout_size;
136         int rval = SUCCESS;
137         apic_get_type_t type_info;

139         /* Version 1 of pcitool_intr_set_t doesn't have flags. */
140         copyinout_size = (size_t)&iset.flags - (size_t)&iset;

142         if (ddi_copyin(arg, &iset, copyinout_size, mode) != DDI_SUCCESS)
143                 return (EFAULT);

145         switch (iset.user_version) {
146         case PCITOOL_V1:
147                 break;

149         case PCITOOL_V2:
150                 copyinout_size = sizeof (pcitool_intr_set_t);
151                 if (ddi_copyin(arg, &iset, copyinout_size, mode) != DDI_SUCCESS)
152                         return (EFAULT);
153                 break;

155         default:
156                 iset.status = PCITOOL_OUT_OF_RANGE;
157                 rval = ENOTSUP;
158                 goto done_set_intr;
159         }

161         if (iset.flags & PCITOOL_INTR_FLAG_SET_MSI) {
162                 rval = ENOTSUP;
163                 iset.status = PCITOOL_IO_ERROR;
164                 goto done_set_intr;
165         }

167         info_hdl.ih_private = &type_info;

169         if ((*psm_intr_ops)(NULL, &info_hdl,
170             PSM_INTR_OP_APIC_TYPE, NULL) != PSM_SUCCESS) {
171                 rval = ENOTSUP;
172                 iset.status = PCITOOL_IO_ERROR;
173                 goto done_set_intr;
174         }

176         if (strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
177                 if (iset.old_cpu > type_info.avgi_num_cpu) {
178                         rval = EINVAL;
179                         iset.status = PCITOOL_INVALID_CPUID;
180                         goto done_set_intr;
181                 }
182                 old_cpu = iset.old_cpu;
183         } else {
184                 if ((old_cpu =
185                     pci_get_cpu_from_vecirq(iset.ino, IS_VEC)) == -1) {
186                         iset.status = PCITOOL_IO_ERROR;
187                         rval = EINVAL;
188                         goto done_set_intr;
189                 }
190         }

192         if (iset.ino > type_info.avgi_num_intr) {
193                 rval = EINVAL;
```



```
194                 iset.status = PCITOOL_INVALID_INO;
195                 goto done_set_intr;
196         }

198         iset.status = PCITOOL_SUCCESS;

200         old_cpu &= ~PSMGI_CPU_USER_BOUND;

202         /*
203          * For this locally-declared and used handle, ih_private will contain a
204          * CPU value, not an ihdl_plat_t as used for global interrupt handling.
205          */
206         if (strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
207                 info_hdl.ih_vector = APIX_VIRTVECTOR(old_cpu, iset.ino);
208         } else {
209                 info_hdl.ih_vector = iset.ino;
210         }
211         info_hdl.ih_private = (void *)(uintptr_t)iset.cpu_id;
212         info_hdl.ih_flags = PSMGI_INTRBY_VEC;
213         if (pcitool_debug)
214                 prom_printf("user version:%d, flags:0x%x\n",
215                     iset.user_version, iset.flags);

217         result = ENOTSUP;
218         if ((iset.user_version >= PCITOOL_V2) &&
219             (iset.flags & PCITOOL_INTR_FLAG_SET_GROUP)) {
220                 ret = (*psm_intr_ops)(NULL, &info_hdl, PSM_INTR_OP_GRP_SET_CPU,
221                     &result);
222         } else {
223                 ret = (*psm_intr_ops)(NULL, &info_hdl, PSM_INTR_OP_SET_CPU,
224                     &result);
225         }

227         if (ret != PSM_SUCCESS) {
228                 switch (result) {
229                 case EIO:               /* Error making the change */
230                         rval = EIO;
231                         iset.status = PCITOOL_IO_ERROR;
232                         break;
233                 case ENXIO:             /* Couldn't convert vector to irq */
234                         rval = EINVAL;
235                         iset.status = PCITOOL_INVALID_INO;
236                         break;
237                 case EINVAL:            /* CPU out of range */
238                         rval = EINVAL;
239                         iset.status = PCITOOL_INVALID_CPUID;
240                         break;
241                 case ENOTSUP:           /* Requested PSM intr ops missing */
242                         rval = ENOTSUP;
243                         iset.status = PCITOOL_IO_ERROR;
244                         break;
245                 }
246         }

248         /* Return original CPU. */
249         iset.cpu_id = old_cpu;

251         /* Return new vector */
252         if (strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
253                 iset.ino = APIX_VIRTVEC_VECTOR(info_hdl.ih_vector);
254         }

256 done_set_intr:
257         iset.drvr_version = PCITOOL_VERSION;
258         if (ddi_copyout(&iset, arg, copyinout_size, mode) != DDI_SUCCESS)
259                 rval = EFAULT;
```

```
260         return (rval);
261 }


264 /* It is assumed that dip != NULL */
265 static void
266 pcitool_get_intr_dev_info(dev_info_t *dip, pcitool_intr_dev_t *devs)
267 {
268         (void) strncpy(devs->driver_name,
269             ddi_driver_name(dip), MAXMODCONFNAME-2);
270         devs->driver_name[MAXMODCONFNAME-1] = '\0';
271         (void) ddi_pathname(dip, devs->path);
272         devs->dev_inst = ddi_get_instance(dip);
273 }

275 static int
276 pcitool_get_intr(dev_info_t *dip, void *arg, int mode)
277 {
278         /* Array part isn't used here, but oh well... */
279         pcitool_intr_get_t partial_iget;
280         pcitool_intr_get_t *iget = &partial_iget;
281         size_t  iget_kmem_alloc_size = 0;
282         uint8_t num_devs_ret;
283         int copyout_rval;
284         int rval = SUCCESS;
285         int circ;
286         int i;

288         ddi_intr_handle_impl_t info_hdl;
289         apic_get_intr_t intr_info;
290         apic_get_type_t type_info;

292         /* Read in just the header part, no array section. */
293         if (ddi_copyin(arg, &partial_iget, PCITOOL_IGET_SIZE(0), mode) !=
294             DDI_SUCCESS)
295                 return (EFAULT);

297         if (partial_iget.flags & PCITOOL_INTR_FLAG_GET_MSI) {
298                 partial_iget.status = PCITOOL_IO_ERROR;
299                 partial_iget.num_devs_ret = 0;
300                 rval = ENOTSUP;
301                 goto done_get_intr;
302         }

304         info_hdl.ih_private = &type_info;

306         if ((*psm_intr_ops)(NULL, &info_hdl,
307             PSM_INTR_OP_APIC_TYPE, NULL) != PSM_SUCCESS) {
308                 iget->status = PCITOOL_IO_ERROR;
309                 iget->num_devs_ret = 0;
310                 rval = EINVAL;
311                 goto done_get_intr;
312         }

314         if (strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
315                 if (partial_iget.cpu_id > type_info.avgi_num_cpu) {
316                         partial_iget.status = PCITOOL_INVALID_CPUID;
317                         partial_iget.num_devs_ret = 0;
318                         rval = EINVAL;
319                         goto done_get_intr;
320                 }
321         }

323         /* Validate argument. */
324         if ((partial_iget.ino & APIX_VIRTVEC_VECMASK) >
325             type_info.avgi_num_intr) {
```



```
326                 partial_iget.status = PCITOOL_INVALID_INO;
327                 partial_iget.num_devs_ret = 0;
328                 rval = EINVAL;
329                 goto done_get_intr;
330         }

332         num_devs_ret = partial_iget.num_devs_ret;
333         intr_info.avgi_dip_list = NULL;
334         intr_info.avgi_req_flags =
335             PSMGI_REQ_CPUID | PSMGI_REQ_NUM_DEVS | PSMGI_INTRBY_VEC;
336         /*
337          * For this locally-declared and used handle, ih_private will contain a
338          * pointer to apic_get_intr_t, not an ihdl_plat_t as used for
339          * global interrupt handling.
340          */
341         info_hdl.ih_private = &intr_info;

343         if (strcmp(type_info.avgi_type, APIC_APIX_NAME) == 0) {
344                 info_hdl.ih_vector =
345                     APIX_VIRTVECTOR(partial_iget.cpu_id, partial_iget.ino);
346         } else {
347                 info_hdl.ih_vector = partial_iget.ino;
348         }

350         /* Caller wants device information returned. */
351         if (num_devs_ret > 0) {

353                 intr_info.avgi_req_flags |= PSMGI_REQ_GET_DEVS;

355                 /*
356                  * Allocate room.
357                  * If num_devs_ret == 0 iget remains pointing to partial_iget.
358                  */
359                 iget_kmem_alloc_size = PCITOOL_IGET_SIZE(num_devs_ret);
360                 iget = kmem_alloc(iget_kmem_alloc_size, KM_SLEEP);

362                 /* Read in whole structure to verify there's room. */
363                 if (ddi_copyin(arg, iget, iget_kmem_alloc_size, mode) !=
364                     SUCCESS) {

366                         /* Be consistent and just return EFAULT here. */
367                         kmem_free(iget, iget_kmem_alloc_size);

369                         return (EFAULT);
370                 }
371         }

373         bzero(iget, PCITOOL_IGET_SIZE(num_devs_ret));
374         iget->ino = info_hdl.ih_vector;

376         /*
377          * Lock device tree branch from the pci root nexus on down if info will
378          * be extracted from dips returned from the tree.
379          */
380         if (intr_info.avgi_req_flags & PSMGI_REQ_GET_DEVS) {
381                 ndi_devi_enter(dip, &circ);
382         }

384         /* Call psm_intr_ops(PSM_INTR_OP_GET_INTR) to get information. */
385         if ((rval = (*psm_intr_ops)(NULL, &info_hdl,
386             PSM_INTR_OP_GET_INTR, NULL)) != PSM_SUCCESS) {
387                 iget->status = PCITOOL_IO_ERROR;
388                 iget->num_devs_ret = 0;
389                 rval = EINVAL;
390                 goto done_get_intr;
391         }
```

```
393         /*
394          * Fill in the pcitool_intr_get_t to be returned,
395          * with the CPU, num_devs_ret and num_devs.
396          */
397         if (intr_info.avgi_cpu_id == IRQ_UNBOUND ||
398             intr_info.avgi_cpu_id == IRQ_UNINIT)
399                 iget->cpu_id = 0;
400         else
401                 iget->cpu_id = intr_info.avgi_cpu_id & ~PSMGI_CPU_USER_BOUND;

403         /* Number of devices returned by apic. */
404         iget->num_devs = intr_info.avgi_num_devs;

406         /* Device info was returned. */
407         if (intr_info.avgi_req_flags & PSMGI_REQ_GET_DEVS) {

409                 /*
410                  * num devs returned is num devs ret by apic,
411                  * space permitting.
412                  */
413                 iget->num_devs_ret = min(num_devs_ret, intr_info.avgi_num_devs);

415                 /*
416                  * Loop thru list of dips and extract driver, name and instance.
417                  * Fill in the pcitool_intr_dev_t's with this info.
418                  */
419                 for (i = 0; i < iget->num_devs_ret; i++)
420                         pcitool_get_intr_dev_info(intr_info.avgi_dip_list[i],
421                             &iget->dev[i]);

423                 /* Free kmem_alloc'ed memory of the apic_get_intr_t */
424                 kmem_free(intr_info.avgi_dip_list,
425                     intr_info.avgi_num_devs * sizeof (dev_info_t *));
426         }

428 done_get_intr:

430         if (intr_info.avgi_req_flags & PSMGI_REQ_GET_DEVS) {
431                 ndi_devi_exit(dip, circ);
432         }

434         iget->drvr_version = PCITOOL_VERSION;
435         copyout_rval = ddi_copyout(iget, arg,
436             PCITOOL_IGET_SIZE(num_devs_ret), mode);

438         if (iget_kmem_alloc_size > 0)
439                 kmem_free(iget, iget_kmem_alloc_size);

441         if (copyout_rval != DDI_SUCCESS)
442                 rval = EFAULT;

444         return (rval);
445 }

447 /*ARGSUSED*/
448 static int
449 pcitool_intr_info(dev_info_t *dip, void *arg, int mode)
450 {
451         pcitool_intr_info_t intr_info;
452         ddi_intr_handle_impl_t info_hdl;
453         int rval = SUCCESS;
454         apic_get_type_t type_info;

456         /* If we need user_version, and to ret same user version as passed in */
457         if (ddi_copyin(arg, &intr_info, sizeof (pcitool_intr_info_t), mode) !=
```



```
458             DDI_SUCCESS) {
459                 if (pcitool_debug)
460                         prom_printf("Error reading arguments\n");
461                 return (EFAULT);
462         }

464         if (intr_info.flags & PCITOOL_INTR_FLAG_GET_MSI)
465                 return (ENOTSUP);

467         info_hdl.ih_private = &type_info;

469         /* For UPPC systems, psm_intr_ops has no entry for APIC_TYPE. */
470         if ((rval = (*psm_intr_ops)(NULL, &info_hdl,
471             PSM_INTR_OP_APIC_TYPE, NULL)) != PSM_SUCCESS) {
472                 intr_info.ctlr_type = PCITOOL_CTLR_TYPE_UPPC;
473                 intr_info.ctlr_version = 0;
474                 intr_info.num_intr = APIC_MAX_VECTOR;
475         } else {
476                 intr_info.ctlr_version = (uint32_t)info_hdl.ih_ver;
477                 intr_info.num_cpu = type_info.avgi_num_cpu;
478                 if (strcmp(type_info.avgi_type,
479                     APIC_PCPLUSMP_NAME) == 0) {
480                         intr_info.ctlr_type = PCITOOL_CTLR_TYPE_PCPLUSMP;
481                         intr_info.num_intr = type_info.avgi_num_intr;
482                 } else if (strcmp(type_info.avgi_type,
483                     APIC_APIX_NAME) == 0) {
484                         intr_info.ctlr_type = PCITOOL_CTLR_TYPE_APIX;
485                         intr_info.num_intr = type_info.avgi_num_intr;
486                 } else {
487                         intr_info.ctlr_type = PCITOOL_CTLR_TYPE_UNKNOWN;
488                         intr_info.num_intr = APIC_MAX_VECTOR;
489                 }
490         }

492         intr_info.drvr_version = PCITOOL_VERSION;
493         if (ddi_copyout(&intr_info, arg, sizeof (pcitool_intr_info_t), mode) !=
494             DDI_SUCCESS) {
495                 if (pcitool_debug)
496                         prom_printf("Error returning arguments.\n");
497                 rval = EFAULT;
498         }

500         return (rval);
501 }



505 /*
506  * Main function for handling interrupt CPU binding requests and queries.
507  * Need to implement later
508  */
509 int
510 pcitool_intr_admn(dev_info_t *dip, void *arg, int cmd, int mode)
511 {
512         int rval;

514         switch (cmd) {

516         /* Associate a new CPU with a given vector */
517         case PCITOOL_DEVICE_SET_INTR:
518                 rval = pcitool_set_intr(dip, arg, mode);
519                 break;

521         case PCITOOL_DEVICE_GET_INTR:
522                 rval = pcitool_get_intr(dip, arg, mode);
523                 break;
```

```
525         case PCITOOL_SYSTEM_INTR_INFO:
526                 rval = pcitool_intr_info(dip, arg, mode);
527                 break;
529         default:
530                 rval = ENOTSUP;
531         }
533         return (rval);
534 }
536 /*
537  * Perform register accesses on the nexus device itself.
538  * No explicit PCI nexus device for X86, so not applicable.
539  */
541 /*ARGSUSED*/
542 int
543 pcitool_bus_reg_ops(dev_info_t *dip, void *arg, int cmd, int mode)
544 {
545         return (ENOTSUP);
546 }
548 /* Swap endianness. */
549 static uint64_t
550 pcitool_swap_endian(uint64_t data, int size)
551 {
552         typedef union {
553                 uint64_t data64;
554                 uint8_t data8[8];
555         } data_split_t;
557         data_split_t orig_data;
558         data_split_t returned_data;
559         int i;
561         orig_data.data64 = data;
562         returned_data.data64 = 0;
564         for (i = 0; i < size; i++) {
565                 returned_data.data8[i] = orig_data.data8[size - 1 - i];
566         }
568         return (returned_data.data64);
569 }
571 /*
572  * A note about ontrap handling:
573  *
574  * X86 systems on which this module was tested return FFs instead of bus errors
575  * when accessing devices with invalid addresses.  Ontrap handling, which
576  * gracefully handles kernel bus errors, is installed anyway for I/O and mem
577  * space accessing (not for pci config space), in case future X86 platforms
578  * require it.
579  */
581 /* Access device.  prg is modified. */
582 static int
583 pcitool_cfg_access(pcitool_reg_t *prg, boolean_t write_flag,
584     boolean_t io_access)
585 {
586         int size = PCITOOL_ACC_ATTR_SIZE(prg->acc_attr);
587         boolean_t big_endian = PCITOOL_ACC_IS_BIG_ENDIAN(prg->acc_attr);
588         int rval = SUCCESS;
589         uint64_t local_data;
```



```
590         pci_cfgacc_req_t req;
591         uint32_t max_offset;
593         if ((size <= 0) || (size > 8) || !ISP2(size)) {
 25         if ((size <= 0) || (size > 8) || ((size & (size - 1)) != 0)) {
594                 prg->status = PCITOOL_INVALID_SIZE;
595                 return (ENOTSUP);
596         }
598         /*
599          * NOTE: there is no way to verify whether or not the address is
600          * valid other than that it is within the maximum offset.  The
601          * put functions return void and the get functions return -1 on error.
602          */
604         if (io_access)
605                 max_offset = 0xFF;
606         else
607                 max_offset = 0xFFF;
608         if (prg->offset + size - 1 > max_offset) {
609                 prg->status = PCITOOL_INVALID_ADDRESS;
610                 return (ENOTSUP);
611         }
613         prg->status = PCITOOL_SUCCESS;
615         req.rcdip = NULL;
616         req.bdf = PCI_GETBDF(prg->bus_no, prg->dev_no, prg->func_no);
617         req.offset = prg->offset;
618         req.size = size;
619         req.write = write_flag;
620         req.ioacc = io_access;
621         if (write_flag) {
622                 if (big_endian) {
623                         local_data = pcitool_swap_endian(prg->data, size);
624                 } else {
625                         local_data = prg->data;
626                 }
627                 VAL64(&req) = local_data;
628                 pci_cfgacc_acc(&req);
629         } else {
630                 pci_cfgacc_acc(&req);
631                 switch (size) {
632                 case 1:
633                         local_data = VAL8(&req);
634                         break;
635                 case 2:
636                         local_data = VAL16(&req);
637                         break;
638                 case 4:
639                         local_data = VAL32(&req);
640                         break;
641                 case 8:
642                         local_data = VAL64(&req);
643                         break;
644                 }
645                 if (big_endian) {
646                         prg->data =
647                             pcitool_swap_endian(local_data, size);
648                 } else {
649                         prg->data = local_data;
650                 }
651         }
652         /*
653          * Check if legacy IO config access is used, in which case
654          * only first 256 bytes are valid.
```

```
655          */
656         if (req.ioacc && (prg->offset + size - 1 > 0xFF)) {
657                 prg->status = PCITOOL_INVALID_ADDRESS;
658                 return (ENOTSUP);
659         }

661         /* Set phys_addr only if MMIO is used */
662         prg->phys_addr = 0;
663         if (!req.ioacc && mcfg_mem_base != 0) {
664                 prg->phys_addr = mcfg_mem_base + prg->offset +
665                     ((prg->bus_no << PCIEX_REG_BUS_SHIFT) |
666                     (prg->dev_no << PCIEX_REG_DEV_SHIFT) |
667                     (prg->func_no << PCIEX_REG_FUNC_SHIFT));
668         }

670         return (rval);
671 }
```

unchanged_portion_omitted_

```
**********************************************************
  142119 Thu Oct 23 10:42:17 2014
new/usr/src/uts/i86pc/io/rootnex.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_


1708 /*
1709  * ******************
1710  *  dma related code
1711  * ******************
1712  */

1714 /*ARGSUSED*/
1715 static int
1716 rootnex_coredma_allochdl(dev_info_t *dip, dev_info_t *rdip,
1717     ddi_dma_attr_t *attr, int (*waitfp)(caddr_t), caddr_t arg,
1718     ddi_dma_handle_t *handlep)
1719 {
1720         uint64_t maxsegmentsize_ll;
1721         uint_t maxsegmentsize;
1722         ddi_dma_impl_t *hp;
1723         rootnex_dma_t *dma;
1724         uint64_t count_max;
1725         uint64_t seg;
1726         int kmflag;
1727         int e;


1730         /* convert our sleep flags */
1731         if (waitfp == DDI_DMA_SLEEP) {
1732                 kmflag = KM_SLEEP;
1733         } else {
1734                 kmflag = KM_NOSLEEP;
1735         }

1737         /*
1738          * We try to do only one memory allocation here. We'll do a little
1739          * pointer manipulation later. If the bind ends up taking more than
1740          * our prealloc's space, we'll have to allocate more memory in the
1741          * bind operation. Not great, but much better than before and the
1742          * best we can do with the current bind interfaces.
1743          */
1744         hp = kmem_cache_alloc(rootnex_state->r_dmahdl_cache, kmflag);
1745         if (hp == NULL)
1746                 return (DDI_DMA_NORESOURCES);

1748         /* Do our pointer manipulation now, align the structures */
1749         hp->dmai_private = (void *)(((uintptr_t)hp +
1750             (uintptr_t)sizeof (ddi_dma_impl_t) + 0x7) & ~0x7);
1751         dma = (rootnex_dma_t *)hp->dmai_private;
1752         dma->dp_prealloc_buffer = (uchar_t *)(((uintptr_t)dma +
1753             sizeof (rootnex_dma_t) + 0x7) & ~0x7);

1755         /* setup the handle */
1756         rootnex_clean_dmahdl(hp);
1757         hp->dmai_error.err_fep = NULL;
1758         hp->dmai_error.err_cf = NULL;
1759         dma->dp_dip = rdip;
1760         dma->dp_sglinfo.si_flags = attr->dma_attr_flags;
1761         dma->dp_sglinfo.si_min_addr = attr->dma_attr_addr_lo;

1763         /*
1764          * The BOUNCE_ON_SEG workaround is not needed when an IOMMU
1765          * is being used. Set the upper limit to the seg value.
```



```
1766          * There will be enough DVMA space to always get addresses
1767          * that will match the constraints.
1768          */
1769         if (IOMMU_USED(rdip) &&
1770             (attr->dma_attr_flags & _DDI_DMA_BOUNCE_ON_SEG)) {
1771                 dma->dp_sglinfo.si_max_addr = attr->dma_attr_seg;
1772                 dma->dp_sglinfo.si_flags &= ~_DDI_DMA_BOUNCE_ON_SEG;
1773         } else
1774                 dma->dp_sglinfo.si_max_addr = attr->dma_attr_addr_hi;

1776         hp->dmai_minxfer = attr->dma_attr_minxfer;
1777         hp->dmai_burstsizes = attr->dma_attr_burstsizes;
1778         hp->dmai_rdip = rdip;
1779         hp->dmai_attr = *attr;

1781         if (attr->dma_attr_seg >= dma->dp_sglinfo.si_max_addr)
1782                 dma->dp_sglinfo.si_cancross = B_FALSE;
1783         else
1784                 dma->dp_sglinfo.si_cancross = B_TRUE;

1786         /* we don't need to worry about the SPL since we do a tryenter */
1787         mutex_init(&dma->dp_mutex, NULL, MUTEX_DRIVER, NULL);

1789         /*
1790          * Figure out our maximum segment size. If the segment size is greater
1791          * than 4G, we will limit it to (4G - 1) since the max size of a dma
1792          * object (ddi_dma_obj_t.dmao_size) is 32 bits. dma_attr_seg and
1793          * dma_attr_count_max are size-1 type values.
1794          *
1795          * Maximum segment size is the largest physically contiguous chunk of
1796          * memory that we can return from a bind (i.e. the maximum size of a
1797          * single cookie).
1798          */

1800         /* handle the rollover cases */
1801         seg = attr->dma_attr_seg + 1;
1802         if (seg < attr->dma_attr_seg) {
1803                 seg = attr->dma_attr_seg;
1804         }
1805         count_max = attr->dma_attr_count_max + 1;
1806         if (count_max < attr->dma_attr_count_max) {
1807                 count_max = attr->dma_attr_count_max;
1808         }

1810         /*
1811          * granularity may or may not be a power of two. If it isn't, we can't
1812          * use a simple mask.
1813          */
1814         if (!ISP2(attr->dma_attr_granular)) {
1814         if (attr->dma_attr_granular & (attr->dma_attr_granular - 1)) {
1815                 dma->dp_granularity_power_2 = B_FALSE;
1816         } else {
1817                 dma->dp_granularity_power_2 = B_TRUE;
1818         }

1820         /*
1821          * maxxfer should be a whole multiple of granularity. If we're going to
1822          * break up a window because we're greater than maxxfer, we might as
1823          * well make sure it's maxxfer is a whole multiple so we don't have to
1824          * worry about triming the window later on for this case.
1825          */
1826         if (attr->dma_attr_granular > 1) {
1827                 if (dma->dp_granularity_power_2) {
1828                         dma->dp_maxxfer = attr->dma_attr_maxxfer -
1829                             (attr->dma_attr_maxxfer &
1830                             (attr->dma_attr_granular - 1));
```

```
1831                 } else {
1832                         dma->dp_maxxfer = attr->dma_attr_maxxfer -
1833                             (attr->dma_attr_maxxfer % attr->dma_attr_granular);
1834                 }
1835         } else {
1836                 dma->dp_maxxfer = attr->dma_attr_maxxfer;
1837         }

1839         maxsegmentsize_ll = MIN(seg, dma->dp_maxxfer);
1840         maxsegmentsize_ll = MIN(maxsegmentsize_ll, count_max);
1841         if (maxsegmentsize_ll == 0 || (maxsegmentsize_ll > 0xFFFFFFFF)) {
1842                 maxsegmentsize = 0xFFFFFFFF;
1843         } else {
1844                 maxsegmentsize = maxsegmentsize_ll;
1845         }
1846         dma->dp_sglinfo.si_max_cookie_size = maxsegmentsize;
1847         dma->dp_sglinfo.si_segmask = attr->dma_attr_seg;

1849         /* check the ddi_dma_attr arg to make sure it makes a little sense */
1850         if (rootnex_alloc_check_parms) {
1851                 e = rootnex_valid_alloc_parms(attr, maxsegmentsize);
1852                 if (e != DDI_SUCCESS) {
1853                         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_ALLOC_FAIL]);
1854                         (void) rootnex_dma_freehdl(dip, rdip,
1855                             (ddi_dma_handle_t)hp);
1856                         return (e);
1857                 }
1858         }

1860         *handlep = (ddi_dma_handle_t)hp;

1862         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_ACTIVE_HDLS]);
1863         ROOTNEX_DPROBE1(rootnex__alloc__handle, uint64_t,
1864             rootnex_cnt[ROOTNEX_CNT_ACTIVE_HDLS]);

1866         return (DDI_SUCCESS);
1867 }
______unchanged_portion_omitted_


3817 /*
3818  * rootnex_setup_cookie()
3819  *    Called in the bind slow path when the sgl uses the copy buffer. If any of
3820  *    the sgl uses the copy buffer, we need to go through each cookie, figure
3821  *    out if it uses the copy buffer, and if it does, save away everything we'll
3822  *    need during sync.
3823  */
3824 static void
3825 rootnex_setup_cookie(ddi_dma_obj_t *dmar_object, rootnex_dma_t *dma,
3826     ddi_dma_cookie_t *cookie, off_t cur_offset, size_t *copybuf_used,
3827     page_t **cur_pp)
3828 {
3829         boolean_t copybuf_sz_power_2;
3830         rootnex_sglinfo_t *sinfo;
3831         paddr_t paddr;
3832         uint_t pidx;
3833         uint_t pcnt;
3834         off_t poff;
3835 #if defined(__amd64)
3836         pfn_t pfn;
3837 #else
3838         page_t **pplist;
3839 #endif

3841         ASSERT(dmar_object->dmao_type != DMA_OTYP_DVADDR);
```



```
3843         sinfo = &dma->dp_sglinfo;

3845         /*
3846          * Calculate the page index relative to the start of the buffer. The
3847          * index to the current page for our buffer is the offset into the
3848          * first page of the buffer plus our current offset into the buffer
3849          * itself, shifted of course...
3850          */
3851         pidx = (sinfo->si_buf_offset + cur_offset) >> MMU_PAGESHIFT;
3852         ASSERT(pidx < sinfo->si_max_pages);

3854         /* if this cookie uses the copy buffer */
3855         if (cookie->dmac_type & ROOTNEX_USES_COPYBUF) {
3856                 /*
3857                  * NOTE: we know that since this cookie uses the copy buffer, it
3858                  * is <= MMU_PAGESIZE.
3859                  */

3861                 /*
3862                  * get the offset into the page. For the 64-bit kernel, get the
3863                  * pfn which we'll use with seg kpm.
3864                  */
3865                 poff = cookie->dmac_laddress & MMU_PAGEOFFSET;
3866 #if defined(__amd64)
3867                 /* mfn_to_pfn() is a NOP on i86pc */
3868                 pfn = mfn_to_pfn(cookie->dmac_laddress >> MMU_PAGESHIFT);
3869 #endif /* __amd64 */

3871                 /* figure out if the copybuf size is a power of 2 */
3872                 if (!ISP2(dma->dp_copybuf_size)) {
3872                 if (dma->dp_copybuf_size & (dma->dp_copybuf_size - 1)) {
3873                         copybuf_sz_power_2 = B_FALSE;
3874                 } else {
3875                         copybuf_sz_power_2 = B_TRUE;
3876                 }

3878                 /* This page uses the copy buffer */
3879                 dma->dp_pgmap[pidx].pm_uses_copybuf = B_TRUE;

3881                 /*
3882                  * save the copy buffer KVA that we'll use with this page.
3883                  * if we still fit within the copybuf, it's a simple add.
3884                  * otherwise, we need to wrap over using & or % accordingly.
3885                  */
3886                 if ((*copybuf_used + MMU_PAGESIZE) <= dma->dp_copybuf_size) {
3887                         dma->dp_pgmap[pidx].pm_cbaddr = dma->dp_cbaddr +
3888                             *copybuf_used;
3889                 } else {
3890                         if (copybuf_sz_power_2) {
3891                                 dma->dp_pgmap[pidx].pm_cbaddr = (caddr_t)(
3892                                     (uintptr_t)dma->dp_cbaddr +
3893                                     (*copybuf_used &
3894                                     (dma->dp_copybuf_size - 1)));
3895                         } else {
3896                                 dma->dp_pgmap[pidx].pm_cbaddr = (caddr_t)(
3897                                     (uintptr_t)dma->dp_cbaddr +
3898                                     (*copybuf_used % dma->dp_copybuf_size));
3899                         }
3900                 }

3902                 /*
3903                  * over write the cookie physical address with the address of
3904                  * the physical address of the copy buffer page that we will
3905                  * use.
3906                  */
3907                 paddr = pfn_to_pa(hat_getpfnum(kas.a_hat,
```

```
3908                        dma->dp_pgmap[pidx].pm_cbaddr)) + poff;

3910                cookie->dmac_laddress = ROOTNEX_PADDR_TO_RBASE(paddr);

3912                /* if we have a kernel VA, it's easy, just save that address */
3913                if ((dmar_object->dmao_type != DMA_OTYP_PAGES) &&
3914                    (sinfo->si_asp == &kas)) {
3915                        /*
3916                         * save away the page aligned virtual address of the
3917                         * driver buffer. Offsets are handled in the sync code.
3918                         */
3919                        dma->dp_pgmap[pidx].pm_kaddr = (caddr_t)(((uintptr_t)
3920                            dmar_object->dmao_obj.virt_obj.v_addr + cur_offset)
3921                            & MMU_PAGEMASK);
3922 #if !defined(__amd64)
3923                        /*
3924                         * we didn't need to, and will never need to map this
3925                         * page.
3926                         */
3927                        dma->dp_pgmap[pidx].pm_mapped = B_FALSE;
3928 #endif

3930                /* we don't have a kernel VA. We need one for the bcopy. */
3931                } else {
3932 #if defined(__amd64)
3933                        /*
3934                         * for the 64-bit kernel, it's easy. We use seg kpm to
3935                         * get a Kernel VA for the corresponding pfn.
3936                         */
3937                        dma->dp_pgmap[pidx].pm_kaddr = hat_kpm_pfn2va(pfn);
3938 #else
3939                        /*
3940                         * for the 32-bit kernel, this is a pain. First we'll
3941                         * save away the page_t or user VA for this page. This
3942                         * is needed in rootnex_dma_win() when we switch to a
3943                         * new window which requires us to re-map the copy
3944                         * buffer.
3945                         */
3946                        pplist = dmar_object->dmao_obj.virt_obj.v_priv;
3947                        if (dmar_object->dmao_type == DMA_OTYP_PAGES) {
3948                                dma->dp_pgmap[pidx].pm_pp = *cur_pp;
3949                                dma->dp_pgmap[pidx].pm_vaddr = NULL;
3950                        } else if (pplist != NULL) {
3951                                dma->dp_pgmap[pidx].pm_pp = pplist[pidx];
3952                                dma->dp_pgmap[pidx].pm_vaddr = NULL;
3953                        } else {
3954                                dma->dp_pgmap[pidx].pm_pp = NULL;
3955                                dma->dp_pgmap[pidx].pm_vaddr = (caddr_t)
3956                                    (((uintptr_t)
3957                                    dmar_object->dmao_obj.virt_obj.v_addr +
3958                                    cur_offset) & MMU_PAGEMASK);
3959                        }

3961                        /*
3962                         * save away the page aligned virtual address which was
3963                         * allocated from the kernel heap arena (taking into
3964                         * account if we need more copy buffer than we alloced
3965                         * and use multiple windows to handle this, i.e. &,%).
3966                         * NOTE: there isn't and physical memory backing up this
3967                         * virtual address space currently.
3968                         */
3969                        if ((*copybuf_used + MMU_PAGESIZE) <=
3970                            dma->dp_copybuf_size) {
3971                                dma->dp_pgmap[pidx].pm_kaddr = (caddr_t)
3972                                    (((uintptr_t)dma->dp_kva + *copybuf_used) &
3973                                    MMU_PAGEMASK);
```



```
3974                        } else {
3975                                if (copybuf_sz_power_2) {
3976                                        dma->dp_pgmap[pidx].pm_kaddr = (caddr_t)
3977                                            (((uintptr_t)dma->dp_kva +
3978                                            (*copybuf_used &
3979                                            (dma->dp_copybuf_size - 1))) &
3980                                            MMU_PAGEMASK);
3981                                } else {
3982                                        dma->dp_pgmap[pidx].pm_kaddr = (caddr_t)
3983                                            (((uintptr_t)dma->dp_kva +
3984                                            (*copybuf_used %
3985                                            dma->dp_copybuf_size)) &
3986                                            MMU_PAGEMASK);
3987                                }
3988                        }

3990                        /*
3991                         * if we haven't used up the available copy buffer yet,
3992                         * map the kva to the physical page.
3993                         */
3994                        if (!dma->dp_cb_remaping && ((*copybuf_used +
3995                            MMU_PAGESIZE) <= dma->dp_copybuf_size)) {
3996                                dma->dp_pgmap[pidx].pm_mapped = B_TRUE;
3997                                if (dma->dp_pgmap[pidx].pm_pp != NULL) {
3998                                        i86_pp_map(dma->dp_pgmap[pidx].pm_pp,
3999                                            dma->dp_pgmap[pidx].pm_kaddr);
4000                                } else {
4001                                        i86_va_map(dma->dp_pgmap[pidx].pm_vaddr,
4002                                            sinfo->si_asp,
4003                                            dma->dp_pgmap[pidx].pm_kaddr);
4004                                }

4006                        /*
4007                         * we've used up the available copy buffer, this page
4008                         * will have to be mapped during rootnex_dma_win() when
4009                         * we switch to a new window which requires a re-map
4010                         * the copy buffer. (32-bit kernel only)
4011                         */
4012                        } else {
4013                                dma->dp_pgmap[pidx].pm_mapped = B_FALSE;
4014                        }
4015 #endif
4016                        /* go to the next page_t */
4017                        if (dmar_object->dmao_type == DMA_OTYP_PAGES) {
4018                                *cur_pp = (*cur_pp)->p_next;
4019                        }
4020                }

4022                /* add to the copy buffer count */
4023                *copybuf_used += MMU_PAGESIZE;

4025        /*
4026         * This cookie doesn't use the copy buffer. Walk through the pages this
4027         * cookie occupies to reflect this.
4028         */
4029        } else {
4030                /*
4031                 * figure out how many pages the cookie occupies. We need to
4032                 * use the original page offset of the buffer and the cookies
4033                 * offset in the buffer to do this.
4034                 */
4035                poff = (sinfo->si_buf_offset + cur_offset) & MMU_PAGEOFFSET;
4036                pcnt = mmu_btopr(cookie->dmac_size + poff);

4038                while (pcnt > 0) {
4039 #if !defined(__amd64)
```

```
4040                        /*
4041                         * the 32-bit kernel doesn't have seg kpm, so we need
4042                         * to map in the driver buffer (if it didn't come down
4043                         * with a kernel VA) on the fly. Since this page doesn't
4044                         * use the copy buffer, it's not, or will it ever, have
4045                         * to be mapped in.
4046                         */
4047                        dma->dp_pgmap[pidx].pm_mapped = B_FALSE;
4048 #endif
4049                        dma->dp_pgmap[pidx].pm_uses_copybuf = B_FALSE;

4051                        /*
4052                         * we need to update pidx and cur_pp or we'll loose
4053                         * track of where we are.
4054                         */
4055                        if (dmar_object->dmao_type == DMA_OTYP_PAGES) {
4056                                *cur_pp = (*cur_pp)->p_next;
4057                        }
4058                        pidx++;
4059                        pcnt--;
4060                }
4061        }
4062 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   73388 Thu Oct 23 10:42:17 2014
new/usr/src/uts/i86pc/os/ddi_impl.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

1527 /*
1528  * Check if the specified cache attribute is supported on the platform.
1529  * This function must be called before i_ddi_cacheattr_to_hatacc().
1530  */
1531 boolean_t
1532 i_ddi_check_cache_attr(uint_t flags)
1533 {
1534         /*
1535          * The cache attributes are mutually exclusive. Any combination of
1536          * the attributes leads to a failure.
1537          */
1538         uint_t cache_attr = IOMEM_CACHE_ATTR(flags);
1539         if ((cache_attr != 0) && !ISP2(cache_attr))
1539         if ((cache_attr != 0) && ((cache_attr & (cache_attr - 1)) != 0))
1540                 return (B_FALSE);

1542         /* All cache attributes are supported on X86/X64 */
1543         if (cache_attr & (IOMEM_DATA_UNCACHED | IOMEM_DATA_CACHED |
1544             IOMEM_DATA_UC_WR_COMBINE))
1545                 return (B_TRUE);

1547         /* undefined attributes */
1548         return (B_FALSE);
1549 }
______unchanged_portion_omitted_

1597 /*
1598  * This should actually be called i_ddi_dma_mem_alloc. There should
1599  * also be an i_ddi_pio_mem_alloc. i_ddi_dma_mem_alloc should call
1600  * through the device tree with the DDI_CTLOPS_DMA_ALIGN ctl ops to
1601  * get alignment requirements for DMA memory. i_ddi_pio_mem_alloc
1602  * should use DDI_CTLOPS_PIO_ALIGN. Since we only have i_ddi_mem_alloc
1603  * so far which is used for both, DMA and PIO, we have to use the DMA
1604  * ctl ops to make everybody happy.
1605  */
1606 /*ARGSUSED*/
1607 int
1608 i_ddi_mem_alloc(dev_info_t *dip, ddi_dma_attr_t *attr,
1609         size_t length, int cansleep, int flags,
1610         ddi_device_acc_attr_t *accattrp, caddr_t *kaddrp,
1611         size_t *real_length, ddi_acc_hdl_t *ap)
1612 {
1613         caddr_t a;
1614         int iomin;
1615         ddi_acc_impl_t *iap;
1616         int physcontig = 0;
1617         pgcnt_t npages;
1618         pgcnt_t minctg;
1619         uint_t order;
1620         int e;

1622         /*
1623          * Check legality of arguments
1624          */
1625         if (length == 0 || kaddrp == NULL || attr == NULL) {
1626                 return (DDI_FAILURE);
1627         }

1629         if (attr->dma_attr_minxfer == 0 || attr->dma_attr_align == 0 ||
```



```
1630             !ISP2(attr->dma_attr_align) || !ISP2(attr->dma_attr_minxfer)) {
1630             (attr->dma_attr_align & (attr->dma_attr_align - 1)) ||
1631             (attr->dma_attr_minxfer & (attr->dma_attr_minxfer - 1))) {
1631                 return (DDI_FAILURE);
1632         }

1634         /*
1635          * figure out most restrictive alignment requirement
1636          */
1637         iomin = attr->dma_attr_minxfer;
1638         iomin = maxbit(iomin, attr->dma_attr_align);
1639         if (iomin == 0)
1640                 return (DDI_FAILURE);

1642         ASSERT((iomin & (iomin - 1)) == 0);

1644         /*
1645          * if we allocate memory with IOMEM_DATA_UNCACHED or
1646          * IOMEM_DATA_UC_WR_COMBINE, make sure we allocate a page aligned
1647          * memory that ends on a page boundry.
1648          * Don't want to have to different cache mappings to the same
1649          * physical page.
1650          */
1651         if (OVERRIDE_CACHE_ATTR(flags)) {
1652                 iomin = (iomin + MMU_PAGEOFFSET) & MMU_PAGEMASK;
1653                 length = (length + MMU_PAGEOFFSET) & (size_t)MMU_PAGEMASK;
1654         }

1656         /*
1657          * Determine if we need to satisfy the request for physically
1658          * contiguous memory or alignments larger than pagesize.
1659          */
1660         npages = btopr(length + attr->dma_attr_align);
1661         minctg = howmany(npages, attr->dma_attr_sgllen);

1663         if (minctg > 1) {
1664                 uint64_t pfnseg = attr->dma_attr_seg >> PAGESHIFT;
1665                 /*
1666                  * verify that the minimum contig requirement for the
1667                  * actual length does not cross segment boundary.
1668                  */
1669                 length = P2ROUNDUP_TYPED(length, attr->dma_attr_minxfer,
1670                     size_t);
1671                 npages = btopr(length);
1672                 minctg = howmany(npages, attr->dma_attr_sgllen);
1673                 if (minctg > pfnseg + 1)
1674                         return (DDI_FAILURE);
1675                 physcontig = 1;
1676         } else {
1677                 length = P2ROUNDUP_TYPED(length, iomin, size_t);
1678         }

1680         /*
1681          * Allocate the requested amount from the system.
1682          */
1683         a = kalloca(length, iomin, cansleep, physcontig, attr);

1685         if ((*kaddrp = a) == NULL)
1686                 return (DDI_FAILURE);

1688         /*
1689          * if we to modify the cache attributes, go back and muck with the
1690          * mappings.
1691          */
1692         if (OVERRIDE_CACHE_ATTR(flags)) {
1693                 order = 0;
```

```
1694                 i_ddi_cacheattr_to_hatacc(flags, &order);
1695                 e = kmem_override_cache_attrs(a, length, order);
1696                 if (e != 0) {
1697                         kfreea(a);
1698                         return (DDI_FAILURE);
1699                 }
1700         }

1702         if (real_length) {
1703                 *real_length = length;
1704         }
1705         if (ap) {
1706                 /*
1707                  * initialize access handle
1708                  */
1709                 iap = (ddi_acc_impl_t *)ap->ah_platform_private;
1710                 iap->ahi_acc_attr |= DDI_ACCATTR_CPU_VADDR;
1711                 impl_acc_hdl_init(ap);
1712         }

1714         return (DDI_SUCCESS);
1715 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   12005 Thu Oct 23 10:42:18 2014
new/usr/src/uts/intel/io/drm/i915_gem_tiling.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
   1 /* BEGIN CSTYLED */

   3 /*
   4  * Copyright (c) 2009, Intel Corporation.
   5  * All Rights Reserved.
   6  *
   7  * Permission is hereby granted, free of charge, to any person obtaining a
   8  * copy of this software and associated documentation files (the "Software"),
   9  * to deal in the Software without restriction, including without limitation
  10  * the rights to use, copy, modify, merge, publish, distribute, sublicense,
  11  * and/or sell copies of the Software, and to permit persons to whom the
  12  * Software is furnished to do so, subject to the following conditions:
  13  *
  14  * The above copyright notice and this permission notice (including the next
  15  * paragraph) shall be included in all copies or substantial portions of the
  16  * Software.
  17  *
  18  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
  19  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
  20  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
  21  * THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
  22  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
  23  * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
  24  * IN THE SOFTWARE.
  25  *
  26  * Authors:
  27  *    Eric Anholt <eric@anholt.net>
  28  *
  29  */

  31 /*
  32  * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
  33  * Use is subject to license terms.
  34  */

  36 #include <sys/sysmacros.h>
  37 #endif /* ! codereview */
  38 #include "drmP.h"
  39 #include "drm.h"
  40 #include "i915_drm.h"
  41 #include "i915_drv.h"

  43 /** @file i915_gem_tiling.c
  44  *
  45  * Support for managing tiling state of buffer objects.
  46  *
  47  * The idea behind tiling is to increase cache hit rates by rearranging
  48  * pixel data so that a group of pixel accesses are in the same cacheline.
  49  * Performance improvement from doing this on the back/depth buffer are on
  50  * the order of 30%.
  51  *
  52  * Intel architectures make this somewhat more complicated, though, by
  53  * adjustments made to addressing of data when the memory is in interleaved
  54  * mode (matched pairs of DIMMS) to improve memory bandwidth.
  55  * For interleaved memory, the CPU sends every sequential 64 bytes
  56  * to an alternate memory channel so it can get the bandwidth from both.
  57  *
  58  * The GPU also rearranges its accesses for increased bandwidth to interleaved
  59  * memory, and it matches what the CPU does for non-tiled.  However, when tiled
  60  * it does it a little differently, since one walks addresses not just in the
  61  * X direction but also Y.  So, along with alternating channels when bit
```



```
  62  * 6 of the address flips, it also alternates when other bits flip --  Bits 9
  63  * (every 512 bytes, an X tile scanline) and 10 (every two X tile scanlines)
  64  * are common to both the 915 and 965-class hardware.
  65  *
  66  * The CPU also sometimes XORs in higher bits as well, to improve
  67  * bandwidth doing strided access like we do so frequently in graphics.  This
  68  * is called "Channel XOR Randomization" in the MCH documentation.  The result
  69  * is that the CPU is XORing in either bit 11 or bit 17 to bit 6 of its address
  70  * decode.
  71  *
  72  * All of this bit 6 XORing has an effect on our memory management,
  73  * as we need to make sure that the 3d driver can correctly address object
  74  * contents.
  75  *
  76  * If we don't have interleaved memory, all tiling is safe and no swizzling is
  77  * required.
  78  *
  79  * When bit 17 is XORed in, we simply refuse to tile at all.  Bit
  80  * 17 is not just a page offset, so as we page an objet out and back in,
  81  * individual pages in it will have different bit 17 addresses, resulting in
  82  * each 64 bytes being swapped with its neighbor!
  83  *
  84  * Otherwise, if interleaved, we have to tell the 3d driver what the address
  85  * swizzling it needs to do is, since it's writing with the CPU to the pages
  86  * (bit 6 and potentially bit 11 XORed in), and the GPU is reading from the
  87  * pages (bit 6, 9, and 10 XORed in), resulting in a cumulative bit swizzling
  88  * required by the CPU of XORing in bit 6, 9, 10, and potentially 11, in order
  89  * to match what the GPU expects.
  90  */

  92 /**
  93  * Detects bit 6 swizzling of address lookup between IGD access and CPU
  94  * access through main memory.
  95  */
  96 void
  97 i915_gem_detect_bit_6_swizzle(struct drm_device *dev)
  98 {
  99         drm_i915_private_t *dev_priv = dev->dev_private;
 100         uint32_t swizzle_x = I915_BIT_6_SWIZZLE_UNKNOWN;
 101         uint32_t swizzle_y = I915_BIT_6_SWIZZLE_UNKNOWN;

 103         if (!IS_I9XX(dev)) {
 104                 /* As far as we know, the 865 doesn't have these bit 6
 105                  * swizzling issues.
 106                  */
 107                 swizzle_x = I915_BIT_6_SWIZZLE_NONE;
 108                 swizzle_y = I915_BIT_6_SWIZZLE_NONE;
 109                 } else if (IS_MOBILE(dev)) {
 110                 uint32_t dcc;

 112                 /* On mobile 9xx chipsets, channel interleave by the CPU is
 113                  * determined by DCC.  For single-channel, neither the CPU
 114                  * nor the GPU do swizzling.  For dual channel interleaved,
 115                  * the GPU's interleave is bit 9 and 10 for X tiled, and bit
 116                  * 9 for Y tiled.  The CPU's interleave is independent, and
 117                  * can be based on either bit 11 (haven't seen this yet) or
 118                  * bit 17 (common).
 119                  */

 121                 dcc = I915_READ(DCC);
 122                 switch (dcc & DCC_ADDRESSING_MODE_MASK) {
 123                 case DCC_ADDRESSING_MODE_SINGLE_CHANNEL:
 124                 case DCC_ADDRESSING_MODE_DUAL_CHANNEL_ASYMMETRIC:
 125                         swizzle_x = I915_BIT_6_SWIZZLE_NONE;
 126                         swizzle_y = I915_BIT_6_SWIZZLE_NONE;
 127                         break;
```

```
128                 case DCC_ADDRESSING_MODE_DUAL_CHANNEL_INTERLEAVED:
129                         if (dcc & DCC_CHANNEL_XOR_DISABLE) {
130                                 /* This is the base swizzling by the GPU for
131                                  * tiled buffers.
132                                  */
133                                 swizzle_x = I915_BIT_6_SWIZZLE_9_10;
134                                 swizzle_y = I915_BIT_6_SWIZZLE_9;
135                         } else if ((dcc & DCC_CHANNEL_XOR_BIT_17) == 0) {
136                                 /* Bit 11 swizzling by the CPU in addition. */
137                                 swizzle_x = I915_BIT_6_SWIZZLE_9_10_11;
138                                 swizzle_y = I915_BIT_6_SWIZZLE_9_11;
139                         } else {
140                                 /* Bit 17 swizzling by the CPU in addition. */
141                                 swizzle_x = I915_BIT_6_SWIZZLE_UNKNOWN;
142                                 swizzle_y = I915_BIT_6_SWIZZLE_UNKNOWN;
143                         }
144                         break;
145                 }
146                 if (dcc == 0xffffffff) {
147                         DRM_ERROR("Couldn't read from MCHBAR.  "
148                                   "Disabling tiling.\n");
149                         swizzle_x = I915_BIT_6_SWIZZLE_UNKNOWN;
150                         swizzle_y = I915_BIT_6_SWIZZLE_UNKNOWN;
151                 }
152         } else {
153                 /* The 965, G33, and newer, have a very flexible memory
154                  * configuration.  It will enable dual-channel mode
155                  * (interleaving) on as much memory as it can, and the GPU
156                  * will additionally sometimes enable different bit 6
157                  * swizzling for tiled objects from the CPU.
158                  *
159                  * Here's what I found on the G965:
160                  *    slot fill         memory size  swizzling
161                  * 0A   0B   1A   1B    1-ch   2-ch
162                  * 512  0    0    0     512    0     O
163                  * 512  0    512  0     16     1008  X
164                  * 512  0    0    512   16     1008  X
165                  * 0    512  0    512   16     1008  X
166                  * 1024 1024 1024 0     2048   1024  O
167                  *
168                  * We could probably detect this based on either the DRB
169                  * matching, which was the case for the swizzling required in
170                  * the table above, or from the 1-ch value being less than
171                  * the minimum size of a rank.
172                  */
173                 if (I915_READ16(C0DRB3) != I915_READ16(C1DRB3)) {
174                         swizzle_x = I915_BIT_6_SWIZZLE_NONE;
175                         swizzle_y = I915_BIT_6_SWIZZLE_NONE;
176                 } else {
177                         swizzle_x = I915_BIT_6_SWIZZLE_9_10;
178                         swizzle_y = I915_BIT_6_SWIZZLE_9;
179                 }
180         }
181
182         /* FIXME: check with memory config on IGDNG */
183         if (IS_IGDNG(dev)) {
184                 swizzle_x = I915_BIT_6_SWIZZLE_9_10;
185                 swizzle_y = I915_BIT_6_SWIZZLE_9;
186         }
187
188         dev_priv->mm.bit_6_swizzle_x = swizzle_x;
189         dev_priv->mm.bit_6_swizzle_y = swizzle_y;
190 }
191
192
193 /**
```



```
194  * Returns the size of the fence for a tiled object of the given size.
195  */
196 static int
197 i915_get_fence_size(struct drm_device *dev, int size)
198 {
199         int i;
200         int start;
201
202         if (IS_I965G(dev)) {
203                 /* The 965 can have fences at any page boundary. */
204
205                 return (size + PAGE_SIZE-1) & ~(PAGE_SIZE-1);
206         } else {
207                 /* Align the size to a power of two greater than the smallest
208                  * fence size.
209                  */
210                 if (IS_I9XX(dev))
211                         start = 1024 * 1024;
212                 else
213                         start = 512 * 1024;
214
215                 for (i = start; i < size; i <<= 1)
216                         ;
217
218                 return i;
219         }
220 }
221
222 /* Check pitch constriants for all chips & tiling formats */
223 static int
224 i915_tiling_ok(struct drm_device *dev, int stride, int size, int tiling_mode)
225 {
226         int tile_width;
227
228         /* Linear is always fine */
229         if (tiling_mode == I915_TILING_NONE)
230                 return 1;
231
232         if (tiling_mode == I915_TILING_Y && HAS_128_BYTE_Y_TILING(dev))
233                 tile_width = 128;
234         else
235                 tile_width = 512;
236
237         if (stride == 0)
238                 return 0;
239
240         /* 965+ just needs multiples of tile width */
241         if (IS_I965G(dev)) {
242                 if (stride & (tile_width - 1))
243                         return 0;
244                 return 1;
245         }
246
247         /* Pre-965 needs power of two tile widths */
248         if (stride < tile_width)
249                 return 0;
250
251         if (!ISP2(stride))
 36         if (stride & (stride - 1))
252                 return 0;
253
254         /* We don't handle the aperture area covered by the fence being bigger
255          * than the object size.
256          */
257         if (i915_get_fence_size(dev, size) != size)
258                 return 0;
```

```
 260         return 1;
 261 }
______unchanged_portion_omitted_
```

```
**********************************************************
    6968 Thu Oct 23 10:42:18 2014
new/usr/src/uts/sun4/io/px/px_debug.c
5255 uts shouldn't open-code ISP2
**********************************************************
/*
 * 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 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * PCI nexus driver general debug support
 */
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/async.h>
#include <sys/sunddi.h>         /* dev_info_t */
#include <sys/ddi_impldefs.h>
#include <sys/disp.h>
#include <sys/archsystm.h>      /* getpil() */
#include "px_obj.h"

/*LINTLIBRARY*/

#ifdef  DEBUG
uint64_t px_debug_flags = 0;

static char *px_debug_sym [] = {        /* same sequence as px_debug_bit */
        /*  0 */ "attach",
        /*  1 */ "detach",
        /*  2 */ "map",
        /*  3 */ "nex-ctlops",

        /*  4 */ "introps",
        /*  5 */ "intx-add",
        /*  6 */ "intx-rem",
        /*  7 */ "intx-intr",

        /*  8 */ "msiq",
        /*  9 */ "msiq-intr",
        /* 10 */ "msg",
        /* 11 */ "msg-intr",

        /* 12 */ "msix-add",
```



```
        /* 13 */ "msix-rem",
        /* 14 */ "msix-intr",
        /* 15 */ "err",

        /* 16 */ "dma-alloc",
        /* 17 */ "dma-free",
        /* 18 */ "dma-bind",
        /* 19 */ "dma-unbind",

        /* 20 */ "chk-dma-mode",
        /* 21 */ "bypass-dma",
        /* 22 */ "fast-dvma",
        /* 23 */ "init_child",

        /* 24 */ "dma-map",
        /* 25 */ "dma-win",
        /* 26 */ "map-win",
        /* 27 */ "unmap-win",

        /* 28 */ "dma-ctl",
        /* 29 */ "dma-sync",
        /* 30 */ NULL,
        /* 31 */ NULL,

        /* 32 */ "ib",
        /* 33 */ "cb",
        /* 34 */ "dmc",
        /* 35 */ "pec",

        /* 36 */ "ilu",
        /* 37 */ "tlu",
        /* 38 */ "lpu",
        /* 39 */ "mmu",

        /* 40 */ "open",
        /* 41 */ "close",
        /* 42 */ "ioctl",
        /* 43 */ "pwr",

        /* 44 */ "lib-cfg",
        /* 45 */ "lib-intr",
        /* 46 */ "lib-dma",
        /* 47 */ "lib-msiq",

        /* 48 */ "lib-msi",
        /* 49 */ "lib-msg",
        /* 50 */ "NULL",
        /* 51 */ "NULL",

        /* 52 */ "tools",
        /* 53 */ "phys_acc",

        /* 54 */ "hotplug",
        /* LAST */ "unknown"
};

/* Tunables */
static int px_dbg_msg_size = 16;                /* # of Qs.  Must be ^2 */

/* Non-Tunables */
static int px_dbg_qmask = 0xFFFF;               /* Mask based on Q size */
static px_dbg_msg_t *px_dbg_msgq = NULL;        /* Debug Msg Queue */
static uint8_t px_dbg_reference = 0;            /* Reference Counter */
static kmutex_t px_dbg_mutex;                   /* Mutex for dequeuing */
static uint8_t px_dbg_qtail = 0;                /* Pointer to q tail */
static uint8_t px_dbg_qhead = 0;                /* Pointer to q head */
```

```
126 static uint_t px_dbg_qsize = 0;                /* # of pending messages */
127 static uint_t px_dbg_failed = 0;               /* # of overflows */

129 /* Forward Declarations */
130 static void px_dbg_print(px_debug_bit_t bit, dev_info_t *dip, char *fmt,
131     va_list args);
132 static void px_dbg_queue(px_debug_bit_t bit, dev_info_t *dip, char *fmt,
133     va_list args);
134 static uint_t px_dbg_drain(caddr_t arg1, caddr_t arg2);

136 /*
137  * Print function called either directly by px_dbg or through soft interrupt.
138  * This function cannot be called directly in threads with PIL above clock.
139  */
140 static void
141 px_dbg_print(px_debug_bit_t bit, dev_info_t *dip, char *fmt, va_list args)
142 {
143         int cont = bit >> DBG_BITS;

145         if (cont)
146                 goto body;

148         if (dip)
149                 prom_printf("%s(%d): %s: ", ddi_driver_name(dip),
150                     ddi_get_instance(dip), px_debug_sym[bit]);
151         else
152                 prom_printf("px: %s: ", px_debug_sym[bit]);
153 body:
154         if (args)
155                 prom_vprintf(fmt, args);
156         else
157                 prom_printf(fmt);
158 }

160 /*
161  * Queueing mechanism to log px_dbg messages if calling thread is running with a
162  * PIL above clock. It's Multithreaded safe.
163  */
164 static void
165 px_dbg_queue(px_debug_bit_t bit, dev_info_t *dip, char *fmt, va_list args)
166 {
167         int             instance = DIP_TO_INST(dip);
168         px_t            *px_p = INST_TO_STATE(instance);
169         uint8_t         q_no;
170         px_dbg_msg_t    *msg_p;

172         /* Check to make sure the queue hasn't overflowed */
173         if (atomic_inc_uint_nv(&px_dbg_qsize) >= px_dbg_msg_size) {
174                 px_dbg_failed++;
175                 atomic_dec_uint(&px_dbg_qsize);
176                 return;
177         }

179         /*
180          * Grab the next available queue bucket. Incrementing the tail here
181          * doesn't need to be protected, as it is guaranteed to not overflow.
182          */
183         q_no = ++px_dbg_qtail & px_dbg_qmask;
184         msg_p = &px_dbg_msgq[q_no];

186         ASSERT(msg_p->active == B_FALSE);

188         /* Print the message in the buffer */
189         vsnprintf(msg_p->msg, DBG_MSG_SIZE, fmt, args);
190         msg_p->bit = bit;
191         msg_p->dip = dip;
```



```
192         msg_p->active = B_TRUE;

194         /* Trigger Soft Int */
195         ddi_intr_trigger_softint(px_p->px_dbg_hdl, (caddr_t)NULL);
196 }

198 /*
199  * Callback function for queuing px_dbg in high PIL by soft intr.  This code
200  * assumes it will be called serially for every msg.
201  */
202 static uint_t
203 px_dbg_drain(caddr_t arg1, caddr_t arg2) {
204         uint8_t         q_no;
205         px_dbg_msg_t    *msg_p;
206         uint_t          ret = DDI_INTR_UNCLAIMED;

208         mutex_enter(&px_dbg_mutex);
209         while (px_dbg_qsize) {
210                 atomic_dec_uint(&px_dbg_qsize);
211                 if (px_dbg_failed) {
212                         cmn_err(CE_WARN, "%d msg(s) were lost",
213                             px_dbg_failed);
214                         px_dbg_failed = 0;
215                 }

217                 q_no = ++px_dbg_qhead & px_dbg_qmask;
218                 msg_p = &px_dbg_msgq[q_no];

220                 if (msg_p->active) {
221                         px_dbg_print(msg_p->bit, msg_p->dip, msg_p->msg, NULL);
222                         msg_p->active = B_FALSE;
223                 }
224                 ret = DDI_INTR_CLAIMED;
225         }

227         mutex_exit(&px_dbg_mutex);
228         return (ret);
229 }

231 void
232 px_dbg(px_debug_bit_t bit, dev_info_t *dip, char *fmt, ...)
233 {
234         va_list ap;

236         bit &= DBG_MASK;
237         if (bit >= sizeof (px_debug_sym) / sizeof (char *))
238                 return;
239         if (!(1ull << bit & px_debug_flags))
240                 return;

242         va_start(ap, fmt);
243         if (getpil() > LOCK_LEVEL)
244                 px_dbg_queue(bit, dip, fmt, ap);
245         else
246                 px_dbg_print(bit, dip, fmt, ap);
247         va_end(ap);
248 }
249 #endif  /* DEBUG */

251 void
252 px_dbg_attach(dev_info_t *dip, ddi_softint_handle_t *dbg_hdl)
253 {
254 #ifdef  DEBUG
255         if (px_dbg_reference++ == 0) {
256                 int size = px_dbg_msg_size;
```

```
258                 /* Check if px_dbg_msg_size is ^2 */
259                 size = !ISP2(size) ? ((size | ~size) + 1) : size;
 31                 size = (size & (size - 1)) ? ((size | ~size) + 1) : size;
260                 px_dbg_msg_size = size;
261                 px_dbg_qmask = size - 1;
262                 px_dbg_msgq = kmem_zalloc(sizeof (px_dbg_msg_t) * size,
263                     KM_SLEEP);

265                 mutex_init(&px_dbg_mutex, NULL, MUTEX_DRIVER, NULL);
266         }

268         if (ddi_intr_add_softint(dip, dbg_hdl,
269                 DDI_INTR_SOFTPRI_MAX, px_dbg_drain, NULL) != DDI_SUCCESS) {
270                 DBG(DBG_ATTACH, dip,
271                     "Unable to allocate soft int for DBG printing.\n");
272                 dbg_hdl = NULL;
273         }
274 #endif  /* DEBUG */
275 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   49366 Thu Oct 23 10:42:18 2014
new/usr/src/uts/sun4/os/ddi_impl.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 992 /*
 993  * Check if the specified cache attribute is supported on the platform.
 994  * This function must be called before i_ddi_cacheattr_to_hatacc().
 995  */
 996 boolean_t
 997 i_ddi_check_cache_attr(uint_t flags)
 998 {
 999         /*
1000          * The cache attributes are mutually exclusive. Any combination of
1001          * the attributes leads to a failure.
1002          */
1003         uint_t cache_attr = IOMEM_CACHE_ATTR(flags);
1004         if ((cache_attr != 0) && !ISP2(cache_attr))
1004         if ((cache_attr != 0) && ((cache_attr & (cache_attr - 1)) != 0))
1005                 return (B_FALSE);

1007         /*
1008          * On the sparc architecture, only IOMEM_DATA_CACHED is meaningful,
1009          * but others lead to a failure.
1010          */
1011         if (cache_attr & IOMEM_DATA_CACHED)
1012                 return (B_TRUE);
1013         else
1014                 return (B_FALSE);
1015 }
______unchanged_portion_omitted_

1134 /*
1135  * This used to be ddi_iomin, but we were the only remaining caller, so
1136  * we've made it private and moved it here.
1137  */
1138 static int
1139 i_ddi_iomin(dev_info_t *a, int i, int stream)
1140 {
1141         int r;

1143         /*
1144          * Make sure that the initial value is sane
1145          */
1146         if (!ISP2(i))
1146         if (i & (i - 1))
1147                 return (0);
1148         if (i == 0)
1149                 i = (stream) ? 4 : 1;

1151         r = ddi_ctlops(a, a,
1152             DDI_CTLOPS_IOMIN, (void *)(uintptr_t)stream, (void *)&i);
1153         if (r != DDI_SUCCESS || !ISP2(i))
1153         if (r != DDI_SUCCESS || (i & (i - 1)))
1154                 return (0);
1155         return (i);
1156 }

1158 int
1159 i_ddi_mem_alloc(dev_info_t *dip, ddi_dma_attr_t *attr,
1160     size_t length, int cansleep, int flags,
1161     ddi_device_acc_attr_t *accattrp,
1162     caddr_t *kaddrp, size_t *real_length, ddi_acc_hdl_t *handlep)
1163 {
```



```
1164         caddr_t a;
1165         int iomin, align, streaming;
1166         uint_t endian_flags = DDI_NEVERSWAP_ACC;

1168 #if defined(lint)
1169         *handlep = *handlep;
1170 #endif

1172         /*
1173          * Check legality of arguments
1174          */
1175         if (length == 0 || kaddrp == NULL || attr == NULL) {
1176                 return (DDI_FAILURE);
1177         }

1179         if (attr->dma_attr_minxfer == 0 || attr->dma_attr_align == 0 ||
1180             !ISP2(attr->dma_attr_align) || !ISP2(attr->dma_attr_minxfer)) {
1180             (attr->dma_attr_align & (attr->dma_attr_align - 1)) ||
1181             (attr->dma_attr_minxfer & (attr->dma_attr_minxfer - 1))) {
1181                 return (DDI_FAILURE);
1182         }

1184         /*
1185          * check if a streaming sequential xfer is requested.
1186          */
1187         streaming = (flags & DDI_DMA_STREAMING) ? 1 : 0;

1189         /*
1190          * Drivers for 64-bit capable SBus devices will encode
1191          * the burtsizes for 64-bit xfers in the upper 16-bits.
1192          * For DMA alignment, we use the most restrictive
1193          * alignment of 32-bit and 64-bit xfers.
1194          */
1195         iomin = (attr->dma_attr_burstsizes & 0xffff) |
1196             ((attr->dma_attr_burstsizes >> 16) & 0xffff);
1197         /*
1198          * If a driver set burtsizes to 0, we give him byte alignment.
1199          * Otherwise align at the burtsizes boundary.
1200          */
1201         if (iomin == 0)
1202                 iomin = 1;
1203         else
1204                 iomin = 1 << (ddi_fls(iomin) - 1);
1205         iomin = maxbit(iomin, attr->dma_attr_minxfer);
1206         iomin = maxbit(iomin, attr->dma_attr_align);
1207         iomin = i_ddi_iomin(dip, iomin, streaming);
1208         if (iomin == 0)
1209                 return (DDI_FAILURE);

1211         ASSERT((iomin & (iomin - 1)) == 0);
1212         ASSERT(iomin >= attr->dma_attr_minxfer);
1213         ASSERT(iomin >= attr->dma_attr_align);

1215         length = P2ROUNDUP(length, iomin);
1216         align = iomin;

1218         if (accattrp != NULL)
1219                 endian_flags = accattrp->devacc_attr_endian_flags;

1221         a = kalloca(length, align, cansleep, endian_flags);
1222         if ((*kaddrp = a) == 0) {
1223                 return (DDI_FAILURE);
1224         } else {
1225                 if (real_length) {
1226                         *real_length = length;
1227                 }
```

```
1228                 if (handlep) {
1229                         /*
1230                          * assign handle information
1231                          */
1232                         impl_acc_hdl_init(handlep);
1233                 }
1234                 return (DDI_SUCCESS);
1235         }
1236 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21105 Thu Oct 23 10:42:18 2014
new/usr/src/uts/sun4v/io/ds_pri.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_

 872 /*
 873  * Routine to get static PRI data from the Hypervisor.
 874  * If successful, this PRI data is the last known PRI
 875  * data generated since the last poweron reset.
 876  */
 877 static uint64_t
 878 ds_get_hv_pri(ds_pri_state_t *sp)
 879 {
 880         uint64_t        status;
 881         uint64_t        pri_size;
 882         uint64_t        buf_size;
 883         uint64_t        buf_pa;
 884         caddr_t         buf_va = NULL;
 885         caddr_t         pri_data;

 887         /*
 888          * Get pri buffer size by calling hcall with buffer size 0.
 889          */
 890         pri_size = 0LL;
 891         status = hv_mach_pri((uint64_t)0, &pri_size);
 892         if (status == H_ENOTSUPPORTED || status == H_ENOACCESS) {
 893                 /*
 894                  * hv_mach_pri() is not supported on a guest domain.
 895                  * Unregister pboot API group to prevent failures.
 896                  */
 897                 (void) hsvc_unregister(&pboot_hsvc);
 898                 hsvc_pboot_available = B_FALSE;
 899                 DS_PRI_DBG("ds_get_hv_pri: hv_mach_pri service is not "
 900                     "available. errorno: 0x%lx\n", status);
 901                 return (0);
 902         } else if (pri_size == 0) {
 903                 return (1);
 904         } else {
 905                 DS_PRI_DBG("ds_get_hv_pri: hv_mach_pri pri size: 0x%lx\n",
 906                     pri_size);
 907         }

 909         /*
 910          * contig_mem_alloc requires size to be a power of 2.
 911          * Increase size to next power of 2 if necessary.
 912          */
 913         if (!ISP2(pri_size))
 913         if ((pri_size & (pri_size - 1)) != 0)
 914                 buf_size = 1 << highbit(pri_size);
 915         DS_PRI_DBG("ds_get_hv_pri: buf_size = 0x%lx\n", buf_size);

 917         buf_va = contig_mem_alloc(buf_size);
 918         if (buf_va == NULL)
 919                 return (1);

 921         buf_pa = va_to_pa(buf_va);
 922         DS_PRI_DBG("ds_get_hv_pri: buf_pa 0x%lx\n", buf_pa);
 923         status = hv_mach_pri(buf_pa, &pri_size);
 924         DS_PRI_DBG("ds_get_hv_pri: hv_mach_pri status = 0x%lx\n", status);

 926         if (status == H_EOK) {
 927                 pri_data = kmem_alloc(pri_size, KM_SLEEP);
 928                 sp->ds_pri = pri_data;
 929                 sp->ds_pri_len = pri_size;
```



```
 930                 bcopy(buf_va, pri_data, sp->ds_pri_len);
 931                 sp->state |= DS_PRI_HAS_PRI;
 932                 sp->gencount++;
 933         }

 935         contig_mem_free(buf_va, buf_size);

 937         return (status);
 938 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36563 Thu Oct 23 10:42:18 2014
new/usr/src/uts/sun4v/os/fillsysinfo.c
5255 uts shouldn't open-code ISP2
**********************************************************
______unchanged_portion_omitted_
```

```
 363 /*
 364  * All the common setup of sun4v CPU modules is done by this routine.
 365  */
 366 void
 367 cpu_setup_common(char **cpu_module_isa_set)
 368 {
 369         extern int mmu_exported_pagesize_mask;
 370         int nocpus, i;
 371         size_t ra_limit;
 372         mde_cookie_t *cpulist;
 373         md_t *mdp;

 375         if ((mdp = md_get_handle()) == NULL)
 376                 cmn_err(CE_PANIC, "Unable to initialize machine description");

 378         boot_ncpus = nocpus = md_alloc_scan_dag(mdp,
 379             md_root_node(mdp), "cpu", "fwd", &cpulist);
 380         if (nocpus < 1) {
 381                 cmn_err(CE_PANIC, "cpu_common_setup: cpulist allocation "
 382                     "failed or incorrect number of CPUs in MD");
 383         }

 385         init_md_broken(mdp, cpulist);

 387         if (use_page_coloring) {
 388                 do_pg_coloring = 1;
 389         }

 391         /*
 392          * Get the valid mmu page sizes mask, Q sizes and isalist/r
 393          * from the MD for the first available CPU in cpulist.
 394          *
 395          * Do not expect the MMU page sizes mask to be more than 32-bit.
 396          */
 397         mmu_exported_pagesize_mask = (int)get_cpu_pagesizes(mdp, cpulist[0]);

 399         /*
 400          * Get the number of contexts and tsbs supported.
 401          */
 402         if (get_mmu_shcontexts(mdp, cpulist[0]) >= MIN_NSHCONTEXTS &&
 403             get_mmu_tsbs(mdp, cpulist[0]) >= MIN_NTSBS) {
 404                 shctx_on = 1;
 405         }

 407         for (i = 0; i < nocpus; i++)
 408                 fill_cpu(mdp, cpulist[i]);

 410         /* setup l2 cache count. */
 411         n_l2_caches = get_l2_cache_node_count(mdp);

 413         setup_chip_mappings(mdp);
 414         setup_exec_unit_mappings(mdp);

 416         /*
 417          * If MD is broken then append the passed ISA set,
 418          * otherwise trust the MD.
 419          */

 421         if (broken_md_flag)
```



```
 422                 isa_list = construct_isalist(mdp, cpulist[0],
 423                     cpu_module_isa_set);
 424         else
 425                 isa_list = construct_isalist(mdp, cpulist[0], NULL);

 427         get_hwcaps(mdp, cpulist[0]);
 428         get_weakest_mem_model(mdp, cpulist[0]);
 429         get_q_sizes(mdp, cpulist[0]);
 430         get_va_bits(mdp, cpulist[0]);

 432         /*
 433          * ra_limit is the highest real address in the machine.
 434          */
 435         ra_limit = get_ra_limit(mdp, cpulist[0]);

 437         md_free_scan_dag(mdp, &cpulist);

 439         (void) md_fini_handle(mdp);

 441         /*
 442          * Block stores invalidate all pages of the d$ so pagecopy
 443          * et. al. do not need virtual translations with virtual
 444          * coloring taken into consideration.
 445          */
 446         pp_consistent_coloring = 0;

 448         /*
 449          * The kpm mapping window.
 450          * kpm_size:
 451          *      The size of a single kpm range.
 452          *      The overall size will be: kpm_size * vac_colors.
 453          * kpm_vbase:
 454          *      The virtual start address of the kpm range within the kernel
 455          *      virtual address space. kpm_vbase has to be kpm_size aligned.
 456          */

 458         /*
 459          * Make kpm_vbase, kpm_size aligned to kpm_size_shift.
 460          * To do this find the nearest power of 2 size that the
 461          * actual ra_limit fits within.
 462          * If it is an even power of two use that, otherwise use the
 463          * next power of two larger than ra_limit.
 464          */

 466         ASSERT(ra_limit != 0);

 468         kpm_size_shift = !ISP2(ra_limit) ?
 468         kpm_size_shift = (ra_limit & (ra_limit - 1)) != 0 ?
 469             highbit(ra_limit) : highbit(ra_limit) - 1;

 471         /*
 472          * No virtual caches on sun4v so size matches size shift
 473          */
 474         kpm_size = 1ul << kpm_size_shift;

 476         if (va_bits < VA_ADDRESS_SPACE_BITS) {
 477                 /*
 478                  * In case of VA hole
 479                  * kpm_base = hole_end + 1TB
 480                  * Starting 1TB beyond where VA hole ends because on Niagara
 481                  * processor software must not use pages within 4GB of the
 482                  * VA hole as instruction pages to avoid problems with
 483                  * prefetching into the VA hole.
 484                  */
 485                 kpm_vbase = (caddr_t)((0ull - (1ull << (va_bits - 1))) +
 486                     (1ull << 40));
```

```
        } else {                /* Number of VA bits 64 ... no VA hole */
                kpm_vbase = (caddr_t)0x8000000000000000ull;     /* 8 EB */
        }

        /*
         * The traptrace code uses either %tick or %stick for
         * timestamping.  The sun4v require use of %stick.
         */
        traptrace_use_stick = 1;
}
```

unchanged_portion_omitted_

```
/*
 * This routine sets the globals for CPU and DEV mondo queue entries and
 * resumable and non-resumable error queue entries.
 *
 * First, look up the number of bits available to pass an entry number.
 * This can vary by platform and may result in allocating an unreasonably
 * (or impossibly) large amount of memory for the corresponding table,
 * so we clamp it by 'max_entries'.  Finally, since the q size is used when
 * calling contig_mem_alloc(), which expects a power of 2, clamp the q size
 * down to a power of 2.  If the prop is missing, use 'default_entries'.
 */
static uint64_t
get_single_q_size(md_t *mdp, mde_cookie_t cpu_node_cookie,
    char *qnamep, uint64_t default_entries, uint64_t max_entries)
{
        uint64_t entries;

        if (default_entries > max_entries)
                cmn_err(CE_CONT, "!get_single_q_size: dflt %ld > "
                    "max %ld for %s\n", default_entries, max_entries, qnamep);

        if (md_get_prop_val(mdp, cpu_node_cookie, qnamep, &entries)) {
                if (!broken_md_flag)
                        cmn_err(CE_PANIC, "Missing %s property in MD cpu node",
                            qnamep);
                entries = default_entries;
        } else {
                entries = 1 << entries;
        }

        entries = MIN(entries, max_entries);
        /* If not a power of 2, truncate to a power of 2. */
        if (!ISP2(entries)) {
        if ((entries & (entries - 1)) != 0) {
                entries = 1 << (highbit(entries) - 1);
        }

        return (entries);
}
```

unchanged_portion_omitted_

```
**********************************************************
    6737 Thu Oct 23 10:42:19 2014
new/usr/src/uts/sun4v/os/intrq.c
5255 uts shouldn't open-code ISP2
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/machsystm.h>
#include <sys/cpu.h>
#include <sys/intreg.h>
#include <sys/machcpuvar.h>
#include <vm/hat_sfmmu.h>
#include <sys/error.h>
#include <sys/hypervisor_api.h>

void
cpu_intrq_register(struct cpu *cpu)
{
        struct machcpu *mcpup = &cpu->cpu_m;
        uint64_t ret;

        ret = hv_cpu_qconf(INTR_CPU_Q, mcpup->cpu_q_base_pa, cpu_q_entries);
        if (ret != H_EOK)
                cmn_err(CE_PANIC, "cpu%d: cpu_mondo queue configuration "
                    "failed, error %lu", cpu->cpu_id, ret);

        ret = hv_cpu_qconf(INTR_DEV_Q, mcpup->dev_q_base_pa, dev_q_entries);
        if (ret != H_EOK)
                cmn_err(CE_PANIC, "cpu%d: dev_mondo queue configuration "
                    "failed, error %lu", cpu->cpu_id, ret);

        ret = hv_cpu_qconf(CPU_RQ, mcpup->cpu_rq_base_pa, cpu_rq_entries);
        if (ret != H_EOK)
                cmn_err(CE_PANIC, "cpu%d: resumable error queue configuration "
                    "failed, error %lu", cpu->cpu_id, ret);

        ret = hv_cpu_qconf(CPU_NRQ, mcpup->cpu_nrq_base_pa, cpu_nrq_entries);
        if (ret != H_EOK)
                cmn_err(CE_PANIC, "cpu%d: non-resumable error queue "
                    "configuration failed, error %lu", cpu->cpu_id, ret);
}
```



```
int
cpu_intrq_setup(struct cpu *cpu)
{
        struct machcpu *mcpup = &cpu->cpu_m;
        size_t size;

        /*
         * This routine will return with an error return if any
         * contig_mem_alloc() fails.  It is expected that the caller will
         * call cpu_intrq_cleanup() (or cleanup_cpu_common() which will).
         * That will cleanly free only those blocks that were alloc'd.
         */

        /*
         * Allocate mondo data for xcalls.
         */
        mcpup->mondo_data = contig_mem_alloc(INTR_REPORT_SIZE);

        if (mcpup->mondo_data == NULL) {
                cmn_err(CE_NOTE, "cpu%d: cpu mondo_data allocation failed",
                    cpu->cpu_id);
                return (ENOMEM);
        }
        /*
         * va_to_pa() is too expensive to call for every crosscall
         * so we do it here at init time and save it in machcpu.
         */
        mcpup->mondo_data_ra = va_to_pa(mcpup->mondo_data);

        /*
         *  Allocate a per-cpu list of ncpu_guest_max for xcalls
         */
        size = ncpu_guest_max * sizeof (uint16_t);
        if (size < INTR_REPORT_SIZE)
                size = INTR_REPORT_SIZE;

        /*
         * contig_mem_alloc() requires size to be a power of 2.
         * Increase size to a power of 2 if necessary.
         */
        if (!ISP2(size)) {
        if ((size & (size - 1)) != 0) {
                size = 1 << highbit(size);
        }

        mcpup->cpu_list = contig_mem_alloc(size);

        if (mcpup->cpu_list == NULL) {
                cmn_err(CE_NOTE, "cpu%d: cpu cpu_list allocation failed",
                    cpu->cpu_id);
                return (ENOMEM);
        }
        mcpup->cpu_list_ra = va_to_pa(mcpup->cpu_list);

        /*
         * Allocate sun4v interrupt and error queues.
         */
        size = cpu_q_entries * INTR_REPORT_SIZE;

        mcpup->cpu_q_va = contig_mem_alloc(size);

        if (mcpup->cpu_q_va == NULL) {
                cmn_err(CE_NOTE, "cpu%d: cpu intrq allocation failed",
                    cpu->cpu_id);
                return (ENOMEM);
```

```
127         }
128         mcpup->cpu_q_base_pa = va_to_pa(mcpup->cpu_q_va);
129         mcpup->cpu_q_size = size;

131         /*
132          * Allocate device queues
133          */
134         size = dev_q_entries * INTR_REPORT_SIZE;

136         mcpup->dev_q_va = contig_mem_alloc(size);

138         if (mcpup->dev_q_va == NULL) {
139                 cmn_err(CE_NOTE, "cpu%d: dev intrq allocation failed",
140                     cpu->cpu_id);
141                 return (ENOMEM);
142         }
143         mcpup->dev_q_base_pa = va_to_pa(mcpup->dev_q_va);
144         mcpup->dev_q_size = size;

146         /*
147          * Allocate resumable queue and its kernel buffer
148          */
149         size = cpu_rq_entries * Q_ENTRY_SIZE;

151         mcpup->cpu_rq_va = contig_mem_alloc(2 * size);

153         if (mcpup->cpu_rq_va == NULL) {
154                 cmn_err(CE_NOTE, "cpu%d: resumable queue allocation failed",
155                     cpu->cpu_id);
156                 return (ENOMEM);
157         }
158         mcpup->cpu_rq_base_pa = va_to_pa(mcpup->cpu_rq_va);
159         mcpup->cpu_rq_size = size;
160         /* zero out the memory */
161         bzero(mcpup->cpu_rq_va, 2 * size);

163         /*
164          * Allocate non-resumable queues
165          */
166         size = cpu_nrq_entries * Q_ENTRY_SIZE;

168         mcpup->cpu_nrq_va = contig_mem_alloc(2 * size);

170         if (mcpup->cpu_nrq_va == NULL) {
171                 cmn_err(CE_NOTE, "cpu%d: nonresumable queue allocation failed",
172                     cpu->cpu_id);
173                 return (ENOMEM);
174         }
175         mcpup->cpu_nrq_base_pa = va_to_pa(mcpup->cpu_nrq_va);
176         mcpup->cpu_nrq_size = size;
177         /* zero out the memory */
178         bzero(mcpup->cpu_nrq_va, 2 * size);

180         return (0);
181 }

183 void
184 cpu_intrq_cleanup(struct cpu *cpu)
185 {
186         struct machcpu *mcpup = &cpu->cpu_m;
187         int cpu_list_size;
188         uint64_t cpu_q_size;
189         uint64_t dev_q_size;
190         uint64_t cpu_rq_size;
191         uint64_t cpu_nrq_size;
```



```
193         /*
194          * Free mondo data for xcalls.
195          */
196         if (mcpup->mondo_data) {
197                 contig_mem_free(mcpup->mondo_data, INTR_REPORT_SIZE);
198                 mcpup->mondo_data = NULL;
199                 mcpup->mondo_data_ra = NULL;
200         }

202         /*
203          *  Free per-cpu list of ncpu_guest_max for xcalls
204          */
205         cpu_list_size = ncpu_guest_max * sizeof (uint16_t);
206         if (cpu_list_size < INTR_REPORT_SIZE)
207                 cpu_list_size = INTR_REPORT_SIZE;

209         /*
210          * contig_mem_alloc() requires size to be a power of 2.
211          * Increase size to a power of 2 if necessary.
212          */
213         if (!ISP2(cpu_list_size)) {
136         if ((cpu_list_size & (cpu_list_size - 1)) != 0) {
214                 cpu_list_size = 1 << highbit(cpu_list_size);
215         }

217         if (mcpup->cpu_list) {
218                 contig_mem_free(mcpup->cpu_list, cpu_list_size);
219                 mcpup->cpu_list = NULL;
220                 mcpup->cpu_list_ra = NULL;
221         }

223         /*
224          * Free sun4v interrupt and error queues.
225          */
226         if (mcpup->cpu_q_va) {
227                 cpu_q_size = cpu_q_entries * INTR_REPORT_SIZE;
228                 contig_mem_free(mcpup->cpu_q_va, cpu_q_size);
229                 mcpup->cpu_q_va = NULL;
230                 mcpup->cpu_q_base_pa = NULL;
231                 mcpup->cpu_q_size = 0;
232         }

234         if (mcpup->dev_q_va) {
235                 dev_q_size = dev_q_entries * INTR_REPORT_SIZE;
236                 contig_mem_free(mcpup->dev_q_va, dev_q_size);
237                 mcpup->dev_q_va = NULL;
238                 mcpup->dev_q_base_pa = NULL;
239                 mcpup->dev_q_size = 0;
240         }

242         if (mcpup->cpu_rq_va) {
243                 cpu_rq_size = cpu_rq_entries * Q_ENTRY_SIZE;
244                 contig_mem_free(mcpup->cpu_rq_va, 2 * cpu_rq_size);
245                 mcpup->cpu_rq_va = NULL;
246                 mcpup->cpu_rq_base_pa = NULL;
247                 mcpup->cpu_rq_size = 0;
248         }

250         if (mcpup->cpu_nrq_va) {
251                 cpu_nrq_size = cpu_nrq_entries * Q_ENTRY_SIZE;
252                 contig_mem_free(mcpup->cpu_nrq_va, 2 * cpu_nrq_size);
253                 mcpup->cpu_nrq_va = NULL;
254                 mcpup->cpu_nrq_base_pa = NULL;
255                 mcpup->cpu_nrq_size = 0;
256         }
257 }
```