```
**********************************************************
   61235 Thu Oct 23 11:04:51 2014
new/usr/src/uts/common/avs/ns/rdc/rdc_bitmap.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 620 static int
 621 rdc_read_bitmap(rdc_k_info_t *krdc, struct bitmapdata *data)
 622 {
 623         rdc_u_info_t *urdc;
 624         int sts;

 626         if (krdc == NULL) {
 627                 return (-1);
 628         }

 630         if (data != NULL) {
 631                 data->data = kmem_alloc(krdc->bitmap_size, KM_SLEEP);
 632                 data->len = krdc->bitmap_size;

 634                 if (data->data == NULL) {
 635                         return (-1);
 636                 }
 633         }

 635         mutex_enter(&krdc->bmapmutex);

 637         urdc = &rdc_u_info[krdc->index];
 638         if (rdc_get_vflags(urdc) & RDC_BMP_FAILED) {
 639                 mutex_exit(&krdc->bmapmutex);
 640                 return (-1);
 641         }

 643         if (krdc->bitmapfd == NULL) {
 644                 mutex_exit(&krdc->bmapmutex);
 645                 return (-1);
 646         }

 648         if (data == NULL && krdc->dcio_bitmap == NULL) {
 649                 mutex_exit(&krdc->bmapmutex);
 650                 return (-1);
 651         }

 653         if (_rdc_rsrv_devs(krdc, RDC_BMP, RDC_INTERNAL)) {
 654                 cmn_err(CE_WARN, "!rdc_read_bitmap: %s reserve failed",
 655                     urdc->primary.file);
 656                 rdc_set_flags_log(urdc, RDC_BMP_FAILED, "reserve failed");
 657                 mutex_exit(&krdc->bmapmutex);
 658                 return (-1);
 659         }

 661         if (krdc->bmp_kstats) {
 662                 mutex_enter(krdc->bmp_kstats->ks_lock);
 663                 kstat_runq_enter(KSTAT_IO_PTR(krdc->bmp_kstats));
 664                 mutex_exit(krdc->bmp_kstats->ks_lock);
 665         }

 667         sts = rdc_ns_io(krdc->bitmapfd, NSC_RDBUF, RDC_BITMAP_FBA,
 668             data ? data->data : krdc->dcio_bitmap, krdc->bitmap_size);

 670         if (krdc->bmp_kstats) {
 671                 mutex_enter(krdc->bmp_kstats->ks_lock);
 672                 kstat_runq_exit(KSTAT_IO_PTR(krdc->bmp_kstats));
 673                 mutex_exit(krdc->bmp_kstats->ks_lock);
```



```
 674                 KSTAT_IO_PTR(krdc->bmp_kstats)->reads++;
 675                 KSTAT_IO_PTR(krdc->bmp_kstats)->nread += krdc->bitmap_size;
 676         }

 678         _rdc_rlse_devs(krdc, RDC_BMP);

 680         if (!RDC_SUCCESS(sts)) {
 681                 cmn_err(CE_WARN, "!rdc_read_bitmap: %s read failed",
 682                     urdc->primary.file);
 683                 rdc_set_flags_log(urdc, RDC_BMP_FAILED, "read failed");
 684                 mutex_exit(&krdc->bmapmutex);
 685                 return (-1);
 686         }

 688         mutex_exit(&krdc->bmapmutex);
 689         return (0);
 690 }
______unchanged_portion_omitted_
```

```
**********************************************************
  160953 Thu Oct 23 11:04:51 2014
new/usr/src/uts/common/avs/ns/rdc/rdc_io.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_


/*
 * _rdc_load() - rdc is being loaded, Allocate anything
 * that will be needed while the cache is loaded but doesn't really
 * depend on configuration parameters.
 *
 */
int
_rdc_load(void)
{
        int i;
        rdc_k_info_t *krdc;

        mutex_init(&rdc_ping_lock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&net_blk_lock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&rdc_conf_lock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&rdc_many_lock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&rdc_net_hnd_id_lock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&rdc_clnt_lock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&sync_info.lock, NULL, MUTEX_DRIVER, NULL);

#ifdef DEBUG
        mutex_init(&rdc_cntlock, NULL, MUTEX_DRIVER, NULL);
#endif

        if ((i = nsc_max_devices()) < rdc_max_sets)
                rdc_max_sets = i;
        /* following case for partial installs that may fail */
        if (!rdc_max_sets)
                rdc_max_sets = 1024;

        rdc_k_info = kmem_zalloc(sizeof (*rdc_k_info) * rdc_max_sets, KM_SLEEP);
        if (!rdc_k_info)
                return (ENOMEM);

        rdc_u_info = kmem_zalloc(sizeof (*rdc_u_info) * rdc_max_sets, KM_SLEEP);
        if (!rdc_u_info) {
                kmem_free(rdc_k_info, sizeof (*rdc_k_info) * rdc_max_sets);
                return (ENOMEM);
        }

        net_exit = ATM_NONE;
        for (i = 0; i < rdc_max_sets; i++) {
                krdc = &rdc_k_info[i];
                bzero(krdc, sizeof (*krdc));
                krdc->index = i;
                mutex_init(&krdc->dc_sleep, NULL, MUTEX_DRIVER, NULL);
                mutex_init(&krdc->bmapmutex, NULL, MUTEX_DRIVER, NULL);
                mutex_init(&krdc->kstat_mutex, NULL, MUTEX_DRIVER, NULL);
                mutex_init(&krdc->bmp_kstat_mutex, NULL, MUTEX_DRIVER, NULL);
                mutex_init(&krdc->syncbitmutex, NULL, MUTEX_DRIVER, NULL);
                cv_init(&krdc->busycv, NULL, CV_DRIVER, NULL);
                cv_init(&krdc->closingcv, NULL, CV_DRIVER, NULL);
                cv_init(&krdc->haltcv, NULL, CV_DRIVER, NULL);
                cv_init(&krdc->synccv, NULL, CV_DRIVER, NULL);
        }

        rdc_volume_update = nsc_register_svc("RDCVolumeUpdated",
```



```
            rdc_volume_update_svc);

        return (0);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   16833 Thu Oct 23 11:04:51 2014
new/usr/src/uts/common/avs/ns/solaris/nsc_raw.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 117 int
 118 _nsc_init_raw(int maxdevs)
 119 {
 120         _nsc_raw_files =
 121             kmem_zalloc(sizeof (*_nsc_raw_files) * maxdevs, KM_SLEEP);
 122         if (!_nsc_raw_files)
 123                 return (ENOMEM);

 123         _nsc_raw_maxdevs = maxdevs;
 124         _nsc_raw_majors = NULL;

 126         mutex_init(&_nsc_raw_lock, NULL, MUTEX_DRIVER, NULL);
 127         return (0);
 128 }
______unchanged_portion_omitted_


 275 /*
 276  * _raw_open
 277  *
 278  * Multiple opens, single close.
 279  */

 281 /* ARGSUSED */
 282 static int
 283 _raw_open(char *path, int flag, blind_t *cdp, void *iodev)
 284 {
 285         struct cred *cred;
 286         raw_dev_t *cdi = NULL;
 287         char *spath;
 288         dev_t rdev;
 289         int rc, cd, the_cd;
 290         int plen;
 291         ldi_ident_t     li;

 293         if (proc_nskernd == NULL) {
 294                 cmn_err(CE_WARN, "nskern: no nskernd daemon running!");
 295                 return (ENXIO);
 296         }

 298         if (_nsc_raw_maxdevs == 0) {
 299                 cmn_err(CE_WARN, "nskern: _raw_open() before _nsc_init_raw()!");
 300                 return (ENXIO);
 301         }

 303         plen = strlen(path) + 1;
 304         spath = kmem_alloc(plen, KM_SLEEP);
 307         if (spath == NULL) {
 308                 cmn_err(CE_WARN,
 309                     "nskern: unable to alloc memory in _raw_open()");
 310                 return (ENOMEM);
 311         }

 306         (void) strcpy(spath, path);

 308         /*
 309          * Lookup the vnode to extract the dev_t info,
 310          * then release the vnode.
```



```
 311          */
 312         if ((rdev = ldi_get_dev_t_from_path(path)) == 0) {
 313                 kmem_free(spath, plen);
 314                 return (ENXIO);
 315         }

 317         /*
 318          * See if this device is already opened
 319          */

 321         the_cd = -1;

 323         mutex_enter(&_nsc_raw_lock);

 325         for (cd = 0, cdi = _nsc_raw_files; cd < fd_hwm; cd++, cdi++) {
 326                 if (rdev == cdi->rdev) {
 327                         the_cd = cd;
 328                         break;
 329                 } else if (the_cd == -1 && !cdi->in_use)
 330                         the_cd = cd;
 331         }

 333         if (the_cd == -1) {
 334                 if (fd_hwm < _nsc_raw_maxdevs)
 335                         the_cd = fd_hwm++;
 336                 else {
 337                         mutex_exit(&_nsc_raw_lock);
 338                         cmn_err(CE_WARN, "_raw_open: too many open devices");
 339                         kmem_free(spath, plen);
 340                         return (EIO);
 341                 }
 342         }

 344         cdi = &_nsc_raw_files[the_cd];
 345         if (cdi->in_use) {
 346                 /* already set up - just return */
 347                 mutex_exit(&_nsc_raw_lock);
 348                 *cdp = (blind_t)cdi->rdev;
 349                 kmem_free(spath, plen);
 350                 return (0);
 351         }

 353         cdi->partition = -1;
 354         cdi->size = (uint64_t)0;
 355         cdi->rdev = rdev;
 356         cdi->path = spath;
 357         cdi->plen = plen;

 359         cred = ddi_get_cred();

 361         /*
 362          * Layered driver
 363          *
 364          * We use xxx_open_by_dev() since this guarantees that a
 365          * specfs vnode is created and used, not a standard filesystem
 366          * vnode. This is necessary since in a cluster PXFS will block
 367          * vnode operations during switchovers, so we have to use the
 368          * underlying specfs vnode not the PXFS vnode.
 369          *
 370          */

 372         if ((rc = ldi_ident_from_dev(cdi->rdev, &li)) == 0) {
 373                 rc = ldi_open_by_dev(&cdi->rdev,
 374                     OTYP_BLK, FREAD|FWRITE, cred, &cdi->lh, li);
 375         }
 376         if (rc != 0) {
```

```
377                 cdi->lh = NULL;
378                 goto failed;
379         }

381         /*
382          * grab the major_t related information
383          */

385         cdi->major = _raw_get_maj_info(getmajor(rdev));
386         if (cdi->major == NULL) {
387                 /* Out of memory */
388                 cmn_err(CE_WARN,
389                     "_raw_open: cannot alloc major number structure");

391                 rc = ENOMEM;
392                 goto failed;
393         }

395         *cdp = (blind_t)cdi->rdev;
396         cdi->in_use++;

398         mutex_exit(&_nsc_raw_lock);

400         return (rc);

402 failed:

404         if (cdi->lh)
405                 (void) ldi_close(cdi->lh, FWRITE|FREAD, cred);

407         bzero(cdi, sizeof (*cdi));

409         mutex_exit(&_nsc_raw_lock);

411         kmem_free(spath, plen);
412         return (rc);
413 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   60696 Thu Oct 23 11:04:51 2014
new/usr/src/uts/common/avs/ns/sv/sv.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_


1727 static int
1728 svioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *crp, int *rvalp)
1729 {
1730         char itmp1[12], itmp2[12]; /* temp char array for editing ints */
1731         spcs_s_info_t kstatus;  /* Kernel version of spcs status */
1732         spcs_s_info_t ustatus;  /* Address of user version of spcs status */
1733         sv_list32_t svl32;      /* 32 bit Initial structure for SVIOC_LIST */
1734         sv_version_t svv;       /* Version structure */
1735         sv_conf_t svc;          /* User config structure */
1736         sv_list_t svl;          /* Initial structure for SVIOC_LIST */
1737         void *usvn;             /* Address of user sv_name_t */
1738         void *svn = NULL;       /* Array for SVIOC_LIST */
1739         uint64_t phash;         /* pathname hash */
1740         int rc = 0;             /* Return code -- errno */
1741         int size;               /* Number of items in array */
1742         int bytes;              /* Byte size of array */
1743         int ilp32;              /* Convert data structures for ilp32 userland */

1745         *rvalp = 0;

1747         /*
1748          * If sv_mod_status is 0 or SV_PREVENT_UNLOAD, then it will continue.
1749          * else it means it previously was SV_PREVENT_UNLOAD, and now it's
1750          * SV_ALLOW_UNLOAD, expecting the driver to eventually unload.
1751          *
1752          * SV_ALLOW_UNLOAD is final state, so no need to grab sv_mutex.
1753          */
1754         if (sv_mod_status == SV_ALLOW_UNLOAD) {
1755                 return (EBUSY);
1756         }

1758         if ((cmd != SVIOC_LIST) && ((rc = drv_priv(crp)) != 0))
1759                 return (rc);

1761         kstatus = spcs_s_kcreate();
1762         if (!kstatus) {
1763                 DTRACE_PROBE1(sv_ioctl_err_kcreate, dev_t, dev);
1764                 return (ENOMEM);
1765         }

1767         ilp32 = (ddi_model_convert_from((mode & FMODELS)) == DDI_MODEL_ILP32);

1769         switch (cmd) {

1771         case SVIOC_ENABLE:

1773                 if (ilp32) {
1774                         sv_conf32_t svc32;

1776                         if (ddi_copyin((void *)arg, &svc32,
1777                             sizeof (svc32), mode) < 0) {
1778                                 spcs_s_kfree(kstatus);
1779                                 return (EFAULT);
1780                         }

1782                         svc.svc_error = (spcs_s_info_t)svc32.svc_error;
1783                         (void) strcpy(svc.svc_path, svc32.svc_path);
```



```
1784                         svc.svc_flag  = svc32.svc_flag;
1785                         svc.svc_major = svc32.svc_major;
1786                         svc.svc_minor = svc32.svc_minor;
1787                 } else {
1788                         if (ddi_copyin((void *)arg, &svc,
1789                             sizeof (svc), mode) < 0) {
1790                                 spcs_s_kfree(kstatus);
1791                                 return (EFAULT);
1792                         }
1793                 }

1795                 /* force to raw access */
1796                 svc.svc_flag = NSC_DEVICE;

1798                 if (sv_tset == NULL) {
1799                         mutex_enter(&sv_mutex);

1801                         if (sv_tset == NULL) {
1802                                 sv_tset = nst_init("sv_thr", sv_threads);
1803                         }

1805                         mutex_exit(&sv_mutex);

1807                         if (sv_tset == NULL) {
1808                                 cmn_err(CE_WARN,
1809                                     "!sv: could not allocate %d threads",
1810                                     sv_threads);
1811                         }
1812                 }

1814                 rc = sv_enable(svc.svc_path, svc.svc_flag,
1815                     makedevice(svc.svc_major, svc.svc_minor), kstatus);

1817                 if (rc == 0) {
1818                         sv_config_time = nsc_lbolt();

1820                         mutex_enter(&sv_mutex);
1821                         sv_thread_tune(sv_threads_dev);
1822                         mutex_exit(&sv_mutex);
1823                 }

1825                 DTRACE_PROBE3(sv_ioctl_end, dev_t, dev, int, *rvalp, int, rc);

1827                 return (spcs_s_ocopyoutf(&kstatus, svc.svc_error, rc));
1828                 /* NOTREACHED */

1830         case SVIOC_DISABLE:

1832                 if (ilp32) {
1833                         sv_conf32_t svc32;

1835                         if (ddi_copyin((void *)arg, &svc32,
1836                             sizeof (svc32), mode) < 0) {
1837                                 spcs_s_kfree(kstatus);
1838                                 return (EFAULT);
1839                         }

1841                         svc.svc_error = (spcs_s_info_t)svc32.svc_error;
1842                         svc.svc_major = svc32.svc_major;
1843                         svc.svc_minor = svc32.svc_minor;
1844                         (void) strcpy(svc.svc_path, svc32.svc_path);
1845                         svc.svc_flag  = svc32.svc_flag;
1846                 } else {
1847                         if (ddi_copyin((void *)arg, &svc,
1848                             sizeof (svc), mode) < 0) {
1849                                 spcs_s_kfree(kstatus);
```

```
1850                                return (EFAULT);
1851                        }
1852                }

1854                if (svc.svc_major == (major_t)-1 &&
1855                    svc.svc_minor == (minor_t)-1) {
1856                        sv_dev_t *svp;
1857                        int i;

1859                        /*
1860                         * User level could not find the minor device
1861                         * node, so do this the slow way by searching
1862                         * the entire sv config for a matching pathname.
1863                         */

1865                        phash = nsc_strhash(svc.svc_path);

1867                        mutex_enter(&sv_mutex);

1869                        for (i = 0; i < sv_max_devices; i++) {
1870                                svp = &sv_devs[i];

1872                                if (svp->sv_state == SV_DISABLE ||
1873                                    svp->sv_fd == NULL)
1874                                        continue;

1876                                if (nsc_fdpathcmp(svp->sv_fd, phash,
1877                                    svc.svc_path) == 0) {
1878                                        svc.svc_major = getmajor(svp->sv_dev);
1879                                        svc.svc_minor = getminor(svp->sv_dev);
1880                                        break;
1881                                }
1882                        }

1884                        mutex_exit(&sv_mutex);

1886                        if (svc.svc_major == (major_t)-1 &&
1887                            svc.svc_minor == (minor_t)-1)
1888                                return (spcs_s_ocopyoutf(&kstatus,
1889                                    svc.svc_error, SV_ENODEV));
1890                }

1892                rc = sv_disable(makedevice(svc.svc_major, svc.svc_minor),
1893                    kstatus);

1895                if (rc == 0) {
1896                        sv_config_time = nsc_lbolt();

1898                        mutex_enter(&sv_mutex);
1899                        sv_thread_tune(-sv_threads_dev);
1900                        mutex_exit(&sv_mutex);
1901                }

1903                DTRACE_PROBE3(sv_ioctl_2, dev_t, dev, int, *rvalp, int, rc);

1905                return (spcs_s_ocopyoutf(&kstatus, svc.svc_error, rc));
1906                /* NOTREACHED */

1908        case SVIOC_LIST:

1910                if (ilp32) {
1911                        if (ddi_copyin((void *)arg, &svl32,
1912                            sizeof (svl32), mode) < 0) {
1913                                spcs_s_kfree(kstatus);
1914                                return (EFAULT);
1915                        }
```



```
1917                        ustatus = (spcs_s_info_t)svl32.svl_error;
1918                        size = svl32.svl_count;
1919                        usvn = (void *)(unsigned long)svl32.svl_names;
1920                } else {
1921                        if (ddi_copyin((void *)arg, &svl,
1922                            sizeof (svl), mode) < 0) {
1923                                spcs_s_kfree(kstatus);
1924                                return (EFAULT);
1925                        }

1927                        ustatus = svl.svl_error;
1928                        size = svl.svl_count;
1929                        usvn = svl.svl_names;
1930                }

1932                /* Do some boundary checking */
1933                if ((size < 0) || (size > sv_max_devices)) {
1934                        /* Array size is out of range */
1935                        return (spcs_s_ocopyoutf(&kstatus, ustatus,
1936                            SV_EARRBOUNDS, "0",
1937                            spcs_s_inttostring(sv_max_devices, itmp1,
1938                            sizeof (itmp1), 0),
1939                            spcs_s_inttostring(size, itmp2,
1940                            sizeof (itmp2), 0)));
1941                }

1943                if (ilp32)
1944                        bytes = size * sizeof (sv_name32_t);
1945                else
1946                        bytes = size * sizeof (sv_name_t);

1948                /* Allocate memory for the array of structures */
1949                if (bytes != 0) {
1950                        svn = kmem_zalloc(bytes, KM_SLEEP);
1951                        if (!svn) {
1952                                return (spcs_s_ocopyoutf(&kstatus,
1953                                    ustatus, ENOMEM));
1954                        }
1951                }

1953                rc = sv_list(svn, size, rvalp, ilp32);
1954                if (rc) {
1955                        if (svn != NULL)
1956                                kmem_free(svn, bytes);
1957                        return (spcs_s_ocopyoutf(&kstatus, ustatus, rc));
1958                }

1960                if (ilp32) {
1961                        svl32.svl_timestamp = (uint32_t)sv_config_time;
1962                        svl32.svl_maxdevs = (int32_t)sv_max_devices;

1964                        /* Return the list structure */
1965                        if (ddi_copyout(&svl32, (void *)arg,
1966                            sizeof (svl32), mode) < 0) {
1967                                spcs_s_kfree(kstatus);
1968                                if (svn != NULL)
1969                                        kmem_free(svn, bytes);
1970                                return (EFAULT);
1971                        }
1972                } else {
1973                        svl.svl_timestamp = sv_config_time;
1974                        svl.svl_maxdevs = sv_max_devices;

1976                        /* Return the list structure */
1977                        if (ddi_copyout(&svl, (void *)arg,
```

```
1978                            sizeof (svl), mode) < 0) {
1979                                spcs_s_kfree(kstatus);
1980                                if (svn != NULL)
1981                                        kmem_free(svn, bytes);
1982                                return (EFAULT);
1983                        }
1984                }

1986                /* Return the array */
1987                if (svn != NULL) {
1988                        if (ddi_copyout(svn, usvn, bytes, mode) < 0) {
1989                                kmem_free(svn, bytes);
1990                                spcs_s_kfree(kstatus);
1991                                return (EFAULT);
1992                        }
1993                        kmem_free(svn, bytes);
1994                }

1996                DTRACE_PROBE3(sv_ioctl_3, dev_t, dev, int, *rvalp, int, 0);

1998                return (spcs_s_ocopyoutf(&kstatus, ustatus, 0));
1999                /* NOTREACHED */

2001        case SVIOC_VERSION:

2003                if (ilp32) {
2004                        sv_version32_t svv32;

2006                        if (ddi_copyin((void *)arg, &svv32,
2007                            sizeof (svv32), mode) < 0) {
2008                                spcs_s_kfree(kstatus);
2009                                return (EFAULT);
2010                        }

2012                        svv32.svv_major_rev = sv_major_rev;
2013                        svv32.svv_minor_rev = sv_minor_rev;
2014                        svv32.svv_micro_rev = sv_micro_rev;
2015                        svv32.svv_baseline_rev = sv_baseline_rev;

2017                        if (ddi_copyout(&svv32, (void *)arg,
2018                            sizeof (svv32), mode) < 0) {
2019                                spcs_s_kfree(kstatus);
2020                                return (EFAULT);
2021                        }

2023                        ustatus = (spcs_s_info_t)svv32.svv_error;
2024                } else {
2025                        if (ddi_copyin((void *)arg, &svv,
2026                            sizeof (svv), mode) < 0) {
2027                                spcs_s_kfree(kstatus);
2028                                return (EFAULT);
2029                        }

2031                        svv.svv_major_rev = sv_major_rev;
2032                        svv.svv_minor_rev = sv_minor_rev;
2033                        svv.svv_micro_rev = sv_micro_rev;
2034                        svv.svv_baseline_rev = sv_baseline_rev;

2036                        if (ddi_copyout(&svv, (void *)arg,
2037                            sizeof (svv), mode) < 0) {
2038                                spcs_s_kfree(kstatus);
2039                                return (EFAULT);
2040                        }

2042                        ustatus = svv.svv_error;
2043                }
```



```
2045                DTRACE_PROBE3(sv_ioctl_4, dev_t, dev, int, *rvalp, int, 0);

2047                return (spcs_s_ocopyoutf(&kstatus, ustatus, 0));
2048                /* NOTREACHED */

2050        case SVIOC_UNLOAD:
2051                rc = sv_prepare_unload();

2053                if (ddi_copyout(&rc, (void *)arg, sizeof (rc), mode) < 0) {
2054                        rc = EFAULT;
2055                }

2057                spcs_s_kfree(kstatus);
2058                return (rc);

2060        default:
2061                spcs_s_kfree(kstatus);

2063                DTRACE_PROBE3(sv_ioctl_4, dev_t, dev, int, *rvalp, int, EINVAL);

2065                return (EINVAL);
2066                /* NOTREACHED */
2067        }

2069        /* NOTREACHED */
2070 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  128510 Thu Oct 23 11:04:52 2014
new/usr/src/uts/common/crypto/io/dca.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1587 dca_request_t *
1588 dca_newreq(dca_t *dca)
1589 {
1590         dca_request_t           *reqp;
1591         size_t                  size;
1592         ddi_dma_cookie_t        c;
1593         unsigned                nc;
1594         int                     rv;
1595         int                     n_chain = 0;

1597         size = (DESC_SIZE * MAXFRAGS) + CTX_MAXLENGTH;

1599         reqp = kmem_zalloc(sizeof (dca_request_t), KM_SLEEP);

1601         reqp->dr_dca = dca;

1603         /*
1604          * Setup the DMA region for the context and descriptors.
1605          */
1606         rv = ddi_dma_alloc_handle(dca->dca_dip, &dca_dmaattr, DDI_DMA_SLEEP,
1607             NULL, &reqp->dr_ctx_dmah);
1608         if (rv != DDI_SUCCESS) {
1609                 dca_error(dca, "failure allocating request DMA handle");
1610                 dca_destroyreq(reqp);
1611                 return (NULL);
1612         }

1614         /* for driver hardening, allocate in whole pages */
1615         rv = ddi_dma_mem_alloc(reqp->dr_ctx_dmah,
1616             ROUNDUP(size, dca->dca_pagesize), &dca_devattr, DDI_DMA_CONSISTENT,
1617             DDI_DMA_SLEEP, NULL, &reqp->dr_ctx_kaddr, &size,
1618             &reqp->dr_ctx_acch);
1619         if (rv != DDI_SUCCESS) {
1620                 dca_error(dca, "unable to alloc request DMA memory");
1621                 dca_destroyreq(reqp);
1622                 return (NULL);
1623         }

1625         rv = ddi_dma_addr_bind_handle(reqp->dr_ctx_dmah, NULL,
1626             reqp->dr_ctx_kaddr, size, DDI_DMA_CONSISTENT | DDI_DMA_WRITE,
1627             DDI_DMA_SLEEP, 0, &c, &nc);
1628         if (rv != DDI_DMA_MAPPED) {
1629                 dca_error(dca, "failed binding request DMA handle");
1630                 dca_destroyreq(reqp);
1631                 return (NULL);
1632         }
1633         reqp->dr_ctx_paddr = c.dmac_address;

1635         reqp->dr_dma_size = size;

1637         /*
1638          * Set up the dma for our scratch/shared buffers.
1639          */
1640         rv = ddi_dma_alloc_handle(dca->dca_dip, &dca_dmaattr,
1641             DDI_DMA_SLEEP, NULL, &reqp->dr_ibuf_dmah);
1642         if (rv != DDI_SUCCESS) {
1643                 dca_error(dca, "failure allocating ibuf DMA handle");
1644                 dca_destroyreq(reqp);
```



```
1645                 return (NULL);
1646         }
1647         rv = ddi_dma_alloc_handle(dca->dca_dip, &dca_dmaattr,
1648             DDI_DMA_SLEEP, NULL, &reqp->dr_obuf_dmah);
1649         if (rv != DDI_SUCCESS) {
1650                 dca_error(dca, "failure allocating obuf DMA handle");
1651                 dca_destroyreq(reqp);
1652                 return (NULL);
1653         }

1655         rv = ddi_dma_alloc_handle(dca->dca_dip, &dca_dmaattr,
1656             DDI_DMA_SLEEP, NULL, &reqp->dr_chain_in_dmah);
1657         if (rv != DDI_SUCCESS) {
1658                 dca_error(dca, "failure allocating chain_in DMA handle");
1659                 dca_destroyreq(reqp);
1660                 return (NULL);
1661         }

1663         rv = ddi_dma_alloc_handle(dca->dca_dip, &dca_dmaattr,
1664             DDI_DMA_SLEEP, NULL, &reqp->dr_chain_out_dmah);
1665         if (rv != DDI_SUCCESS) {
1666                 dca_error(dca, "failure allocating chain_out DMA handle");
1667                 dca_destroyreq(reqp);
1668                 return (NULL);
1669         }

1671         /*
1672          * for driver hardening, allocate in whole pages.
1673          */
1674         size = ROUNDUP(MAXPACKET, dca->dca_pagesize);
1675 #if defined(i386) || defined(__i386)
1676         /*
1677          * Use kmem_alloc instead of ddi_dma_mem_alloc here since the latter
1678          * may fail on x86 platform if a physically contiguous memory chunk
1679          * cannot be found. From initial testing, we did not see performance
1680          * degradation as seen on Sparc.
1681          */
1682         reqp->dr_ibuf_kaddr = kmem_alloc(size, KM_SLEEP);
1683         reqp->dr_obuf_kaddr = kmem_alloc(size, KM_SLEEP);
1682         if ((reqp->dr_ibuf_kaddr = kmem_alloc(size, KM_SLEEP)) == NULL) {
1683                 dca_error(dca, "unable to alloc request ibuf memory");
1684                 dca_destroyreq(reqp);
1685                 return (NULL);
1686         }
1687         if ((reqp->dr_obuf_kaddr = kmem_alloc(size, KM_SLEEP)) == NULL) {
1688                 dca_error(dca, "unable to alloc request obuf memory");
1689                 dca_destroyreq(reqp);
1690                 return (NULL);
1691         }
1684 #else
1685         /*
1686          * We could kmem_alloc for Sparc too. However, it gives worse
1687          * performance when transferring more than one page data. For example,
1688          * using 4 threads and 12032 byte data and 3DES on 900MHZ Sparc system,
1689          * kmem_alloc uses 80% CPU and ddi_dma_mem_alloc uses 50% CPU for
1690          * the same throughput.
1691          */
1692         rv = ddi_dma_mem_alloc(reqp->dr_ibuf_dmah,
1693             size, &dca_bufattr,
1694             DDI_DMA_STREAMING, DDI_DMA_SLEEP, NULL, &reqp->dr_ibuf_kaddr,
1695             &size, &reqp->dr_ibuf_acch);
1696         if (rv != DDI_SUCCESS) {
1697                 dca_error(dca, "unable to alloc request DMA memory");
1698                 dca_destroyreq(reqp);
1699                 return (NULL);
1700         }
```

```
1702         rv = ddi_dma_mem_alloc(reqp->dr_obuf_dmah,
1703             size, &dca_bufattr,
1704             DDI_DMA_STREAMING, DDI_DMA_SLEEP, NULL, &reqp->dr_obuf_kaddr,
1705             &size, &reqp->dr_obuf_acch);
1706         if (rv != DDI_SUCCESS) {
1707                 dca_error(dca, "unable to alloc request DMA memory");
1708                 dca_destroyreq(reqp);
1709                 return (NULL);
1710         }
1711 #endif

1713         /* Skip the used portion in the context page */
1714         reqp->dr_offset = CTX_MAXLENGTH;
1715         if ((rv = dca_bindchains_one(reqp, size, reqp->dr_offset,
1716             reqp->dr_ibuf_kaddr, reqp->dr_ibuf_dmah,
1717             DDI_DMA_WRITE | DDI_DMA_STREAMING,
1718             &reqp->dr_ibuf_head, &n_chain)) != DDI_SUCCESS) {
1719                 (void) dca_destroyreq(reqp);
1720                 return (NULL);
1721         }
1722         reqp->dr_ibuf_paddr = reqp->dr_ibuf_head.dc_buffer_paddr;
1723         /* Skip the space used by the input buffer */
1724         reqp->dr_offset += DESC_SIZE * n_chain;

1726         if ((rv = dca_bindchains_one(reqp, size, reqp->dr_offset,
1727             reqp->dr_obuf_kaddr, reqp->dr_obuf_dmah,
1728             DDI_DMA_READ | DDI_DMA_STREAMING,
1729             &reqp->dr_obuf_head, &n_chain)) != DDI_SUCCESS) {
1730                 (void) dca_destroyreq(reqp);
1731                 return (NULL);
1732         }
1733         reqp->dr_obuf_paddr = reqp->dr_obuf_head.dc_buffer_paddr;
1734         /* Skip the space used by the output buffer */
1735         reqp->dr_offset += DESC_SIZE * n_chain;

1737         DBG(dca, DCHATTY, "CTX is 0x%p, phys 0x%x, len %d",
1738             reqp->dr_ctx_kaddr, reqp->dr_ctx_paddr, CTX_MAXLENGTH);
1739         return (reqp);
1740 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    7815 Thu Oct 23 11:04:52 2014
new/usr/src/uts/common/crypto/io/dca_rng.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
```

```
/*
 * 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.
 */

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

/*
 * Deimos - cryptographic acceleration based upon Broadcom 582x.
 */

#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/kmem.h>
#include <sys/crypto/dca.h>
#include <sys/atomic.h>

/*
 * Random number implementation.
 */

static int dca_rngstart(dca_t *, dca_request_t *);
static void dca_rngdone(dca_request_t *, int);

static void dca_random_done();
int dca_random_buffer(dca_t *dca, caddr_t buf, int len);
int dca_random_init();
void dca_random_fini();

int
dca_rng(dca_t *dca, uchar_t *buf, size_t len, crypto_req_handle_t req)
{
        dca_request_t   *reqp;
        int             rv;
        crypto_data_t   *data;

        if ((reqp = dca_getreq(dca, MCR2, 1)) == NULL) {
                dca_error(dca, "unable to allocate request for RNG");
```



```
                return (CRYPTO_HOST_MEMORY);
        }

        reqp->dr_kcf_req = req;

        data = &reqp->dr_ctx.in_dup;
        data->cd_format = CRYPTO_DATA_RAW;
        data->cd_offset = 0;
        data->cd_length = 0;
        data->cd_raw.iov_base = (char *)buf;
        data->cd_raw.iov_len = len;
        reqp->dr_out = data;
        reqp->dr_in = NULL;

        rv = dca_rngstart(dca, reqp);
        if (rv != CRYPTO_QUEUED) {
                if (reqp->destroy)
                        dca_destroyreq(reqp);
                else
                        dca_freereq(reqp);
        }
        return (rv);
}
```

_____unchanged_portion_omitted_

```
/*
 * This gives a 32k random bytes per buffer. The two buffers will switch back
 * and forth. When a buffer is used up, a request will be submitted to refill
 * this buffer before switching to the other one
 */

#define RANDOM_BUFFER_SIZE              (1<<15)
#define DCA_RANDOM_MAX_WAIT             10000

int
dca_random_init(dca_t *dca)
{
        /* Mutex for the local random number pool */
        mutex_init(&dca->dca_random_lock, NULL, MUTEX_DRIVER, NULL);

        dca->dca_buf1 = kmem_alloc(RANDOM_BUFFER_SIZE, KM_SLEEP);
        if ((dca->dca_buf1 = kmem_alloc(RANDOM_BUFFER_SIZE, KM_SLEEP)) ==
            NULL) {
                mutex_destroy(&dca->dca_random_lock);
                return (CRYPTO_FAILED);
        }

        dca->dca_buf2 = kmem_alloc(RANDOM_BUFFER_SIZE, KM_SLEEP);
        if ((dca->dca_buf2 = kmem_alloc(RANDOM_BUFFER_SIZE, KM_SLEEP)) ==
            NULL) {
                mutex_destroy(&dca->dca_random_lock);
                kmem_free(dca->dca_buf1, RANDOM_BUFFER_SIZE);
                return (CRYPTO_FAILED);
        }

        return (CRYPTO_SUCCESS);
}
```

_____unchanged_portion_omitted_

```
**********************************************************
  298742 Thu Oct 23 11:04:52 2014
new/usr/src/uts/common/crypto/io/dprov.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

9102 /*
9103  * Create an object from the specified template. Checks whether the
9104  * object can be created according to its attributes and the state
9105  * of the session. The new session object id is returned. If the
9106  * object is a token object, it is added to the per-instance object
9107  * table as well.
9108  */
9109 static int
9110 dprov_create_object_from_template(dprov_state_t *softc,
9111     dprov_session_t *session, crypto_object_attribute_t *template,
9112     uint_t nattr, crypto_object_id_t *object_id, boolean_t check_for_secret,
9113     boolean_t force)
9114 {
9115         dprov_object_t *object;
9116         boolean_t is_token = B_FALSE;
9117         boolean_t extractable_attribute_present = B_FALSE;
9118         boolean_t sensitive_attribute_present = B_FALSE;
9119         boolean_t private_attribute_present = B_FALSE;
9120         boolean_t token_attribute_present = B_FALSE;
9121         uint_t i;
9122         int error;
9123         uint_t attr;
9124         uint_t oattr;
9125         crypto_attr_type_t type;
9126         size_t old_len, new_len;
9127         offset_t offset;

9129         if (nattr > DPROV_MAX_ATTR)
9130                 return (CRYPTO_HOST_MEMORY);

9132         if (!force) {
9133                 /* verify that object can be created */
9134                 if ((error = dprov_template_can_create(session, template,
9135                     nattr, check_for_secret)) != CRYPTO_SUCCESS)
9136                         return (error);
9137         }

9139         /* allocate new object */
9140         object = kmem_zalloc(sizeof (dprov_object_t), KM_SLEEP);
9141         if (object == NULL)
9142                 return (CRYPTO_HOST_MEMORY);

9142         /* is it a token object? */
9143         /* check CKA_TOKEN attribute value */
9144         error = dprov_get_template_attr_boolean(template, nattr,
9145             DPROV_CKA_TOKEN, &is_token);
9146         if (error == CRYPTO_SUCCESS && is_token) {
9147                 /* token object, add it to the per-instance object table */
9148                 for (i = 0; i < DPROV_MAX_OBJECTS; i++)
9149                         if (softc->ds_objects[i] == NULL)
9150                                 break;
9151                 if (i == DPROV_MAX_OBJECTS)
9152                         /* no free slot */
9153                         return (CRYPTO_HOST_MEMORY);
9154                 softc->ds_objects[i] = object;
9155                 object->do_token_idx = i;
9156                 DPROV_OBJECT_REFHOLD(object);
9157         }
```



```
9159         /* add object to session object table */
9160         for (i = 0; i < DPROV_MAX_OBJECTS; i++)
9161                 if (session->ds_objects[i] == NULL)
9162                         break;
9163         if (i == DPROV_MAX_OBJECTS) {
9164                 /* no more session object slots */
9165                 DPROV_OBJECT_REFRELE(object);
9166                 return (CRYPTO_HOST_MEMORY);
9167         }
9168         session->ds_objects[i] = object;
9169         DPROV_OBJECT_REFHOLD(object);
9170         *object_id = i;

9172         /* initialize object from template */
9173         for (attr = 0, oattr = 0; attr < nattr; attr++) {
9174                 if (template[attr].oa_value == NULL)
9175                         continue;
9176                 type = template[attr].oa_type;
9177                 old_len = template[attr].oa_value_len;
9178                 new_len = attribute_size(type, old_len);

9180                 if (type == DPROV_CKA_EXTRACTABLE) {
9181                         extractable_attribute_present = B_TRUE;
9182                 } else if (type == DPROV_CKA_PRIVATE) {
9183                         private_attribute_present = B_TRUE;
9184                 } else if (type == DPROV_CKA_TOKEN) {
9185                         token_attribute_present = B_TRUE;
9186                 }
9187                 object->do_attr[oattr].oa_type = type;
9188                 object->do_attr[oattr].oa_value_len = new_len;

9190                 object->do_attr[oattr].oa_value = kmem_zalloc(new_len,
9191                     KM_SLEEP);

9193                 offset = 0;
9194 #ifdef _BIG_ENDIAN
9195                 if (fixed_size_attribute(type)) {
9196                         offset = old_len - new_len;
9197                 }
9198 #endif
9199                 bcopy(&template[attr].oa_value[offset],
9200                     object->do_attr[oattr].oa_value, new_len);
9201                 oattr++;
9202         }

9204         /* add boolean attributes that must be present */
9205         if (extractable_attribute_present == B_FALSE) {
9206                 object->do_attr[oattr].oa_type = DPROV_CKA_EXTRACTABLE;
9207                 object->do_attr[oattr].oa_value_len = 1;
9208                 object->do_attr[oattr].oa_value = kmem_alloc(1, KM_SLEEP);
9209                 object->do_attr[oattr].oa_value[0] = B_TRUE;
9210                 oattr++;
9211         }

9213         if (private_attribute_present == B_FALSE) {
9214                 object->do_attr[oattr].oa_type = DPROV_CKA_PRIVATE;
9215                 object->do_attr[oattr].oa_value_len = 1;
9216                 object->do_attr[oattr].oa_value = kmem_alloc(1, KM_SLEEP);
9217                 object->do_attr[oattr].oa_value[0] = B_FALSE;
9218                 oattr++;
9219         }

9221         if (token_attribute_present == B_FALSE) {
9222                 object->do_attr[oattr].oa_type = DPROV_CKA_TOKEN;
9223                 object->do_attr[oattr].oa_value_len = 1;
```

```
9224                 object->do_attr[oattr].oa_value = kmem_alloc(1, KM_SLEEP);
9225                 object->do_attr[oattr].oa_value[0] = B_FALSE;
9226                 oattr++;
9227         }

9229         if (sensitive_attribute_present == B_FALSE) {
9230                 object->do_attr[oattr].oa_type = DPROV_CKA_SENSITIVE;
9231                 object->do_attr[oattr].oa_value_len = 1;
9232                 object->do_attr[oattr].oa_value = kmem_alloc(1, KM_SLEEP);
9233                 object->do_attr[oattr].oa_value[0] = B_FALSE;
9234                 oattr++;
9235         }
9236         return (CRYPTO_SUCCESS);
9237 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10050 Thu Oct 23 11:04:53 2014
new/usr/src/uts/common/fs/nfs/nfs_sys.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

  92 int
  93 nfssys(enum nfssys_op opcode, void *arg)
  94 {
  95         int error = 0;

  97         if (!(opcode == NFS_REVAUTH || opcode == NFS4_SVC) &&
  98             secpolicy_nfs(CRED()) != 0)
  99                 return (set_errno(EPERM));

 101         switch (opcode) {
 102         case NFS4_CLR_STATE: { /* Clear NFS4 client state */
 103                 struct nfs4clrst_args clr;
 104                 STRUCT_DECL(nfs4clrst_args, u_clr);

 106                 /*
 107                  * If the server is not loaded then no point in
 108                  * clearing nothing :-)
 109                  */
 110                 if (rfs4_client_clrst == NULL) {
 111                         break;
 112                 }

 114                 if (!INGLOBALZONE(curproc))
 115                         return (set_errno(EPERM));

 117                 STRUCT_INIT(u_clr, get_udatamodel());

 119                 if (copyin(arg, STRUCT_BUF(u_clr), STRUCT_SIZE(u_clr)))
 120                         return (set_errno(EFAULT));

 122                 clr.vers = STRUCT_FGET(u_clr, vers);

 124                 if (clr.vers != NFS4_CLRST_VERSION)
 125                         return (set_errno(EINVAL));

 127                 clr.addr_type = STRUCT_FGET(u_clr, addr_type);
 128                 clr.ap = STRUCT_FGETP(u_clr, ap);
 129                 rfs4_client_clrst(&clr);
 130                 break;
 131         }

 133         case SVCPOOL_CREATE: { /* setup an RPC server thread pool */
 134                 struct svcpool_args p;

 136                 if (copyin(arg, &p, sizeof (p)))
 137                         return (set_errno(EFAULT));

 139                 error = svc_pool_create(&p);
 140                 break;
 141         }

 143         case SVCPOOL_WAIT: { /* wait in kernel for threads to be needed */
 144                 int id;

 146                 if (copyin(arg, &id, sizeof (id)))
 147                         return (set_errno(EFAULT));

 149                 error = svc_wait(id);
```



```
 150                 break;
 151         }

 153         case SVCPOOL_RUN: { /* give work to a runnable thread */
 154                 int id;

 156                 if (copyin(arg, &id, sizeof (id)))
 157                         return (set_errno(EFAULT));

 159                 error = svc_do_run(id);
 160                 break;
 161         }

 163         case RDMA_SVC_INIT: {
 164                 struct rdma_svc_args rsa;
 165                 char netstore[20] = "tcp";

 167                 if (!INGLOBALZONE(curproc))
 168                         return (set_errno(EPERM));
 169                 if (get_udatamodel() != DATAMODEL_NATIVE) {
 170                         STRUCT_DECL(rdma_svc_args, ursa);

 172                         STRUCT_INIT(ursa, get_udatamodel());
 173                         if (copyin(arg, STRUCT_BUF(ursa), STRUCT_SIZE(ursa)))
 174                                 return (set_errno(EFAULT));

 176                         rsa.poolid = STRUCT_FGET(ursa, poolid);
 177                         rsa.nfs_versmin = STRUCT_FGET(ursa, nfs_versmin);
 178                         rsa.nfs_versmax = STRUCT_FGET(ursa, nfs_versmax);
 179                         rsa.delegation = STRUCT_FGET(ursa, delegation);
 180                 } else {
 181                         if (copyin(arg, &rsa, sizeof (rsa)))
 182                                 return (set_errno(EFAULT));
 183                 }
 184                 rsa.netid = netstore;

 186                 error = rdma_start(&rsa);
 187                 break;
 188         }

 190         case NFS_SVC: { /* NFS server daemon */
 191                 STRUCT_DECL(nfs_svc_args, nsa);

 193                 if (!INGLOBALZONE(curproc))
 194                         return (set_errno(EPERM));
 195                 STRUCT_INIT(nsa, get_udatamodel());

 197                 if (copyin(arg, STRUCT_BUF(nsa), STRUCT_SIZE(nsa)))
 198                         return (set_errno(EFAULT));

 200                 error = nfs_svc(STRUCT_BUF(nsa), get_udatamodel());
 201                 break;
 202         }

 204         case EXPORTFS: { /* export a file system */
 205                 error = nfs_export(arg);
 206                 break;
 207         }

 209         case NFS_GETFH: { /* get a file handle */
 210                 STRUCT_DECL(nfs_getfh_args, nga);

 212                 if (!INGLOBALZONE(curproc))
 213                         return (set_errno(EPERM));
 214                 STRUCT_INIT(nga, get_udatamodel());
 215                 if (copyin(arg, STRUCT_BUF(nga), STRUCT_SIZE(nga)))
```

```
216                         return (set_errno(EFAULT));

218                 error = nfs_getfh(STRUCT_BUF(nga), get_udatamodel(), CRED());
219                 break;
220         }

222         case NFS_REVAUTH: { /* revoke the cached credentials for the uid */
223                 STRUCT_DECL(nfs_revauth_args, nra);

225                 STRUCT_INIT(nra, get_udatamodel());
226                 if (copyin(arg, STRUCT_BUF(nra), STRUCT_SIZE(nra)))
227                         return (set_errno(EFAULT));

229                 /* This call performs its own privilege checking */
230                 error = sec_clnt_revoke(STRUCT_FGET(nra, authtype),
231                     STRUCT_FGET(nra, uid), CRED(), NULL, get_udatamodel());
232                 break;
233         }

235         case LM_SVC: { /* LM server daemon */
236                 struct lm_svc_args lsa;

238                 if (get_udatamodel() != DATAMODEL_NATIVE) {
239                         STRUCT_DECL(lm_svc_args, ulsa);

241                         STRUCT_INIT(ulsa, get_udatamodel());
242                         if (copyin(arg, STRUCT_BUF(ulsa), STRUCT_SIZE(ulsa)))
243                                 return (set_errno(EFAULT));

245                         lsa.version = STRUCT_FGET(ulsa, version);
246                         lsa.fd = STRUCT_FGET(ulsa, fd);
247                         lsa.n_fmly = STRUCT_FGET(ulsa, n_fmly);
248                         lsa.n_proto = STRUCT_FGET(ulsa, n_proto);
249                         lsa.n_rdev = expldev(STRUCT_FGET(ulsa, n_rdev));
250                         lsa.debug = STRUCT_FGET(ulsa, debug);
251                         lsa.timout = STRUCT_FGET(ulsa, timout);
252                         lsa.grace = STRUCT_FGET(ulsa, grace);
253                         lsa.retransmittimeout = STRUCT_FGET(ulsa,
254                             retransmittimeout);
255                 } else {
256                         if (copyin(arg, &lsa, sizeof (lsa)))
257                                 return (set_errno(EFAULT));
258                 }

260                 error = lm_svc(&lsa);
261                 break;
262         }

264         case KILL_LOCKMGR: {
265                 error = lm_shutdown();
266                 break;
267         }

269         case LOG_FLUSH: {       /* Flush log buffer and possibly rename */
270                 STRUCT_DECL(nfsl_flush_args, nfa);

272                 STRUCT_INIT(nfa, get_udatamodel());
273                 if (copyin(arg, STRUCT_BUF(nfa), STRUCT_SIZE(nfa)))
274                         return (set_errno(EFAULT));

276                 error = nfsl_flush(STRUCT_BUF(nfa), get_udatamodel());
277                 break;
278         }

280         case NFS4_SVC: { /* NFS client callback daemon */
```



```
282                 STRUCT_DECL(nfs4_svc_args, nsa);

284                 STRUCT_INIT(nsa, get_udatamodel());

286                 if (copyin(arg, STRUCT_BUF(nsa), STRUCT_SIZE(nsa)))
287                         return (set_errno(EFAULT));

289                 error = nfs4_svc(STRUCT_BUF(nsa), get_udatamodel());
290                 break;
291         }

293         /* Request that NFSv4 server quiesce on next shutdown */
294         case NFS4_SVC_REQUEST_QUIESCE: {
295                 int id;

297                 /* check that nfssrv module is loaded */
298                 if (nfs_srv_quiesce_func == NULL)
299                         return (set_errno(ENOTSUP));

301                 if (copyin(arg, &id, sizeof (id)))
302                         return (set_errno(EFAULT));

304                 error = svc_pool_control(id, SVCPSET_SHUTDOWN_PROC,
305                     (void *)nfs_srv_quiesce_func);
306                 break;
307         }

309         case NFS_IDMAP: {
310                 struct nfsidmap_args idm;

312                 if (copyin(arg, &idm, sizeof (idm)))
313                         return (set_errno(EFAULT));

315                 nfs_idmap_args(&idm);
316                 error = 0;
317                 break;
318         }

320         case NFS4_DSS_SETPATHS_SIZE: {
321                 /* crosses ILP32/LP64 boundary */
322                 uint32_t nfs4_dss_bufsize = 0;

324                 if (copyin(arg, &nfs4_dss_bufsize, sizeof (nfs4_dss_bufsize)))
325                         return (set_errno(EFAULT));
326                 nfs4_dss_buflen = (long)nfs4_dss_bufsize;
327                 error = 0;
328                 break;
329         }

331         case NFS4_DSS_SETPATHS: {
332                 char *nfs4_dss_bufp;

334                 /* check that nfssrv module is loaded */
335                 if (nfs_srv_dss_func == NULL)
336                         return (set_errno(ENOTSUP));

338                 /*
339                  * NFS4_DSS_SETPATHS_SIZE must be called before
340                  * NFS4_DSS_SETPATHS, to tell us how big a buffer we need
341                  * to allocate.
342                  */
343                 if (nfs4_dss_buflen == 0)
344                         return (set_errno(EINVAL));
345                 nfs4_dss_bufp = kmem_alloc(nfs4_dss_buflen, KM_SLEEP);
346                 if (nfs4_dss_bufp == NULL)
347                         return (set_errno(ENOMEM));
```

```
347                 if (copyin(arg, nfs4_dss_bufp, nfs4_dss_buflen)) {
348                         kmem_free(nfs4_dss_bufp, nfs4_dss_buflen);
349                         return (set_errno(EFAULT));
350                 }

352                 /* unpack the buffer and extract the pathnames */
353                 error = nfs_srv_dss_func(nfs4_dss_bufp, nfs4_dss_buflen);
354                 kmem_free(nfs4_dss_bufp, nfs4_dss_buflen);

356                 break;
357         }

359         case NFS4_EPHEMERAL_MOUNT_TO: {
360                 uint_t  mount_to;

362                 /*
363                  * Not a very complicated call.
364                  */
365                 if (copyin(arg, &mount_to, sizeof (mount_to)))
366                         return (set_errno(EFAULT));
367                 nfs4_ephemeral_set_mount_to(mount_to);
368                 error = 0;
369                 break;
370         }

372         case MOUNTD_ARGS: {
373                 uint_t  did;

375                 /*
376                  * For now, only passing down the door fd; if we
377                  * ever need to pass down more info, we can use
378                  * a (properly aligned) struct.
379                  */
380                 if (copyin(arg, &did, sizeof (did)))
381                         return (set_errno(EFAULT));
382                 mountd_args(did);
383                 error = 0;
384                 break;
385         }

387         case NFSCMD_ARGS: {
388                 uint_t  did;

390                 /*
391                  * For now, only passing down the door fd; if we
392                  * ever need to pass down more info, we can use
393                  * a (properly aligned) struct.
394                  */
395                 if (copyin(arg, &did, sizeof (did)))
396                         return (set_errno(EFAULT));
397                 nfscmd_args(did);
398                 error = 0;
399                 break;
400         }

402         default:
403                 error = EINVAL;
404                 break;
405         }

407         return ((error != 0) ? set_errno(error) : 0);
408 }
```

unchanged_portion_omitted_

```
**********************************************************
   53529 Thu Oct 23 11:04:53 2014
new/usr/src/uts/common/fs/ufs/ufs_acl.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 712 /*ARGSUSED2*/
 713 int
 714 ufs_acl_get(struct inode *ip, vsecattr_t *vsap, int flag, cred_t *cr)
 715 {
 716         aclent_t        *aclentp;

 718         ASSERT(RW_LOCK_HELD(&ip->i_contents));

 720         /* XXX Range check, sanity check, shadow check */
 721         /* If an ACL is present, get the data from the shadow inode info */
 722         if (ip->i_ufs_acl)
 723                 return (aclentry2vsecattr(ip->i_ufs_acl, vsap));

 725         /*
 726          * If no ACLs are present, fabricate one from the mode bits.
 727          * This code is almost identical to fs_fab_acl(), but we
 728          * already have the mode bits handy, so we'll avoid going
 729          * through VOP_GETATTR() again.
 730          */

 732         vsap->vsa_aclcnt    = 0;
 733         vsap->vsa_aclentp   = NULL;
 734         vsap->vsa_dfaclcnt  = 0;        /* Default ACLs are not fabricated */
 735         vsap->vsa_dfaclentp = NULL;

 737         if (vsap->vsa_mask & (VSA_ACLCNT | VSA_ACL))
 738                 vsap->vsa_aclcnt    = 4;  /* USER, GROUP, OTHER, and CLASS */

 740         if (vsap->vsa_mask & VSA_ACL) {
 741                 vsap->vsa_aclentp = kmem_zalloc(4 * sizeof (aclent_t),
 742                     KM_SLEEP);

 743                 if (vsap->vsa_aclentp == NULL)
 744                         return (ENOMEM);
 744                 aclentp = vsap->vsa_aclentp;

 746                 /* Owner */
 747                 aclentp->a_type = USER_OBJ;
 748                 aclentp->a_perm = ((ushort_t)(ip->i_mode & 0700)) >> 6;
 749                 aclentp->a_id = ip->i_uid;      /* Really undefined */
 750                 aclentp++;

 752                 /* Group */
 753                 aclentp->a_type = GROUP_OBJ;
 754                 aclentp->a_perm = ((ushort_t)(ip->i_mode & 0070)) >> 3;
 755                 aclentp->a_id = ip->i_gid;      /* Really undefined */
 756                 aclentp++;

 758                 /* Other */
 759                 aclentp->a_type = OTHER_OBJ;
 760                 aclentp->a_perm = ip->i_mode & 0007;
 761                 aclentp->a_id = 0;              /* Really undefined */
 762                 aclentp++;

 764                 /* Class */
 765                 aclentp->a_type = CLASS_OBJ;
 766                 aclentp->a_perm = ((ushort_t)(ip->i_mode & 0070)) >> 3;
 767                 aclentp->a_id = 0;              /* Really undefined */
```



```
 768                 ksort((caddr_t)vsap->vsa_aclentp, vsap->vsa_aclcnt,
 769                     sizeof (aclent_t), cmp2acls);
 770         }

 772         return (0);
 773 }
______unchanged_portion_omitted_
```

```
**********************************************************
  242489 Thu Oct 23 11:04:53 2014
new/usr/src/uts/common/io/aac/aac.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

6189 static int
6190 aac_create_slots(struct aac_softstate *softs)
6191 {
6192         int i;

6194         softs->total_slots = softs->aac_max_fibs;
6195         softs->io_slot = kmem_zalloc(sizeof (struct aac_slot) * \
6196             softs->total_slots, KM_SLEEP);
6197         if (softs->io_slot == NULL) {
6198                 AACDB_PRINT(softs, CE_WARN, "Cannot allocate slot");
6199                 return (AACERR);
6200         }
6197         for (i = 0; i < softs->total_slots; i++)
6198                 softs->io_slot[i].index = i;
6199         softs->free_io_slot_head = NULL;
6200         softs->total_fibs = 0;
6201         return (AACOK);
6202 }
______unchanged_portion_omitted_
```

```
**********************************************************
   12211 Thu Oct 23 11:04:53 2014
new/usr/src/uts/common/io/arn/arn_phy.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 335 boolean_t
 336 ath9k_hw_init_rf(struct ath_hal *ah, int *status)
 337 {
 338         struct ath_hal_5416 *ahp = AH5416(ah);

 340         if (!AR_SREV_9280_10_OR_LATER(ah)) {

 342                 ahp->ah_analogBank0Data =
 343                     kmem_zalloc((sizeof (uint32_t) *
 344                     ahp->ah_iniBank0.ia_rows), KM_SLEEP);
 345                 ahp->ah_analogBank1Data =
 346                     kmem_zalloc((sizeof (uint32_t) *
 347                     ahp->ah_iniBank1.ia_rows), KM_SLEEP);
 348                 ahp->ah_analogBank2Data =
 349                     kmem_zalloc((sizeof (uint32_t) *
 350                     ahp->ah_iniBank2.ia_rows), KM_SLEEP);
 351                 ahp->ah_analogBank3Data =
 352                     kmem_zalloc((sizeof (uint32_t) *
 353                     ahp->ah_iniBank3.ia_rows), KM_SLEEP);
 354                 ahp->ah_analogBank6Data =
 355                     kmem_zalloc((sizeof (uint32_t) *
 356                     ahp->ah_iniBank6.ia_rows), KM_SLEEP);
 357                 ahp->ah_analogBank6TPCData =
 358                     kmem_zalloc((sizeof (uint32_t) *
 359                     ahp->ah_iniBank6TPC.ia_rows), KM_SLEEP);
 360                 ahp->ah_analogBank7Data =
 361                     kmem_zalloc((sizeof (uint32_t) *
 362                     ahp->ah_iniBank7.ia_rows), KM_SLEEP);

 364                 if (ahp->ah_analogBank0Data == NULL ||
 365                     ahp->ah_analogBank1Data == NULL ||
 366                     ahp->ah_analogBank2Data == NULL ||
 367                     ahp->ah_analogBank3Data == NULL ||
 368                     ahp->ah_analogBank6Data == NULL ||
 369                     ahp->ah_analogBank6TPCData == NULL ||
 370                     ahp->ah_analogBank7Data == NULL) {
 371                         ARN_DBG((ARN_DBG_FATAL, "arn: ath9k_hw_init_rf(): "
 372                             "cannot allocate RF banks\n"));
 373                         *status = ENOMEM;
 374                         return (B_FALSE);
 375                 }

 377                 ahp->ah_addac5416_21 =
 378                     kmem_zalloc((sizeof (uint32_t) *
 379                     ahp->ah_iniAddac.ia_rows *
 380                     ahp->ah_iniAddac.ia_columns), KM_SLEEP);
 381                 if (ahp->ah_addac5416_21 == NULL) {
 382                         ARN_DBG((ARN_DBG_FATAL, "arn: ath9k_hw_init_rf(): "
 383                             "cannot allocate ah_addac5416_21\n"));
 384                         *status = ENOMEM;
 385                         return (B_FALSE);
 386                 }

 382                 ahp->ah_bank6Temp =
 383                     kmem_zalloc((sizeof (uint32_t) *
 384                     ahp->ah_iniBank6.ia_rows), KM_SLEEP);
 391                 if (ahp->ah_bank6Temp == NULL) {
 392                         ARN_DBG((ARN_DBG_FATAL, "arn: ath9k_hw_init_rf(): "
```



```
 393                             "cannot allocate ah_bank6Temp\n"));
 394                         *status = ENOMEM;
 395                         return (B_FALSE);
 396                 }
 385         }

 387         return (B_TRUE);
 388 }
______unchanged_portion_omitted_
```

```
**********************************************************
   10755 Thu Oct 23 11:04:54 2014
new/usr/src/uts/common/io/beep.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
```

```
/*
 * 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"

/*
 * This is the Beep module for supporting keyboard beep for keyboards
 * that do not have the beeping feature within themselves
 *
 */

#include <sys/types.h>
#include <sys/conf.h>

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/ddi_impldefs.h>
#include <sys/kmem.h>

#include <sys/beep.h>
#include <sys/inttypes.h>

/*
 * Debug stuff
 * BEEP_DEBUG used for errors
 * BEEP_DEBUG1 prints when beep_debug > 1 and used for normal messages
 */
#ifdef DEBUG
int beep_debug = 0;
#define BEEP_DEBUG(args)        if (beep_debug) cmn_err args
#define BEEP_DEBUG1(args)       if (beep_debug > 1) cmn_err args
#else
#define BEEP_DEBUG(args)
#define BEEP_DEBUG1(args)
#endif

int beep_queue_size = BEEP_QUEUE_SIZE;
```



```
/*
 * Note that mutex_init is not called on the mutex in beep_state,
 * But assumes that zeroed memory does not need to call mutex_init,
 * as documented in mutex.c
 */

beep_state_t beep_state;

beep_params_t beep_params[] = {
        {BEEP_CONSOLE,  900,    200},
        {BEEP_TYPE4,    2000,   0},
        {BEEP_DEFAULT,  1000,   200},   /* Must be last */
};


/*
 * beep_init:
 * Allocate the beep_queue structure
 * Initialize beep_state structure
 * Called from beep driver attach routine
 */

int
beep_init(void *arg,
    beep_on_func_t beep_on_func,
    beep_off_func_t beep_off_func,
    beep_freq_func_t beep_freq_func)
{
        beep_entry_t *queue;

        BEEP_DEBUG1((CE_CONT,
            "beep_init(0x%lx, 0x%lx, 0x%lx, 0x%lx) : start.",
            (unsigned long) arg,
            (unsigned long) beep_on_func,
            (unsigned long) beep_off_func,
            (unsigned long) beep_freq_func));

        mutex_enter(&beep_state.mutex);

        if (beep_state.mode != BEEP_UNINIT) {
                mutex_exit(&beep_state.mutex);
                BEEP_DEBUG((CE_WARN,
                    "beep_init : beep_state already initialized."));
                return (DDI_SUCCESS);
        }

        queue = kmem_zalloc(sizeof (beep_entry_t) * beep_queue_size,
            KM_SLEEP);

        BEEP_DEBUG1((CE_CONT,
            "beep_init : beep_queue kmem_zalloc(%d) = 0x%lx.",
            (int)sizeof (beep_entry_t) * beep_queue_size,
            (unsigned long)queue));

        if (queue == NULL) {
                BEEP_DEBUG((CE_WARN,
                    "beep_init : kmem_zalloc of beep_queue failed."));
                return (DDI_FAILURE);
        }

        beep_state.arg = arg;
        beep_state.mode = BEEP_OFF;
        beep_state.beep_freq = beep_freq_func;
        beep_state.beep_on = beep_on_func;
        beep_state.beep_off = beep_off_func;
```

```
 119         beep_state.timeout_id = 0;

 121         beep_state.queue_head = 0;
 122         beep_state.queue_tail = 0;
 123         beep_state.queue_size = beep_queue_size;
 124         beep_state.queue = queue;

 126         mutex_exit(&beep_state.mutex);

 128         BEEP_DEBUG1((CE_CONT, "beep_init : done."));
 129         return (DDI_SUCCESS);
 130 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   39281 Thu Oct 23 11:04:54 2014
new/usr/src/uts/common/io/blkdev/blkdev.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1475 static int
1476 bd_flush_write_cache(bd_t *bd, struct dk_callback *dkc)
1477 {
1478         buf_t                   *bp;
1479         struct dk_callback      *dc;
1480         bd_xfer_impl_t          *xi;
1481         int                     rv;

1483         if (bd->d_ops.o_sync_cache == NULL) {
1484                 return (ENOTSUP);
1485         }
1486         bp = getrbuf(KM_SLEEP);
1486         if ((bp = getrbuf(KM_SLEEP)) == NULL) {
1487                 return (ENOMEM);
1488         }
1487         bp->b_resid = 0;
1488         bp->b_bcount = 0;

1490         xi = bd_xfer_alloc(bd, bp, bd->d_ops.o_sync_cache, KM_SLEEP);
1491         if (xi == NULL) {
1492                 rv = geterror(bp);
1493                 freerbuf(bp);
1494                 return (rv);
1495         }

1497         /* Make an asynchronous flush, but only if there is a callback */
1498         if (dkc != NULL && dkc->dkc_callback != NULL) {
1499                 /* Make a private copy of the callback structure */
1500                 dc = kmem_alloc(sizeof (*dc), KM_SLEEP);
1501                 *dc = *dkc;
1502                 bp->b_private = dc;
1503                 bp->b_iodone = bd_flush_write_cache_done;

1505                 bd_submit(bd, xi);
1506                 return (0);
1507         }

1509         /* In case there is no callback, perform a synchronous flush */
1510         bd_submit(bd, xi);
1511         (void) biowait(bp);
1512         rv = geterror(bp);
1513         freerbuf(bp);

1515         return (rv);
1516 }
______unchanged_portion_omitted_
```

****
97023 Thu Oct 23 11:04:54 2014
new/usr/src/uts/common/io/comstar/lu/stmf_sbd/sbd.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
****
_____***unchanged_portion_omitted_***

```
3360 /*
3361  * this function creates a local metadata zvol property
3362  */
3363 sbd_status_t
3364 sbd_create_zfs_meta_object(sbd_lu_t *sl)
3365 {
3366         /*
3367          * -allocate 1/2 the property size, the zfs property
3368          *  is 8k in size and stored as ascii hex string, all
3369          *  we needed is 4k buffer to store the binary data.
3370          * -initialize reader/write lock
3371          */
3372         sl->sl_zfs_meta = kmem_zalloc(ZAP_MAXVALUELEN / 2, KM_SLEEP);
3372         if ((sl->sl_zfs_meta = kmem_zalloc(ZAP_MAXVALUELEN / 2, KM_SLEEP))
3373             == NULL)
3374                 return (SBD_FAILURE);
3373         rw_init(&sl->sl_zfs_meta_lock, NULL, RW_DRIVER, NULL);
3374         return (SBD_SUCCESS);
3375 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   35222 Thu Oct 23 11:04:54 2014
new/usr/src/uts/common/io/comstar/port/pppt/pppt.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 342 /* ARGSUSED */
 343 static int
 344 pppt_drv_ioctl(dev_t drv, int cmd, intptr_t argp, int flag, cred_t *cred,
 345     int *retval)
 346 {
 347         int                             rc;
 348         void                            *buf;
 349         size_t                          buf_size;
 350         pppt_iocdata_t                  iocd;
 351         door_handle_t                   new_handle;

 353         if (drv_priv(cred) != 0) {
 354                 return (EPERM);
 355         }

 357         rc = ddi_copyin((void *)argp, &iocd, sizeof (iocd), flag);
 358         if (rc)
 359                 return (EFAULT);

 361         if (iocd.pppt_version != PPPT_VERSION_1)
 362                 return (EINVAL);

 364         switch (cmd) {
 365         case PPPT_MESSAGE:

 367                 /* XXX limit buf_size ? */
 368                 buf_size = (size_t)iocd.pppt_buf_size;
 369                 buf = kmem_alloc(buf_size, KM_SLEEP);
 370                 if (buf == NULL)
 371                         return (ENOMEM);

 371                 rc = ddi_copyin((void *)(unsigned long)iocd.pppt_buf,
 372                     buf, buf_size, flag);
 373                 if (rc) {
 374                         kmem_free(buf, buf_size);
 375                         return (EFAULT);
 376                 }

 378                 stmf_ic_rx_msg(buf, buf_size);

 380                 kmem_free(buf, buf_size);
 381                 break;
 382         case PPPT_INSTALL_DOOR:

 384                 new_handle = door_ki_lookup((int)iocd.pppt_door_fd);
 385                 if (new_handle == NULL)
 386                         return (EINVAL);

 388                 mutex_enter(&pppt_global.global_door_lock);
 389                 ASSERT(pppt_global.global_svc_state == PSS_ENABLED);
 390                 if (pppt_global.global_door != NULL) {
 391                         /*
 392                          * There can only be one door installed
 393                          */
 394                         mutex_exit(&pppt_global.global_door_lock);
 395                         door_ki_rele(new_handle);
 396                         return (EBUSY);
 397                 }
```



```
 398                 pppt_global.global_door = new_handle;
 399                 mutex_exit(&pppt_global.global_door_lock);
 400                 break;
 401         }

 403         return (rc);
 404 }
______unchanged_portion_omitted_
```

```
**********************************************************
  116791 Thu Oct 23 11:04:54 2014
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_init.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

/*
 * ql_configure_device_d_id
 *      Updates device loop ID.
 *      Also adds to device queue any new devices found on private loop.
 *
 * Input:
 *      ha = adapter state pointer.
 *
 * Returns:
 *      ql local function return status code.
 *
 * Context:
 *      Kernel context.
 */
static int
ql_configure_device_d_id(ql_adapter_state_t *ha)
{
        port_id_t               d_id;
        ql_link_t               *link;
        int                     rval;
        int                     loop;
        ql_tgt_t                *tq;
        ql_dev_id_list_t        *list;
        uint32_t                list_size;
        uint16_t                index, loop_id;
        ql_mbx_data_t           mr;
        uint8_t                 retries = MAX_DEVICE_LOST_RETRY;

        QL_PRINT_3(CE_CONT, "(%d): started\n", ha->instance);

        list_size = sizeof (ql_dev_id_list_t) * DEVICE_LIST_ENTRIES;
        list = kmem_zalloc(list_size, KM_SLEEP);
        if (list == NULL) {
                rval = QL_MEMORY_ALLOC_FAILED;
                EL(ha, "failed, rval = %xh\n", rval);
                return (rval);
        }

        do {
                /*
                 * Get data from RISC code d_id list to init each device queue.
                 */
                rval = ql_get_id_list(ha, (caddr_t)list, list_size, &mr);
                if (rval != QL_SUCCESS) {
                        kmem_free(list, list_size);
                        EL(ha, "failed, rval = %xh\n", rval);
                        return (rval);
                }

                /* Acquire adapter state lock. */
                ADAPTER_STATE_LOCK(ha);

                /* Mark all queues as unusable. */
                for (index = 0; index < DEVICE_HEAD_LIST_SIZE; index++) {
                        for (link = ha->dev[index].first; link != NULL;
                            link = link->next) {
                                tq = link->base_address;
                                DEVICE_QUEUE_LOCK(tq);
```



```
                                if (!(tq->flags & TQF_PLOGI_PROGRS) &&
                                    !(ha->topology & QL_N_PORT)) {
                                        tq->loop_id = (uint16_t)
                                            (tq->loop_id | PORT_LOST_ID);
                                }
                                DEVICE_QUEUE_UNLOCK(tq);
                        }
                }

                /* If device not in queues add new queue. */
                for (index = 0; index < mr.mb[1]; index++) {
                        ql_dev_list(ha, list, index, &d_id, &loop_id);

                        if (VALID_DEVICE_ID(ha, loop_id)) {
                                tq = ql_dev_init(ha, d_id, loop_id);
                                if (tq != NULL) {
                                        tq->loop_id = loop_id;

                                        /* Test for fabric device. */
                                        if (d_id.b.domain !=
                                            ha->d_id.b.domain ||
                                            d_id.b.area != ha->d_id.b.area) {
                                                tq->flags |= TQF_FABRIC_DEVICE;
                                        }

                                        ADAPTER_STATE_UNLOCK(ha);
                                        if (ql_get_port_database(ha, tq,
                                            PDF_NONE) == QL_SUCCESS) {
                                                ADAPTER_STATE_LOCK(ha);
                                                tq->loop_id = (uint16_t)
                                                    (tq->loop_id &
                                                    ~PORT_LOST_ID);
                                        } else {
                                                ADAPTER_STATE_LOCK(ha);
                                        }
                                }
                                DEVICE_QUEUE_UNLOCK(tq);
                        }
                }

                /* 24xx does not report switch devices in ID list. */
                if ((CFG_IST(ha, CFG_CTRL_24258081)) &&
                    ha->topology & (QL_F_PORT | QL_FL_PORT)) {
                        d_id.b24 = 0xfffffe;
                        tq = ql_dev_init(ha, d_id, FL_PORT_24XX_HDL);
                        if (tq != NULL) {
                                tq->flags |= TQF_FABRIC_DEVICE;
                                ADAPTER_STATE_UNLOCK(ha);
                                (void) ql_get_port_database(ha, tq, PDF_NONE);
                                ADAPTER_STATE_LOCK(ha);
                        }
                        d_id.b24 = 0xfffffc;
                        tq = ql_dev_init(ha, d_id, SNS_24XX_HDL);
                        if (tq != NULL) {
                                tq->flags |= TQF_FABRIC_DEVICE;
                                ADAPTER_STATE_UNLOCK(ha);
                                if (ha->vp_index != 0) {
                                        (void) ql_login_fport(ha, tq,
                                            SNS_24XX_HDL, LFF_NONE, NULL);
                                }
                                (void) ql_get_port_database(ha, tq, PDF_NONE);
                                ADAPTER_STATE_LOCK(ha);
                        }
                }

                /* If F_port exists, allocate queue for FL_Port. */
                index = ql_alpa_to_index[0xfe];
```

```
3344                d_id.b24 = 0;
3345                if (ha->dev[index].first != NULL) {
3346                        tq = ql_dev_init(ha, d_id, (uint16_t)
3347                            (CFG_IST(ha, CFG_CTRL_24258081) ?
3348                            FL_PORT_24XX_HDL : FL_PORT_LOOP_ID));
3349                        if (tq != NULL) {
3350                                tq->flags |= TQF_FABRIC_DEVICE;
3351                                ADAPTER_STATE_UNLOCK(ha);
3352                                (void) ql_get_port_database(ha, tq, PDF_NONE);
3353                                ADAPTER_STATE_LOCK(ha);
3354                        }
3355                }

3357                /* Allocate queue for broadcast. */
3358                d_id.b24 = 0xffffff;
3359                (void) ql_dev_init(ha, d_id, (uint16_t)
3360                    (CFG_IST(ha, CFG_CTRL_24258081) ? BROADCAST_24XX_HDL :
3361                    IP_BROADCAST_LOOP_ID));

3363                /* Check for any devices lost. */
3364                loop = FALSE;
3365                for (index = 0; index < DEVICE_HEAD_LIST_SIZE; index++) {
3366                        for (link = ha->dev[index].first; link != NULL;
3367                            link = link->next) {
3368                                tq = link->base_address;

3370                                if ((tq->loop_id & PORT_LOST_ID) &&
3371                                    !(tq->flags & (TQF_INITIATOR_DEVICE |
3372                                    TQF_FABRIC_DEVICE))) {
3373                                        loop = TRUE;
3374                                }
3375                        }
3376                }

3378                /* Release adapter state lock. */
3379                ADAPTER_STATE_UNLOCK(ha);

3381                /* Give devices time to recover. */
3382                if (loop == TRUE) {
3383                        drv_usecwait(1000000);
3384                }
3385        } while (retries-- && loop == TRUE &&
3386            !(ha->pha->task_daemon_flags & LOOP_RESYNC_NEEDED));

3388        kmem_free(list, list_size);

3390        if (rval != QL_SUCCESS) {
3391                EL(ha, "failed=%xh\n", rval);
3392        } else {
3393                /*EMPTY*/
3394                QL_PRINT_3(CE_CONT, "(%d): done\n", ha->instance);
3395        }

3397        return (rval);
3398 }
```

*unchanged_portion_omitted_*

```
4063 /*
4064  * ql_vport_control
4065  *      Issue Virtual Port Control command.
4066  *
4067  * Input:
4068  *      ha = virtual adapter state pointer.
4069  *      cmd = control command.
4070  *
4071  * Returns:
```



```
4072  *      ql local function return status code.
4073  *
4074  * Context:
4075  *      Kernel context.
4076  */
4077 int
4078 ql_vport_control(ql_adapter_state_t *ha, uint8_t cmd)
4079 {
4080         ql_mbx_iocb_t   *pkt;
4081         uint8_t         bit;
4082         int             rval;
4083         uint32_t        pkt_size;

4085         QL_PRINT_10(CE_CONT, "(%d,%d): started\n", ha->instance, ha->vp_index);

4087         if (ha->vp_index != 0) {
4088                 pkt_size = sizeof (ql_mbx_iocb_t);
4089                 pkt = kmem_zalloc(pkt_size, KM_SLEEP);
4095                 if (pkt == NULL) {
4096                         EL(ha, "failed, kmem_zalloc\n");
4097                         return (QL_MEMORY_ALLOC_FAILED);
4098                 }

4091                 pkt->vpc.entry_type = VP_CONTROL_TYPE;
4092                 pkt->vpc.entry_count = 1;
4093                 pkt->vpc.command = cmd;
4094                 pkt->vpc.vp_count = 1;
4095                 bit = (uint8_t)(ha->vp_index - 1);
4096                 pkt->vpc.vp_index[bit / 8] = (uint8_t)
4097                     (pkt->vpc.vp_index[bit / 8] | BIT_0 << bit % 8);

4099                 rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, pkt_size);
4100                 if (rval == QL_SUCCESS && pkt->vpc.status != 0) {
4101                         rval = QL_COMMAND_ERROR;
4102                 }

4104                 kmem_free(pkt, pkt_size);
4105         } else {
4106                 rval = QL_SUCCESS;
4107         }

4109         if (rval != QL_SUCCESS) {
4110                 EL(ha, "failed, rval = %xh\n", rval);
4111         } else {
4112                 /*EMPTY*/
4113                 QL_PRINT_10(CE_CONT, "(%d,%d): done\n", ha->instance,
4114                     ha->vp_index);
4115         }
4116         return (rval);
4117 }

4119 /*
4120  * ql_vport_modify
4121  *      Issue of Modify Virtual Port command.
4122  *
4123  * Input:
4124  *      ha = virtual adapter state pointer.
4125  *      cmd = command.
4126  *      opt = option.
4127  *
4128  * Context:
4129  *      Interrupt or Kernel context, no mailbox commands allowed.
4130  */
4131 int
4132 ql_vport_modify(ql_adapter_state_t *ha, uint8_t cmd, uint8_t opt)
4133 {
```

```
4134         ql_mbx_iocb_t   *pkt;
4135         int             rval;
4136         uint32_t        pkt_size;

4138         QL_PRINT_10(CE_CONT, "(%d,%d): started\n", ha->instance, ha->vp_index);

4140         pkt_size = sizeof (ql_mbx_iocb_t);
4141         pkt = kmem_zalloc(pkt_size, KM_SLEEP);
4151         if (pkt == NULL) {
4152                 EL(ha, "failed, kmem_zalloc\n");
4153                 return (QL_MEMORY_ALLOC_FAILED);
4154         }

4143         pkt->vpm.entry_type = VP_MODIFY_TYPE;
4144         pkt->vpm.entry_count = 1;
4145         pkt->vpm.command = cmd;
4146         pkt->vpm.vp_count = 1;
4147         pkt->vpm.first_vp_index = ha->vp_index;
4148         pkt->vpm.first_options = opt;
4149         bcopy(ha->loginparams.nport_ww_name.raw_wwn, pkt->vpm.first_port_name,
4150             8);
4151         bcopy(ha->loginparams.node_ww_name.raw_wwn, pkt->vpm.first_node_name,
4152             8);

4154         rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, pkt_size);
4155         if (rval == QL_SUCCESS && pkt->vpm.status != 0) {
4156                 EL(ha, "failed, ql_issue_mbx_iocb=%xh, status=%xh\n", rval,
4157                     pkt->vpm.status);
4158                 rval = QL_COMMAND_ERROR;
4159         }

4161         kmem_free(pkt, pkt_size);

4163         if (rval != QL_SUCCESS) {
4164                 EL(ha, "failed, rval = %xh\n", rval);
4165         } else {
4166                 /*EMPTY*/
4167                 QL_PRINT_10(CE_CONT, "(%d,%d): done\n", ha->instance,
4168                     ha->vp_index);
4169         }
4170         return (rval);
4171 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   52613 Thu Oct 23 11:04:55 2014
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_ioctl.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 485 /*
 486  * Set adapter feature bits in NVRAM
 487  */
 488 static int
 489 ql_set_feature_bits(ql_adapter_state_t *ha, uint16_t features)
 490 {
 491         int             rval;
 492         uint32_t        count;
 493         nvram_t         *nv;
 494         uint16_t        *wptr;
 495         uint8_t         *bptr;
 496         uint8_t         csum;
 497         uint32_t        start_addr;

 499         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

 501         if (CFG_IST(ha, CFG_CTRL_24258081)) {
 502                 EL(ha, "Not supported for 24xx\n");
 503                 return (EINVAL);
 504         }

 506         nv = kmem_zalloc(sizeof (*nv), KM_SLEEP);
 507         if (nv == NULL) {
 508                 EL(ha, "failed, kmem_zalloc\n");
 509                 return (ENOMEM);
 510         }

 508         rval = ql_lock_nvram(ha, &start_addr, LNF_NVRAM_DATA);
 509         if (rval != QL_SUCCESS) {
 510                 EL(ha, "failed, ql_lock_nvram=%xh\n", rval);
 511                 kmem_free(nv, sizeof (*nv));
 512                 return (EIO);
 513         }
 514         rval = 0;

 516         /*
 517          * Read off the whole NVRAM
 518          */
 519         wptr = (uint16_t *)nv;
 520         csum = 0;
 521         for (count = 0; count < sizeof (nvram_t) / 2; count++) {
 522                 *wptr = (uint16_t)ql_get_nvram_word(ha, count + start_addr);
 523                 csum = (uint8_t)(csum + (uint8_t)*wptr);
 524                 csum = (uint8_t)(csum + (uint8_t)(*wptr >> 8));
 525                 wptr++;
 526         }

 528         /*
 529          * If the checksum is BAD then fail it right here.
 530          */
 531         if (csum) {
 532                 kmem_free(nv, sizeof (*nv));
 533                 ql_release_nvram(ha);
 534                 return (EBADF);
 535         }

 537         nv->adapter_features[0] = (uint8_t)((features & 0xFF00) >> 8);
 538         nv->adapter_features[1] = (uint8_t)(features & 0xFF);
```



```
 540         /*
 541          * Recompute the chesksum now
 542          */
 543         bptr = (uint8_t *)nv;
 544         for (count = 0; count < sizeof (nvram_t) - 1; count++) {
 545                 csum = (uint8_t)(csum + *bptr++);
 546         }
 547         csum = (uint8_t)(~csum + 1);
 548         nv->checksum = csum;

 550         /*
 551          * Now load the NVRAM
 552          */
 553         wptr = (uint16_t *)nv;
 554         for (count = 0; count < sizeof (nvram_t) / 2; count++) {
 555                 ql_load_nvram(ha, (uint8_t)(count + start_addr), *wptr++);
 556         }

 558         /*
 559          * Read NVRAM and verify the contents
 560          */
 561         wptr = (uint16_t *)nv;
 562         csum = 0;
 563         for (count = 0; count < sizeof (nvram_t) / 2; count++) {
 564                 if (ql_get_nvram_word(ha, count + start_addr) != *wptr) {
 565                         rval = EIO;
 566                         break;
 567                 }
 568                 csum = (uint8_t)(csum + (uint8_t)*wptr);
 569                 csum = (uint8_t)(csum + (uint8_t)(*wptr >> 8));
 570                 wptr++;
 571         }

 573         if (csum) {
 574                 rval = EINVAL;
 575         }

 577         kmem_free(nv, sizeof (*nv));
 578         ql_release_nvram(ha);

 580         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

 582         return (rval);
 583 }

 585 /*
 586  * Fix this function to update just feature bits and checksum in NVRAM
 587  */
 588 static int
 589 ql_set_nvram_adapter_defaults(ql_adapter_state_t *ha)
 590 {
 591         int             rval;
 592         uint32_t        count;
 593         uint32_t        start_addr;

 595         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

 597         rval = ql_lock_nvram(ha, &start_addr, LNF_NVRAM_DATA);
 598         if (rval != QL_SUCCESS) {
 599                 EL(ha, "failed, ql_lock_nvram=%xh\n", rval);
 600                 return (EIO);
 601         }
 602         rval = 0;

 604         if (CFG_IST(ha, CFG_CTRL_24258081)) {
```

```
605                 nvram_24xx_t    *nv;
606                 uint32_t        *longptr;
607                 uint32_t        csum = 0;

609                 nv = kmem_zalloc(sizeof (*nv), KM_SLEEP);
614                 if (nv == NULL) {
615                         EL(ha, "failed, kmem_zalloc\n");
616                         return (ENOMEM);
617                 }

611                 nv->nvram_version[0] = LSB(ICB_24XX_VERSION);
612                 nv->nvram_version[1] = MSB(ICB_24XX_VERSION);

614                 nv->version[0] = 1;
615                 nv->max_frame_length[1] = 8;
616                 nv->execution_throttle[0] = 16;
617                 nv->login_retry_count[0] = 8;

619                 nv->firmware_options_1[0] = BIT_2 | BIT_1;
620                 nv->firmware_options_1[1] = BIT_5;
621                 nv->firmware_options_2[0] = BIT_5;
622                 nv->firmware_options_2[1] = BIT_4;
623                 nv->firmware_options_3[1] = BIT_6;

625                 /*
626                  * Set default host adapter parameters
627                  */
628                 nv->host_p[0] = BIT_4 | BIT_1;
629                 nv->host_p[1] = BIT_3 | BIT_2;
630                 nv->reset_delay = 5;
631                 nv->max_luns_per_target[0] = 128;
632                 nv->port_down_retry_count[0] = 30;
633                 nv->link_down_timeout[0] = 30;

635                 /*
636                  * compute the chesksum now
637                  */
638                 longptr = (uint32_t *)nv;
639                 csum = 0;
640                 for (count = 0; count < (sizeof (nvram_24xx_t)/4)-1; count++) {
641                         csum += *longptr;
642                         longptr++;
643                 }
644                 csum = (uint32_t)(~csum + 1);
645                 LITTLE_ENDIAN_32((long)csum);
646                 *longptr = csum;

648                 /*
649                  * Now load the NVRAM
650                  */
651                 longptr = (uint32_t *)nv;
652                 for (count = 0; count < sizeof (nvram_24xx_t) / 4; count++) {
653                         (void) ql_24xx_load_nvram(ha,
654                             (uint32_t)(count + start_addr), *longptr++);
655                 }

657                 /*
658                  * Read NVRAM and verify the contents
659                  */
660                 csum = 0;
661                 longptr = (uint32_t *)nv;
662                 for (count = 0; count < sizeof (nvram_24xx_t) / 4; count++) {
663                         rval = ql_24xx_read_flash(ha, count + start_addr,
664                             longptr);
665                         if (rval != QL_SUCCESS) {
666                                 EL(ha, "24xx_read_flash failed=%xh\n", rval);
```



```
667                                 break;
668                         }
669                         csum += *longptr;
670                 }

672                 if (csum) {
673                         rval = EINVAL;
674                 }
675                 kmem_free(nv, sizeof (nvram_24xx_t));
676         } else {
677                 nvram_t         *nv;
678                 uint16_t        *wptr;
679                 uint8_t         *bptr;
680                 uint8_t         csum;

682                 nv = kmem_zalloc(sizeof (*nv), KM_SLEEP);
691                 if (nv == NULL) {
692                         EL(ha, "failed, kmem_zalloc\n");
693                         return (ENOMEM);
694                 }
683                 /*
684                  * Set default initialization control block.
685                  */
686                 nv->parameter_block_version = ICB_VERSION;
687                 nv->firmware_options[0] = BIT_4 | BIT_3 | BIT_2 | BIT_1;
688                 nv->firmware_options[1] = BIT_7 | BIT_5 | BIT_2;

690                 nv->max_frame_length[1] = 4;
691                 nv->max_iocb_allocation[1] = 1;
692                 nv->execution_throttle[0] = 16;
693                 nv->login_retry_count = 8;
694                 nv->port_name[0] = 33;
695                 nv->port_name[3] = 224;
696                 nv->port_name[4] = 139;
697                 nv->login_timeout = 4;

699                 /*
700                  * Set default host adapter parameters
701                  */
702                 nv->host_p[0] = BIT_1;
703                 nv->host_p[1] = BIT_2;
704                 nv->reset_delay = 5;
705                 nv->port_down_retry_count = 8;
706                 nv->maximum_luns_per_target[0] = 8;

708                 /*
709                  * compute the chesksum now
710                  */
711                 bptr = (uint8_t *)nv;
712                 csum = 0;
713                 for (count = 0; count < sizeof (nvram_t) - 1; count++) {
714                         csum = (uint8_t)(csum + *bptr++);
715                 }
716                 csum = (uint8_t)(~csum + 1);
717                 nv->checksum = csum;

719                 /*
720                  * Now load the NVRAM
721                  */
722                 wptr = (uint16_t *)nv;
723                 for (count = 0; count < sizeof (nvram_t) / 2; count++) {
724                         ql_load_nvram(ha, (uint8_t)(count + start_addr),
725                             *wptr++);
726                 }

728                 /*
```

```
729                  * Read NVRAM and verify the contents
730                  */
731                 wptr = (uint16_t *)nv;
732                 csum = 0;
733                 for (count = 0; count < sizeof (nvram_t) / 2; count++) {
734                         if (ql_get_nvram_word(ha, count + start_addr) !=
735                             *wptr) {
736                                 rval = EIO;
737                                 break;
738                         }
739                         csum = (uint8_t)(csum + (uint8_t)*wptr);
740                         csum = (uint8_t)(csum + (uint8_t)(*wptr >> 8));
741                         wptr++;
742                 }
743                 if (csum) {
744                         rval = EINVAL;
745                 }
746                 kmem_free(nv, sizeof (*nv));
747         }
748         ql_release_nvram(ha);

750         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

752         return (rval);
753 }
_____unchanged_portion_omitted_

905 /*
906  * ql_nv_util_load
907  *      Loads NVRAM from application.
908  *
909  * Input:
910  *      ha = adapter state pointer.
911  *      bp = user buffer address.
912  *
913  * Returns:
914  *
915  * Context:
916  *      Kernel context.
917  */
918 int
919 ql_nv_util_load(ql_adapter_state_t *ha, void *bp, int mode)
920 {
921         uint8_t         cnt;
922         void            *nv;
923         uint16_t        *wptr;
924         uint16_t        data;
925         uint32_t        start_addr, *lptr, data32;
926         nvram_t         *nptr;
927         int             rval;

929         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

931         nv = kmem_zalloc(ha->nvram_cache->size, KM_SLEEP);
943         if ((nv = kmem_zalloc(ha->nvram_cache->size, KM_SLEEP)) == NULL) {
944                 EL(ha, "failed, kmem_zalloc\n");
945                 return (ENOMEM);
946         }

933         if (ddi_copyin(bp, nv, ha->nvram_cache->size, mode) != 0) {
934                 EL(ha, "Buffer copy failed\n");
935                 kmem_free(nv, ha->nvram_cache->size);
936                 return (EFAULT);
937         }

939         /* See if the buffer passed to us looks sane */
```



```
940         nptr = (nvram_t *)nv;
941         if (nptr->id[0] != 'I' || nptr->id[1] != 'S' || nptr->id[2] != 'P' ||
942             nptr->id[3] != ' ') {
943                 EL(ha, "failed, buffer sanity check\n");
944                 kmem_free(nv, ha->nvram_cache->size);
945                 return (EINVAL);
946         }

948         /* Quiesce I/O */
949         if (ql_stall_driver(ha, 0) != QL_SUCCESS) {
950                 EL(ha, "ql_stall_driver failed\n");
951                 kmem_free(nv, ha->nvram_cache->size);
952                 return (EBUSY);
953         }

955         rval = ql_lock_nvram(ha, &start_addr, LNF_NVRAM_DATA);
956         if (rval != QL_SUCCESS) {
957                 EL(ha, "failed, ql_lock_nvram=%xh\n", rval);
958                 kmem_free(nv, ha->nvram_cache->size);
959                 ql_restart_driver(ha);
960                 return (EIO);
961         }

963         /* Load NVRAM. */
964         if (CFG_IST(ha, CFG_CTRL_258081)) {
965                 GLOBAL_HW_UNLOCK();
966                 start_addr &= ~ha->flash_data_addr;
967                 start_addr <<= 2;
968                 if ((rval = ql_r_m_w_flash(ha, bp, ha->nvram_cache->size,
969                     start_addr, mode)) != QL_SUCCESS) {
970                         EL(ha, "nvram load failed, rval = %0xh\n", rval);
971                 }
972                 GLOBAL_HW_LOCK();
973         } else if (CFG_IST(ha, CFG_CTRL_2422)) {
974                 lptr = (uint32_t *)nv;
975                 for (cnt = 0; cnt < ha->nvram_cache->size / 4; cnt++) {
976                         data32 = *lptr++;
977                         LITTLE_ENDIAN_32(&data32);
978                         rval = ql_24xx_load_nvram(ha, cnt + start_addr,
979                             data32);
980                         if (rval != QL_SUCCESS) {
981                                 EL(ha, "failed, 24xx_load_nvram=%xh\n", rval);
982                                 break;
983                         }
984                 }
985         } else {
986                 wptr = (uint16_t *)nv;
987                 for (cnt = 0; cnt < ha->nvram_cache->size / 2; cnt++) {
988                         data = *wptr++;
989                         LITTLE_ENDIAN_16(&data);
990                         ql_load_nvram(ha, (uint8_t)(cnt + start_addr), data);
991                 }
992         }
993         /* switch to the new one */
994         NVRAM_CACHE_LOCK(ha);

996         kmem_free(ha->nvram_cache->cache, ha->nvram_cache->size);
997         ha->nvram_cache->cache = (void *)nptr;

999         NVRAM_CACHE_UNLOCK(ha);

1001        ql_release_nvram(ha);
1002        ql_restart_driver(ha);

1004        QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
```

```
1006         if (rval == QL_SUCCESS) {
1007                 return (0);
1008         }

1010         return (EFAULT);
1011 }
_____unchanged_portion_omitted_

1120 /*
1121  * ql_vpd_load
1122  *      Loads VPD from application.
1123  *
1124  * Input:
1125  *      ha = adapter state pointer.
1126  *      bp = user buffer address.
1127  *
1128  * Returns:
1129  *
1130  * Context:
1131  *      Kernel context.
1132  */
1133 int
1134 ql_vpd_load(ql_adapter_state_t *ha, void *bp, int mode)
1135 {
1136         uint8_t         cnt;
1137         uint8_t         *vpd, *vpdptr, *vbuf;
1138         uint32_t        start_addr, vpd_size, *lptr, data32;
1139         int             rval;

1141         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

1143         if ((CFG_IST(ha, CFG_CTRL_24258081)) == 0) {
1144                 EL(ha, "unsupported adapter feature\n");
1145                 return (ENOTSUP);
1146         }

1148         vpd_size = QL_24XX_VPD_SIZE;

1150         vpd = kmem_zalloc(vpd_size, KM_SLEEP);
1165         if ((vpd = kmem_zalloc(vpd_size, KM_SLEEP)) == NULL) {
1166                 EL(ha, "failed, kmem_zalloc\n");
1167                 return (ENOMEM);
1168         }

1152         if (ddi_copyin(bp, vpd, vpd_size, mode) != 0) {
1153                 EL(ha, "Buffer copy failed\n");
1154                 kmem_free(vpd, vpd_size);
1155                 return (EFAULT);
1156         }

1158         /* Sanity check the user supplied data via checksum */
1159         if ((vpdptr = ql_vpd_findtag(ha, vpd, "RV")) == NULL) {
1160                 EL(ha, "vpd RV tag missing\n");
1161                 kmem_free(vpd, vpd_size);
1162                 return (EINVAL);
1163         }

1165         vpdptr += 3;
1166         cnt = 0;
1167         vbuf = vpd;
1168         while (vbuf <= vpdptr) {
1169                 cnt += *vbuf++;
1170         }
1171         if (cnt != 0) {
1172                 EL(ha, "mismatched checksum, cal=%xh, passed=%xh\n",
1173                     (uint8_t)cnt, (uintptr_t)vpdptr);
```



```
1174                 kmem_free(vpd, vpd_size);
1175                 return (EINVAL);
1176         }

1178         /* Quiesce I/O */
1179         if (ql_stall_driver(ha, 0) != QL_SUCCESS) {
1180                 EL(ha, "ql_stall_driver failed\n");
1181                 kmem_free(vpd, vpd_size);
1182                 return (EBUSY);
1183         }

1185         rval = ql_lock_nvram(ha, &start_addr, LNF_VPD_DATA);
1186         if (rval != QL_SUCCESS) {
1187                 EL(ha, "failed, ql_lock_nvram=%xh\n", rval);
1188                 kmem_free(vpd, vpd_size);
1189                 ql_restart_driver(ha);
1190                 return (EIO);
1191         }

1193         /* Load VPD. */
1194         if (CFG_IST(ha, CFG_CTRL_258081)) {
1195                 GLOBAL_HW_UNLOCK();
1196                 start_addr &= ~ha->flash_data_addr;
1197                 start_addr <<= 2;
1198                 if ((rval = ql_r_m_w_flash(ha, bp, vpd_size, start_addr,
1199                     mode)) != QL_SUCCESS) {
1200                         EL(ha, "vpd load error: %xh\n", rval);
1201                 }
1202                 GLOBAL_HW_LOCK();
1203         } else {
1204                 lptr = (uint32_t *)vpd;
1205                 for (cnt = 0; cnt < vpd_size / 4; cnt++) {
1206                         data32 = *lptr++;
1207                         LITTLE_ENDIAN_32(&data32);
1208                         rval = ql_24xx_load_nvram(ha, cnt + start_addr,
1209                             data32);
1210                         if (rval != QL_SUCCESS) {
1211                                 EL(ha, "failed, 24xx_load_nvram=%xh\n", rval);
1212                                 break;
1213                         }
1214                 }
1215         }

1217         kmem_free(vpd, vpd_size);

1219         /* Update the vcache */
1220         CACHE_LOCK(ha);

1222         if (rval != QL_SUCCESS) {
1223                 EL(ha, "failed, load\n");
1224         } else if ((ha->vcache == NULL) && ((ha->vcache =
1225             kmem_zalloc(vpd_size, KM_SLEEP)) == NULL)) {
1226                 EL(ha, "failed, kmem_zalloc2\n");
1227         } else if (ddi_copyin(bp, ha->vcache, vpd_size, mode) != 0) {
1228                 EL(ha, "Buffer copy2 failed\n");
1229                 kmem_free(ha->vcache, vpd_size);
1230                 ha->vcache = NULL;
1231         }

1233         CACHE_UNLOCK(ha);

1235         ql_release_nvram(ha);
1236         ql_restart_driver(ha);

1238         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
```

```
1240         if (rval == QL_SUCCESS) {
1241                 return (0);
1242         }

1244         return (EFAULT);
1245 }

1247 /*
1248  * ql_vpd_dump
1249  *      Dumps VPD to application buffer.
1250  *
1251  * Input:
1252  *      ha = adapter state pointer.
1253  *      bp = user buffer address.
1254  *
1255  * Returns:
1256  *
1257  * Context:
1258  *      Kernel context.
1259  */
1260 int
1261 ql_vpd_dump(ql_adapter_state_t *ha, void *bp, int mode)
1262 {
1263         uint8_t         cnt;
1264         void            *vpd;
1265         uint32_t        start_addr, vpd_size, *lptr;
1266         int             rval = 0;

1268         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

1270         if ((CFG_IST(ha, CFG_CTRL_24258081)) == 0) {
1271                 EL(ha, "unsupported adapter feature\n");
1272                 return (EACCES);
1273         }

1275         vpd_size = QL_24XX_VPD_SIZE;

1277         CACHE_LOCK(ha);

1279         if (ha->vcache != NULL) {
1280                 /* copy back the vpd cache data */
1281                 if (ddi_copyout(ha->vcache, bp, vpd_size, mode) != 0) {
1282                         EL(ha, "Buffer copy failed\n");
1283                         rval = EFAULT;
1284                 }
1285                 CACHE_UNLOCK(ha);
1286                 return (rval);
1287         }

1289         vpd = kmem_zalloc(vpd_size, KM_SLEEP);
1307         if ((vpd = kmem_zalloc(vpd_size, KM_SLEEP)) == NULL) {
1308                 CACHE_UNLOCK(ha);
1309                 EL(ha, "failed, kmem_zalloc\n");
1310                 return (ENOMEM);
1311         }

1291         /* Quiesce I/O */
1292         if (ql_stall_driver(ha, 0) != QL_SUCCESS) {
1293                 CACHE_UNLOCK(ha);
1294                 EL(ha, "ql_stall_driver failed\n");
1295                 kmem_free(vpd, vpd_size);
1296                 return (EBUSY);
1297         }

1299         rval = ql_lock_nvram(ha, &start_addr, LNF_VPD_DATA);
1300         if (rval != QL_SUCCESS) {
```



```
1301                 CACHE_UNLOCK(ha);
1302                 EL(ha, "failed, ql_lock_nvram=%xh\n", rval);
1303                 kmem_free(vpd, vpd_size);
1304                 ql_restart_driver(ha);
1305                 return (EIO);
1306         }

1308         /* Dump VPD. */
1309         lptr = (uint32_t *)vpd;

1311         for (cnt = 0; cnt < vpd_size / 4; cnt++) {
1312                 rval = ql_24xx_read_flash(ha, start_addr++, lptr);
1313                 if (rval != QL_SUCCESS) {
1314                         EL(ha, "read_flash failed=%xh\n", rval);
1315                         rval = EAGAIN;
1316                         break;
1317                 }
1318                 LITTLE_ENDIAN_32(lptr);
1319                 lptr++;
1320         }

1322         ql_release_nvram(ha);
1323         ql_restart_driver(ha);

1325         if (ddi_copyout(vpd, bp, vpd_size, mode) != 0) {
1326                 CACHE_UNLOCK(ha);
1327                 EL(ha, "Buffer copy failed\n");
1328                 kmem_free(vpd, vpd_size);
1329                 return (EFAULT);
1330         }

1332         ha->vcache = vpd;

1334         CACHE_UNLOCK(ha);

1336         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

1338         if (rval != QL_SUCCESS) {
1339                 return (EFAULT);
1340         } else {
1341                 return (0);
1342         }
1343 }
______unchanged_portion_omitted_

1525 /*
1526  * ql_r_m_w_flash
1527  *      Read modify write from user space to flash.
1528  *
1529  * Input:
1530  *      ha:     adapter state pointer.
1531  *      dp:     source byte pointer.
1532  *      bc:     byte count.
1533  *      faddr:  flash byte address.
1534  *      mode:   flags.
1535  *
1536  * Returns:
1537  *      ql local function return status code.
1538  *
1539  * Context:
1540  *      Kernel context.
1541  */
1542 int
1543 ql_r_m_w_flash(ql_adapter_state_t *ha, caddr_t dp, uint32_t bc, uint32_t faddr,
1544     int mode)
1545 {
```

```
1546         uint8_t         *bp;
1547         uint32_t        xfer, bsize, saddr, ofst;
1548         int             rval = 0;

1550         QL_PRINT_9(CE_CONT, "(%d): started, dp=%ph, faddr=%xh, bc=%xh\n",
1551             ha->instance, (void *)dp, faddr, bc);

1553         bsize = ha->xioctl->fdesc.block_size;
1554         saddr = faddr & ~(bsize - 1);
1555         ofst = faddr & (bsize - 1);

1557         bp = kmem_zalloc(bsize, KM_SLEEP);
1579         if ((bp = kmem_zalloc(bsize, KM_SLEEP)) == NULL) {
1580                 EL(ha, "kmem_zalloc=null\n");
1581                 return (QL_MEMORY_ALLOC_FAILED);
1582         }

1559         while (bc) {
1560                 xfer = bc > bsize ? bsize : bc;
1561                 if (ofst + xfer > bsize) {
1562                         xfer = bsize - ofst;
1563                 }
1564                 QL_PRINT_9(CE_CONT, "(%d): dp=%ph, saddr=%xh, bc=%xh, "
1565                     "ofst=%xh, xfer=%xh\n", ha->instance, (void *)dp, saddr,
1566                     bc, ofst, xfer);

1568                 if (ofst || xfer < bsize) {
1569                         /* Dump Flash sector. */
1570                         if ((rval = ql_dump_fcode(ha, bp, bsize, saddr)) !=
1571                             QL_SUCCESS) {
1572                                 EL(ha, "dump_flash status=%x\n", rval);
1573                                 break;
1574                         }
1575                 }

1577                 /* Set new data. */
1578                 if ((rval = ddi_copyin(dp, (caddr_t)(bp + ofst), xfer,
1579                     mode)) != 0) {
1580                         EL(ha, "ddi_copyin status=%xh, dp=%ph, ofst=%xh, "
1581                             "xfer=%xh\n", rval, (void *)dp, ofst, xfer);
1582                         rval = QL_FUNCTION_FAILED;
1583                         break;
1584                 }

1586                 /* Write to flash. */
1587                 if ((rval = ql_load_fcode(ha, bp, bsize, saddr)) !=
1588                     QL_SUCCESS) {
1589                         EL(ha, "load_flash status=%x\n", rval);
1590                         break;
1591                 }
1592                 bc -= xfer;
1593                 dp += xfer;
1594                 saddr += bsize;
1595                 ofst = 0;
1596         }

1598         kmem_free(bp, bsize);

1600         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

1602         return (rval);
1603 }
```

*_____unchanged_portion_omitted_*

```
1990 /*
1991  * ql_adm_prop_update_int
```



```
1992  *      Performs qladm QL_PROP_UPDATE_INT command
1993  *
1994  * Input:
1995  *      ha:     adapter state pointer.
1996  *      dop:    ql_adm_op_t structure pointer.
1997  *      mode:   flags.
1998  *
1999  * Returns:
2000  *
2001  * Context:
2002  *      Kernel context.
2003  */
2004 static int
2005 ql_adm_prop_update_int(ql_adapter_state_t *ha, ql_adm_op_t *dop, int mode)
2006 {
2007         char    *prop_name;
2008         int     rval;

2010         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

2012         prop_name = kmem_zalloc(dop->length, KM_SLEEP);
2038         if (prop_name == NULL) {
2039                 EL(ha, "failed, kmem_zalloc\n");
2040                 return (ENOMEM);
2041         }

2014         if (ddi_copyin((void *)(uintptr_t)dop->buffer, prop_name, dop->length,
2015             mode) != 0) {
2016                 EL(ha, "failed, prop_name ddi_copyin\n");
2017                 kmem_free(prop_name, dop->length);
2018                 return (EFAULT);
2019         }

2021         /*LINTED [Solaris DDI_DEV_T_ANY Lint warning]*/
2022         if ((rval = ddi_prop_update_int(DDI_DEV_T_NONE, ha->dip, prop_name,
2023             (int)dop->option)) != DDI_PROP_SUCCESS) {
2024                 EL(ha, "failed, prop_update=%xh\n", rval);
2025                 kmem_free(prop_name, dop->length);
2026                 return (EINVAL);
2027         }

2029         kmem_free(prop_name, dop->length);

2031         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

2033         return (0);
2034 }

2036 /*
2037  * ql_adm_fw_dump
2038  *      Performs qladm QL_FW_DUMP command
2039  *
2040  * Input:
2041  *      ha:     adapter state pointer.
2042  *      dop:    ql_adm_op_t structure pointer.
2043  *      udop:   user space ql_adm_op_t structure pointer.
2044  *      mode:   flags.
2045  *
2046  * Returns:
2047  *
2048  * Context:
2049  *      Kernel context.
2050  */
2051 static int
2052 ql_adm_fw_dump(ql_adapter_state_t *ha, ql_adm_op_t *dop, void *udop, int mode)
2053 {
```

```
2054         caddr_t dmp;

2056         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

2058         if (dop->length < ha->risc_dump_size) {
2059                 EL(ha, "failed, incorrect length=%xh, size=%xh\n",
2060                     dop->length, ha->risc_dump_size);
2061                 return (EINVAL);
2062         }

2064         if (ha->ql_dump_state & QL_DUMP_VALID) {
2065                 dmp = kmem_zalloc(ha->risc_dump_size, KM_SLEEP);
2095                 if (dmp == NULL) {
2096                         EL(ha, "failed, kmem_zalloc\n");
2097                         return (ENOMEM);
2098                 }

2067                 dop->length = (uint32_t)ql_ascii_fw_dump(ha, dmp);
2068                 if (ddi_copyout((void *)dmp, (void *)(uintptr_t)dop->buffer,
2069                     dop->length, mode) != 0) {
2070                         EL(ha, "failed, ddi_copyout\n");
2071                         kmem_free(dmp, ha->risc_dump_size);
2072                         return (EFAULT);
2073                 }

2075                 kmem_free(dmp, ha->risc_dump_size);
2076                 ha->ql_dump_state |= QL_DUMP_UPLOADED;

2078         } else {
2079                 EL(ha, "failed, no dump file\n");
2080                 dop->length = 0;
2081         }

2083         if (ddi_copyout(dop, udop, sizeof (ql_adm_op_t), mode) != 0) {
2084                 EL(ha, "failed, driver_op_t ddi_copyout\n");
2085                 return (EFAULT);
2086         }

2088         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

2090         return (0);
2091 }
```

*unchanged_portion_omitted_*

```
2169 /*
2170  * ql_adm_flash_load
2171  *      Performs qladm QL_FLASH_LOAD command
2172  *
2173  * Input:
2174  *      ha:     adapter state pointer.
2175  *      dop:    ql_adm_op_t structure pointer.
2176  *      mode:   flags.
2177  *
2178  * Returns:
2179  *
2180  * Context:
2181  *      Kernel context.
2182  */
2183 static int
2184 ql_adm_flash_load(ql_adapter_state_t *ha, ql_adm_op_t *dop, int mode)
2185 {
2186         uint8_t *dp;
2187         int     rval;

2189         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);
```



```
2191         dp = kmem_zalloc(dop->length, KM_SLEEP);
2224         if ((dp = kmem_zalloc(dop->length, KM_SLEEP)) == NULL) {
2225                 EL(ha, "failed, kmem_zalloc\n");
2226                 return (ENOMEM);
2227         }

2193         if (ddi_copyin((void *)(uintptr_t)dop->buffer, dp, dop->length,
2194             mode) != 0) {
2195                 EL(ha, "ddi_copyin failed\n");
2196                 kmem_free(dp, dop->length);
2197                 return (EFAULT);
2198         }

2200         if (ql_stall_driver(ha, 0) != QL_SUCCESS) {
2201                 EL(ha, "ql_stall_driver failed\n");
2202                 kmem_free(dp, dop->length);
2203                 return (EBUSY);
2204         }

2206         rval = (CFG_IST(ha, CFG_CTRL_24258081) ?
2207             ql_24xx_load_flash(ha, dp, dop->length, dop->option) :
2208             ql_load_flash(ha, dp, dop->length));

2210         ql_restart_driver(ha);

2212         kmem_free(dp, dop->length);

2214         if (rval != QL_SUCCESS) {
2215                 EL(ha, "failed\n");
2216                 return (EIO);
2217         }

2219         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

2221         return (0);
2222 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  112730 Thu Oct 23 11:04:55 2014
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_mbx.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1251 /*
1252  * ql_task_mgmt_iocb
1253  *      Function issues task management IOCB.
1254  *
1255  * Input:
1256  *      ha:     adapter state pointer.
1257  *      tq:     target queue pointer.
1258  *      lun:    LUN.
1259  *      flags:  control flags.
1260  *      delay:  seconds.
1261  *
1262  * Returns:
1263  *      ql local function return status code.
1264  *
1265  * Context:
1266  *      Kernel context
1267  */
1268 static int
1269 ql_task_mgmt_iocb(ql_adapter_state_t *ha, ql_tgt_t *tq, uint16_t lun,
1270     uint32_t flags, uint16_t delay)
1271 {
1272         ql_mbx_iocb_t   *pkt;
1273         int             rval;
1274         uint32_t        pkt_size;

1276         QL_PRINT_3(CE_CONT, "(%d): started\n", ha->instance);

1278         pkt_size = sizeof (ql_mbx_iocb_t);
1279         pkt = kmem_zalloc(pkt_size, KM_SLEEP);
1280         if (pkt == NULL) {
1281                 EL(ha, "failed, kmem_zalloc\n");
1282                 return (QL_MEMORY_ALLOC_FAILED);
1283         }

1281         pkt->mgmt.entry_type = TASK_MGMT_TYPE;
1282         pkt->mgmt.entry_count = 1;

1284         pkt->mgmt.n_port_hdl = (uint16_t)LE_16(tq->loop_id);
1285         pkt->mgmt.delay = (uint16_t)LE_16(delay);
1286         pkt->mgmt.timeout = LE_16(MAILBOX_TOV);
1287         pkt->mgmt.fcp_lun[2] = LSB(lun);
1288         pkt->mgmt.fcp_lun[3] = MSB(lun);
1289         pkt->mgmt.control_flags = LE_32(flags);
1290         pkt->mgmt.target_id[0] = tq->d_id.b.al_pa;
1291         pkt->mgmt.target_id[1] = tq->d_id.b.area;
1292         pkt->mgmt.target_id[2] = tq->d_id.b.domain;
1293         pkt->mgmt.vp_index = ha->vp_index;

1295         rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, pkt_size);
1296         if (rval == QL_SUCCESS && (pkt->sts24.entry_status & 0x3c) != 0) {
1297                 EL(ha, "failed, entry_status=%xh, d_id=%xh\n",
1298                     pkt->sts24.entry_status, tq->d_id.b24);
1299                 rval = QL_FUNCTION_PARAMETER_ERROR;
1300         }

1302         LITTLE_ENDIAN_16(&pkt->sts24.comp_status);

1304         if (rval == QL_SUCCESS && pkt->sts24.comp_status != CS_COMPLETE) {
```



```
1305                 EL(ha, "failed, comp_status=%xh, d_id=%xh\n",
1306                     pkt->sts24.comp_status, tq->d_id.b24);
1307                 rval = QL_FUNCTION_FAILED;
1308         }

1310         kmem_free(pkt, pkt_size);

1312         if (rval != QL_SUCCESS) {
1313                 EL(ha, "failed, rval = %xh\n", rval);
1314         } else {
1315                 /*EMPTY*/
1316                 QL_PRINT_3(CE_CONT, "(%d): done\n", ha->instance);
1317         }

1319         return (rval);
1320 }
______unchanged_portion_omitted_

1617 /*
1618  * ql_log_iocb
1619  *      Function issues login/logout IOCB.
1620  *
1621  * Input:
1622  *      ha:             adapter state pointer.
1623  *      tq:             target queue pointer.
1624  *      loop_id:        FC Loop ID.
1625  *      flags:          control flags.
1626  *      mr:             pointer for mailbox data.
1627  *
1628  * Returns:
1629  *      ql local function return status code.
1630  *
1631  * Context:
1632  *      Kernel context.
1633  */
1634 int
1635 ql_log_iocb(ql_adapter_state_t *ha, ql_tgt_t *tq, uint16_t loop_id,
1636     uint16_t flags, ql_mbx_data_t *mr)
1637 {
1638         ql_mbx_iocb_t   *pkt;
1639         int             rval;
1640         uint32_t        pkt_size;

1642         QL_PRINT_3(CE_CONT, "(%d): started\n", ha->instance);

1644         pkt_size = sizeof (ql_mbx_iocb_t);
1645         pkt = kmem_zalloc(pkt_size, KM_SLEEP);
1650         if (pkt == NULL) {
1651                 EL(ha, "failed, kmem_zalloc\n");
1652                 return (QL_MEMORY_ALLOC_FAILED);
1653         }

1647         pkt->log.entry_type = LOG_TYPE;
1648         pkt->log.entry_count = 1;
1649         pkt->log.n_port_hdl = (uint16_t)LE_16(loop_id);
1650         pkt->log.control_flags = (uint16_t)LE_16(flags);
1651         pkt->log.port_id[0] = tq->d_id.b.al_pa;
1652         pkt->log.port_id[1] = tq->d_id.b.area;
1653         pkt->log.port_id[2] = tq->d_id.b.domain;
1654         pkt->log.vp_index = ha->vp_index;

1656         rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, pkt_size);
1657         if (rval == QL_SUCCESS && (pkt->log.entry_status & 0x3c) != 0) {
1658                 EL(ha, "failed, entry_status=%xh, d_id=%xh\n",
1659                     pkt->log.entry_status, tq->d_id.b24);
1660                 rval = QL_FUNCTION_PARAMETER_ERROR;
```

```
1661         }

1663         if (rval == QL_SUCCESS) {
1664                 if (pkt->log.rsp_size == 0xB) {
1665                         LITTLE_ENDIAN_32(&pkt->log.io_param[5]);
1666                         tq->cmn_features = MSW(pkt->log.io_param[5]);
1667                         LITTLE_ENDIAN_32(&pkt->log.io_param[6]);
1668                         tq->conc_sequences = MSW(pkt->log.io_param[6]);
1669                         tq->relative_offset = LSW(pkt->log.io_param[6]);
1670                         LITTLE_ENDIAN_32(&pkt->log.io_param[9]);
1671                         tq->class3_recipient_ctl = MSW(pkt->log.io_param[9]);
1672                         tq->class3_conc_sequences = LSW(pkt->log.io_param[9]);
1673                         LITTLE_ENDIAN_32(&pkt->log.io_param[10]);
1674                         tq->class3_open_sequences_per_exch =
1675                             MSW(pkt->log.io_param[10]);
1676                         tq->prli_payload_length = 0x14;
1677                 }
1678                 if (mr != NULL) {
1679                         LITTLE_ENDIAN_16(&pkt->log.status);
1680                         LITTLE_ENDIAN_32(&pkt->log.io_param[0]);
1681                         LITTLE_ENDIAN_32(&pkt->log.io_param[1]);

1683                         if (pkt->log.status != CS_COMPLETE) {
1684                                 EL(ha, "failed, status=%xh, iop0=%xh, iop1="
1685                                     "%xh\n", pkt->log.status,
1686                                     pkt->log.io_param[0],
1687                                     pkt->log.io_param[1]);

1689                                 switch (pkt->log.io_param[0]) {
1690                                 case CS0_NO_LINK:
1691                                 case CS0_FIRMWARE_NOT_READY:
1692                                         mr->mb[0] = MBS_COMMAND_ERROR;
1693                                         mr->mb[1] = 1;
1694                                         break;
1695                                 case CS0_NO_IOCB:
1696                                 case CS0_NO_PCB_ALLOCATED:
1697                                         mr->mb[0] = MBS_COMMAND_ERROR;
1698                                         mr->mb[1] = 2;
1699                                         break;
1700                                 case CS0_NO_EXCH_CTRL_BLK:
1701                                         mr->mb[0] = MBS_COMMAND_ERROR;
1702                                         mr->mb[1] = 3;
1703                                         break;
1704                                 case CS0_COMMAND_FAILED:
1705                                         mr->mb[0] = MBS_COMMAND_ERROR;
1706                                         mr->mb[1] = 4;
1707                                         switch (LSB(pkt->log.io_param[1])) {
1708                                         case CS1_PLOGI_RESPONSE_FAILED:
1709                                                 mr->mb[2] = 3;
1710                                                 break;
1711                                         case CS1_PRLI_FAILED:
1712                                                 mr->mb[2] = 4;
1713                                                 break;
1714                                         case CS1_PRLI_RESPONSE_FAILED:
1715                                                 mr->mb[2] = 5;
1716                                                 break;
1717                                         case CS1_COMMAND_LOGGED_OUT:
1718                                                 mr->mb[2] = 7;
1719                                                 break;
1720                                         case CS1_PLOGI_FAILED:
1721                                         default:
1722                                                 EL(ha, "log iop1 = %xh\n",
1723                                                     LSB(pkt->log.io_param[1]))
1724                                                 mr->mb[2] = 2;
1725                                                 break;
1726                                         }
```



```
1727                                         break;
1728                                 case CS0_PORT_NOT_LOGGED_IN:
1729                                         mr->mb[0] = MBS_COMMAND_ERROR;
1730                                         mr->mb[1] = 4;
1731                                         mr->mb[2] = 7;
1732                                         break;
1733                                 case CS0_NO_FLOGI_ACC:
1734                                 case CS0_NO_FABRIC_PRESENT:
1735                                         mr->mb[0] = MBS_COMMAND_ERROR;
1736                                         mr->mb[1] = 5;
1737                                         break;
1738                                 case CS0_ELS_REJECT_RECEIVED:
1739                                         mr->mb[0] = MBS_COMMAND_ERROR;
1740                                         mr->mb[1] = 0xd;
1741                                         break;
1742                                 case CS0_PORT_ID_USED:
1743                                         mr->mb[0] = MBS_PORT_ID_USED;
1744                                         mr->mb[1] = LSW(pkt->log.io_param[1]);
1745                                         break;
1746                                 case CS0_N_PORT_HANDLE_USED:
1747                                         mr->mb[0] = MBS_LOOP_ID_USED;
1748                                         mr->mb[1] = MSW(pkt->log.io_param[1]);
1749                                         mr->mb[2] = LSW(pkt->log.io_param[1]);
1750                                         break;
1751                                 case CS0_NO_N_PORT_HANDLE_AVAILABLE:
1752                                         mr->mb[0] = MBS_ALL_IDS_IN_USE;
1753                                         break;
1754                                 case CS0_CMD_PARAMETER_ERROR:
1755                                 default:
1756                                         EL(ha, "pkt->log iop[0]=%xh\n",
1757                                             pkt->log.io_param[0]);
1758                                         mr->mb[0] =
1759                                             MBS_COMMAND_PARAMETER_ERROR;
1760                                         break;
1761                                 }
1762                         } else {
1763                                 QL_PRINT_3(CE_CONT, "(%d): status=%xh\n",
1764                                     ha->instance, pkt->log.status);

1766                                 mr->mb[0] = MBS_COMMAND_COMPLETE;
1767                                 mr->mb[1] = (uint16_t)
1768                                     (pkt->log.io_param[0] & BIT_4 ? 0 : BIT_0);
1769                                 if (pkt->log.io_param[0] & BIT_8) {
1770                                         mr->mb[1] = (uint16_t)
1771                                             (mr->mb[1] | BIT_1);
1772                                 }
1773                         }
1774                         rval = mr->mb[0];
1775                 }

1777         }

1779         kmem_free(pkt, pkt_size);

1781         if (rval != QL_SUCCESS) {
1782                 EL(ha, "failed=%xh, d_id=%xh\n", rval, tq->d_id.b24);
1783         } else {
1784                 /*EMPTY*/
1785                 QL_PRINT_3(CE_CONT, "(%d): done\n", ha->instance);
1786         }

1788         return (rval);
1789 }
```

_____***unchanged_portion_omitted_***

```
2664 /*
```

```
2665  * ql_abort_cmd_iocb
2666  *      Function issues abort command IOCB.
2667  *
2668  * Input:
2669  *      ha:     adapter state pointer.
2670  *      sp:     SRB structure pointer.
2671  *
2672  * Returns:
2673  *      ql local function return status code.
2674  *
2675  * Context:
2676  *      Interrupt or Kernel context, no mailbox commands allowed.
2677  */
2678 static int
2679 ql_abort_cmd_iocb(ql_adapter_state_t *ha, ql_srb_t *sp)
2680 {
2681         ql_mbx_iocb_t   *pkt;
2682         int             rval;
2683         uint32_t        pkt_size;
2684         uint16_t        comp_status;
2685         ql_tgt_t        *tq = sp->lun_queue->target_queue;

2687         QL_PRINT_3(CE_CONT, "(%d): started\n", ha->instance);

2689         pkt_size = sizeof (ql_mbx_iocb_t);
2690         pkt = kmem_zalloc(pkt_size, KM_SLEEP);
2698         if ((pkt = kmem_zalloc(pkt_size, KM_SLEEP)) == NULL) {
2699                 EL(ha, "failed, kmem_zalloc\n");
2700                 return (QL_MEMORY_ALLOC_FAILED);
2701         }

2692         pkt->abo.entry_type = ABORT_CMD_TYPE;
2693         pkt->abo.entry_count = 1;
2694         pkt->abo.n_port_hdl = (uint16_t)LE_16(tq->loop_id);
2695         if (!CFG_IST(ha, CFG_CTRL_8021)) {
2696                 pkt->abo.options = AF_NO_ABTS;
2697         }
2698         pkt->abo.cmd_handle = LE_32(sp->handle);
2699         pkt->abo.target_id[0] = tq->d_id.b.al_pa;
2700         pkt->abo.target_id[1] = tq->d_id.b.area;
2701         pkt->abo.target_id[2] = tq->d_id.b.domain;
2702         pkt->abo.vp_index = ha->vp_index;

2704         rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, pkt_size);

2706         if (rval == QL_SUCCESS) {
2707                 if ((pkt->abo.entry_status  & 0x3c) != 0) {
2708                         EL(ha, "failed, entry_status=%xh, d_id=%xh\n",
2709                             pkt->abo.entry_status, tq->d_id.b24);
2710                         rval = QL_FUNCTION_PARAMETER_ERROR;
2711                 } else {
2712                         comp_status = (uint16_t)LE_16(pkt->abo.n_port_hdl);
2713                         if (comp_status != CS_COMPLETE) {
2714                                 EL(ha, "failed, comp_status=%xh, d_id=%xh\n",
2715                                     comp_status, tq->d_id.b24);
2716                                 rval = QL_FUNCTION_FAILED;
2717                         }
2718                 }
2719         }

2721         kmem_free(pkt, pkt_size);

2723         if (rval != QL_SUCCESS) {
2724                 EL(ha, "failed=%xh, d_id=%xh\n", rval, tq->d_id.b24);
2725         } else {
2726                 /*EMPTY*/
```



```
2727                 QL_PRINT_3(CE_CONT, "(%d): done\n", ha->instance);
2728         }

2730         return (rval);
2731 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   50767 Thu Oct 23 11:04:55 2014
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_nx.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1370 static int
1371 ql_8021_pinit_from_rom(ql_adapter_state_t *ha)
1372 {
1373         int                     init_delay = 0;
1374         struct crb_addr_pair    *buf;
1375         uint32_t                offset, off, i, n, addr, val;

1377         /* Grab the lock so that no one can read flash when we reset the chip */
1378         (void) ql_8021_rom_lock(ha);
1379         ql_8021_wr_32(ha, UNM_ROMUSB_GLB_SW_RESET, 0xffffffff);
1380         /* Just in case it was held when we reset the chip */
1381         ql_8021_rom_unlock(ha);

1383         if (ql_8021_rom_fast_read(ha, 0, &n) != 0 || n != 0xcafecafe ||
1384             ql_8021_rom_fast_read(ha, 4, &n) != 0) {
1385                 EL(ha, "ERROR Reading crb_init area: n: %08x\n", n);
1386                 return (-1);
1387         }
1388         offset = n & 0xffff;
1389         n = (n >> 16) & 0xffff;
1390         if (n >= 1024) {
1391                 EL(ha, "n=0x%x Error! NetXen card flash not initialized\n", n);
1392                 return (-1);
1393         }

1395         buf = kmem_zalloc(n * sizeof (struct crb_addr_pair), KM_SLEEP);
1396         if (buf == NULL) {
1397                 EL(ha, "Unable to zalloc memory\n");
1398                 return (-1);
1399         }

1397         for (i = 0; i < n; i++) {
1398                 if (ql_8021_rom_fast_read(ha, 8 * i + 4 * offset, &val) != 0 ||
1399                     ql_8021_rom_fast_read(ha, 8 * i + 4 * offset + 4, &addr) !=
1400                     0) {
1401                         kmem_free(buf, n * sizeof (struct crb_addr_pair));
1402                         EL(ha, "ql_8021_rom_fast_read != 0 to zalloc memory\n");
1403                         return (-1);
1404                 }

1406                 buf[i].addr = addr;
1407                 buf[i].data = val;
1408         }

1410         for (i = 0; i < n; i++) {
1411                 off = ql_8021_decode_crb_addr(ha, buf[i].addr);
1412                 if (off == ADDR_ERROR) {
1413                         EL(ha, "Err: Unknown addr: 0x%lx\n", buf[i].addr);
1414                         continue;
1415                 }
1416                 off += UNM_PCI_CRBSPACE;

1418                 if (off & 1) {
1419                         continue;
1420                 }

1422                 /* skipping cold reboot MAGIC */
1423                 if (off == UNM_RAM_COLD_BOOT) {
```



```
1424                         continue;
1425                 }
1426                 if (off == (UNM_CRB_I2C0 + 0x1c)) {
1427                         continue;
1428                 }
1429                 /* do not reset PCI */
1430                 if (off == (ROMUSB_GLB + 0xbc)) {
1431                         continue;
1432                 }
1433                 if (off == (ROMUSB_GLB + 0xa8)) {
1434                         continue;
1435                 }
1436                 if (off == (ROMUSB_GLB + 0xc8)) {       /* core clock */
1437                         continue;
1438                 }
1439                 if (off == (ROMUSB_GLB + 0x24)) {       /* MN clock */
1440                         continue;
1441                 }
1442                 if (off == (ROMUSB_GLB + 0x1c)) {       /* MS clock */
1443                         continue;
1444                 }
1445                 if ((off & 0x0ff00000) == UNM_CRB_DDR_NET) {
1446                         continue;
1447                 }
1448                 if (off == (UNM_CRB_PEG_NET_1 + 0x18) &&
1449                     !NX_IS_REVISION_P3PLUS(ha->rev_id)) {
1450                         buf[i].data = 0x1020;
1451                 }
1452                 /* skip the function enable register */
1453                 if (off == UNM_PCIE_REG(PCIE_SETUP_FUNCTION)) {
1454                         continue;
1455                 }
1456                 if (off == UNM_PCIE_REG(PCIE_SETUP_FUNCTION2)) {
1457                         continue;
1458                 }
1459                 if ((off & 0x0ff00000) == UNM_CRB_SMB) {
1460                         continue;
1461                 }

1463                 /* After writing this register, HW needs time for CRB */
1464                 /* to quiet down (else crb_window returns 0xffffffff) */
1465                 init_delay = 1;
1466                 if (off == UNM_ROMUSB_GLB_SW_RESET) {
1467                         init_delay = 100;       /* Sleep 1000 msecs */
1468                 }

1470                 ql_8021_wr_32(ha, off, buf[i].data);

1472                 delay(init_delay);
1473         }
1474         kmem_free(buf, n * sizeof (struct crb_addr_pair));

1476         /* disable_peg_cache_all */

1478         /* p2dn replyCount */
1479         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_D + 0xec, 0x1e);
1480         /* disable_peg_cache 0 */
1481         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_D + 0x4c, 8);
1482         /* disable_peg_cache 1 */
1483         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_I + 0x4c, 8);

1485         /* peg_clr_all */
1486         /* peg_clr 0 */
1487         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_0 + 0x8, 0);
1488         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_0 + 0xc, 0);
1489         /* peg_clr 1 */
```

```
1490         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_1 + 0x8, 0);
1491         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_1 + 0xc, 0);
1492         /* peg_clr 2 */
1493         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_2 + 0x8, 0);
1494         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_2 + 0xc, 0);
1495         /* peg_clr 3 */
1496         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_3 + 0x8, 0);
1497         ql_8021_wr_32(ha, UNM_CRB_PEG_NET_3 + 0xc, 0);

1499         return (0);
1500 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  235829 Thu Oct 23 11:04:55 2014
new/usr/src/uts/common/io/fibre-channel/fca/qlc/ql_xioctl.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
/*
 * 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 QLogic Corporation */

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

#pragma ident   "Copyright 2010 QLogic Corporation; ql_xioctl.c"

/*
 * ISP2xxx Solaris Fibre Channel Adapter (FCA) driver source file.
 *
 * ***********************************************************************
 * *                                                                     **
 * *                               NOTICE                                **
 * *            COPYRIGHT (C) 1996-2010 QLOGIC CORPORATION               **
 * *                    ALL RIGHTS RESERVED                              **
 * *                                                                     **
 * ***********************************************************************
 *
 */

#include <ql_apps.h>
#include <ql_api.h>
#include <ql_debug.h>
#include <ql_init.h>
#include <ql_iocb.h>
#include <ql_ioctl.h>
#include <ql_mbx.h>
#include <ql_xioctl.h>

/*
 * Local data
 */

/*
```



```
 * Local prototypes
 */
static int ql_sdm_ioctl(ql_adapter_state_t *, int, void *, int);
static int ql_sdm_setup(ql_adapter_state_t *, EXT_IOCTL **, void *, int,
    boolean_t (*)(EXT_IOCTL *));
static boolean_t ql_validate_signature(EXT_IOCTL *);
static int ql_sdm_return(ql_adapter_state_t *, EXT_IOCTL *, void *, int);
static void ql_query(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_hba_node(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_hba_port(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_disc_port(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_disc_tgt(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_fw(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_chip(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_driver(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_fcct(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_aen_reg(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_aen_get(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_scsi_passthru(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_wwpn_to_scsiaddr(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_host_idx(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_host_drvname(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_read_nvram(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_write_nvram(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_read_flash(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_write_flash(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_write_vpd(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_read_vpd(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_diagnostic_loopback(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_send_els_rnid(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_set_host_data(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_host_data(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_qry_cna_port(ql_adapter_state_t *, EXT_IOCTL *, int);

static int ql_lun_count(ql_adapter_state_t *, ql_tgt_t *);
static int ql_report_lun(ql_adapter_state_t *, ql_tgt_t *);
static int ql_inq_scan(ql_adapter_state_t *, ql_tgt_t *, int);
static int ql_inq(ql_adapter_state_t *, ql_tgt_t *, int, ql_mbx_iocb_t *,
    uint8_t);
static uint32_t ql_get_buffer_data(caddr_t, caddr_t, uint32_t, int);
static uint32_t ql_send_buffer_data(caddr_t, caddr_t, uint32_t, int);
static int ql_24xx_flash_desc(ql_adapter_state_t *);
static int ql_setup_flash(ql_adapter_state_t *);
static ql_tgt_t *ql_find_port(ql_adapter_state_t *, uint8_t *, uint16_t);
static int ql_flash_fcode_load(ql_adapter_state_t *, void *, uint32_t, int);
static int ql_flash_fcode_dump(ql_adapter_state_t *, void *, uint32_t,
    uint32_t, int);
static int ql_program_flash_address(ql_adapter_state_t *, uint32_t,
    uint8_t);
static void ql_set_rnid_parameters(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_rnid_parameters(ql_adapter_state_t *, EXT_IOCTL *, int);
static int ql_reset_statistics(ql_adapter_state_t *, EXT_IOCTL *);
static void ql_get_statistics(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_statistics_fc(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_statistics_fc4(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_set_led_state(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_led_state(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_drive_led(ql_adapter_state_t *, uint32_t);
static uint32_t ql_setup_led(ql_adapter_state_t *);
static uint32_t ql_wrapup_led(ql_adapter_state_t *);
static void ql_get_port_summary(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_target_id(ql_adapter_state_t *, EXT_IOCTL *, int);
static void ql_get_sfp(ql_adapter_state_t *, EXT_IOCTL *, int);
static int ql_dump_sfp(ql_adapter_state_t *, void *, int);
static ql_fcache_t *ql_setup_fnode(ql_adapter_state_t *);
static void ql_get_fcache(ql_adapter_state_t *, EXT_IOCTL *, int);
```

```
127 static void ql_get_fcache_ex(ql_adapter_state_t *, EXT_IOCTL *, int);
128 void ql_update_fcache(ql_adapter_state_t *, uint8_t *, uint32_t);
129 static int ql_check_pci(ql_adapter_state_t *, ql_fcache_t *, uint32_t *);
130 static void ql_flash_layout_table(ql_adapter_state_t *, uint32_t);
131 static void ql_process_flt(ql_adapter_state_t *, uint32_t);
132 static void ql_flash_nvram_defaults(ql_adapter_state_t *);
133 static void ql_port_param(ql_adapter_state_t *, EXT_IOCTL *, int);
134 static int ql_check_pci(ql_adapter_state_t *, ql_fcache_t *, uint32_t *);
135 static void ql_get_pci_data(ql_adapter_state_t *, EXT_IOCTL *, int);
136 static void ql_get_fwfcetrace(ql_adapter_state_t *, EXT_IOCTL *, int);
137 static void ql_get_fwexttrace(ql_adapter_state_t *, EXT_IOCTL *, int);
138 static void ql_menlo_reset(ql_adapter_state_t *, EXT_IOCTL *, int);
139 static void ql_menlo_get_fw_version(ql_adapter_state_t *, EXT_IOCTL *, int);
140 static void ql_menlo_update_fw(ql_adapter_state_t *, EXT_IOCTL *, int);
141 static void ql_menlo_manage_info(ql_adapter_state_t *, EXT_IOCTL *, int);
142 static int ql_suspend_hba(ql_adapter_state_t *, uint32_t);
143 static void ql_restart_hba(ql_adapter_state_t *);
144 static void ql_get_vp_cnt_id(ql_adapter_state_t *, EXT_IOCTL *, int);
145 static void ql_vp_ioctl(ql_adapter_state_t *, EXT_IOCTL *, int);
146 static void ql_qry_vport(ql_adapter_state_t *, EXT_IOCTL *, int);
147 static void ql_access_flash(ql_adapter_state_t *, EXT_IOCTL *, int);
148 static void ql_reset_cmd(ql_adapter_state_t *, EXT_IOCTL *);
149 static void ql_update_flash_caches(ql_adapter_state_t *);
150 static void ql_get_dcbx_parameters(ql_adapter_state_t *, EXT_IOCTL *, int);
151 static void ql_get_xgmac_statistics(ql_adapter_state_t *, EXT_IOCTL *, int);
152 static void ql_get_fcf_list(ql_adapter_state_t *, EXT_IOCTL *, int);
153 static void ql_get_resource_counts(ql_adapter_state_t *, EXT_IOCTL *, int);
154 static void ql_qry_adapter_versions(ql_adapter_state_t *, EXT_IOCTL *, int);
155 static int ql_set_loop_point(ql_adapter_state_t *, uint16_t);

157 /* ********************************************************************** */
158 /*                      External IOCTL support.                           */
159 /* ********************************************************************** */

161 /*
162  * ql_alloc_xioctl_resource
163  *      Allocates resources needed by module code.
164  *
165  * Input:
166  *      ha:             adapter state pointer.
167  *
168  * Returns:
169  *      SYS_ERRNO
170  *
171  * Context:
172  *      Kernel context.
173  */
174 int
175 ql_alloc_xioctl_resource(ql_adapter_state_t *ha)
176 {
177         ql_xioctl_t     *xp;

179         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

181         if (ha->xioctl != NULL) {
182                 QL_PRINT_9(CE_CONT, "(%d): already allocated done\n",
183                     ha->instance);
184                 return (0);
185         }

187         xp = kmem_zalloc(sizeof (ql_xioctl_t), KM_SLEEP);
188         if (xp == NULL) {
189                 EL(ha, "failed, kmem_zalloc\n");
190                 return (ENOMEM);
191         }
188         ha->xioctl = xp;
```



```
190         /* Allocate AEN tracking buffer */
191         xp->aen_tracking_queue = kmem_zalloc(EXT_DEF_MAX_AEN_QUEUE *
192             sizeof (EXT_ASYNC_EVENT), KM_SLEEP);
197         if (xp->aen_tracking_queue == NULL) {
198                 EL(ha, "failed, kmem_zalloc-2\n");
199                 ql_free_xioctl_resource(ha);
200                 return (ENOMEM);
201         }

194         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

196         return (0);
197 }
```

_____unchanged_portion_omitted_

```
1438 /*
1439  * ql_fcct
1440  *      IOCTL management server FC-CT passthrough.
1441  *
1442  * Input:
1443  *      ha:     adapter state pointer.
1444  *      cmd:    User space CT arguments pointer.
1445  *      mode:   flags.
1446  *
1447  * Returns:
1448  *      None, request status indicated in cmd->Status.
1449  *
1450  * Context:
1451  *      Kernel context.
1452  */
1453 static void
1454 ql_fcct(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
1455 {
1456         ql_mbx_iocb_t           *pkt;
1457         ql_mbx_data_t           mr;
1458         dma_mem_t               *dma_mem;
1459         caddr_t                 pld;
1460         uint32_t                pkt_size, pld_byte_cnt, *long_ptr;
1461         int                     rval;
1462         ql_ct_iu_preamble_t     *ct;
1463         ql_xioctl_t             *xp = ha->xioctl;
1464         ql_tgt_t                tq;
1465         uint16_t                comp_status, loop_id;

1467         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

1469         /* Get CT argument structure. */
1470         if ((ha->topology & QL_SNS_CONNECTION) == 0) {
1471                 EL(ha, "failed, No switch\n");
1472                 cmd->Status = EXT_STATUS_DEV_NOT_FOUND;
1473                 cmd->ResponseLen = 0;
1474                 return;
1475         }

1477         if (DRIVER_SUSPENDED(ha)) {
1478                 EL(ha, "failed, LOOP_NOT_READY\n");
1479                 cmd->Status = EXT_STATUS_BUSY;
1480                 cmd->ResponseLen = 0;
1481                 return;
1482         }

1484         /* Login management server device. */
1485         if ((xp->flags & QL_MGMT_SERVER_LOGIN) == 0) {
1486                 tq.d_id.b.al_pa = 0xfa;
1487                 tq.d_id.b.area = 0xff;
```

```
1488                tq.d_id.b.domain = 0xff;
1489                tq.loop_id = (uint16_t)(CFG_IST(ha, CFG_CTRL_24258081) ?
1490                    MANAGEMENT_SERVER_24XX_LOOP_ID :
1491                    MANAGEMENT_SERVER_LOOP_ID);
1492                rval = ql_login_fport(ha, &tq, tq.loop_id, LFF_NO_PRLI, &mr);
1493                if (rval != QL_SUCCESS) {
1494                        EL(ha, "failed, server login\n");
1495                        cmd->Status = EXT_STATUS_DEV_NOT_FOUND;
1496                        cmd->ResponseLen = 0;
1497                        return;
1498                } else {
1499                        xp->flags |= QL_MGMT_SERVER_LOGIN;
1500                }
1501        }

1503        QL_PRINT_9(CE_CONT, "(%d): cmd\n", ha->instance);
1504        QL_DUMP_9(cmd, 8, sizeof (EXT_IOCTL));

1506        /* Allocate a DMA Memory Descriptor */
1507        dma_mem = (dma_mem_t *)kmem_zalloc(sizeof (dma_mem_t), KM_SLEEP);
1508        if (dma_mem == NULL) {
1509                EL(ha, "failed, kmem_zalloc\n");
1510                cmd->Status = EXT_STATUS_NO_MEMORY;
1511                cmd->ResponseLen = 0;
1512                return;
1513        }
1514        /* Determine maximum buffer size. */
1515        if (cmd->RequestLen < cmd->ResponseLen) {
1516                pld_byte_cnt = cmd->ResponseLen;
1517        } else {
1518                pld_byte_cnt = cmd->RequestLen;
1519        }

1521        /* Allocate command block. */
1522        pkt_size = (uint32_t)(sizeof (ql_mbx_iocb_t) + pld_byte_cnt);
1523        pkt = kmem_zalloc(pkt_size, KM_SLEEP);
1533        if (pkt == NULL) {
1534                EL(ha, "failed, kmem_zalloc\n");
1535                cmd->Status = EXT_STATUS_NO_MEMORY;
1536                cmd->ResponseLen = 0;
1537                return;
1538        }
1524        pld = (caddr_t)pkt + sizeof (ql_mbx_iocb_t);

1526        /* Get command payload data. */
1527        if (ql_get_buffer_data((caddr_t)(uintptr_t)cmd->RequestAdr, pld,
1528            cmd->RequestLen, mode) != cmd->RequestLen) {
1529                EL(ha, "failed, get_buffer_data\n");
1530                kmem_free(pkt, pkt_size);
1531                cmd->Status = EXT_STATUS_COPY_ERR;
1532                cmd->ResponseLen = 0;
1533                return;
1534        }

1536        /* Get DMA memory for the IOCB */
1537        if (ql_get_dma_mem(ha, dma_mem, pkt_size, LITTLE_ENDIAN_DMA,
1538            QL_DMA_RING_ALIGN) != QL_SUCCESS) {
1539                cmn_err(CE_WARN, "%s(%d): DMA memory "
1540                    "alloc failed", QL_NAME, ha->instance);
1541                kmem_free(pkt, pkt_size);
1542                kmem_free(dma_mem, sizeof (dma_mem_t));
1543                cmd->Status = EXT_STATUS_MS_NO_RESPONSE;
1544                cmd->ResponseLen = 0;
1545                return;
1546        }
```



```
1548        /* Copy out going payload data to IOCB DMA buffer. */
1549        ddi_rep_put8(dma_mem->acc_handle, (uint8_t *)pld,
1550            (uint8_t *)dma_mem->bp, pld_byte_cnt, DDI_DEV_AUTOINCR);

1552        /* Sync IOCB DMA buffer. */
1553        (void) ddi_dma_sync(dma_mem->dma_handle, 0, pld_byte_cnt,
1554            DDI_DMA_SYNC_FORDEV);

1556        /*
1557         * Setup IOCB
1558         */
1559        ct = (ql_ct_iu_preamble_t *)pld;
1560        if (CFG_IST(ha, CFG_CTRL_24258081)) {
1561                pkt->ms24.entry_type = CT_PASSTHRU_TYPE;
1562                pkt->ms24.entry_count = 1;

1564                pkt->ms24.vp_index = ha->vp_index;

1566                /* Set loop ID */
1567                pkt->ms24.n_port_hdl = (uint16_t)
1568                    (ct->gs_type == GS_TYPE_DIR_SERVER ?
1569                    LE_16(SNS_24XX_HDL) :
1570                    LE_16(MANAGEMENT_SERVER_24XX_LOOP_ID));

1572                /* Set ISP command timeout. */
1573                pkt->ms24.timeout = LE_16(120);

1575                /* Set cmd/response data segment counts. */
1576                pkt->ms24.cmd_dseg_count = LE_16(1);
1577                pkt->ms24.resp_dseg_count = LE_16(1);

1579                /* Load ct cmd byte count. */
1580                pkt->ms24.cmd_byte_count = LE_32(cmd->RequestLen);

1582                /* Load ct rsp byte count. */
1583                pkt->ms24.resp_byte_count = LE_32(cmd->ResponseLen);

1585                long_ptr = (uint32_t *)&pkt->ms24.dseg_0_address;

1587                /* Load MS command entry data segments. */
1588                *long_ptr++ = (uint32_t)
1589                    LE_32(LSD(dma_mem->cookie.dmac_laddress));
1590                *long_ptr++ = (uint32_t)
1591                    LE_32(MSD(dma_mem->cookie.dmac_laddress));
1592                *long_ptr++ = (uint32_t)(LE_32(cmd->RequestLen));

1594                /* Load MS response entry data segments. */
1595                *long_ptr++ = (uint32_t)
1596                    LE_32(LSD(dma_mem->cookie.dmac_laddress));
1597                *long_ptr++ = (uint32_t)
1598                    LE_32(MSD(dma_mem->cookie.dmac_laddress));
1599                *long_ptr = (uint32_t)LE_32(cmd->ResponseLen);

1601                rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt,
1602                    sizeof (ql_mbx_iocb_t));

1604                comp_status = (uint16_t)LE_16(pkt->sts24.comp_status);
1605                if (comp_status == CS_DATA_UNDERRUN) {
1606                        if ((BE_16(ct->max_residual_size)) == 0) {
1607                                comp_status = CS_COMPLETE;
1608                        }
1609                }

1611                if (rval != QL_SUCCESS || (pkt->sts24.entry_status & 0x3c) !=
1612                    0) {
1613                        EL(ha, "failed, I/O timeout or "
```

```
1614                            "es=%xh, ss_l=%xh, rval=%xh\n",
1615                            pkt->sts24.entry_status,
1616                            pkt->sts24.scsi_status_l, rval);
1617                    kmem_free(pkt, pkt_size);
1618                    ql_free_dma_resource(ha, dma_mem);
1619                    kmem_free(dma_mem, sizeof (dma_mem_t));
1620                    cmd->Status = EXT_STATUS_MS_NO_RESPONSE;
1621                    cmd->ResponseLen = 0;
1622                    return;
1623            }
1624    } else {
1625            pkt->ms.entry_type = MS_TYPE;
1626            pkt->ms.entry_count = 1;

1628            /* Set loop ID */
1629            loop_id = (uint16_t)(ct->gs_type == GS_TYPE_DIR_SERVER ?
1630                SIMPLE_NAME_SERVER_LOOP_ID : MANAGEMENT_SERVER_LOOP_ID);
1631            if (CFG_IST(ha, CFG_EXT_FW_INTERFACE)) {
1632                    pkt->ms.loop_id_l = LSB(loop_id);
1633                    pkt->ms.loop_id_h = MSB(loop_id);
1634            } else {
1635                    pkt->ms.loop_id_h = LSB(loop_id);
1636            }

1638            /* Set ISP command timeout. */
1639            pkt->ms.timeout = LE_16(120);

1641            /* Set data segment counts. */
1642            pkt->ms.cmd_dseg_count_l = 1;
1643            pkt->ms.total_dseg_count = LE_16(2);

1645            /* Response total byte count. */
1646            pkt->ms.resp_byte_count = LE_32(cmd->ResponseLen);
1647            pkt->ms.dseg_1_length = LE_32(cmd->ResponseLen);

1649            /* Command total byte count. */
1650            pkt->ms.cmd_byte_count = LE_32(cmd->RequestLen);
1651            pkt->ms.dseg_0_length = LE_32(cmd->RequestLen);

1653            /* Load command/response data segments. */
1654            pkt->ms.dseg_0_address[0] = (uint32_t)
1655                LE_32(LSD(dma_mem->cookie.dmac_laddress));
1656            pkt->ms.dseg_0_address[1] = (uint32_t)
1657                LE_32(MSD(dma_mem->cookie.dmac_laddress));
1658            pkt->ms.dseg_1_address[0] = (uint32_t)
1659                LE_32(LSD(dma_mem->cookie.dmac_laddress));
1660            pkt->ms.dseg_1_address[1] = (uint32_t)
1661                LE_32(MSD(dma_mem->cookie.dmac_laddress));

1663            rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt,
1664                sizeof (ql_mbx_iocb_t));

1666            comp_status = (uint16_t)LE_16(pkt->sts.comp_status);
1667            if (comp_status == CS_DATA_UNDERRUN) {
1668                    if ((BE_16(ct->max_residual_size)) == 0) {
1669                            comp_status = CS_COMPLETE;
1670                    }
1671            }
1672            if (rval != QL_SUCCESS || (pkt->sts.entry_status & 0x7e) != 0) {
1673                    EL(ha, "failed, I/O timeout or "
1674                        "es=%xh, rval=%xh\n", pkt->sts.entry_status, rval);
1675                    kmem_free(pkt, pkt_size);
1676                    ql_free_dma_resource(ha, dma_mem);
1677                    kmem_free(dma_mem, sizeof (dma_mem_t));
1678                    cmd->Status = EXT_STATUS_MS_NO_RESPONSE;
1679                    cmd->ResponseLen = 0;
```



```
1680                            return;
1681                    }
1682            }

1684            /* Sync in coming DMA buffer. */
1685            (void) ddi_dma_sync(dma_mem->dma_handle, 0,
1686                pld_byte_cnt, DDI_DMA_SYNC_FORKERNEL);
1687            /* Copy in coming DMA data. */
1688            ddi_rep_get8(dma_mem->acc_handle, (uint8_t *)pld,
1689                (uint8_t *)dma_mem->bp, pld_byte_cnt,
1690                DDI_DEV_AUTOINCR);

1692            /* Copy response payload from DMA buffer to application. */
1693            if (cmd->ResponseLen != 0) {
1694                    QL_PRINT_9(CE_CONT, "(%d): ResponseLen=%d\n", ha->instance,
1695                        cmd->ResponseLen);
1696                    QL_DUMP_9(pld, 8, cmd->ResponseLen);

1698                    /* Send response payload. */
1699                    if (ql_send_buffer_data(pld,
1700                        (caddr_t)(uintptr_t)cmd->ResponseAdr,
1701                        cmd->ResponseLen, mode) != cmd->ResponseLen) {
1702                            EL(ha, "failed, send_buffer_data\n");
1703                            cmd->Status = EXT_STATUS_COPY_ERR;
1704                            cmd->ResponseLen = 0;
1705                    }
1706            }

1708            kmem_free(pkt, pkt_size);
1709            ql_free_dma_resource(ha, dma_mem);
1710            kmem_free(dma_mem, sizeof (dma_mem_t));

1712            QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
1713 }
_____unchanged_portion_omitted_

1940 /*
1941  * ql_scsi_passthru
1942  *      IOCTL SCSI passthrough.
1943  *
1944  * Input:
1945  *      ha:     adapter state pointer.
1946  *      cmd:    User space SCSI command pointer.
1947  *      mode:   flags.
1948  *
1949  * Returns:
1950  *      None, request status indicated in cmd->Status.
1951  *
1952  * Context:
1953  *      Kernel context.
1954  */
1955 static void
1956 ql_scsi_passthru(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
1957 {
1958         ql_mbx_iocb_t           *pkt;
1959         ql_mbx_data_t           mr;
1960         dma_mem_t               *dma_mem;
1961         caddr_t                 pld;
1962         uint32_t                pkt_size, pld_size;
1963         uint16_t                qlnt, retries, cnt, cnt2;
1964         uint8_t                 *name;
1965         EXT_FC_SCSI_PASSTHRU    *ufc_req;
1966         EXT_SCSI_PASSTHRU       *usp_req;
1967         int                     rval;
1968         union _passthru {
1969                 EXT_SCSI_PASSTHRU       sp_cmd;
```

```
1970                EXT_FC_SCSI_PASSTHRU    fc_cmd;
1971        } pt_req;               /* Passthru request */
1972        uint32_t                status, sense_sz = 0;
1973        ql_tgt_t                *tq = NULL;
1974        EXT_SCSI_PASSTHRU       *sp_req = &pt_req.sp_cmd;
1975        EXT_FC_SCSI_PASSTHRU    *fc_req = &pt_req.fc_cmd;

1977        /* SCSI request struct for SCSI passthrough IOs. */
1978        struct {
1979                uint16_t        lun;
1980                uint16_t        sense_length;   /* Sense buffer size */
1981                size_t          resid;          /* Residual */
1982                uint8_t         *cdbp;          /* Requestor's CDB */
1983                uint8_t         *u_sense;       /* Requestor's sense buffer */
1984                uint8_t         cdb_len;        /* Requestor's CDB length */
1985                uint8_t         direction;
1986        } scsi_req;

1988        struct {
1989                uint8_t         *rsp_info;
1990                uint8_t         *req_sense_data;
1991                uint32_t        residual_length;
1992                uint32_t        rsp_info_length;
1993                uint32_t        req_sense_length;
1994                uint16_t        comp_status;
1995                uint8_t         state_flags_l;
1996                uint8_t         state_flags_h;
1997                uint8_t         scsi_status_l;
1998                uint8_t         scsi_status_h;
1999        } sts;

2001        QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

2003        /* Verify Sub Code and set cnt to needed request size. */
2004        if (cmd->SubCode == EXT_SC_SEND_SCSI_PASSTHRU) {
2005                pld_size = sizeof (EXT_SCSI_PASSTHRU);
2006        } else if (cmd->SubCode == EXT_SC_SEND_FC_SCSI_PASSTHRU) {
2007                pld_size = sizeof (EXT_FC_SCSI_PASSTHRU);
2008        } else {
2009                EL(ha, "failed, invalid SubCode=%xh\n", cmd->SubCode);
2010                cmd->Status = EXT_STATUS_UNSUPPORTED_SUBCODE;
2011                cmd->ResponseLen = 0;
2012                return;
2013        }

2015        dma_mem = (dma_mem_t *)kmem_zalloc(sizeof (dma_mem_t), KM_SLEEP);
2016        if (dma_mem == NULL) {
2017                EL(ha, "failed, kmem_zalloc\n");
2018                cmd->Status = EXT_STATUS_NO_MEMORY;
2019                cmd->ResponseLen = 0;
2020                return;
2021        }
2022        /*  Verify the size of and copy in the passthru request structure. */
2023        if (cmd->RequestLen != pld_size) {
2024                /* Return error */
2025                EL(ha, "failed, RequestLen != cnt, is=%xh, expected=%xh\n",
2026                    cmd->RequestLen, pld_size);
2027                cmd->Status = EXT_STATUS_INVALID_PARAM;
2028                cmd->DetailStatus = EXT_DSTATUS_REQUEST_LEN;
2029                cmd->ResponseLen = 0;
2030                return;
2031        }

2033        if (ddi_copyin((void *)(uintptr_t)cmd->RequestAdr, &pt_req,
2034            pld_size, mode) != 0) {
2035                EL(ha, "failed, ddi_copyin\n");
```



```
2036                cmd->Status = EXT_STATUS_COPY_ERR;
2037                cmd->ResponseLen = 0;
2038                return;
2039        }

2041        /*
2042         * Find fc_port from SCSI PASSTHRU structure fill in the scsi_req
2043         * request data structure.
2044         */
2045        if (cmd->SubCode == EXT_SC_SEND_SCSI_PASSTHRU) {
2046                scsi_req.lun = sp_req->TargetAddr.Lun;
2047                scsi_req.sense_length = sizeof (sp_req->SenseData);
2048                scsi_req.cdbp = &sp_req->Cdb[0];
2049                scsi_req.cdb_len = sp_req->CdbLength;
2050                scsi_req.direction = sp_req->Direction;
2051                usp_req = (EXT_SCSI_PASSTHRU *)(uintptr_t)cmd->RequestAdr;
2052                scsi_req.u_sense = &usp_req->SenseData[0];
2053                cmd->DetailStatus = EXT_DSTATUS_TARGET;

2055                qlnt = QLNT_PORT;
2056                name = (uint8_t *)&sp_req->TargetAddr.Target;
2057                QL_PRINT_9(CE_CONT, "(%d): SubCode=%xh, Target=%lld\n",
2058                    ha->instance, cmd->SubCode, sp_req->TargetAddr.Target);
2059                tq = ql_find_port(ha, name, qlnt);
2060        } else {
2061                /*
2062                 * Must be FC PASSTHRU, verified above.
2063                 */
2064                if (fc_req->FCScsiAddr.DestType == EXT_DEF_DESTTYPE_WWPN) {
2065                        qlnt = QLNT_PORT;
2066                        name = &fc_req->FCScsiAddr.DestAddr.WWPN[0];
2067                        QL_PRINT_9(CE_CONT, "(%d): SubCode=%xh, "
2068                            "wwpn=%02x%02x%02x%02x%02x%02x%02x%02x\n",
2069                            ha->instance, cmd->SubCode, name[0], name[1],
2070                            name[2], name[3], name[4], name[5], name[6],
2071                            name[7]);
2072                        tq = ql_find_port(ha, name, qlnt);
2073                } else if (fc_req->FCScsiAddr.DestType ==
2074                    EXT_DEF_DESTTYPE_WWNN) {
2075                        qlnt = QLNT_NODE;
2076                        name = &fc_req->FCScsiAddr.DestAddr.WWNN[0];
2077                        QL_PRINT_9(CE_CONT, "(%d): SubCode=%xh, "
2078                            "wwnn=%02x%02x%02x%02x%02x%02x%02x%02x\n",
2079                            ha->instance, cmd->SubCode, name[0], name[1],
2080                            name[2], name[3], name[4], name[5], name[6],
2081                            name[7]);
2082                        tq = ql_find_port(ha, name, qlnt);
2083                } else if (fc_req->FCScsiAddr.DestType ==
2084                    EXT_DEF_DESTTYPE_PORTID) {
2085                        qlnt = QLNT_PID;
2086                        name = &fc_req->FCScsiAddr.DestAddr.Id[0];
2087                        QL_PRINT_9(CE_CONT, "(%d): SubCode=%xh, PID="
2088                            "%02x%02x%02x\n", ha->instance, cmd->SubCode,
2089                            name[0], name[1], name[2]);
2090                        tq = ql_find_port(ha, name, qlnt);
2091                } else {
2092                        EL(ha, "failed, SubCode=%xh invalid DestType=%xh\n",
2093                            cmd->SubCode, fc_req->FCScsiAddr.DestType);
2094                        cmd->Status = EXT_STATUS_INVALID_PARAM;
2095                        cmd->ResponseLen = 0;
2096                        return;
2097                }
2098                scsi_req.lun = fc_req->FCScsiAddr.Lun;
2099                scsi_req.sense_length = sizeof (fc_req->SenseData);
2100                scsi_req.cdbp = &sp_req->Cdb[0];
2101                scsi_req.cdb_len = sp_req->CdbLength;
```

```
2102                ufc_req = (EXT_FC_SCSI_PASSTHRU *)(uintptr_t)cmd->RequestAdr;
2103                scsi_req.u_sense = &ufc_req->SenseData[0];
2104                scsi_req.direction = fc_req->Direction;
2105        }
2106
2107        if (tq == NULL || !VALID_TARGET_ID(ha, tq->loop_id)) {
2108                EL(ha, "failed, fc_port not found\n");
2109                cmd->Status = EXT_STATUS_DEV_NOT_FOUND;
2110                cmd->ResponseLen = 0;
2111                return;
2112        }
2113
2114        if (tq->flags & TQF_NEED_AUTHENTICATION) {
2115                EL(ha, "target not available; loopid=%xh\n", tq->loop_id);
2116                cmd->Status = EXT_STATUS_DEVICE_OFFLINE;
2117                cmd->ResponseLen = 0;
2118                return;
2119        }
2120
2121        /* Allocate command block. */
2122        if ((scsi_req.direction == EXT_DEF_SCSI_PASSTHRU_DATA_IN ||
2123            scsi_req.direction == EXT_DEF_SCSI_PASSTHRU_DATA_OUT) &&
2124            cmd->ResponseLen) {
2125                pld_size = cmd->ResponseLen;
2126                pkt_size = (uint32_t)(sizeof (ql_mbx_iocb_t) + pld_size);
2127                pkt = kmem_zalloc(pkt_size, KM_SLEEP);
2143                if (pkt == NULL) {
2144                        EL(ha, "failed, kmem_zalloc\n");
2145                        cmd->Status = EXT_STATUS_NO_MEMORY;
2146                        cmd->ResponseLen = 0;
2147                        return;
2148                }
2128                pld = (caddr_t)pkt + sizeof (ql_mbx_iocb_t);
2129
2130                /* Get DMA memory for the IOCB */
2131                if (ql_get_dma_mem(ha, dma_mem, pld_size, LITTLE_ENDIAN_DMA,
2132                    QL_DMA_DATA_ALIGN) != QL_SUCCESS) {
2133                        cmn_err(CE_WARN, "%s(%d): request queue DMA memory "
2134                            "alloc failed", QL_NAME, ha->instance);
2135                        kmem_free(pkt, pkt_size);
2136                        cmd->Status = EXT_STATUS_MS_NO_RESPONSE;
2137                        cmd->ResponseLen = 0;
2138                        return;
2139                }
2140
2141                if (scsi_req.direction == EXT_DEF_SCSI_PASSTHRU_DATA_IN) {
2142                        scsi_req.direction = (uint8_t)
2143                            (CFG_IST(ha, CFG_CTRL_24258081) ?
2144                            CF_RD : CF_DATA_IN | CF_STAG);
2145                } else {
2146                        scsi_req.direction = (uint8_t)
2147                            (CFG_IST(ha, CFG_CTRL_24258081) ?
2148                            CF_WR : CF_DATA_OUT | CF_STAG);
2149                        cmd->ResponseLen = 0;
2150
2151                        /* Get command payload. */
2152                        if (ql_get_buffer_data(
2153                            (caddr_t)(uintptr_t)cmd->ResponseAdr,
2154                            pld, pld_size, mode) != pld_size) {
2155                                EL(ha, "failed, get_buffer_data\n");
2156                                cmd->Status = EXT_STATUS_COPY_ERR;
2157
2158                                kmem_free(pkt, pkt_size);
2159                                ql_free_dma_resource(ha, dma_mem);
2160                                kmem_free(dma_mem, sizeof (dma_mem_t));
2161                                return;
```



```
2162                        }
2163
2164                        /* Copy out going data to DMA buffer. */
2165                        ddi_rep_put8(dma_mem->acc_handle, (uint8_t *)pld,
2166                            (uint8_t *)dma_mem->bp, pld_size,
2167                            DDI_DEV_AUTOINCR);
2168
2169                        /* Sync DMA buffer. */
2170                        (void) ddi_dma_sync(dma_mem->dma_handle, 0,
2171                            dma_mem->size, DDI_DMA_SYNC_FORDEV);
2172                }
2173        } else {
2174                scsi_req.direction = (uint8_t)
2175                    (CFG_IST(ha, CFG_CTRL_24258081) ? 0 : CF_STAG);
2176                cmd->ResponseLen = 0;
2177
2178                pkt_size = sizeof (ql_mbx_iocb_t);
2179                pkt = kmem_zalloc(pkt_size, KM_SLEEP);
2201                if (pkt == NULL) {
2202                        EL(ha, "failed, kmem_zalloc-2\n");
2203                        cmd->Status = EXT_STATUS_NO_MEMORY;
2204                        return;
2205                }
2180                pld = NULL;
2181                pld_size = 0;
2182        }
2183
2184        /* retries = ha->port_down_retry_count; */
2185        retries = 1;
2186        cmd->Status = EXT_STATUS_OK;
2187        cmd->DetailStatus = EXT_DSTATUS_NOADNL_INFO;
2188
2189        QL_PRINT_9(CE_CONT, "(%d): SCSI cdb\n", ha->instance);
2190        QL_DUMP_9(scsi_req.cdbp, 8, scsi_req.cdb_len);
2191
2192        do {
2193                if (DRIVER_SUSPENDED(ha)) {
2194                        sts.comp_status = CS_LOOP_DOWN_ABORT;
2195                        break;
2196                }
2197
2198                if (CFG_IST(ha, CFG_CTRL_24258081)) {
2199                        pkt->cmd24.entry_type = IOCB_CMD_TYPE_7;
2200                        pkt->cmd24.entry_count = 1;
2201
2202                        /* Set LUN number */
2203                        pkt->cmd24.fcp_lun[2] = LSB(scsi_req.lun);
2204                        pkt->cmd24.fcp_lun[3] = MSB(scsi_req.lun);
2205
2206                        /* Set N_port handle */
2207                        pkt->cmd24.n_port_hdl = (uint16_t)LE_16(tq->loop_id);
2208
2209                        /* Set VP Index */
2210                        pkt->cmd24.vp_index = ha->vp_index;
2211
2212                        /* Set target ID */
2213                        pkt->cmd24.target_id[0] = tq->d_id.b.al_pa;
2214                        pkt->cmd24.target_id[1] = tq->d_id.b.area;
2215                        pkt->cmd24.target_id[2] = tq->d_id.b.domain;
2216
2217                        /* Set ISP command timeout. */
2218                        pkt->cmd24.timeout = (uint16_t)LE_16(15);
2219
2220                        /* Load SCSI CDB */
2221                        ddi_rep_put8(ha->hba_buf.acc_handle, scsi_req.cdbp,
2222                            pkt->cmd24.scsi_cdb, scsi_req.cdb_len,
```

```
2223                            DDI_DEV_AUTOINCR);
2224                    for (cnt = 0; cnt < MAX_CMDSZ;
2225                        cnt = (uint16_t)(cnt + 4)) {
2226                            ql_chg_endian((uint8_t *)&pkt->cmd24.scsi_cdb
2227                                + cnt, 4);
2228                    }

2230                    /* Set tag queue control flags */
2231                    pkt->cmd24.task = TA_STAG;

2233                    if (pld_size) {
2234                            /* Set transfer direction. */
2235                            pkt->cmd24.control_flags = scsi_req.direction;

2237                            /* Set data segment count. */
2238                            pkt->cmd24.dseg_count = LE_16(1);

2240                            /* Load total byte count. */
2241                            pkt->cmd24.total_byte_count = LE_32(pld_size);

2243                            /* Load data descriptor. */
2244                            pkt->cmd24.dseg_0_address[0] = (uint32_t)
2245                                LE_32(LSD(dma_mem->cookie.dmac_laddress));
2246                            pkt->cmd24.dseg_0_address[1] = (uint32_t)
2247                                LE_32(MSD(dma_mem->cookie.dmac_laddress));
2248                            pkt->cmd24.dseg_0_length = LE_32(pld_size);
2249                    }
2250            } else if (CFG_IST(ha, CFG_ENABLE_64BIT_ADDRESSING)) {
2251                    pkt->cmd3.entry_type = IOCB_CMD_TYPE_3;
2252                    pkt->cmd3.entry_count = 1;
2253                    if (CFG_IST(ha, CFG_EXT_FW_INTERFACE)) {
2254                            pkt->cmd3.target_l = LSB(tq->loop_id);
2255                            pkt->cmd3.target_h = MSB(tq->loop_id);
2256                    } else {
2257                            pkt->cmd3.target_h = LSB(tq->loop_id);
2258                    }
2259                    pkt->cmd3.lun_l = LSB(scsi_req.lun);
2260                    pkt->cmd3.lun_h = MSB(scsi_req.lun);
2261                    pkt->cmd3.control_flags_l = scsi_req.direction;
2262                    pkt->cmd3.timeout = LE_16(15);
2263                    for (cnt = 0; cnt < scsi_req.cdb_len; cnt++) {
2264                            pkt->cmd3.scsi_cdb[cnt] = scsi_req.cdbp[cnt];
2265                    }
2266                    if (pld_size) {
2267                            pkt->cmd3.dseg_count = LE_16(1);
2268                            pkt->cmd3.byte_count = LE_32(pld_size);
2269                            pkt->cmd3.dseg_0_address[0] = (uint32_t)
2270                                LE_32(LSD(dma_mem->cookie.dmac_laddress));
2271                            pkt->cmd3.dseg_0_address[1] = (uint32_t)
2272                                LE_32(MSD(dma_mem->cookie.dmac_laddress));
2273                            pkt->cmd3.dseg_0_length = LE_32(pld_size);
2274                    }
2275            } else {
2276                    pkt->cmd.entry_type = IOCB_CMD_TYPE_2;
2277                    pkt->cmd.entry_count = 1;
2278                    if (CFG_IST(ha, CFG_EXT_FW_INTERFACE)) {
2279                            pkt->cmd.target_l = LSB(tq->loop_id);
2280                            pkt->cmd.target_h = MSB(tq->loop_id);
2281                    } else {
2282                            pkt->cmd.target_h = LSB(tq->loop_id);
2283                    }
2284                    pkt->cmd.lun_l = LSB(scsi_req.lun);
2285                    pkt->cmd.lun_h = MSB(scsi_req.lun);
2286                    pkt->cmd.control_flags_l = scsi_req.direction;
2287                    pkt->cmd.timeout = LE_16(15);
2288                    for (cnt = 0; cnt < scsi_req.cdb_len; cnt++) {
```



```
2289                            pkt->cmd.scsi_cdb[cnt] = scsi_req.cdbp[cnt];
2290                    }
2291                    if (pld_size) {
2292                            pkt->cmd.dseg_count = LE_16(1);
2293                            pkt->cmd.byte_count = LE_32(pld_size);
2294                            pkt->cmd.dseg_0_address = (uint32_t)
2295                                LE_32(LSD(dma_mem->cookie.dmac_laddress));
2296                            pkt->cmd.dseg_0_length = LE_32(pld_size);
2297                    }
2298            }
2299            /* Go issue command and wait for completion. */
2300            QL_PRINT_9(CE_CONT, "(%d): request pkt\n", ha->instance);
2301            QL_DUMP_9(pkt, 8, pkt_size);

2303            status = ql_issue_mbx_iocb(ha, (caddr_t)pkt, pkt_size);

2305            if (pld_size) {
2306                    /* Sync in coming DMA buffer. */
2307                    (void) ddi_dma_sync(dma_mem->dma_handle, 0,
2308                        dma_mem->size, DDI_DMA_SYNC_FORKERNEL);
2309                    /* Copy in coming DMA data. */
2310                    ddi_rep_get8(dma_mem->acc_handle, (uint8_t *)pld,
2311                        (uint8_t *)dma_mem->bp, pld_size,
2312                        DDI_DEV_AUTOINCR);
2313            }

2315            if (CFG_IST(ha, CFG_CTRL_24258081)) {
2316                    pkt->sts24.entry_status = (uint8_t)
2317                        (pkt->sts24.entry_status & 0x3c);
2318            } else {
2319                    pkt->sts.entry_status = (uint8_t)
2320                        (pkt->sts.entry_status & 0x7e);
2321            }

2323            if (status == QL_SUCCESS && pkt->sts.entry_status != 0) {
2324                    EL(ha, "failed, entry_status=%xh, d_id=%xh\n",
2325                        pkt->sts.entry_status, tq->d_id.b24);
2326                    status = QL_FUNCTION_PARAMETER_ERROR;
2327            }

2329            sts.comp_status = (uint16_t)(CFG_IST(ha, CFG_CTRL_24258081) ?
2330                LE_16(pkt->sts24.comp_status) :
2331                LE_16(pkt->sts.comp_status));

2333            /*
2334             * We have verified about all the request that can be so far.
2335             * Now we need to start verification of our ability to
2336             * actually issue the CDB.
2337             */
2338            if (DRIVER_SUSPENDED(ha)) {
2339                    sts.comp_status = CS_LOOP_DOWN_ABORT;
2340                    break;
2341            } else if (status == QL_SUCCESS &&
2342                (sts.comp_status == CS_PORT_LOGGED_OUT ||
2343                sts.comp_status == CS_PORT_UNAVAILABLE)) {
2344                    EL(ha, "login retry d_id=%xh\n", tq->d_id.b24);
2345                    if (tq->flags & TQF_FABRIC_DEVICE) {
2346                            rval = ql_login_fport(ha, tq, tq->loop_id,
2347                                LFF_NO_PLOGI, &mr);
2348                            if (rval != QL_SUCCESS) {
2349                                    EL(ha, "failed, login_fport=%xh, "
2350                                        "d_id=%xh\n", rval, tq->d_id.b24);
2351                            }
2352                    } else {
2353                            rval = ql_login_lport(ha, tq, tq->loop_id,
2354                                LLF_NONE);
```

```
2355                                if (rval != QL_SUCCESS) {
2356                                        EL(ha, "failed, login_lport=%xh, "
2357                                            "d_id=%xh\n", rval, tq->d_id.b24);
2358                                }
2359                        }
2360                } else {
2361                        break;
2362                }

2364                bzero((caddr_t)pkt, sizeof (ql_mbx_iocb_t));

2366        } while (retries--);

2368        if (sts.comp_status == CS_LOOP_DOWN_ABORT) {
2369                /* Cannot issue command now, maybe later */
2370                EL(ha, "failed, suspended\n");
2371                kmem_free(pkt, pkt_size);
2372                ql_free_dma_resource(ha, dma_mem);
2373                kmem_free(dma_mem, sizeof (dma_mem_t));
2374                cmd->Status = EXT_STATUS_SUSPENDED;
2375                cmd->ResponseLen = 0;
2376                return;
2377        }

2379        if (status != QL_SUCCESS) {
2380                /* Command error */
2381                EL(ha, "failed, I/O\n");
2382                kmem_free(pkt, pkt_size);
2383                ql_free_dma_resource(ha, dma_mem);
2384                kmem_free(dma_mem, sizeof (dma_mem_t));
2385                cmd->Status = EXT_STATUS_ERR;
2386                cmd->DetailStatus = status;
2387                cmd->ResponseLen = 0;
2388                return;
2389        }

2391        /* Setup status. */
2392        if (CFG_IST(ha, CFG_CTRL_24258081)) {
2393                sts.scsi_status_l = pkt->sts24.scsi_status_l;
2394                sts.scsi_status_h = pkt->sts24.scsi_status_h;

2396                /* Setup residuals. */
2397                sts.residual_length = LE_32(pkt->sts24.residual_length);

2399                /* Setup state flags. */
2400                sts.state_flags_l = pkt->sts24.state_flags_l;
2401                sts.state_flags_h = pkt->sts24.state_flags_h;
2402                if (pld_size && sts.comp_status != CS_DATA_UNDERRUN) {
2403                        sts.state_flags_h = (uint8_t)(sts.state_flags_h |
2404                            SF_GOT_BUS | SF_GOT_TARGET | SF_SENT_CMD |
2405                            SF_XFERRED_DATA | SF_GOT_STATUS);
2406                } else {
2407                        sts.state_flags_h = (uint8_t)(sts.state_flags_h |
2408                            SF_GOT_BUS | SF_GOT_TARGET | SF_SENT_CMD |
2409                            SF_GOT_STATUS);
2410                }
2411                if (scsi_req.direction & CF_WR) {
2412                        sts.state_flags_l = (uint8_t)(sts.state_flags_l |
2413                            SF_DATA_OUT);
2414                } else if (scsi_req.direction & CF_RD) {
2415                        sts.state_flags_l = (uint8_t)(sts.state_flags_l |
2416                            SF_DATA_IN);
2417                }
2418                sts.state_flags_l = (uint8_t)(sts.state_flags_l | SF_SIMPLE_Q);

2420                /* Setup FCP response info. */
```



```
2421                sts.rsp_info_length = sts.scsi_status_h & FCP_RSP_LEN_VALID ?
2422                    LE_32(pkt->sts24.fcp_rsp_data_length) : 0;
2423                sts.rsp_info = &pkt->sts24.rsp_sense_data[0];
2424                for (cnt = 0; cnt < sts.rsp_info_length;
2425                    cnt = (uint16_t)(cnt + 4)) {
2426                        ql_chg_endian(sts.rsp_info + cnt, 4);
2427                }

2429                /* Setup sense data. */
2430                if (sts.scsi_status_h & FCP_SNS_LEN_VALID) {
2431                        sts.req_sense_length =
2432                            LE_32(pkt->sts24.fcp_sense_length);
2433                        sts.state_flags_h = (uint8_t)(sts.state_flags_h |
2434                            SF_ARQ_DONE);
2435                } else {
2436                        sts.req_sense_length = 0;
2437                }
2438                sts.req_sense_data =
2439                    &pkt->sts24.rsp_sense_data[sts.rsp_info_length];
2440                cnt2 = (uint16_t)(((uintptr_t)pkt + sizeof (sts_24xx_entry_t)) -
2441                    (uintptr_t)sts.req_sense_data);
2442                for (cnt = 0; cnt < cnt2; cnt = (uint16_t)(cnt + 4)) {
2443                        ql_chg_endian(sts.req_sense_data + cnt, 4);
2444                }
2445        } else {
2446                sts.scsi_status_l = pkt->sts.scsi_status_l;
2447                sts.scsi_status_h = pkt->sts.scsi_status_h;

2449                /* Setup residuals. */
2450                sts.residual_length = LE_32(pkt->sts.residual_length);

2452                /* Setup state flags. */
2453                sts.state_flags_l = pkt->sts.state_flags_l;
2454                sts.state_flags_h = pkt->sts.state_flags_h;

2456                /* Setup FCP response info. */
2457                sts.rsp_info_length = sts.scsi_status_h & FCP_RSP_LEN_VALID ?
2458                    LE_16(pkt->sts.rsp_info_length) : 0;
2459                sts.rsp_info = &pkt->sts.rsp_info[0];

2461                /* Setup sense data. */
2462                sts.req_sense_length = sts.scsi_status_h & FCP_SNS_LEN_VALID ?
2463                    LE_16(pkt->sts.req_sense_length) : 0;
2464                sts.req_sense_data = &pkt->sts.req_sense_data[0];
2465        }

2467        QL_PRINT_9(CE_CONT, "(%d): response pkt\n", ha->instance);
2468        QL_DUMP_9(&pkt->sts, 8, sizeof (sts_entry_t));

2470        switch (sts.comp_status) {
2471        case CS_INCOMPLETE:
2472        case CS_ABORTED:
2473        case CS_DEVICE_UNAVAILABLE:
2474        case CS_PORT_UNAVAILABLE:
2475        case CS_PORT_LOGGED_OUT:
2476        case CS_PORT_CONFIG_CHG:
2477        case CS_PORT_BUSY:
2478        case CS_LOOP_DOWN_ABORT:
2479                cmd->Status = EXT_STATUS_BUSY;
2480                break;
2481        case CS_RESET:
2482        case CS_QUEUE_FULL:
2483                cmd->Status = EXT_STATUS_ERR;
2484                break;
2485        case CS_TIMEOUT:
2486                cmd->Status = EXT_STATUS_ERR;
```

```
2487                break;
2488        case CS_DATA_OVERRUN:
2489                cmd->Status = EXT_STATUS_DATA_OVERRUN;
2490                break;
2491        case CS_DATA_UNDERRUN:
2492                cmd->Status = EXT_STATUS_DATA_UNDERRUN;
2493                break;
2494        }

2496        /*
2497         * If non data transfer commands fix tranfer counts.
2498         */
2499        if (scsi_req.cdbp[0] == SCMD_TEST_UNIT_READY ||
2500            scsi_req.cdbp[0] == SCMD_REZERO_UNIT ||
2501            scsi_req.cdbp[0] == SCMD_SEEK ||
2502            scsi_req.cdbp[0] == SCMD_SEEK_G1 ||
2503            scsi_req.cdbp[0] == SCMD_RESERVE ||
2504            scsi_req.cdbp[0] == SCMD_RELEASE ||
2505            scsi_req.cdbp[0] == SCMD_START_STOP ||
2506            scsi_req.cdbp[0] == SCMD_DOORLOCK ||
2507            scsi_req.cdbp[0] == SCMD_VERIFY ||
2508            scsi_req.cdbp[0] == SCMD_WRITE_FILE_MARK ||
2509            scsi_req.cdbp[0] == SCMD_VERIFY_G0 ||
2510            scsi_req.cdbp[0] == SCMD_SPACE ||
2511            scsi_req.cdbp[0] == SCMD_ERASE ||
2512            (scsi_req.cdbp[0] == SCMD_FORMAT &&
2513            (scsi_req.cdbp[1] & FPB_DATA) == 0)) {
2514                /*
2515                 * Non data transfer command, clear sts_entry residual
2516                 * length.
2517                 */
2518                sts.residual_length = 0;
2519                cmd->ResponseLen = 0;
2520                if (sts.comp_status == CS_DATA_UNDERRUN) {
2521                        sts.comp_status = CS_COMPLETE;
2522                        cmd->Status = EXT_STATUS_OK;
2523                }
2524        } else {
2525                cmd->ResponseLen = pld_size;
2526        }

2528        /* Correct ISP completion status */
2529        if (sts.comp_status == CS_COMPLETE && sts.scsi_status_l == 0 &&
2530            (sts.scsi_status_h & FCP_RSP_MASK) == 0) {
2531                QL_PRINT_9(CE_CONT, "(%d): Correct completion\n",
2532                    ha->instance);
2533                scsi_req.resid = 0;
2534        } else if (sts.comp_status == CS_DATA_UNDERRUN) {
2535                QL_PRINT_9(CE_CONT, "(%d): Correct UNDERRUN\n",
2536                    ha->instance);
2537                scsi_req.resid = sts.residual_length;
2538                if (sts.scsi_status_h & FCP_RESID_UNDER) {
2539                        cmd->Status = (uint32_t)EXT_STATUS_OK;

2541                        cmd->ResponseLen = (uint32_t)
2542                            (pld_size - scsi_req.resid);
2543                } else {
2544                        EL(ha, "failed, Transfer ERROR\n");
2545                        cmd->Status = EXT_STATUS_ERR;
2546                        cmd->ResponseLen = 0;
2547                }
2548        } else {
2549                QL_PRINT_9(CE_CONT, "(%d): error d_id=%xh, comp_status=%xh, "
2550                    "scsi_status_h=%xh, scsi_status_l=%xh\n", ha->instance,
2551                    tq->d_id.b24, sts.comp_status, sts.scsi_status_h,
2552                    sts.scsi_status_l);
```



```
2554                scsi_req.resid = pld_size;
2555                /*
2556                 * Handle residual count on SCSI check
2557                 * condition.
2558                 *
2559                 * - If Residual Under / Over is set, use the
2560                 *   Residual Transfer Length field in IOCB.
2561                 * - If Residual Under / Over is not set, and
2562                 *   Transferred Data bit is set in State Flags
2563                 *   field of IOCB, report residual value of 0
2564                 *   (you may want to do this for tape
2565                 *   Write-type commands only). This takes care
2566                 *   of logical end of tape problem and does
2567                 *   not break Unit Attention.
2568                 * - If Residual Under / Over is not set, and
2569                 *   Transferred Data bit is not set in State
2570                 *   Flags, report residual value equal to
2571                 *   original data transfer length.
2572                 */
2573                if (sts.scsi_status_l & STATUS_CHECK) {
2574                        cmd->Status = EXT_STATUS_SCSI_STATUS;
2575                        cmd->DetailStatus = sts.scsi_status_l;
2576                        if (sts.scsi_status_h &
2577                            (FCP_RESID_OVER | FCP_RESID_UNDER)) {
2578                                scsi_req.resid = sts.residual_length;
2579                        } else if (sts.state_flags_h &
2580                            STATE_XFERRED_DATA) {
2581                                scsi_req.resid = 0;
2582                        }
2583                }
2584        }

2586        if (sts.scsi_status_l & STATUS_CHECK &&
2587            sts.scsi_status_h & FCP_SNS_LEN_VALID &&
2588            sts.req_sense_length) {
2589                /*
2590                 * Check condition with vaild sense data flag set and sense
2591                 * length != 0
2592                 */
2593                if (sts.req_sense_length > scsi_req.sense_length) {
2594                        sense_sz = scsi_req.sense_length;
2595                } else {
2596                        sense_sz = sts.req_sense_length;
2597                }

2599                EL(ha, "failed, Check Condition Status, d_id=%xh\n",
2600                    tq->d_id.b24);
2601                QL_DUMP_2(sts.req_sense_data, 8, sts.req_sense_length);

2603                if (ddi_copyout(sts.req_sense_data, scsi_req.u_sense,
2604                    (size_t)sense_sz, mode) != 0) {
2605                        EL(ha, "failed, request sense ddi_copyout\n");
2606                }

2608                cmd->Status = EXT_STATUS_SCSI_STATUS;
2609                cmd->DetailStatus = sts.scsi_status_l;
2610        }

2612        /* Copy response payload from DMA buffer to application. */
2613        if (scsi_req.direction & (CF_RD | CF_DATA_IN) &&
2614            cmd->ResponseLen != 0) {
2615                QL_PRINT_9(CE_CONT, "(%d): Data Return resid=%lu, "
2616                    "byte_count=%u, ResponseLen=%xh\n", ha->instance,
2617                    scsi_req.resid, pld_size, cmd->ResponseLen);
2618                QL_DUMP_9(pld, 8, cmd->ResponseLen);
```

```
2620                 /* Send response payload. */
2621                 if (ql_send_buffer_data(pld,
2622                     (caddr_t)(uintptr_t)cmd->ResponseAdr,
2623                     cmd->ResponseLen, mode) != cmd->ResponseLen) {
2624                         EL(ha, "failed, send_buffer_data\n");
2625                         cmd->Status = EXT_STATUS_COPY_ERR;
2626                         cmd->ResponseLen = 0;
2627                 }
2628         }

2630         if (cmd->Status != EXT_STATUS_OK) {
2631                 EL(ha, "failed, cmd->Status=%xh, comp_status=%xh, "
2632                     "d_id=%xh\n", cmd->Status, sts.comp_status, tq->d_id.b24);
2633         } else {
2634                 /*EMPTY*/
2635                 QL_PRINT_9(CE_CONT, "(%d): done, ResponseLen=%d\n",
2636                     ha->instance, cmd->ResponseLen);
2637         }

2639         kmem_free(pkt, pkt_size);
2640         ql_free_dma_resource(ha, dma_mem);
2641         kmem_free(dma_mem, sizeof (dma_mem_t));
2642 }
```

unchanged_portion_omitted_

```
3288 /*
3289  * ql_diagnostic_loopback
3290  *      Performs EXT_CC_LOOPBACK Command
3291  *
3292  * Input:
3293  *      ha:     adapter state pointer.
3294  *      cmd:    Local EXT_IOCTL cmd struct pointer.
3295  *      mode:   flags.
3296  *
3297  * Returns:
3298  *      None, request status indicated in cmd->Status.
3299  *
3300  * Context:
3301  *      Kernel context.
3302  */
3303 static void
3304 ql_diagnostic_loopback(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
3305 {
3306         EXT_LOOPBACK_REQ        plbreq;
3307         EXT_LOOPBACK_RSP        plbrsp;
3308         ql_mbx_data_t           mr;
3309         uint32_t                rval;
3310         caddr_t                 bp;
3311         uint16_t                opt;

3313         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

3315         /* Get loop back request. */
3316         if (ddi_copyin((void *)(uintptr_t)cmd->RequestAdr,
3317             (void *)&plbreq, sizeof (EXT_LOOPBACK_REQ), mode) != 0) {
3318                 EL(ha, "failed, ddi_copyin\n");
3319                 cmd->Status = EXT_STATUS_COPY_ERR;
3320                 cmd->ResponseLen = 0;
3321                 return;
3322         }

3324         opt = (uint16_t)(plbreq.Options & MBC_LOOPBACK_POINT_MASK);

3326         /* Check transfer length fits in buffer. */
3327         if (plbreq.BufferLength < plbreq.TransferCount &&
```



```
3328             plbreq.TransferCount < MAILBOX_BUFFER_SIZE) {
3329                 EL(ha, "failed, BufferLength=%d, xfercnt=%d, "
3330                     "mailbox_buffer_size=%d\n", plbreq.BufferLength,
3331                     plbreq.TransferCount, MAILBOX_BUFFER_SIZE);
3332                 cmd->Status = EXT_STATUS_INVALID_PARAM;
3333                 cmd->ResponseLen = 0;
3334                 return;
3335         }

3337         /* Allocate command memory. */
3338         bp = kmem_zalloc(plbreq.TransferCount, KM_SLEEP);
3365         if (bp == NULL) {
3366                 EL(ha, "failed, kmem_zalloc\n");
3367                 cmd->Status = EXT_STATUS_NO_MEMORY;
3368                 cmd->ResponseLen = 0;
3369                 return;
3370         }

3340         /* Get loopback data. */
3341         if (ql_get_buffer_data((caddr_t)(uintptr_t)plbreq.BufferAddress,
3342             bp, plbreq.TransferCount, mode) != plbreq.TransferCount) {
3343                 EL(ha, "failed, ddi_copyin-2\n");
3344                 kmem_free(bp, plbreq.TransferCount);
3345                 cmd->Status = EXT_STATUS_COPY_ERR;
3346                 cmd->ResponseLen = 0;
3347                 return;
3348         }

3350         if ((ha->task_daemon_flags & (QL_LOOP_TRANSITION | DRIVER_STALL)) ||
3351             ql_stall_driver(ha, 0) != QL_SUCCESS) {
3352                 EL(ha, "failed, LOOP_NOT_READY\n");
3353                 kmem_free(bp, plbreq.TransferCount);
3354                 cmd->Status = EXT_STATUS_BUSY;
3355                 cmd->ResponseLen = 0;
3356                 return;
3357         }

3359         /* Shutdown IP. */
3360         if (ha->flags & IP_INITIALIZED) {
3361                 (void) ql_shutdown_ip(ha);
3362         }

3364         /* determine topology so we can send the loopback or the echo */
3365         /* Echo is supported on 2300's only and above */

3367         if (CFG_IST(ha, CFG_CTRL_8081)) {
3368                 if (!(ha->task_daemon_flags & LOOP_DOWN) && opt ==
3369                     MBC_LOOPBACK_POINT_EXTERNAL) {
3370                         if (plbreq.TransferCount > 252) {
3371                                 EL(ha, "transfer count (%d) > 252\n",
3372                                     plbreq.TransferCount);
3373                                 kmem_free(bp, plbreq.TransferCount);
3374                                 cmd->Status = EXT_STATUS_INVALID_PARAM;
3375                                 cmd->ResponseLen = 0;
3376                                 return;
3377                         }
3378                         plbrsp.CommandSent = INT_DEF_LB_ECHO_CMD;
3379                         rval = ql_diag_echo(ha, 0, bp, plbreq.TransferCount,
3380                             MBC_ECHO_ELS, &mr);
3381                 } else {
3382                         if (CFG_IST(ha, CFG_CTRL_81XX)) {
3383                                 (void) ql_set_loop_point(ha, opt);
3384                         }
3385                         plbrsp.CommandSent = INT_DEF_LB_LOOPBACK_CMD;
3386                         rval = ql_diag_loopback(ha, 0, bp, plbreq.TransferCount,
3387                             opt, plbreq.IterationCount, &mr);
```

```
3388                        if (CFG_IST(ha, CFG_CTRL_81XX)) {
3389                                (void) ql_set_loop_point(ha, 0);
3390                        }
3391                }
3392        } else {
3393                if (!(ha->task_daemon_flags & LOOP_DOWN) &&
3394                    (ha->topology & QL_F_PORT) &&
3395                    ha->device_id >= 0x2300) {
3396                        QL_PRINT_9(CE_CONT, "(%d): F_PORT topology -- using "
3397                            "echo\n", ha->instance);
3398                        plbrsp.CommandSent = INT_DEF_LB_ECHO_CMD;
3399                        rval = ql_diag_echo(ha, 0, bp, plbreq.TransferCount,
3400                            (uint16_t)(CFG_IST(ha, CFG_CTRL_8081) ?
3401                            MBC_ECHO_ELS : MBC_ECHO_64BIT), &mr);
3402                } else {
3403                        plbrsp.CommandSent = INT_DEF_LB_LOOPBACK_CMD;
3404                        rval = ql_diag_loopback(ha, 0, bp, plbreq.TransferCount,
3405                            opt, plbreq.IterationCount, &mr);
3406                }
3407        }

3409        ql_restart_driver(ha);

3411        /* Restart IP if it was shutdown. */
3412        if (ha->flags & IP_ENABLED && !(ha->flags & IP_INITIALIZED)) {
3413                (void) ql_initialize_ip(ha);
3414                ql_isp_rcvbuf(ha);
3415        }

3417        if (rval != QL_SUCCESS) {
3418                EL(ha, "failed, diagnostic_loopback_mbx=%xh\n", rval);
3419                kmem_free(bp, plbreq.TransferCount);
3420                cmd->Status = EXT_STATUS_MAILBOX;
3421                cmd->DetailStatus = rval;
3422                cmd->ResponseLen = 0;
3423                return;
3424        }

3426        /* Return loopback data. */
3427        if (ql_send_buffer_data(bp, (caddr_t)(uintptr_t)plbreq.BufferAddress,
3428            plbreq.TransferCount, mode) != plbreq.TransferCount) {
3429                EL(ha, "failed, ddi_copyout\n");
3430                kmem_free(bp, plbreq.TransferCount);
3431                cmd->Status = EXT_STATUS_COPY_ERR;
3432                cmd->ResponseLen = 0;
3433                return;
3434        }
3435        kmem_free(bp, plbreq.TransferCount);

3437        /* Return loopback results. */
3438        plbrsp.BufferAddress = plbreq.BufferAddress;
3439        plbrsp.BufferLength = plbreq.TransferCount;
3440        plbrsp.CompletionStatus = mr.mb[0];

3442        if (plbrsp.CommandSent == INT_DEF_LB_ECHO_CMD) {
3443                plbrsp.CrcErrorCount = 0;
3444                plbrsp.DisparityErrorCount = 0;
3445                plbrsp.FrameLengthErrorCount = 0;
3446                plbrsp.IterationCountLastError = 0;
3447        } else {
3448                plbrsp.CrcErrorCount = mr.mb[1];
3449                plbrsp.DisparityErrorCount = mr.mb[2];
3450                plbrsp.FrameLengthErrorCount = mr.mb[3];
3451                plbrsp.IterationCountLastError = (mr.mb[19] >> 16) | mr.mb[18];
3452        }
```



```
3454        rval = ddi_copyout((void *)&plbrsp,
3455            (void *)(uintptr_t)cmd->ResponseAdr,
3456            sizeof (EXT_LOOPBACK_RSP), mode);
3457        if (rval != 0) {
3458                EL(ha, "failed, ddi_copyout-2\n");
3459                cmd->Status = EXT_STATUS_COPY_ERR;
3460                cmd->ResponseLen = 0;
3461                return;
3462        }
3463        cmd->ResponseLen = sizeof (EXT_LOOPBACK_RSP);

3465        QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
3466 }
______unchanged_portion_omitted_

3534 /*
3535  * ql_send_els_rnid
3536  *      IOCTL for extended link service RNID command.
3537  *
3538  * Input:
3539  *      ha:     adapter state pointer.
3540  *      cmd:    User space CT arguments pointer.
3541  *      mode:   flags.
3542  *
3543  * Returns:
3544  *      None, request status indicated in cmd->Status.
3545  *
3546  * Context:
3547  *      Kernel context.
3548  */
3549 static void
3550 ql_send_els_rnid(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
3551 {
3552        EXT_RNID_REQ    tmp_rnid;
3553        port_id_t       tmp_fcid;
3554        caddr_t         tmp_buf, bptr;
3555        uint32_t        copy_len;
3556        ql_tgt_t        *tq;
3557        EXT_RNID_DATA   rnid_data;
3558        uint32_t        loop_ready_wait = 10 * 60 * 10;
3559        int             rval = 0;
3560        uint32_t        local_hba = 0;

3562        QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

3564        if (DRIVER_SUSPENDED(ha)) {
3565                EL(ha, "failed, LOOP_NOT_READY\n");
3566                cmd->Status = EXT_STATUS_BUSY;
3567                cmd->ResponseLen = 0;
3568                return;
3569        }

3571        if (cmd->RequestLen != sizeof (EXT_RNID_REQ)) {
3572                /* parameter error */
3573                EL(ha, "failed, RequestLen < EXT_RNID_REQ, Len=%xh\n",
3574                    cmd->RequestLen);
3575                cmd->Status = EXT_STATUS_INVALID_PARAM;
3576                cmd->DetailStatus = EXT_DSTATUS_REQUEST_LEN;
3577                cmd->ResponseLen = 0;
3578                return;
3579        }

3581        if (ddi_copyin((void*)(uintptr_t)cmd->RequestAdr,
3582            &tmp_rnid, cmd->RequestLen, mode) != 0) {
3583                EL(ha, "failed, ddi_copyin\n");
3584                cmd->Status = EXT_STATUS_COPY_ERR;
```

```
3585                cmd->ResponseLen = 0;
3586                return;
3587        }

3589        /* Find loop ID of the device */
3590        if (tmp_rnid.Addr.Type == EXT_DEF_TYPE_WWNN) {
3591                bptr = CFG_IST(ha, CFG_CTRL_24258081) ?
3592                    (caddr_t)&ha->init_ctrl_blk.cb24.node_name :
3593                    (caddr_t)&ha->init_ctrl_blk.cb.node_name;
3594                if (bcmp((void *)bptr, (void *)tmp_rnid.Addr.FcAddr.WWNN,
3595                    EXT_DEF_WWN_NAME_SIZE) == 0) {
3596                        local_hba = 1;
3597                } else {
3598                        tq = ql_find_port(ha,
3599                            (uint8_t *)tmp_rnid.Addr.FcAddr.WWNN, QLNT_NODE);
3600                }
3601        } else if (tmp_rnid.Addr.Type == EXT_DEF_TYPE_WWPN) {
3602                bptr = CFG_IST(ha, CFG_CTRL_24258081) ?
3603                    (caddr_t)&ha->init_ctrl_blk.cb24.port_name :
3604                    (caddr_t)&ha->init_ctrl_blk.cb.port_name;
3605                if (bcmp((void *)bptr, (void *)tmp_rnid.Addr.FcAddr.WWPN,
3606                    EXT_DEF_WWN_NAME_SIZE) == 0) {
3607                        local_hba = 1;
3608                } else {
3609                        tq = ql_find_port(ha,
3610                            (uint8_t *)tmp_rnid.Addr.FcAddr.WWPN, QLNT_PORT);
3611                }
3612        } else if (tmp_rnid.Addr.Type == EXT_DEF_TYPE_PORTID) {
3613                /*
3614                 * Copy caller's d_id to tmp space.
3615                 */
3616                bcopy(&tmp_rnid.Addr.FcAddr.Id[1], tmp_fcid.r.d_id,
3617                    EXT_DEF_PORTID_SIZE_ACTUAL);
3618                BIG_ENDIAN_24(&tmp_fcid.r.d_id[0]);

3620                if (bcmp((void *)&ha->d_id, (void *)tmp_fcid.r.d_id,
3621                    EXT_DEF_PORTID_SIZE_ACTUAL) == 0) {
3622                        local_hba = 1;
3623                } else {
3624                        tq = ql_find_port(ha, (uint8_t *)tmp_fcid.r.d_id,
3625                            QLNT_PID);
3626                }
3627        }

3629        /* Allocate memory for command. */
3630        tmp_buf = kmem_zalloc(SEND_RNID_RSP_SIZE, KM_SLEEP);
3663        if (tmp_buf == NULL) {
3664                EL(ha, "failed, kmem_zalloc\n");
3665                cmd->Status = EXT_STATUS_NO_MEMORY;
3666                cmd->ResponseLen = 0;
3667                return;
3668        }

3632        if (local_hba) {
3633                rval = ql_get_rnid_params(ha, SEND_RNID_RSP_SIZE, tmp_buf);
3634                if (rval != QL_SUCCESS) {
3635                        EL(ha, "failed, get_rnid_params_mbx=%xh\n", rval);
3636                        kmem_free(tmp_buf, SEND_RNID_RSP_SIZE);
3637                        cmd->Status = EXT_STATUS_ERR;
3638                        cmd->ResponseLen = 0;
3639                        return;
3640                }

3642                /* Save gotten RNID data. */
3643                bcopy(tmp_buf, &rnid_data, sizeof (EXT_RNID_DATA));
```



```
3645                /* Now build the Send RNID response */
3646                tmp_buf[0] = (char)(EXT_DEF_RNID_DFORMAT_TOPO_DISC);
3647                tmp_buf[1] = (2 * EXT_DEF_WWN_NAME_SIZE);
3648                tmp_buf[2] = 0;
3649                tmp_buf[3] = sizeof (EXT_RNID_DATA);

3651                if (CFG_IST(ha, CFG_CTRL_24258081)) {
3652                        bcopy(ha->init_ctrl_blk.cb24.port_name, &tmp_buf[4],
3653                            EXT_DEF_WWN_NAME_SIZE);
3654                        bcopy(ha->init_ctrl_blk.cb24.node_name,
3655                            &tmp_buf[4 + EXT_DEF_WWN_NAME_SIZE],
3656                            EXT_DEF_WWN_NAME_SIZE);
3657                } else {
3658                        bcopy(ha->init_ctrl_blk.cb.port_name, &tmp_buf[4],
3659                            EXT_DEF_WWN_NAME_SIZE);
3660                        bcopy(ha->init_ctrl_blk.cb.node_name,
3661                            &tmp_buf[4 + EXT_DEF_WWN_NAME_SIZE],
3662                            EXT_DEF_WWN_NAME_SIZE);
3663                }

3665                bcopy((uint8_t *)&rnid_data,
3666                    &tmp_buf[4 + 2 * EXT_DEF_WWN_NAME_SIZE],
3667                    sizeof (EXT_RNID_DATA));
3668        } else {
3669                if (tq == NULL) {
3670                        /* no matching device */
3671                        EL(ha, "failed, device not found\n");
3672                        kmem_free(tmp_buf, SEND_RNID_RSP_SIZE);
3673                        cmd->Status = EXT_STATUS_DEV_NOT_FOUND;
3674                        cmd->DetailStatus = EXT_DSTATUS_TARGET;
3675                        cmd->ResponseLen = 0;
3676                        return;
3677                }

3679                /* Send command */
3680                rval = ql_send_rnid_els(ha, tq->loop_id,
3681                    (uint8_t)tmp_rnid.DataFormat, SEND_RNID_RSP_SIZE, tmp_buf);
3682                if (rval != QL_SUCCESS) {
3683                        EL(ha, "failed, send_rnid_mbx=%xh, id=%xh\n",
3684                            rval, tq->loop_id);
3685                        while (LOOP_NOT_READY(ha)) {
3686                                ql_delay(ha, 100000);
3687                                if (loop_ready_wait-- == 0) {
3688                                        EL(ha, "failed, loop not ready\n");
3689                                        cmd->Status = EXT_STATUS_ERR;
3690                                        cmd->ResponseLen = 0;
3691                                }
3692                        }
3693                        rval = ql_send_rnid_els(ha, tq->loop_id,
3694                            (uint8_t)tmp_rnid.DataFormat, SEND_RNID_RSP_SIZE,
3695                            tmp_buf);
3696                        if (rval != QL_SUCCESS) {
3697                                /* error */
3698                                EL(ha, "failed, send_rnid_mbx=%xh, id=%xh\n",
3699                                    rval, tq->loop_id);
3700                                kmem_free(tmp_buf, SEND_RNID_RSP_SIZE);
3701                                cmd->Status = EXT_STATUS_ERR;
3702                                cmd->ResponseLen = 0;
3703                                return;
3704                        }
3705                }
3706        }

3708        /* Copy the response */
3709        copy_len = (cmd->ResponseLen > SEND_RNID_RSP_SIZE) ?
3710            SEND_RNID_RSP_SIZE : cmd->ResponseLen;
```

```
3712         if (ql_send_buffer_data(tmp_buf, (caddr_t)(uintptr_t)cmd->ResponseAdr,
3713             copy_len, mode) != copy_len) {
3714                 cmd->Status = EXT_STATUS_COPY_ERR;
3715                 EL(ha, "failed, ddi_copyout\n");
3716         } else {
3717                 cmd->ResponseLen = copy_len;
3718                 if (copy_len < SEND_RNID_RSP_SIZE) {
3719                         cmd->Status = EXT_STATUS_DATA_OVERRUN;
3720                         EL(ha, "failed, EXT_STATUS_DATA_OVERRUN\n");

3722                 } else if (cmd->ResponseLen > SEND_RNID_RSP_SIZE) {
3723                         cmd->Status = EXT_STATUS_DATA_UNDERRUN;
3724                         EL(ha, "failed, EXT_STATUS_DATA_UNDERRUN\n");
3725                 } else {
3726                         cmd->Status = EXT_STATUS_OK;
3727                         QL_PRINT_9(CE_CONT, "(%d): done\n",
3728                             ha->instance);
3729                 }
3730         }

3732         kmem_free(tmp_buf, SEND_RNID_RSP_SIZE);
3733 }
```

unchanged_portion_omitted_

```
3961 /*
3962  * ql_report_lun
3963  *      Get numbers of LUNS using report LUN command.
3964  *
3965  * Input:
3966  *      ha:     adapter state pointer.
3967  *      q:      target queue pointer.
3968  *
3969  * Returns:
3970  *      Number of LUNs.
3971  *
3972  * Context:
3973  *      Kernel context.
3974  */
3975 static int
3976 ql_report_lun(ql_adapter_state_t *ha, ql_tgt_t *tq)
3977 {
3978         int                     rval;
3979         uint8_t                 retries;
3980         ql_mbx_iocb_t           *pkt;
3981         ql_rpt_lun_lst_t        *rpt;
3982         dma_mem_t               dma_mem;
3983         uint32_t                pkt_size, cnt;
3984         uint16_t                comp_status;
3985         uint8_t                 scsi_status_h, scsi_status_l, *reqs;

3987         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

3989         if (DRIVER_SUSPENDED(ha)) {
3990                 EL(ha, "failed, LOOP_NOT_READY\n");
3991                 return (0);
3992         }

3994         pkt_size = sizeof (ql_mbx_iocb_t) + sizeof (ql_rpt_lun_lst_t);
3995         pkt = kmem_zalloc(pkt_size, KM_SLEEP);
4034         if (pkt == NULL) {
4035                 EL(ha, "failed, kmem_zalloc\n");
4036                 return (0);
4037         }
3996         rpt = (ql_rpt_lun_lst_t *)((caddr_t)pkt + sizeof (ql_mbx_iocb_t));
```



```
3998         /* Get DMA memory for the IOCB */
3999         if (ql_get_dma_mem(ha, &dma_mem, sizeof (ql_rpt_lun_lst_t),
4000             LITTLE_ENDIAN_DMA, QL_DMA_RING_ALIGN) != QL_SUCCESS) {
4001                 cmn_err(CE_WARN, "%s(%d): DMA memory "
4002                     "alloc failed", QL_NAME, ha->instance);
4003                 kmem_free(pkt, pkt_size);
4004                 return (0);
4005         }

4007         for (retries = 0; retries < 4; retries++) {
4008                 if (CFG_IST(ha, CFG_CTRL_24258081)) {
4009                         pkt->cmd24.entry_type = IOCB_CMD_TYPE_7;
4010                         pkt->cmd24.entry_count = 1;

4012                         /* Set N_port handle */
4013                         pkt->cmd24.n_port_hdl = (uint16_t)LE_16(tq->loop_id);

4015                         /* Set target ID */
4016                         pkt->cmd24.target_id[0] = tq->d_id.b.al_pa;
4017                         pkt->cmd24.target_id[1] = tq->d_id.b.area;
4018                         pkt->cmd24.target_id[2] = tq->d_id.b.domain;

4020                         /* Set Virtual Port ID */
4021                         pkt->cmd24.vp_index = ha->vp_index;

4023                         /* Set ISP command timeout. */
4024                         pkt->cmd24.timeout = LE_16(15);

4026                         /* Load SCSI CDB */
4027                         pkt->cmd24.scsi_cdb[0] = SCMD_REPORT_LUNS;
4028                         pkt->cmd24.scsi_cdb[6] =
4029                             MSB(MSW(sizeof (ql_rpt_lun_lst_t)));
4030                         pkt->cmd24.scsi_cdb[7] =
4031                             LSB(MSW(sizeof (ql_rpt_lun_lst_t)));
4032                         pkt->cmd24.scsi_cdb[8] =
4033                             MSB(LSW(sizeof (ql_rpt_lun_lst_t)));
4034                         pkt->cmd24.scsi_cdb[9] =
4035                             LSB(LSW(sizeof (ql_rpt_lun_lst_t)));
4036                         for (cnt = 0; cnt < MAX_CMDSZ; cnt += 4) {
4037                                 ql_chg_endian((uint8_t *)&pkt->cmd24.scsi_cdb
4038                                     + cnt, 4);
4039                         }

4041                         /* Set tag queue control flags */
4042                         pkt->cmd24.task = TA_STAG;

4044                         /* Set transfer direction. */
4045                         pkt->cmd24.control_flags = CF_RD;

4047                         /* Set data segment count. */
4048                         pkt->cmd24.dseg_count = LE_16(1);

4050                         /* Load total byte count. */
4051                         /* Load data descriptor. */
4052                         pkt->cmd24.dseg_0_address[0] = (uint32_t)
4053                             LE_32(LSD(dma_mem.cookie.dmac_laddress));
4054                         pkt->cmd24.dseg_0_address[1] = (uint32_t)
4055                             LE_32(MSD(dma_mem.cookie.dmac_laddress));
4056                         pkt->cmd24.total_byte_count =
4057                             LE_32(sizeof (ql_rpt_lun_lst_t));
4058                         pkt->cmd24.dseg_0_length =
4059                             LE_32(sizeof (ql_rpt_lun_lst_t));
4060                 } else if (CFG_IST(ha, CFG_ENABLE_64BIT_ADDRESSING)) {
4061                         pkt->cmd3.entry_type = IOCB_CMD_TYPE_3;
4062                         pkt->cmd3.entry_count = 1;
4063                         if (CFG_IST(ha, CFG_EXT_FW_INTERFACE)) {
```

```
4064                                pkt->cmd3.target_l = LSB(tq->loop_id);
4065                                pkt->cmd3.target_h = MSB(tq->loop_id);
4066                        } else {
4067                                pkt->cmd3.target_h = LSB(tq->loop_id);
4068                        }
4069                        pkt->cmd3.control_flags_l = CF_DATA_IN | CF_STAG;
4070                        pkt->cmd3.timeout = LE_16(15);
4071                        pkt->cmd3.dseg_count = LE_16(1);
4072                        pkt->cmd3.scsi_cdb[0] = SCMD_REPORT_LUNS;
4073                        pkt->cmd3.scsi_cdb[6] =
4074                            MSB(MSW(sizeof (ql_rpt_lun_lst_t)));
4075                        pkt->cmd3.scsi_cdb[7] =
4076                            LSB(MSW(sizeof (ql_rpt_lun_lst_t)));
4077                        pkt->cmd3.scsi_cdb[8] =
4078                            MSB(LSW(sizeof (ql_rpt_lun_lst_t)));
4079                        pkt->cmd3.scsi_cdb[9] =
4080                            LSB(LSW(sizeof (ql_rpt_lun_lst_t)));
4081                        pkt->cmd3.byte_count =
4082                            LE_32(sizeof (ql_rpt_lun_lst_t));
4083                        pkt->cmd3.dseg_0_address[0] = (uint32_t)
4084                            LE_32(LSD(dma_mem.cookie.dmac_laddress));
4085                        pkt->cmd3.dseg_0_address[1] = (uint32_t)
4086                            LE_32(MSD(dma_mem.cookie.dmac_laddress));
4087                        pkt->cmd3.dseg_0_length =
4088                            LE_32(sizeof (ql_rpt_lun_lst_t));
4089                } else {
4090                        pkt->cmd.entry_type = IOCB_CMD_TYPE_2;
4091                        pkt->cmd.entry_count = 1;
4092                        if (CFG_IST(ha, CFG_EXT_FW_INTERFACE)) {
4093                                pkt->cmd.target_l = LSB(tq->loop_id);
4094                                pkt->cmd.target_h = MSB(tq->loop_id);
4095                        } else {
4096                                pkt->cmd.target_h = LSB(tq->loop_id);
4097                        }
4098                        pkt->cmd.control_flags_l = CF_DATA_IN | CF_STAG;
4099                        pkt->cmd.timeout = LE_16(15);
4100                        pkt->cmd.dseg_count = LE_16(1);
4101                        pkt->cmd.scsi_cdb[0] = SCMD_REPORT_LUNS;
4102                        pkt->cmd.scsi_cdb[6] =
4103                            MSB(MSW(sizeof (ql_rpt_lun_lst_t)));
4104                        pkt->cmd.scsi_cdb[7] =
4105                            LSB(MSW(sizeof (ql_rpt_lun_lst_t)));
4106                        pkt->cmd.scsi_cdb[8] =
4107                            MSB(LSW(sizeof (ql_rpt_lun_lst_t)));
4108                        pkt->cmd.scsi_cdb[9] =
4109                            LSB(LSW(sizeof (ql_rpt_lun_lst_t)));
4110                        pkt->cmd.byte_count =
4111                            LE_32(sizeof (ql_rpt_lun_lst_t));
4112                        pkt->cmd.dseg_0_address = (uint32_t)
4113                            LE_32(LSD(dma_mem.cookie.dmac_laddress));
4114                        pkt->cmd.dseg_0_length =
4115                            LE_32(sizeof (ql_rpt_lun_lst_t));
4116                }

4118                rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt,
4119                    sizeof (ql_mbx_iocb_t));

4121                /* Sync in coming DMA buffer. */
4122                (void) ddi_dma_sync(dma_mem.dma_handle, 0, dma_mem.size,
4123                    DDI_DMA_SYNC_FORKERNEL);
4124                /* Copy in coming DMA data. */
4125                ddi_rep_get8(dma_mem.acc_handle, (uint8_t *)rpt,
4126                    (uint8_t *)dma_mem.bp, dma_mem.size, DDI_DEV_AUTOINCR);

4128                if (CFG_IST(ha, CFG_CTRL_24258081)) {
4129                        pkt->sts24.entry_status = (uint8_t)
```



```
4130                            (pkt->sts24.entry_status & 0x3c);
4131                        comp_status = (uint16_t)LE_16(pkt->sts24.comp_status);
4132                        scsi_status_h = pkt->sts24.scsi_status_h;
4133                        scsi_status_l = pkt->sts24.scsi_status_l;
4134                        cnt = scsi_status_h & FCP_RSP_LEN_VALID ?
4135                            LE_32(pkt->sts24.fcp_rsp_data_length) : 0;
4136                        reqs = &pkt->sts24.rsp_sense_data[cnt];
4137                } else {
4138                        pkt->sts.entry_status = (uint8_t)
4139                            (pkt->sts.entry_status & 0x7e);
4140                        comp_status = (uint16_t)LE_16(pkt->sts.comp_status);
4141                        scsi_status_h = pkt->sts.scsi_status_h;
4142                        scsi_status_l = pkt->sts.scsi_status_l;
4143                        reqs = &pkt->sts.req_sense_data[0];
4144                }
4145                if (rval == QL_SUCCESS && pkt->sts.entry_status != 0) {
4146                        EL(ha, "failed, entry_status=%xh, d_id=%xh\n",
4147                            pkt->sts.entry_status, tq->d_id.b24);
4148                        rval = QL_FUNCTION_PARAMETER_ERROR;
4149                }

4151                if (rval != QL_SUCCESS || comp_status != CS_COMPLETE ||
4152                    scsi_status_l & STATUS_CHECK) {
4153                        /* Device underrun, treat as OK. */
4154                        if (rval == QL_SUCCESS &&
4155                            comp_status == CS_DATA_UNDERRUN &&
4156                            scsi_status_h & FCP_RESID_UNDER) {
4157                                break;
4158                        }

4160                        EL(ha, "failed, issue_iocb=%xh, d_id=%xh, cs=%xh, "
4161                            "ss_h=%xh, ss_l=%xh\n", rval, tq->d_id.b24,
4162                            comp_status, scsi_status_h, scsi_status_l);

4164                        if (rval == QL_SUCCESS) {
4165                                if ((comp_status == CS_TIMEOUT) ||
4166                                    (comp_status == CS_PORT_UNAVAILABLE) ||
4167                                    (comp_status == CS_PORT_LOGGED_OUT)) {
4168                                        rval = QL_FUNCTION_TIMEOUT;
4169                                        break;
4170                                }
4171                                rval = QL_FUNCTION_FAILED;
4172                        } else if (rval == QL_ABORTED) {
4173                                break;
4174                        }

4176                        if (scsi_status_l & STATUS_CHECK) {
4177                                EL(ha, "STATUS_CHECK Sense Data\n%2xh%3xh"
4178                                    "%3xh%3xh%3xh%3xh%3xh%3xh%3xh%3xh"
4179                                    "%3xh%3xh%3xh%3xh%3xh%3xh%3xh%3xh\n", reqs[0],
4180                                    reqs[1], reqs[2], reqs[3], reqs[4],
4181                                    reqs[5], reqs[6], reqs[7], reqs[8],
4182                                    reqs[9], reqs[10], reqs[11], reqs[12],
4183                                    reqs[13], reqs[14], reqs[15], reqs[16],
4184                                    reqs[17]);
4185                        }
4186                } else {
4187                        break;
4188                }
4189                bzero((caddr_t)pkt, pkt_size);
4190        }

4192        if (rval != QL_SUCCESS) {
4193                EL(ha, "failed=%xh\n", rval);
4194                rval = 0;
4195        } else {
```

```
4196                 QL_PRINT_9(CE_CONT, "(%d): LUN list\n", ha->instance);
4197                 QL_DUMP_9(rpt, 8, rpt->hdr.len + 8);
4198                 rval = (int)(BE_32(rpt->hdr.len) / 8);
4199         }

4201         kmem_free(pkt, pkt_size);
4202         ql_free_dma_resource(ha, &dma_mem);

4204         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

4206         return (rval);
4207 }

4209 /*
4210  * ql_inq_scan
4211  *      Get numbers of LUNS using inquiry command.
4212  *
4213  * Input:
4214  *      ha:             adapter state pointer.
4215  *      tq:             target queue pointer.
4216  *      count:          scan for the number of existing LUNs.
4217  *
4218  * Returns:
4219  *      Number of LUNs.
4220  *
4221  * Context:
4222  *      Kernel context.
4223  */
4224 static int
4225 ql_inq_scan(ql_adapter_state_t *ha, ql_tgt_t *tq, int count)
4226 {
4227         int             lun, cnt, rval;
4228         ql_mbx_iocb_t   *pkt;
4229         uint8_t         *inq;
4230         uint32_t        pkt_size;

4232         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

4234         pkt_size = sizeof (ql_mbx_iocb_t) + INQ_DATA_SIZE;
4235         pkt = kmem_zalloc(pkt_size, KM_SLEEP);
4278         if (pkt == NULL) {
4279                 EL(ha, "failed, kmem_zalloc\n");
4280                 return (0);
4281         }
4236         inq = (uint8_t *)((caddr_t)pkt + sizeof (ql_mbx_iocb_t));

4238         cnt = 0;
4239         for (lun = 0; lun < MAX_LUNS; lun++) {

4241                 if (DRIVER_SUSPENDED(ha)) {
4242                         rval = QL_LOOP_DOWN;
4243                         cnt = 0;
4244                         break;
4245                 }

4247                 rval = ql_inq(ha, tq, lun, pkt, INQ_DATA_SIZE);
4248                 if (rval == QL_SUCCESS) {
4249                         switch (*inq) {
4250                         case DTYPE_DIRECT:
4251                         case DTYPE_PROCESSOR:   /* Appliance. */
4252                         case DTYPE_WORM:
4253                         case DTYPE_RODIRECT:
4254                         case DTYPE_SCANNER:
4255                         case DTYPE_OPTICAL:
4256                         case DTYPE_CHANGER:
4257                         case DTYPE_ESI:
```



```
4258                                 cnt++;
4259                                 break;
4260                         case DTYPE_SEQUENTIAL:
4261                                 cnt++;
4262                                 tq->flags |= TQF_TAPE_DEVICE;
4263                                 break;
4264                         default:
4265                                 QL_PRINT_9(CE_CONT, "(%d): failed, "
4266                                     "unsupported device id=%xh, lun=%d, "
4267                                     "type=%xh\n", ha->instance, tq->loop_id,
4268                                     lun, *inq);
4269                                 break;
4270                         }

4272                         if (*inq == DTYPE_ESI || cnt >= count) {
4273                                 break;
4274                         }
4275                 } else if (rval == QL_ABORTED || rval == QL_FUNCTION_TIMEOUT) {
4276                         cnt = 0;
4277                         break;
4278                 }
4279         }

4281         kmem_free(pkt, pkt_size);

4283         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

4285         return (cnt);
4286 }
```

*unchanged_portion_omitted_*

```
4626 /*
4627  * ql_24xx_flash_desc
4628  *      Get flash descriptor table.
4629  *
4630  * Input:
4631  *      ha:             adapter state pointer.
4632  *
4633  * Returns:
4634  *      ql local function return status code.
4635  *
4636  * Context:
4637  *      Kernel context.
4638  */
4639 static int
4640 ql_24xx_flash_desc(ql_adapter_state_t *ha)
4641 {
4642         uint32_t        cnt;
4643         uint16_t        chksum, *bp, data;
4644         int             rval;
4645         flash_desc_t    *fdesc;
4646         ql_xioctl_t     *xp = ha->xioctl;

4648         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

4650         if (ha->flash_desc_addr == 0) {
4651                 QL_PRINT_9(CE_CONT, "(%d): desc ptr=0\n", ha->instance);
4652                 return (QL_FUNCTION_FAILED);
4653         }

4655         fdesc = kmem_zalloc(sizeof (flash_desc_t), KM_SLEEP);
4701         if ((fdesc = kmem_zalloc(sizeof (flash_desc_t), KM_SLEEP)) == NULL) {
4702                 EL(ha, "kmem_zalloc=null\n");
4703                 return (QL_MEMORY_ALLOC_FAILED);
4704         }
4656         rval = ql_dump_fcode(ha, (uint8_t *)fdesc, sizeof (flash_desc_t),
```

```
4657             ha->flash_desc_addr << 2);
4658         if (rval != QL_SUCCESS) {
4659                 EL(ha, "read status=%xh\n", rval);
4660                 kmem_free(fdesc, sizeof (flash_desc_t));
4661                 return (rval);
4662         }

4664         chksum = 0;
4665         bp = (uint16_t *)fdesc;
4666         for (cnt = 0; cnt < (sizeof (flash_desc_t)) / 2; cnt++) {
4667                 data = *bp++;
4668                 LITTLE_ENDIAN_16(&data);
4669                 chksum += data;
4670         }

4672         LITTLE_ENDIAN_32(&fdesc->flash_valid);
4673         LITTLE_ENDIAN_16(&fdesc->flash_version);
4674         LITTLE_ENDIAN_16(&fdesc->flash_len);
4675         LITTLE_ENDIAN_16(&fdesc->flash_checksum);
4676         LITTLE_ENDIAN_16(&fdesc->flash_manuf);
4677         LITTLE_ENDIAN_16(&fdesc->flash_id);
4678         LITTLE_ENDIAN_32(&fdesc->block_size);
4679         LITTLE_ENDIAN_32(&fdesc->alt_block_size);
4680         LITTLE_ENDIAN_32(&fdesc->flash_size);
4681         LITTLE_ENDIAN_32(&fdesc->write_enable_data);
4682         LITTLE_ENDIAN_32(&fdesc->read_timeout);

4684         /* flash size in desc table is in 1024 bytes */
4685         fdesc->flash_size = fdesc->flash_size * 0x400;

4687         if (chksum != 0 || fdesc->flash_valid != FLASH_DESC_VAILD ||
4688             fdesc->flash_version != FLASH_DESC_VERSION) {
4689                 EL(ha, "invalid descriptor table\n");
4690                 kmem_free(fdesc, sizeof (flash_desc_t));
4691                 return (QL_FUNCTION_FAILED);
4692         }

4694         bcopy(fdesc, &xp->fdesc, sizeof (flash_desc_t));
4695         kmem_free(fdesc, sizeof (flash_desc_t));

4697         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

4699         return (QL_SUCCESS);
4700 }
```

unchanged_portion_omitted

```
5342 /*
5343  * ql_set_rnid_parameters
5344  *      Set RNID parameters.
5345  *
5346  * Input:
5347  *      ha:     adapter state pointer.
5348  *      cmd:    User space CT arguments pointer.
5349  *      mode:   flags.
5350  */
5351 static void
5352 ql_set_rnid_parameters(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
5353 {
5354         EXT_SET_RNID_REQ        tmp_set;
5355         EXT_RNID_DATA           *tmp_buf;
5356         int                     rval = 0;

5358         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

5360         if (DRIVER_SUSPENDED(ha)) {
5361                 EL(ha, "failed, LOOP_NOT_READY\n");
```



```
5362                 cmd->Status = EXT_STATUS_BUSY;
5363                 cmd->ResponseLen = 0;
5364                 return;
5365         }

5367         cmd->ResponseLen = 0; /* NO response to caller. */
5368         if (cmd->RequestLen != sizeof (EXT_SET_RNID_REQ)) {
5369                 /* parameter error */
5370                 EL(ha, "failed, RequestLen < EXT_SET_RNID_REQ, Len=%xh\n",
5371                     cmd->RequestLen);
5372                 cmd->Status = EXT_STATUS_INVALID_PARAM;
5373                 cmd->DetailStatus = EXT_DSTATUS_REQUEST_LEN;
5374                 cmd->ResponseLen = 0;
5375                 return;
5376         }

5378         rval = ddi_copyin((void*)(uintptr_t)cmd->RequestAdr, &tmp_set,
5379             cmd->RequestLen, mode);
5380         if (rval != 0) {
5381                 EL(ha, "failed, ddi_copyin\n");
5382                 cmd->Status = EXT_STATUS_COPY_ERR;
5383                 cmd->ResponseLen = 0;
5384                 return;
5385         }

5387         /* Allocate memory for command. */
5388         tmp_buf = kmem_zalloc(sizeof (EXT_RNID_DATA), KM_SLEEP);
5438         if (tmp_buf == NULL) {
5439                 EL(ha, "failed, kmem_zalloc\n");
5440                 cmd->Status = EXT_STATUS_NO_MEMORY;
5441                 cmd->ResponseLen = 0;
5442                 return;
5443         }

5390         rval = ql_get_rnid_params(ha, sizeof (EXT_RNID_DATA),
5391             (caddr_t)tmp_buf);
5392         if (rval != QL_SUCCESS) {
5393                 /* error */
5394                 EL(ha, "failed, get_rnid_params_mbx=%xh\n", rval);
5395                 kmem_free(tmp_buf, sizeof (EXT_RNID_DATA));
5396                 cmd->Status = EXT_STATUS_ERR;
5397                 cmd->ResponseLen = 0;
5398                 return;
5399         }

5401         /* Now set the requested params. */
5402         bcopy(tmp_set.IPVersion, tmp_buf->IPVersion, 2);
5403         bcopy(tmp_set.UDPPortNumber, tmp_buf->UDPPortNumber, 2);
5404         bcopy(tmp_set.IPAddress, tmp_buf->IPAddress, 16);

5406         rval = ql_set_rnid_params(ha, sizeof (EXT_RNID_DATA),
5407             (caddr_t)tmp_buf);
5408         if (rval != QL_SUCCESS) {
5409                 /* error */
5410                 EL(ha, "failed, set_rnid_params_mbx=%xh\n", rval);
5411                 cmd->Status = EXT_STATUS_ERR;
5412                 cmd->ResponseLen = 0;
5413         }

5415         kmem_free(tmp_buf, sizeof (EXT_RNID_DATA));

5417         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
5418 }

5420 /*
5421  * ql_get_rnid_parameters
```

```
5422  *      Get RNID parameters.
5423  *
5424  * Input:
5425  *      ha:     adapter state pointer.
5426  *      cmd:    User space CT arguments pointer.
5427  *      mode:   flags.
5428  */
5429 static void
5430 ql_get_rnid_parameters(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
5431 {
5432         EXT_RNID_DATA   *tmp_buf;
5433         uint32_t        rval;

5435         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

5437         if (DRIVER_SUSPENDED(ha)) {
5438                 EL(ha, "failed, LOOP_NOT_READY\n");
5439                 cmd->Status = EXT_STATUS_BUSY;
5440                 cmd->ResponseLen = 0;
5441                 return;
5442         }

5444         /* Allocate memory for command. */
5445         tmp_buf = kmem_zalloc(sizeof (EXT_RNID_DATA), KM_SLEEP);
5501         if (tmp_buf == NULL) {
5502                 EL(ha, "failed, kmem_zalloc\n");
5503                 cmd->Status = EXT_STATUS_NO_MEMORY;
5504                 cmd->ResponseLen = 0;
5505                 return;
5506         }

5447         /* Send command */
5448         rval = ql_get_rnid_params(ha, sizeof (EXT_RNID_DATA),
5449             (caddr_t)tmp_buf);
5450         if (rval != QL_SUCCESS) {
5451                 /* error */
5452                 EL(ha, "failed, get_rnid_params_mbx=%xh\n", rval);
5453                 kmem_free(tmp_buf, sizeof (EXT_RNID_DATA));
5454                 cmd->Status = EXT_STATUS_ERR;
5455                 cmd->ResponseLen = 0;
5456                 return;
5457         }

5459         /* Copy the response */
5460         if (ql_send_buffer_data((caddr_t)tmp_buf,
5461             (caddr_t)(uintptr_t)cmd->ResponseAdr,
5462             sizeof (EXT_RNID_DATA), mode) != sizeof (EXT_RNID_DATA)) {
5463                 EL(ha, "failed, ddi_copyout\n");
5464                 cmd->Status = EXT_STATUS_COPY_ERR;
5465                 cmd->ResponseLen = 0;
5466         } else {
5467                 QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
5468                 cmd->ResponseLen = sizeof (EXT_RNID_DATA);
5469         }

5471         kmem_free(tmp_buf, sizeof (EXT_RNID_DATA));
5472 }
```

*unchanged_portion_omitted_*

```
5535 /*
5536  * ql_get_statistics
5537  *      Performs EXT_SC_GET_STATISTICS subcommand. of EXT_CC_GET_DATA.
5538  *
5539  * Input:
5540  *      ha:     adapter state pointer.
5541  *      cmd:    Local EXT_IOCTL cmd struct pointer.
```



```
5542  *      mode:   flags.
5543  *
5544  * Returns:
5545  *      None, request status indicated in cmd->Status.
5546  *
5547  * Context:
5548  *      Kernel context.
5549  */
5550 static void
5551 ql_get_statistics(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
5552 {
5553         EXT_HBA_PORT_STAT       ps = {0};
5554         ql_link_stats_t         *ls;
5555         int                     rval;
5556         ql_xioctl_t             *xp = ha->xioctl;
5557         int                     retry = 10;

5559         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

5561         while (ha->task_daemon_flags &
5562             (ABORT_ISP_ACTIVE | LOOP_RESYNC_ACTIVE | DRIVER_STALL)) {
5563                 ql_delay(ha, 10000000); /* 10 second delay */

5565                 retry--;

5567                 if (retry == 0) { /* effectively 100 seconds */
5568                         EL(ha, "failed, LOOP_NOT_READY\n");
5569                         cmd->Status = EXT_STATUS_BUSY;
5570                         cmd->ResponseLen = 0;
5571                         return;
5572                 }
5573         }

5575         /* Allocate memory for command. */
5576         ls = kmem_zalloc(sizeof (ql_link_stats_t), KM_SLEEP);
5638         if (ls == NULL) {
5639                 EL(ha, "failed, kmem_zalloc\n");
5640                 cmd->Status = EXT_STATUS_NO_MEMORY;
5641                 cmd->ResponseLen = 0;
5642                 return;
5643         }

5578         /*
5579          * I think these are supposed to be port statistics
5580          * the loop ID or port ID should be in cmd->Instance.
5581          */
5582         rval = ql_get_status_counts(ha, (uint16_t)
5583             (ha->task_daemon_flags & LOOP_DOWN ? 0xFF : ha->loop_id),
5584             sizeof (ql_link_stats_t), (caddr_t)ls, 0);
5585         if (rval != QL_SUCCESS) {
5586                 EL(ha, "failed, get_link_status=%xh, id=%xh\n", rval,
5587                     ha->loop_id);
5588                 cmd->Status = EXT_STATUS_MAILBOX;
5589                 cmd->DetailStatus = rval;
5590                 cmd->ResponseLen = 0;
5591         } else {
5592                 ps.ControllerErrorCount = xp->ControllerErrorCount;
5593                 ps.DeviceErrorCount = xp->DeviceErrorCount;
5594                 ps.IoCount = (uint32_t)(xp->IOInputRequests +
5595                     xp->IOOutputRequests + xp->IOControlRequests);
5596                 ps.MBytesCount = (uint32_t)(xp->IOInputMByteCnt +
5597                     xp->IOOutputMByteCnt);
5598                 ps.LipResetCount = xp->TotalLipResets;
5599                 ps.InterruptCount = xp->TotalInterrupts;
5600                 ps.LinkFailureCount = LE_32(ls->link_fail_cnt);
5601                 ps.LossOfSyncCount = LE_32(ls->sync_loss_cnt);
```

```
5602                ps.LossOfSignalsCount = LE_32(ls->signal_loss_cnt);
5603                ps.PrimitiveSeqProtocolErrorCount = LE_32(ls->prot_err_cnt);
5604                ps.InvalidTransmissionWordCount = LE_32(ls->inv_xmit_cnt);
5605                ps.InvalidCRCCount = LE_32(ls->inv_crc_cnt);

5607                rval = ddi_copyout((void *)&ps,
5608                    (void *)(uintptr_t)cmd->ResponseAdr,
5609                    sizeof (EXT_HBA_PORT_STAT), mode);
5610                if (rval != 0) {
5611                        EL(ha, "failed, ddi_copyout\n");
5612                        cmd->Status = EXT_STATUS_COPY_ERR;
5613                        cmd->ResponseLen = 0;
5614                } else {
5615                        cmd->ResponseLen = sizeof (EXT_HBA_PORT_STAT);
5616                }
5617        }

5619        kmem_free(ls, sizeof (ql_link_stats_t));

5621        QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
5622 }

5624 /*
5625  * ql_get_statistics_fc
5626  *      Performs EXT_SC_GET_FC_STATISTICS subcommand. of EXT_CC_GET_DATA.
5627  *
5628  * Input:
5629  *      ha:     adapter state pointer.
5630  *      cmd:    Local EXT_IOCTL cmd struct pointer.
5631  *      mode:   flags.
5632  *
5633  * Returns:
5634  *      None, request status indicated in cmd->Status.
5635  *
5636  * Context:
5637  *      Kernel context.
5638  */
5639 static void
5640 ql_get_statistics_fc(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
5641 {
5642        EXT_HBA_PORT_STAT       ps = {0};
5643        ql_link_stats_t         *ls;
5644        int                     rval;
5645        uint16_t                qlnt;
5646        EXT_DEST_ADDR           pextdestaddr;
5647        uint8_t                 *name;
5648        ql_tgt_t                *tq = NULL;
5649        int                     retry = 10;

5651        QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

5653        if (ddi_copyin((void *)(uintptr_t)cmd->RequestAdr,
5654            (void *)&pextdestaddr, sizeof (EXT_DEST_ADDR), mode) != 0) {
5655                EL(ha, "failed, ddi_copyin\n");
5656                cmd->Status = EXT_STATUS_COPY_ERR;
5657                cmd->ResponseLen = 0;
5658                return;
5659        }

5661        qlnt = QLNT_PORT;
5662        name = pextdestaddr.DestAddr.WWPN;

5664        QL_PRINT_9(CE_CONT, "(%d): wwpn=%02x%02x%02x%02x%02x%02x%02x%02x\n",
5665            ha->instance, name[0], name[1], name[2], name[3], name[4],
5666            name[5], name[6], name[7]);
```



```
5668        tq = ql_find_port(ha, name, qlnt);

5670        if (tq == NULL || !VALID_TARGET_ID(ha, tq->loop_id)) {
5671                EL(ha, "failed, fc_port not found\n");
5672                cmd->Status = EXT_STATUS_DEV_NOT_FOUND;
5673                cmd->ResponseLen = 0;
5674                return;
5675        }

5677        while (ha->task_daemon_flags &
5678            (ABORT_ISP_ACTIVE | LOOP_RESYNC_ACTIVE  | DRIVER_STALL)) {
5679                ql_delay(ha, 10000000); /* 10 second delay */

5681                retry--;

5683                if (retry == 0) { /* effectively 100 seconds */
5684                        EL(ha, "failed, LOOP_NOT_READY\n");
5685                        cmd->Status = EXT_STATUS_BUSY;
5686                        cmd->ResponseLen = 0;
5687                        return;
5688                }
5689        }

5691        /* Allocate memory for command. */
5692        ls = kmem_zalloc(sizeof (ql_link_stats_t), KM_SLEEP);
5760        if (ls == NULL) {
5761                EL(ha, "failed, kmem_zalloc\n");
5762                cmd->Status = EXT_STATUS_NO_MEMORY;
5763                cmd->ResponseLen = 0;
5764                return;
5765        }

5694        rval = ql_get_link_status(ha, tq->loop_id, sizeof (ql_link_stats_t),
5695            (caddr_t)ls, 0);
5696        if (rval != QL_SUCCESS) {
5697                EL(ha, "failed, get_link_status=%xh, d_id=%xh\n", rval,
5698                    tq->d_id.b24);
5699                cmd->Status = EXT_STATUS_MAILBOX;
5700                cmd->DetailStatus = rval;
5701                cmd->ResponseLen = 0;
5702        } else {
5703                ps.LinkFailureCount = LE_32(ls->link_fail_cnt);
5704                ps.LossOfSyncCount = LE_32(ls->sync_loss_cnt);
5705                ps.LossOfSignalsCount = LE_32(ls->signal_loss_cnt);
5706                ps.PrimitiveSeqProtocolErrorCount = LE_32(ls->prot_err_cnt);
5707                ps.InvalidTransmissionWordCount = LE_32(ls->inv_xmit_cnt);
5708                ps.InvalidCRCCount = LE_32(ls->inv_crc_cnt);

5710                rval = ddi_copyout((void *)&ps,
5711                    (void *)(uintptr_t)cmd->ResponseAdr,
5712                    sizeof (EXT_HBA_PORT_STAT), mode);

5714                if (rval != 0) {
5715                        EL(ha, "failed, ddi_copyout\n");
5716                        cmd->Status = EXT_STATUS_COPY_ERR;
5717                        cmd->ResponseLen = 0;
5718                } else {
5719                        cmd->ResponseLen = sizeof (EXT_HBA_PORT_STAT);
5720                }
5721        }

5723        kmem_free(ls, sizeof (ql_link_stats_t));

5725        QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
5726 }
```

_____unchanged_portion_omitted_

```
6878 /*
6879  * ql_flash_layout_table
6880  *      Obtains flash addresses from table
6881  *
6882  * Input:
6883  *      ha:             adapter state pointer.
6884  *      flt_paddr:      flash layout pointer address.
6885  *
6886  * Context:
6887  *      Kernel context.
6888  */
6889 static void
6890 ql_flash_layout_table(ql_adapter_state_t *ha, uint32_t flt_paddr)
6891 {
6892         ql_flt_ptr_t    *fptr;
6893         uint8_t         *bp;
6894         int             rval;
6895         uint32_t        len, faddr, cnt;
6896         uint16_t        chksum, w16;

6898         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

6900         /* Process flash layout table header */
6901         len = sizeof (ql_flt_ptr_t);
6902         bp = kmem_zalloc(len, KM_SLEEP);
6975         if ((bp = kmem_zalloc(len, KM_SLEEP)) == NULL) {
6976                 EL(ha, "kmem_zalloc=null\n");
6977                 return;
6978         }

6904         /* Process pointer to flash layout table */
6905         if ((rval = ql_dump_fcode(ha, bp, len, flt_paddr)) != QL_SUCCESS) {
6906                 EL(ha, "fptr dump_flash pos=%xh, status=%xh\n", flt_paddr,
6907                     rval);
6908                 kmem_free(bp, len);
6909                 return;
6910         }
6911         fptr = (ql_flt_ptr_t *)bp;

6913         /* Verify pointer to flash layout table. */
6914         for (chksum = 0, cnt = 0; cnt < len; cnt += 2) {
6915                 w16 = (uint16_t)CHAR_TO_SHORT(bp[cnt], bp[cnt + 1]);
6916                 chksum += w16;
6917         }
6918         if (chksum != 0 || fptr->sig[0] != 'Q' || fptr->sig[1] != 'F' ||
6919             fptr->sig[2] != 'L' || fptr->sig[3] != 'T') {
6920                 EL(ha, "ptr chksum=%xh, sig=%c%c%c%c\n", chksum, fptr->sig[0],
6921                     fptr->sig[1], fptr->sig[2], fptr->sig[3]);
6922                 kmem_free(bp, len);
6923                 return;
6924         }
6925         faddr = CHAR_TO_LONG(fptr->addr[0], fptr->addr[1], fptr->addr[2],
6926             fptr->addr[3]);

6928         kmem_free(bp, len);

6930         ql_process_flt(ha, faddr);

6932         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
6933 }

6935 /*
6936  * ql_process_flt
6937  *      Obtains flash addresses from flash layout table
6938  *
```



```
6939  * Input:
6940  *      ha:     adapter state pointer.
6941  *      faddr:  flash layout table byte address.
6942  *
6943  * Context:
6944  *      Kernel context.
6945  */
6946 static void
6947 ql_process_flt(ql_adapter_state_t *ha, uint32_t faddr)
6948 {
6949         ql_flt_hdr_t    *fhdr;
6950         ql_flt_region_t *frgn;
6951         uint8_t         *bp, *eaddr, nv_rg, vpd_rg;
6952         int             rval;
6953         uint32_t        len, cnt, fe_addr;
6954         uint16_t        chksum, w16;

6956         QL_PRINT_9(CE_CONT, "(%d): started faddr=%xh\n", ha->instance, faddr);

6958         /* Process flash layout table header */
6959         bp = kmem_zalloc(FLASH_LAYOUT_TABLE_SIZE, KM_SLEEP);
7035         if ((bp = kmem_zalloc(FLASH_LAYOUT_TABLE_SIZE, KM_SLEEP)) == NULL) {
7036                 EL(ha, "kmem_zalloc=null\n");
7037                 return;
7038         }
6960         fhdr = (ql_flt_hdr_t *)bp;

6962         /* Process flash layout table. */
6963         if ((rval = ql_dump_fcode(ha, bp, FLASH_LAYOUT_TABLE_SIZE, faddr)) !=
6964             QL_SUCCESS) {
6965                 EL(ha, "fhdr dump_flash pos=%xh, status=%xh\n", faddr, rval);
6966                 kmem_free(bp, FLASH_LAYOUT_TABLE_SIZE);
6967                 return;
6968         }

6970         /* Verify flash layout table. */
6971         len = (uint32_t)(CHAR_TO_SHORT(fhdr->len[0], fhdr->len[1]) +
6972             sizeof (ql_flt_hdr_t) + sizeof (ql_flt_region_t));
6973         if (len > FLASH_LAYOUT_TABLE_SIZE) {
6974                 chksum = 0xffff;
6975         } else {
6976                 for (chksum = 0, cnt = 0; cnt < len; cnt += 2) {
6977                         w16 = (uint16_t)CHAR_TO_SHORT(bp[cnt], bp[cnt + 1]);
6978                         chksum += w16;
6979                 }
6980         }
6981         w16 = CHAR_TO_SHORT(fhdr->version[0], fhdr->version[1]);
6982         if (chksum != 0 || w16 != 1) {
6983                 EL(ha, "table chksum=%xh, version=%d\n", chksum, w16);
6984                 kmem_free(bp, FLASH_LAYOUT_TABLE_SIZE);
6985                 return;
6986         }
6987         eaddr = bp + len;

6989         /* Process Function/Port Configuration Map. */
6990         nv_rg = vpd_rg = 0;
6991         if (CFG_IST(ha, CFG_CTRL_8021)) {
6992                 uint16_t        i;
6993                 uint8_t         *mbp = eaddr;
6994                 ql_fp_cfg_map_t *cmp = (ql_fp_cfg_map_t *)mbp;

6996                 len = (uint32_t)(CHAR_TO_SHORT(cmp->hdr.len[0],
6997                     cmp->hdr.len[1]));
6998                 if (len > FLASH_LAYOUT_TABLE_SIZE) {
6999                         chksum = 0xffff;
7000                 } else {
```

```
7001                        for (chksum = 0, cnt = 0; cnt < len; cnt += 2) {
7002                                w16 = (uint16_t)CHAR_TO_SHORT(mbp[cnt],
7003                                    mbp[cnt + 1]);
7004                                chksum += w16;
7005                        }
7006                }
7007                w16 = CHAR_TO_SHORT(cmp->hdr.version[0], cmp->hdr.version[1]);
7008                if (chksum != 0 || w16 != 1 ||
7009                    cmp->hdr.Signature[0] != 'F' ||
7010                    cmp->hdr.Signature[1] != 'P' ||
7011                    cmp->hdr.Signature[2] != 'C' ||
7012                    cmp->hdr.Signature[3] != 'M') {
7013                        EL(ha, "cfg_map chksum=%xh, version=%d, "
7014                            "sig=%c%c%c%c\n", chksum, w16,
7015                            cmp->hdr.Signature[0], cmp->hdr.Signature[1],
7016                            cmp->hdr.Signature[2], cmp->hdr.Signature[3]);
7017                } else {
7018                        cnt = (uint16_t)
7019                            (CHAR_TO_SHORT(cmp->hdr.NumberEntries[0],
7020                            cmp->hdr.NumberEntries[1]));
7021                        /* Locate entry for function. */
7022                        for (i = 0; i < cnt; i++) {
7023                                if (cmp->cfg[i].FunctionType == FT_FC &&
7024                                    cmp->cfg[i].FunctionNumber[0] ==
7025                                    ha->function_number &&
7026                                    cmp->cfg[i].FunctionNumber[1] == 0) {
7027                                        nv_rg = cmp->cfg[i].ConfigRegion;
7028                                        vpd_rg = cmp->cfg[i].VpdRegion;
7029                                        break;
7030                                }
7031                        }

7033                        if (nv_rg == 0 || vpd_rg == 0) {
7034                                EL(ha, "cfg_map nv_rg=%d, vpd_rg=%d\n", nv_rg,
7035                                    vpd_rg);
7036                                nv_rg = vpd_rg = 0;
7037                        }
7038                }
7039        }

7041        /* Process flash layout table regions */
7042        for (frgn = (ql_flt_region_t *)(bp + sizeof (ql_flt_hdr_t));
7043            (uint8_t *)frgn < eaddr; frgn++) {
7044                faddr = CHAR_TO_LONG(frgn->beg_addr[0], frgn->beg_addr[1],
7045                    frgn->beg_addr[2], frgn->beg_addr[3]);
7046                faddr >>= 2;
7047                fe_addr = CHAR_TO_LONG(frgn->end_addr[0], frgn->end_addr[1],
7048                    frgn->end_addr[2], frgn->end_addr[3]);
7049                fe_addr >>= 2;

7051                switch (frgn->region) {
7052                case FLASH_8021_BOOTLOADER_REGION:
7053                        ha->bootloader_addr = faddr;
7054                        ha->bootloader_size = (fe_addr - faddr) + 1;
7055                        QL_PRINT_9(CE_CONT, "(%d): bootloader_addr=%xh, "
7056                            "size=%xh\n", ha->instance, faddr,
7057                            ha->bootloader_size);
7058                        break;
7059                case FLASH_FW_REGION:
7060                case FLASH_8021_FW_REGION:
7061                        ha->flash_fw_addr = faddr;
7062                        ha->flash_fw_size = (fe_addr - faddr) + 1;
7063                        QL_PRINT_9(CE_CONT, "(%d): flash_fw_addr=%xh, "
7064                            "size=%xh\n", ha->instance, faddr,
7065                            ha->flash_fw_size);
7066                        break;
```



```
7067                case FLASH_GOLDEN_FW_REGION:
7068                case FLASH_8021_GOLDEN_FW_REGION:
7069                        ha->flash_golden_fw_addr = faddr;
7070                        QL_PRINT_9(CE_CONT, "(%d): flash_golden_fw_addr=%xh\n",
7071                            ha->instance, faddr);
7072                        break;
7073                case FLASH_8021_VPD_REGION:
7074                        if (!vpd_rg || vpd_rg == FLASH_8021_VPD_REGION) {
7075                                ha->flash_vpd_addr = faddr;
7076                                QL_PRINT_9(CE_CONT, "(%d): 8021_flash_vpd_"
7077                                    "addr=%xh\n", ha->instance, faddr);
7078                        }
7079                        break;
7080                case FLASH_VPD_0_REGION:
7081                        if (vpd_rg) {
7082                                if (vpd_rg == FLASH_VPD_0_REGION) {
7083                                        ha->flash_vpd_addr = faddr;
7084                                        QL_PRINT_9(CE_CONT, "(%d): vpd_rg  "
7085                                            "flash_vpd_addr=%xh\n",
7086                                            ha->instance, faddr);
7087                                }
7088                        } else if (!(ha->flags & FUNCTION_1) &&
7089                            !(CFG_IST(ha, CFG_CTRL_8021))) {
7090                                ha->flash_vpd_addr = faddr;
7091                                QL_PRINT_9(CE_CONT, "(%d): flash_vpd_addr=%xh"
7092                                    "\n", ha->instance, faddr);
7093                        }
7094                        break;
7095                case FLASH_NVRAM_0_REGION:
7096                        if (nv_rg) {
7097                                if (nv_rg == FLASH_NVRAM_0_REGION) {
7098                                        ADAPTER_STATE_LOCK(ha);
7099                                        ha->flags &= ~FUNCTION_1;
7100                                        ADAPTER_STATE_UNLOCK(ha);
7101                                        ha->flash_nvram_addr = faddr;
7102                                        QL_PRINT_9(CE_CONT, "(%d): nv_rg "
7103                                            "flash_nvram_addr=%xh\n",
7104                                            ha->instance, faddr);
7105                                }
7106                        } else if (!(ha->flags & FUNCTION_1)) {
7107                                ha->flash_nvram_addr = faddr;
7108                                QL_PRINT_9(CE_CONT, "(%d): flash_nvram_addr="
7109                                    "%xh\n", ha->instance, faddr);
7110                        }
7111                        break;
7112                case FLASH_VPD_1_REGION:
7113                        if (vpd_rg) {
7114                                if (vpd_rg == FLASH_VPD_1_REGION) {
7115                                        ha->flash_vpd_addr = faddr;
7116                                        QL_PRINT_9(CE_CONT, "(%d): vpd_rg "
7117                                            "flash_vpd_addr=%xh\n",
7118                                            ha->instance, faddr);
7119                                }
7120                        } else if (ha->flags & FUNCTION_1 &&
7121                            !(CFG_IST(ha, CFG_CTRL_8021))) {
7122                                ha->flash_vpd_addr = faddr;
7123                                QL_PRINT_9(CE_CONT, "(%d): flash_vpd_addr=%xh"
7124                                    "\n", ha->instance, faddr);
7125                        }
7126                        break;
7127                case FLASH_NVRAM_1_REGION:
7128                        if (nv_rg) {
7129                                if (nv_rg == FLASH_NVRAM_1_REGION) {
7130                                        ADAPTER_STATE_LOCK(ha);
7131                                        ha->flags |= FUNCTION_1;
7132                                        ADAPTER_STATE_UNLOCK(ha);
```

```
7133                                    ha->flash_nvram_addr = faddr;
7134                                    QL_PRINT_9(CE_CONT, "(%d): nv_rg "
7135                                        "flash_nvram_addr=%xh\n",
7136                                        ha->instance, faddr);
7137                            }
7138                    } else if (ha->flags & FUNCTION_1) {
7139                            ha->flash_nvram_addr = faddr;
7140                            QL_PRINT_9(CE_CONT, "(%d): flash_nvram_addr="
7141                                "%xh\n", ha->instance, faddr);
7142                    }
7143                    break;
7144            case FLASH_DESC_TABLE_REGION:
7145                    if (!(CFG_IST(ha, CFG_CTRL_8021))) {
7146                            ha->flash_desc_addr = faddr;
7147                            QL_PRINT_9(CE_CONT, "(%d): flash_desc_addr="
7148                                "%xh\n", ha->instance, faddr);
7149                    }
7150                    break;
7151            case FLASH_ERROR_LOG_0_REGION:
7152                    if (!(ha->flags & FUNCTION_1)) {
7153                            ha->flash_errlog_start = faddr;
7154                            QL_PRINT_9(CE_CONT, "(%d): flash_errlog_addr="
7155                                "%xh\n", ha->instance, faddr);
7156                    }
7157                    break;
7158            case FLASH_ERROR_LOG_1_REGION:
7159                    if (ha->flags & FUNCTION_1) {
7160                            ha->flash_errlog_start = faddr;
7161                            QL_PRINT_9(CE_CONT, "(%d): flash_errlog_addr="
7162                                "%xh\n", ha->instance, faddr);
7163                    }
7164                    break;
7165            default:
7166                    break;
7167            }
7168    }
7169    kmem_free(bp, FLASH_LAYOUT_TABLE_SIZE);

7171    QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
7172 }
```

*unchanged_portion_omitted_*

```
7567 /*
7568  * ql_get_fwexttrace
7569  *      Dumps f/w extended trace buffer
7570  *
7571  * Input:
7572  *      ha:     adapter state pointer.
7573  *      bp:     buffer address.
7574  *      mode:   flags
7575  *
7576  * Returns:
7577  *
7578  * Context:
7579  *      Kernel context.
7580  */
7581 /* ARGSUSED */
7582 static void
7583 ql_get_fwexttrace(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
7584 {
7585    int     rval;
7586    caddr_t payload;

7588    QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

7590    if (CFG_IST(ha, CFG_CTRL_24258081) == 0) {
```



```
7591            EL(ha, "invalid request for this HBA\n");
7592            cmd->Status = EXT_STATUS_INVALID_REQUEST;
7593            cmd->ResponseLen = 0;
7594            return;
7595    }

7597    if ((CFG_IST(ha, CFG_ENABLE_FWEXTTRACE) == 0) ||
7598        (ha->fwexttracebuf.bp == NULL)) {
7599            EL(ha, "f/w extended trace is not enabled\n");
7600            cmd->Status = EXT_STATUS_INVALID_REQUEST;
7601            cmd->ResponseLen = 0;
7602            return;
7603    }

7605    if (cmd->ResponseLen < FWEXTSIZE) {
7606            cmd->Status = EXT_STATUS_BUFFER_TOO_SMALL;
7607            cmd->DetailStatus = FWEXTSIZE;
7608            EL(ha, "failed, ResponseLen (%xh) < %xh (FWEXTSIZE)\n",
7609                cmd->ResponseLen, FWEXTSIZE);
7610            cmd->ResponseLen = 0;
7611            return;
7612    }

7614    /* Time Stamp */
7615    rval = ql_fw_etrace(ha, &ha->fwexttracebuf, FTO_INSERT_TIME_STAMP);
7616    if (rval != QL_SUCCESS) {
7617            EL(ha, "f/w extended trace insert"
7618                "time stamp failed: %xh\n", rval);
7619            cmd->Status = EXT_STATUS_ERR;
7620            cmd->ResponseLen = 0;
7621            return;
7622    }

7624    /* Disable Tracing */
7625    rval = ql_fw_etrace(ha, &ha->fwexttracebuf, FTO_EXT_TRACE_DISABLE);
7626    if (rval != QL_SUCCESS) {
7627            EL(ha, "f/w extended trace disable failed: %xh\n", rval);
7628            cmd->Status = EXT_STATUS_ERR;
7629            cmd->ResponseLen = 0;
7630            return;
7631    }

7633    /* Allocate payload buffer */
7634    payload = kmem_zalloc(FWEXTSIZE, KM_SLEEP);
7714    if (payload == NULL) {
7715            EL(ha, "failed, kmem_zalloc\n");
7716            cmd->Status = EXT_STATUS_NO_MEMORY;
7717            cmd->ResponseLen = 0;
7718            return;
7719    }

7636    /* Sync DMA buffer. */
7637    (void) ddi_dma_sync(ha->fwexttracebuf.dma_handle, 0,
7638        FWEXTSIZE, DDI_DMA_SYNC_FORKERNEL);

7640    /* Copy trace buffer data. */
7641    ddi_rep_get8(ha->fwexttracebuf.acc_handle, (uint8_t *)payload,
7642        (uint8_t *)ha->fwexttracebuf.bp, FWEXTSIZE,
7643        DDI_DEV_AUTOINCR);

7645    /* Send payload to application. */
7646    if (ql_send_buffer_data(payload, (caddr_t)(uintptr_t)cmd->ResponseAdr,
7647        cmd->ResponseLen, mode) != cmd->ResponseLen) {
7648            EL(ha, "failed, send_buffer_data\n");
7649            cmd->Status = EXT_STATUS_COPY_ERR;
7650            cmd->ResponseLen = 0;
```

```
7651         } else {
7652                 cmd->Status = EXT_STATUS_OK;
7653         }

7655         kmem_free(payload, FWEXTSIZE);

7657         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
7658 }

7660 /*
7661  * ql_get_fwfcetrace
7662  *      Dumps f/w fibre channel event trace buffer
7663  *
7664  * Input:
7665  *      ha:     adapter state pointer.
7666  *      bp:     buffer address.
7667  *      mode:   flags
7668  *
7669  * Returns:
7670  *
7671  * Context:
7672  *      Kernel context.
7673  */
7674 /* ARGSUSED */
7675 static void
7676 ql_get_fwfcetrace(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
7677 {
7678         int     rval;
7679         caddr_t payload;

7681         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

7683         if (CFG_IST(ha, CFG_CTRL_24258081) == 0) {
7684                 EL(ha, "invalid request for this HBA\n");
7685                 cmd->Status = EXT_STATUS_INVALID_REQUEST;
7686                 cmd->ResponseLen = 0;
7687                 return;
7688         }

7690         if ((CFG_IST(ha, CFG_ENABLE_FWFCETRACE) == 0) ||
7691             (ha->fwfcetracebuf.bp == NULL)) {
7692                 EL(ha, "f/w FCE trace is not enabled\n");
7693                 cmd->Status = EXT_STATUS_INVALID_REQUEST;
7694                 cmd->ResponseLen = 0;
7695                 return;
7696         }

7698         if (cmd->ResponseLen < FWFCESIZE) {
7699                 cmd->Status = EXT_STATUS_BUFFER_TOO_SMALL;
7700                 cmd->DetailStatus = FWFCESIZE;
7701                 EL(ha, "failed, ResponseLen (%xh) < %xh (FWFCESIZE)\n",
7702                     cmd->ResponseLen, FWFCESIZE);
7703                 cmd->ResponseLen = 0;
7704                 return;
7705         }

7707         /* Disable Tracing */
7708         rval = ql_fw_etrace(ha, &ha->fwfcetracebuf, FTO_FCE_TRACE_DISABLE);
7709         if (rval != QL_SUCCESS) {
7710                 EL(ha, "f/w FCE trace disable failed: %xh\n", rval);
7711                 cmd->Status = EXT_STATUS_ERR;
7712                 cmd->ResponseLen = 0;
7713                 return;
7714         }

7716         /* Allocate payload buffer */
```



```
7717         payload = kmem_zalloc(FWEXTSIZE, KM_SLEEP);
7803         if (payload == NULL) {
7804                 EL(ha, "failed, kmem_zalloc\n");
7805                 cmd->Status = EXT_STATUS_NO_MEMORY;
7806                 cmd->ResponseLen = 0;
7807                 return;
7808         }

7719         /* Sync DMA buffer. */
7720         (void) ddi_dma_sync(ha->fwfcetracebuf.dma_handle, 0,
7721             FWFCESIZE, DDI_DMA_SYNC_FORKERNEL);

7723         /* Copy trace buffer data. */
7724         ddi_rep_get8(ha->fwfcetracebuf.acc_handle, (uint8_t *)payload,
7725             (uint8_t *)ha->fwfcetracebuf.bp, FWFCESIZE,
7726             DDI_DEV_AUTOINCR);

7728         /* Send payload to application. */
7729         if (ql_send_buffer_data(payload, (caddr_t)(uintptr_t)cmd->ResponseAdr,
7730             cmd->ResponseLen, mode) != cmd->ResponseLen) {
7731                 EL(ha, "failed, send_buffer_data\n");
7732                 cmd->Status = EXT_STATUS_COPY_ERR;
7733                 cmd->ResponseLen = 0;
7734         } else {
7735                 cmd->Status = EXT_STATUS_OK;
7736         }

7738         kmem_free(payload, FWFCESIZE);

7740         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
7741 }
```

*unchanged_portion_omitted_*

```
7812 /*
7813  * ql_pci_dump
7814  *      Dumps PCI config data to application buffer.
7815  *
7816  * Input:
7817  *      ha = adapter state pointer.
7818  *      bp = user buffer address.
7819  *
7820  * Returns:
7821  *
7822  * Context:
7823  *      Kernel context.
7824  */
7825 int
7826 ql_pci_dump(ql_adapter_state_t *ha, uint32_t *bp, uint32_t pci_size, int mode)
7827 {
7828         uint32_t        pci_os;
7829         uint32_t        *ptr32, *org_ptr32;

7831         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

7833         ptr32 = kmem_zalloc(pci_size, KM_SLEEP);
7925         if (ptr32 == NULL) {
7926                 EL(ha, "failed kmem_zalloc\n");
7927                 return (ENOMEM);
7928         }

7835         /* store the initial value of ptr32 */
7836         org_ptr32 = ptr32;
7837         for (pci_os = 0; pci_os < pci_size; pci_os += 4) {
7838                 *ptr32 = (uint32_t)ql_pci_config_get32(ha, pci_os);
7839                 LITTLE_ENDIAN_32(ptr32);
7840                 ptr32++;
```

```
7841         }

7843         if (ddi_copyout((void *)org_ptr32, (void *)bp, pci_size, mode) !=
7844             0) {
7845                 EL(ha, "failed ddi_copyout\n");
7846                 kmem_free(org_ptr32, pci_size);
7847                 return (EFAULT);
7848         }

7850         QL_DUMP_9(org_ptr32, 8, pci_size);

7852         kmem_free(org_ptr32, pci_size);

7854         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);

7856         return (0);
7857 }
_____unchanged_portion_omitted_

7940 /*
7941  * ql_menlo_get_fw_version
7942  *      Get Menlo firmware version.
7943  *
7944  * Input:
7945  *      ha:     adapter state pointer.
7946  *      bp:     buffer address.
7947  *      mode:   flags
7948  *
7949  * Returns:
7950  *
7951  * Context:
7952  *      Kernel context.
7953  */
7954 static void
7955 ql_menlo_get_fw_version(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
7956 {
7957         int                             rval;
7958         ql_mbx_iocb_t                   *pkt;
7959         EXT_MENLO_GET_FW_VERSION        ver = {0};

7961         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

7963         if ((CFG_IST(ha, CFG_CTRL_MENLO)) == 0) {
7964                 EL(ha, "failed, invalid request for HBA\n");
7965                 cmd->Status = EXT_STATUS_INVALID_REQUEST;
7966                 cmd->ResponseLen = 0;
7967                 return;
7968         }

7970         if (cmd->ResponseLen < sizeof (EXT_MENLO_GET_FW_VERSION)) {
7971                 cmd->Status = EXT_STATUS_BUFFER_TOO_SMALL;
7972                 cmd->DetailStatus = sizeof (EXT_MENLO_GET_FW_VERSION);
7973                 EL(ha, "ResponseLen=%d < %d\n", cmd->ResponseLen,
7974                     sizeof (EXT_MENLO_GET_FW_VERSION));
7975                 cmd->ResponseLen = 0;
7976                 return;
7977         }

7979         /* Allocate packet. */
7980         pkt = kmem_zalloc(sizeof (ql_mbx_iocb_t), KM_SLEEP);
8076         if (pkt == NULL) {
8077                 EL(ha, "failed, kmem_zalloc\n");
8078                 cmd->Status = EXT_STATUS_NO_MEMORY;
8079                 cmd->ResponseLen = 0;
8080                 return;
8081         }
```



```
7982         pkt->mvfy.entry_type = VERIFY_MENLO_TYPE;
7983         pkt->mvfy.entry_count = 1;
7984         pkt->mvfy.options_status = LE_16(VMF_DO_NOT_UPDATE_FW);

7986         rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, sizeof (ql_mbx_iocb_t));
7987         LITTLE_ENDIAN_16(&pkt->mvfy.options_status);
7988         LITTLE_ENDIAN_16(&pkt->mvfy.failure_code);
7989         ver.FwVersion = LE_32(pkt->mvfy.fw_version);

7991         if (rval != QL_SUCCESS || (pkt->mvfy.entry_status & 0x3c) != 0 ||
7992             pkt->mvfy.options_status != CS_COMPLETE) {
7993                 /* Command error */
7994                 EL(ha, "failed, status=%xh, es=%xh, cs=%xh, fc=%xh\n", rval,
7995                     pkt->mvfy.entry_status & 0x3c, pkt->mvfy.options_status,
7996                     pkt->mvfy.failure_code);
7997                 cmd->Status = EXT_STATUS_ERR;
7998                 cmd->DetailStatus = rval != QL_SUCCESS ? rval :
7999                     QL_FUNCTION_FAILED;
8000                 cmd->ResponseLen = 0;
8001         } else if (ddi_copyout((void *)&ver,
8002             (void *)(uintptr_t)cmd->ResponseAdr,
8003             sizeof (EXT_MENLO_GET_FW_VERSION), mode) != 0) {
8004                 EL(ha, "failed, ddi_copyout\n");
8005                 cmd->Status = EXT_STATUS_COPY_ERR;
8006                 cmd->ResponseLen = 0;
8007         } else {
8008                 cmd->ResponseLen = sizeof (EXT_MENLO_GET_FW_VERSION);
8009         }

8011         kmem_free(pkt, sizeof (ql_mbx_iocb_t));

8013         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
8014 }

8016 /*
8017  * ql_menlo_update_fw
8018  *      Get Menlo update firmware.
8019  *
8020  * Input:
8021  *      ha:     adapter state pointer.
8022  *      bp:     buffer address.
8023  *      mode:   flags
8024  *
8025  * Returns:
8026  *
8027  * Context:
8028  *      Kernel context.
8029  */
8030 static void
8031 ql_menlo_update_fw(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
8032 {
8033         ql_mbx_iocb_t           *pkt;
8034         dma_mem_t               *dma_mem;
8035         EXT_MENLO_UPDATE_FW     fw;
8036         uint32_t                *ptr32;
8037         int                     rval;

8039         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

8041         if ((CFG_IST(ha, CFG_CTRL_MENLO)) == 0) {
8042                 EL(ha, "failed, invalid request for HBA\n");
8043                 cmd->Status = EXT_STATUS_INVALID_REQUEST;
8044                 cmd->ResponseLen = 0;
8045                 return;
8046         }
```

```
8048         /*
8049          * TODO: only vp_index 0 can do this (?)
8050          */
8051
8052         /*  Verify the size of request structure. */
8053         if (cmd->RequestLen < sizeof (EXT_MENLO_UPDATE_FW)) {
8054                 /* Return error */
8055                 EL(ha, "RequestLen=%d < %d\n", cmd->RequestLen,
8056                     sizeof (EXT_MENLO_UPDATE_FW));
8057                 cmd->Status = EXT_STATUS_INVALID_PARAM;
8058                 cmd->DetailStatus = EXT_DSTATUS_REQUEST_LEN;
8059                 cmd->ResponseLen = 0;
8060                 return;
8061         }
8062
8063         /* Get update fw request. */
8064         if (ddi_copyin((caddr_t)(uintptr_t)cmd->RequestAdr, (caddr_t)&fw,
8065             sizeof (EXT_MENLO_UPDATE_FW), mode) != 0) {
8066                 EL(ha, "failed, ddi_copyin\n");
8067                 cmd->Status = EXT_STATUS_COPY_ERR;
8068                 cmd->ResponseLen = 0;
8069                 return;
8070         }
8071
8072         /* Wait for I/O to stop and daemon to stall. */
8073         if (ql_suspend_hba(ha, 0) != QL_SUCCESS) {
8074                 EL(ha, "ql_stall_driver failed\n");
8075                 ql_restart_hba(ha);
8076                 cmd->Status = EXT_STATUS_BUSY;
8077                 cmd->ResponseLen = 0;
8078                 return;
8079         }
8080
8081         /* Allocate packet. */
8082         dma_mem = (dma_mem_t *)kmem_zalloc(sizeof (dma_mem_t), KM_SLEEP);
8083         if (dma_mem == NULL) {
8084                 EL(ha, "failed, kmem_zalloc\n");
8085                 cmd->Status = EXT_STATUS_NO_MEMORY;
8086                 cmd->ResponseLen = 0;
8087                 return;
8088         }
8089         pkt = kmem_zalloc(sizeof (ql_mbx_iocb_t), KM_SLEEP);
8090         if (pkt == NULL) {
8091                 EL(ha, "failed, kmem_zalloc\n");
8092                 kmem_free(dma_mem, sizeof (dma_mem_t));
8093                 ql_restart_hba(ha);
8094                 cmd->Status = EXT_STATUS_NO_MEMORY;
8095                 cmd->ResponseLen = 0;
8096                 return;
8097         }
8098
8099         /* Get DMA memory for the IOCB */
8100         if (ql_get_dma_mem(ha, dma_mem, fw.TotalByteCount, LITTLE_ENDIAN_DMA,
8101             QL_DMA_DATA_ALIGN) != QL_SUCCESS) {
8102                 cmn_err(CE_WARN, "%s(%d): request queue DMA memory "
8103                     "alloc failed", QL_NAME, ha->instance);
8104                 kmem_free(pkt, sizeof (ql_mbx_iocb_t));
8105                 kmem_free(dma_mem, sizeof (dma_mem_t));
8106                 ql_restart_hba(ha);
8107                 cmd->Status = EXT_STATUS_MS_NO_RESPONSE;
8108                 cmd->ResponseLen = 0;
8109                 return;
8110         }
8111
8112         /* Get firmware data. */
```



```
8113         if (ql_get_buffer_data((caddr_t)(uintptr_t)fw.pFwDataBytes, dma_mem->bp,
8114             fw.TotalByteCount, mode) != fw.TotalByteCount) {
8115                 EL(ha, "failed, get_buffer_data\n");
8116                 ql_free_dma_resource(ha, dma_mem);
8117                 kmem_free(pkt, sizeof (ql_mbx_iocb_t));
8118                 kmem_free(dma_mem, sizeof (dma_mem_t));
8119                 ql_restart_hba(ha);
8120                 cmd->Status = EXT_STATUS_COPY_ERR;
8121                 cmd->ResponseLen = 0;
8122                 return;
8123         }
8124
8125         /* Sync DMA buffer. */
8126         (void) ddi_dma_sync(dma_mem->dma_handle, 0, dma_mem->size,
8127             DDI_DMA_SYNC_FORDEV);
8128
8129         pkt->mvfy.entry_type = VERIFY_MENLO_TYPE;
8130         pkt->mvfy.entry_count = 1;
8131         pkt->mvfy.options_status = (uint16_t)LE_16(fw.Flags);
8132         ptr32 = dma_mem->bp;
8133         pkt->mvfy.fw_version = LE_32(ptr32[2]);
8134         pkt->mvfy.fw_size = LE_32(fw.TotalByteCount);
8135         pkt->mvfy.fw_sequence_size = LE_32(fw.TotalByteCount);
8136         pkt->mvfy.dseg_count = LE_16(1);
8137         pkt->mvfy.dseg_0_address[0] = (uint32_t)
8138             LE_32(LSD(dma_mem->cookie.dmac_laddress));
8139         pkt->mvfy.dseg_0_address[1] = (uint32_t)
8140             LE_32(MSD(dma_mem->cookie.dmac_laddress));
8141         pkt->mvfy.dseg_0_length = LE_32(fw.TotalByteCount);
8142
8143         rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, sizeof (ql_mbx_iocb_t));
8144         LITTLE_ENDIAN_16(&pkt->mvfy.options_status);
8145         LITTLE_ENDIAN_16(&pkt->mvfy.failure_code);
8146
8147         if (rval != QL_SUCCESS || (pkt->mvfy.entry_status & 0x3c) != 0 ||
8148             pkt->mvfy.options_status != CS_COMPLETE) {
8149                 /* Command error */
8150                 EL(ha, "failed, status=%xh, es=%xh, cs=%xh, fc=%xh\n", rval,
8151                     pkt->mvfy.entry_status & 0x3c, pkt->mvfy.options_status,
8152                     pkt->mvfy.failure_code);
8153                 cmd->Status = EXT_STATUS_ERR;
8154                 cmd->DetailStatus = rval != QL_SUCCESS ? rval :
8155                     QL_FUNCTION_FAILED;
8156                 cmd->ResponseLen = 0;
8157         }
8158
8159         ql_free_dma_resource(ha, dma_mem);
8160         kmem_free(pkt, sizeof (ql_mbx_iocb_t));
8161         kmem_free(dma_mem, sizeof (dma_mem_t));
8162         ql_restart_hba(ha);
8163
8164         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
8165 }
8166
8167 /*
8168  * ql_menlo_manage_info
8169  *      Get Menlo manage info.
8170  *
8171  * Input:
8172  *      ha:     adapter state pointer.
8173  *      bp:     buffer address.
8174  *      mode:   flags
8175  *
8176  * Returns:
8177  *
8178  * Context:
```

```
 *      Kernel context.
 */
static void
ql_menlo_manage_info(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
{
        ql_mbx_iocb_t           *pkt;
        dma_mem_t               *dma_mem = NULL;
        EXT_MENLO_MANAGE_INFO   info;
        int                     rval;

        QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);


        /* The call is only supported for Schultz right now */
        if (CFG_IST(ha, CFG_CTRL_8081)) {
                ql_get_xgmac_statistics(ha, cmd, mode);
                QL_PRINT_9(CE_CONT, "(%d): CFG_CTRL_81XX done\n",
                    ha->instance);
                return;
        }

        if (!CFG_IST(ha, CFG_CTRL_8081) || !CFG_IST(ha, CFG_CTRL_MENLO)) {
                EL(ha, "failed, invalid request for HBA\n");
                cmd->Status = EXT_STATUS_INVALID_REQUEST;
                cmd->ResponseLen = 0;
                return;
        }

        /*  Verify the size of request structure. */
        if (cmd->RequestLen < sizeof (EXT_MENLO_MANAGE_INFO)) {
                /* Return error */
                EL(ha, "RequestLen=%d < %d\n", cmd->RequestLen,
                    sizeof (EXT_MENLO_MANAGE_INFO));
                cmd->Status = EXT_STATUS_INVALID_PARAM;
                cmd->DetailStatus = EXT_DSTATUS_REQUEST_LEN;
                cmd->ResponseLen = 0;
                return;
        }

        /* Get manage info request. */
        if (ddi_copyin((caddr_t)(uintptr_t)cmd->RequestAdr,
            (caddr_t)&info, sizeof (EXT_MENLO_MANAGE_INFO), mode) != 0) {
                EL(ha, "failed, ddi_copyin\n");
                cmd->Status = EXT_STATUS_COPY_ERR;
                cmd->ResponseLen = 0;
                return;
        }

        /* Allocate packet. */
        pkt = kmem_zalloc(sizeof (ql_mbx_iocb_t), KM_SLEEP);
        if (pkt == NULL) {
                EL(ha, "failed, kmem_zalloc\n");
                ql_restart_driver(ha);
                cmd->Status = EXT_STATUS_NO_MEMORY;
                cmd->ResponseLen = 0;
                return;
        }

        pkt->mdata.entry_type = MENLO_DATA_TYPE;
        pkt->mdata.entry_count = 1;
        pkt->mdata.options_status = (uint16_t)LE_16(info.Operation);

        /* Get DMA memory for the IOCB */
        if (info.Operation == MENLO_OP_READ_MEM ||
            info.Operation == MENLO_OP_WRITE_MEM) {
                pkt->mdata.total_byte_count = LE_32(info.TotalByteCount);
```



```
                pkt->mdata.parameter_1 =
                    LE_32(info.Parameters.ap.MenloMemory.StartingAddr);
                dma_mem = (dma_mem_t *)kmem_zalloc(sizeof (dma_mem_t),
                    KM_SLEEP);
                if (dma_mem == NULL) {
                        EL(ha, "failed, kmem_zalloc\n");
                        kmem_free(pkt, sizeof (ql_mbx_iocb_t));
                        cmd->Status = EXT_STATUS_NO_MEMORY;
                        cmd->ResponseLen = 0;
                        return;
                }
                if (ql_get_dma_mem(ha, dma_mem, info.TotalByteCount,
                    LITTLE_ENDIAN_DMA, QL_DMA_DATA_ALIGN) != QL_SUCCESS) {
                        cmn_err(CE_WARN, "%s(%d): request queue DMA memory "
                            "alloc failed", QL_NAME, ha->instance);
                        kmem_free(dma_mem, sizeof (dma_mem_t));
                        kmem_free(pkt, sizeof (ql_mbx_iocb_t));
                        cmd->Status = EXT_STATUS_MS_NO_RESPONSE;
                        cmd->ResponseLen = 0;
                        return;
                }
                if (info.Operation == MENLO_OP_WRITE_MEM) {
                        /* Get data. */
                        if (ql_get_buffer_data(
                            (caddr_t)(uintptr_t)info.pDataBytes,
                            dma_mem->bp, info.TotalByteCount, mode) !=
                            info.TotalByteCount) {
                                EL(ha, "failed, get_buffer_data\n");
                                ql_free_dma_resource(ha, dma_mem);
                                kmem_free(dma_mem, sizeof (dma_mem_t));
                                kmem_free(pkt, sizeof (ql_mbx_iocb_t));
                                cmd->Status = EXT_STATUS_COPY_ERR;
                                cmd->ResponseLen = 0;
                                return;
                        }
                        (void) ddi_dma_sync(dma_mem->dma_handle, 0,
                            dma_mem->size, DDI_DMA_SYNC_FORDEV);
                }
                pkt->mdata.dseg_count = LE_16(1);
                pkt->mdata.dseg_0_address[0] = (uint32_t)
                    LE_32(LSD(dma_mem->cookie.dmac_laddress));
                pkt->mdata.dseg_0_address[1] = (uint32_t)
                    LE_32(MSD(dma_mem->cookie.dmac_laddress));
                pkt->mdata.dseg_0_length = LE_32(info.TotalByteCount);
        } else if (info.Operation & MENLO_OP_CHANGE_CONFIG) {
                pkt->mdata.parameter_1 =
                    LE_32(info.Parameters.ap.MenloConfig.ConfigParamID);
                pkt->mdata.parameter_2 =
                    LE_32(info.Parameters.ap.MenloConfig.ConfigParamData0);
                pkt->mdata.parameter_3 =
                    LE_32(info.Parameters.ap.MenloConfig.ConfigParamData1);
        } else if (info.Operation & MENLO_OP_GET_INFO) {
                pkt->mdata.parameter_1 =
                    LE_32(info.Parameters.ap.MenloInfo.InfoDataType);
                pkt->mdata.parameter_2 =
                    LE_32(info.Parameters.ap.MenloInfo.InfoContext);
        }

        rval = ql_issue_mbx_iocb(ha, (caddr_t)pkt, sizeof (ql_mbx_iocb_t));
        LITTLE_ENDIAN_16(&pkt->mdata.options_status);
        LITTLE_ENDIAN_16(&pkt->mdata.failure_code);

        if (rval != QL_SUCCESS || (pkt->mdata.entry_status & 0x3c) != 0 ||
            pkt->mdata.options_status != CS_COMPLETE) {
                /* Command error */
                EL(ha, "failed, status=%xh, es=%xh, cs=%xh, fc=%xh\n", rval,
```

```
8296                    pkt->mdata.entry_status & 0x3c, pkt->mdata.options_status,
8297                    pkt->mdata.failure_code);
8298                cmd->Status = EXT_STATUS_ERR;
8299                cmd->DetailStatus = rval != QL_SUCCESS ? rval :
8300                    QL_FUNCTION_FAILED;
8301                cmd->ResponseLen = 0;
8302        } else if (info.Operation == MENLO_OP_READ_MEM) {
8303                (void) ddi_dma_sync(dma_mem->dma_handle, 0, dma_mem->size,
8304                    DDI_DMA_SYNC_FORKERNEL);
8305                if (ql_send_buffer_data((caddr_t)(uintptr_t)info.pDataBytes,
8306                    dma_mem->bp, info.TotalByteCount, mode) !=
8307                    info.TotalByteCount) {
8308                        cmd->Status = EXT_STATUS_COPY_ERR;
8309                        cmd->ResponseLen = 0;
8310                }
8311        }

8313        ql_free_dma_resource(ha, dma_mem);
8314        kmem_free(dma_mem, sizeof (dma_mem_t));
8315        kmem_free(pkt, sizeof (ql_mbx_iocb_t));

8317        QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
8318 }
```

unchanged_portion_omitted_

```
8684 /*
8685  * ql_get_dcbx_parameters
8686  *      Get DCBX parameters.
8687  *
8688  * Input:
8689  *      ha:     adapter state pointer.
8690  *      cmd:    User space CT arguments pointer.
8691  *      mode:   flags.
8692  */
8693 static void
8694 ql_get_dcbx_parameters(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
8695 {
8696        uint8_t         *tmp_buf;
8697        int             rval;

8699        QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

8701        if (!(CFG_IST(ha, CFG_CTRL_8081))) {
8702                EL(ha, "invalid request for HBA\n");
8703                cmd->Status = EXT_STATUS_INVALID_REQUEST;
8704                cmd->ResponseLen = 0;
8705                return;
8706        }

8708        /* Allocate memory for command. */
8709        tmp_buf = kmem_zalloc(EXT_DEF_DCBX_PARAM_BUF_SIZE, KM_SLEEP);
8826        if (tmp_buf == NULL) {
8827                EL(ha, "failed, kmem_zalloc\n");
8828                cmd->Status = EXT_STATUS_NO_MEMORY;
8829                cmd->ResponseLen = 0;
8830                return;
8831        }
8710        /* Send command */
8711        rval = ql_get_dcbx_params(ha, EXT_DEF_DCBX_PARAM_BUF_SIZE,
8712            (caddr_t)tmp_buf);
8713        if (rval != QL_SUCCESS) {
8714                /* error */
8715                EL(ha, "failed, get_dcbx_params_mbx=%xh\n", rval);
8716                kmem_free(tmp_buf, EXT_DEF_DCBX_PARAM_BUF_SIZE);
8717                cmd->Status = EXT_STATUS_ERR;
8718                cmd->ResponseLen = 0;
```



```
8719                return;
8720        }

8722        /* Copy the response */
8723        if (ql_send_buffer_data((caddr_t)tmp_buf,
8724            (caddr_t)(uintptr_t)cmd->ResponseAdr,
8725            EXT_DEF_DCBX_PARAM_BUF_SIZE, mode) != EXT_DEF_DCBX_PARAM_BUF_SIZE) {
8726                EL(ha, "failed, ddi_copyout\n");
8727                cmd->Status = EXT_STATUS_COPY_ERR;
8728                cmd->ResponseLen = 0;
8729        } else {
8730                cmd->ResponseLen = EXT_DEF_DCBX_PARAM_BUF_SIZE;
8731                QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
8732        }
8733        kmem_free(tmp_buf, EXT_DEF_DCBX_PARAM_BUF_SIZE);

8735 }
```

unchanged_portion_omitted_

```
8895 /*
8896  * ql_get_xgmac_statistics
8897  *      Get XgMac information
8898  *
8899  * Input:
8900  *      ha:     adapter state pointer.
8901  *      cmd:    EXT_IOCTL cmd struct pointer.
8902  *      mode:   flags.
8903  *
8904  * Returns:
8905  *      None, request status indicated in cmd->Status.
8906  *
8907  * Context:
8908  *      Kernel context.
8909  */
8910 static void
8911 ql_get_xgmac_statistics(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
8912 {
8913        int                     rval;
8914        uint32_t                size;
8915        int8_t                  *tmp_buf;
8916        EXT_MENLO_MANAGE_INFO   info;

8918        QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

8920        /*  Verify the size of request structure. */
8921        if (cmd->RequestLen < sizeof (EXT_MENLO_MANAGE_INFO)) {
8922                /* Return error */
8923                EL(ha, "RequestLen=%d < %d\n", cmd->RequestLen,
8924                    sizeof (EXT_MENLO_MANAGE_INFO));
8925                cmd->Status = EXT_STATUS_INVALID_PARAM;
8926                cmd->DetailStatus = EXT_DSTATUS_REQUEST_LEN;
8927                cmd->ResponseLen = 0;
8928                return;
8929        }

8931        /* Get manage info request. */
8932        if (ddi_copyin((caddr_t)(uintptr_t)cmd->RequestAdr,
8933            (caddr_t)&info, sizeof (EXT_MENLO_MANAGE_INFO), mode) != 0) {
8934                EL(ha, "failed, ddi_copyin\n");
8935                cmd->Status = EXT_STATUS_COPY_ERR;
8936                cmd->ResponseLen = 0;
8937                return;
8938        }

8940        size = info.TotalByteCount;
8941        if (!size) {
```

```
8942                 /* parameter error */
8943                 cmd->Status = EXT_STATUS_INVALID_PARAM;
8944                 cmd->DetailStatus = 0;
8945                 EL(ha, "failed, size=%xh\n", size);
8946                 cmd->ResponseLen = 0;
8947                 return;
8948         }

8950         /* Allocate memory for command. */
8951         tmp_buf = kmem_zalloc(size, KM_SLEEP);
9074         if (tmp_buf == NULL) {
9075                 EL(ha, "failed, kmem_zalloc\n");
9076                 cmd->Status = EXT_STATUS_NO_MEMORY;
9077                 cmd->ResponseLen = 0;
9078                 return;
9079         }

8953         if (!(info.Operation & MENLO_OP_GET_INFO)) {
8954                 EL(ha, "Invalid request for 81XX\n");
8955                 kmem_free(tmp_buf, size);
8956                 cmd->Status = EXT_STATUS_ERR;
8957                 cmd->ResponseLen = 0;
8958                 return;
8959         }

8961         rval = ql_get_xgmac_stats(ha, size, (caddr_t)tmp_buf);

8963         if (rval != QL_SUCCESS) {
8964                 /* error */
8965                 EL(ha, "failed, get_xgmac_stats =%xh\n", rval);
8966                 kmem_free(tmp_buf, size);
8967                 cmd->Status = EXT_STATUS_ERR;
8968                 cmd->ResponseLen = 0;
8969                 return;
8970         }

8972         if (ql_send_buffer_data(tmp_buf, (caddr_t)(uintptr_t)info.pDataBytes,
8973             size, mode) != size) {
8974                 EL(ha, "failed, ddi_copyout\n");
8975                 cmd->Status = EXT_STATUS_COPY_ERR;
8976                 cmd->ResponseLen = 0;
8977         } else {
8978                 cmd->ResponseLen = info.TotalByteCount;
8979                 QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
8980         }
8981         kmem_free(tmp_buf, size);
8982         QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
8983 }

8985 /*
8986  * ql_get_fcf_list
8987  *      Get FCF list.
8988  *
8989  * Input:
8990  *      ha:     adapter state pointer.
8991  *      cmd:    User space CT arguments pointer.
8992  *      mode:   flags.
8993  */
8994 static void
8995 ql_get_fcf_list(ql_adapter_state_t *ha, EXT_IOCTL *cmd, int mode)
8996 {
8997         uint8_t                 *tmp_buf;
8998         int                     rval;
8999         EXT_FCF_LIST            fcf_list = {0};
9000         ql_fcf_list_desc_t      mb_fcf_list = {0};
```



```
9002         QL_PRINT_9(CE_CONT, "(%d): started\n", ha->instance);

9004         if (!(CFG_IST(ha, CFG_CTRL_81XX))) {
9005                 EL(ha, "invalid request for HBA\n");
9006                 cmd->Status = EXT_STATUS_INVALID_REQUEST;
9007                 cmd->ResponseLen = 0;
9008                 return;
9009         }
9010         /* Get manage info request. */
9011         if (ddi_copyin((caddr_t)(uintptr_t)cmd->RequestAdr,
9012             (caddr_t)&fcf_list, sizeof (EXT_FCF_LIST), mode) != 0) {
9013                 EL(ha, "failed, ddi_copyin\n");
9014                 cmd->Status = EXT_STATUS_COPY_ERR;
9015                 cmd->ResponseLen = 0;
9016                 return;
9017         }

9019         if (!(fcf_list.BufSize)) {
9020                 /* Return error */
9021                 EL(ha, "failed, fcf_list BufSize is=%xh\n",
9022                     fcf_list.BufSize);
9023                 cmd->Status = EXT_STATUS_INVALID_PARAM;
9024                 cmd->ResponseLen = 0;
9025                 return;
9026         }
9027         /* Allocate memory for command. */
9028         tmp_buf = kmem_zalloc(fcf_list.BufSize, KM_SLEEP);
9157         if (tmp_buf == NULL) {
9158                 EL(ha, "failed, kmem_zalloc\n");
9159                 cmd->Status = EXT_STATUS_NO_MEMORY;
9160                 cmd->ResponseLen = 0;
9161                 return;
9162         }
9029         /* build the descriptor */
9030         if (fcf_list.Options) {
9031                 mb_fcf_list.options = FCF_LIST_RETURN_ONE;
9032         } else {
9033                 mb_fcf_list.options = FCF_LIST_RETURN_ALL;
9034         }
9035         mb_fcf_list.fcf_index = (uint16_t)fcf_list.FcfIndex;
9036         mb_fcf_list.buffer_size = fcf_list.BufSize;

9038         /* Send command */
9039         rval = ql_get_fcf_list_mbx(ha, &mb_fcf_list, (caddr_t)tmp_buf);
9040         if (rval != QL_SUCCESS) {
9041                 /* error */
9042                 EL(ha, "failed, get_fcf_list_mbx=%xh\n", rval);
9043                 kmem_free(tmp_buf, fcf_list.BufSize);
9044                 cmd->Status = EXT_STATUS_ERR;
9045                 cmd->ResponseLen = 0;
9046                 return;
9047         }

9049         /* Copy the response */
9050         if (ql_send_buffer_data((caddr_t)tmp_buf,
9051             (caddr_t)(uintptr_t)cmd->ResponseAdr,
9052             fcf_list.BufSize, mode) != fcf_list.BufSize) {
9053                 EL(ha, "failed, ddi_copyout\n");
9054                 cmd->Status = EXT_STATUS_COPY_ERR;
9055                 cmd->ResponseLen = 0;
9056         } else {
9057                 cmd->ResponseLen = mb_fcf_list.buffer_size;
9058                 QL_PRINT_9(CE_CONT, "(%d): done\n", ha->instance);
9059         }

9061         kmem_free(tmp_buf, fcf_list.BufSize);
```

```
9062 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   38588 Thu Oct 23 11:04:56 2014
new/usr/src/uts/common/io/fibre-channel/fca/qlge/qlge_flash.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 213 /*
 214  * qlge_load_flash
 215  * Write "size" bytes from memory "dp" to flash address "faddr".
 216  * faddr = 32bit word flash address.
 217  */
 218 int
 219 qlge_load_flash(qlge_t *qlge, uint8_t *dp, uint32_t len, uint32_t faddr)
 220 {
 221         int rval = DDI_FAILURE;
 222         uint32_t start_block, end_block;
 223         uint32_t start_byte, end_byte;
 224         uint32_t num;
 225         uint32_t sector_size, addr_src, addr_desc;
 226         uint8_t *temp;
 227         caddr_t bp, bdesc;

 229         QL_PRINT(DBG_FLASH, ("%s(%d) entered to write addr %x, %d bytes\n",
 230             __func__, qlge->instance, faddr, len));

 232         sector_size = qlge->fdesc.block_size;

 234         if (faddr > qlge->fdesc.flash_size) {
 235                 cmn_err(CE_WARN, "%s(%d): invalid flash write address %x",
 236                     __func__, qlge->instance, faddr);
 237                 return (DDI_FAILURE);
 238         }
 239         /* Get semaphore to access Flash Address and Flash Data Registers */
 240         if (ql_sem_spinlock(qlge, QL_FLASH_SEM_MASK) != DDI_SUCCESS) {
 241                 return (DDI_FAILURE);
 242         }
 243         temp = kmem_zalloc(sector_size, KM_SLEEP);
 244         if (temp == NULL) {
 245                 cmn_err(CE_WARN, "%s(%d): Unable to allocate buffer",
 246                     __func__, qlge->instance);
 247                 ql_sem_unlock(qlge, QL_FLASH_SEM_MASK);
 248                 return (DDI_FAILURE);
 249         }

 245         (void) ql_unprotect_flash(qlge);

 247         get_sector_number(qlge, faddr, &start_block);
 248         get_sector_number(qlge, faddr + len - 1, &end_block);

 250         QL_PRINT(DBG_FLASH, ("%s(%d) start_block %x, end_block %x\n",
 251             __func__, qlge->instance, start_block, end_block));

 253         for (num = start_block; num <= end_block; num++) {
 254                 QL_PRINT(DBG_FLASH,
 255                     ("%s(%d) sector_size 0x%x, sector read addr %x\n",
 256                     __func__, qlge->instance, sector_size, num * sector_size));
 257                 /* read one whole sector flash data to buffer */
 258                 rval = qlge_dump_fcode(qlge, (uint8_t *)temp, sector_size,
 259                     num * sector_size);

 261                 start_byte = num * sector_size;
 262                 end_byte = start_byte + sector_size -1;
 263                 if (start_byte < faddr)
 264                         start_byte = faddr;
```



```
 265                 if (end_byte > (faddr + len))
 266                         end_byte = (faddr + len - 1);

 268                 addr_src = start_byte - faddr;
 269                 addr_desc = start_byte - num * sector_size;
 270                 bp = (caddr_t)dp + addr_src;
 271                 bdesc = (caddr_t)temp + addr_desc;
 272                 bcopy(bp, bdesc, (end_byte - start_byte + 1));

 274                 /* write the whole sector data to flash */
 275                 if (ql_erase_and_write_to_flash(qlge, temp, sector_size,
 276                     num * sector_size) != DDI_SUCCESS)
 277                         goto out;
 278         }
 279         rval = DDI_SUCCESS;
 280 out:
 281         (void) ql_protect_flash(qlge);
 282         kmem_free(temp, sector_size);

 284         ql_sem_unlock(qlge, QL_FLASH_SEM_MASK);

 286         if (rval != DDI_SUCCESS) {
 287                 cmn_err(CE_WARN, "%s(%d failed=%xh",
 288                     __func__, qlge->instance, rval);
 289         }

 291         return (rval);
 292 }
______unchanged_portion_omitted_
```

```
**********************************************************
  153507 Thu Oct 23 11:04:56 2014
new/usr/src/uts/common/io/iwh/iwh.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1795 /* ARGSUSED */
1796 static ieee80211_node_t *
1797 iwh_node_alloc(ieee80211com_t *ic)
1798 {
1799         iwh_amrr_t *amrr;

1801         amrr = kmem_zalloc(sizeof (iwh_amrr_t), KM_SLEEP);
1802         if (NULL == amrr) {
1803                 cmn_err(CE_WARN, "iwh_node_alloc(): "
1804                     "failed to allocate memory for amrr structure\n");
1805                 return (NULL);
1806         }

1803         iwh_amrr_init(amrr);

1805         return (&amrr->in);
1806 }
______unchanged_portion_omitted_
```

```
**********************************************************
  126658 Thu Oct 23 11:04:56 2014
new/usr/src/uts/common/io/iwp/iwp.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1732 /* ARGSUSED */
1733 static ieee80211_node_t *
1734 iwp_node_alloc(ieee80211com_t *ic)
1735 {
1736         iwp_amrr_t *amrr;

1738         amrr = kmem_zalloc(sizeof (iwp_amrr_t), KM_SLEEP);
1739         if (NULL == amrr) {
1740                 cmn_err(CE_WARN, "iwp_node_alloc(): "
1741                     "failed to allocate memory for amrr structure\n");
1742                 return (NULL);
1743         }

1740         iwp_amrr_init(amrr);

1742         return (&amrr->in);
1743 }
______unchanged_portion_omitted_
```

```
**********************************************************
  114548 Thu Oct 23 11:04:57 2014
new/usr/src/uts/common/io/mwl/mwl.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 542 static int
 543 mwl_alloc_tx_ring(struct mwl_softc *sc, struct mwl_tx_ring *ring,
 544     int count)
 545 {
 546         struct mwl_txdesc *ds;
 547         struct mwl_txbuf *bf;
 548         int i, err, datadlen;

 550         ring->count = count;
 551         ring->queued = 0;
 552         ring->cur = ring->next = ring->stat = 0;
 553         err = mwl_alloc_dma_mem(sc->sc_dev, &mwl_dma_attr,
 554             count * sizeof (struct mwl_txdesc), &mwl_desc_accattr,
 555             DDI_DMA_CONSISTENT, DDI_DMA_RDWR | DDI_DMA_CONSISTENT,
 556             &ring->txdesc_dma);
 557         if (err) {
 558                 MWL_DBG(MWL_DBG_DMA, "mwl: mwl_alloc_tx_ring(): "
 559                     "alloc tx ring failed, size %d\n",
 560                     (uint32_t)(count * sizeof (struct mwl_txdesc)));
 561                 return (DDI_FAILURE);
 562         }

 564         MWL_DBG(MWL_DBG_DMA, "mwl: mwl_alloc_tx_ring(): "
 565             "dma len = %d\n", (uint32_t)(ring->txdesc_dma.alength));
 566         ring->desc = (struct mwl_txdesc *)ring->txdesc_dma.mem_va;
 567         ring->physaddr = ring->txdesc_dma.cookie.dmac_address;
 568         bzero(ring->desc, count * sizeof (struct mwl_txdesc));

 570         datadlen = count * sizeof (struct mwl_txbuf);
 571         ring->buf = kmem_zalloc(datadlen, KM_SLEEP);
 572         if (ring->buf == NULL) {
 573                 MWL_DBG(MWL_DBG_DMA, "mwl: mwl_alloc_tx_ring(): "
 574                     "could not alloc tx ring data buffer\n");
 575                 return (DDI_FAILURE);
 576         }
 572         bzero(ring->buf, count * sizeof (struct mwl_txbuf));

 574         for (i = 0; i < count; i++) {
 575                 ds = &ring->desc[i];
 576                 bf = &ring->buf[i];
 577                 /* alloc DMA memory */
 578                 (void) mwl_alloc_dma_mem(sc->sc_dev, &mwl_dma_attr,
 579                     sc->sc_dmabuf_size,
 580                     &mwl_buf_accattr,
 581                     DDI_DMA_STREAMING,
 582                     DDI_DMA_WRITE | DDI_DMA_STREAMING,
 583                     &bf->txbuf_dma);
 584                 bf->bf_baddr = bf->txbuf_dma.cookie.dmac_address;
 585                 bf->bf_mem = (uint8_t *)(bf->txbuf_dma.mem_va);
 586                 bf->bf_daddr = ring->physaddr + _PTRDIFF(ds, ring->desc);
 587                 bf->bf_desc = ds;
 588         }

 590         (void) ddi_dma_sync(ring->txdesc_dma.dma_hdl,
 591             0,
 592             ring->txdesc_dma.alength,
 593             DDI_DMA_SYNC_FORDEV);
```



```
 595         return (0);
 596 }
______unchanged_portion_omitted_

2217 /* ARGSUSED */
2218 static struct ieee80211_node *
2219 mwl_node_alloc(struct ieee80211com *ic)
2220 {
2221         struct mwl_node *mn;

2223         mn = kmem_zalloc(sizeof (struct mwl_node), KM_SLEEP);
2229         if (mn == NULL) {
2230                 /* XXX stat+msg */
2231                 MWL_DBG(MWL_DBG_MSG, "mwl: mwl_node_alloc(): "
2232                     "alloc node failed\n");
2233                 return (NULL);
2234         }
2224         return (&mn->mn_node);
2225 }
______unchanged_portion_omitted_
```

****
13141 Thu Oct 23 11:04:57 2014
new/usr/src/uts/common/io/net80211/net80211_crypto_ccmp.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
****
_____***unchanged_portion_omitted_***

```
 83 /* ARGSUSED */
 84 static void *
 85 ccmp_attach(struct ieee80211com *ic, struct ieee80211_key *k)
 86 {
 87         struct ccmp_ctx *ctx;

 89         ctx = kmem_zalloc(sizeof (struct ccmp_ctx), KM_SLEEP);
 90         if (ctx == NULL)
 91                 return (NULL);

 91         ctx->cc_ic = ic;
 92         return (ctx);
 93 }
```

_____***unchanged_portion_omitted_***

****
21087 Thu Oct 23 11:04:57 2014
new/usr/src/uts/common/io/net80211/net80211_crypto_tkip.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
****
_____***unchanged_portion_omitted_***

```
 89 static void michael_mic(struct tkip_ctx *, const uint8_t *,
 90     mblk_t *, uint_t, size_t, uint8_t[]);
 91 static int tkip_encrypt(struct tkip_ctx *, struct ieee80211_key *,
 92     mblk_t *, int);
 93 static int tkip_decrypt(struct tkip_ctx *, struct ieee80211_key *,
 94     mblk_t *, int);

 96 extern int rc4_init(crypto_context_t *, const uint8_t *, int);
 97 extern int rc4_crypt(crypto_context_t, const uint8_t *, uint8_t *, int);
 98 extern int rc4_final(crypto_context_t, uint8_t *, int);

100 /* ARGSUSED */
101 static void *
102 tkip_attach(struct ieee80211com *ic, struct ieee80211_key *k)
103 {
104         struct tkip_ctx *ctx;

106         ctx = kmem_zalloc(sizeof (struct tkip_ctx), KM_SLEEP);
107         if (ctx == NULL)
108                 return (NULL);

108         ctx->tc_ic = ic;
109         return (ctx);
110 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   12883 Thu Oct 23 11:04:57 2014
new/usr/src/uts/common/io/ntxn/unm_nic_init.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 253 int
 254 pinit_from_rom(struct unm_adapter_s *adapter, int verbose)
 255 {
 256         int     addr, val, status, i, init_delay = 0, n;
 257         struct crb_addr_pair    *buf;
 258         unsigned long   off;
 259         unsigned int    offset;

 261         status = unm_nic_get_board_info(adapter);
 262         if (status)
 263                 cmn_err(CE_WARN, "%s: pinit_from_rom: Error getting brdinfo\n",
 264                     unm_nic_driver_name);

 266         UNM_CRB_WRITELIT_ADAPTER(UNM_ROMUSB_GLB_SW_RESET, 0xffffffff, adapter);

 268         if (verbose) {
 269                 int     val;
 270                 if (rom_fast_read(adapter, 0x4008, &val) == 0)
 271                         cmn_err(CE_WARN, "P2 ROM board type: 0x%08x\n", val);
 272                 else
 273                         cmn_err(CE_WARN, "Could not read board type\n");
 274                 if (rom_fast_read(adapter, 0x400c, &val) == 0)
 275                         cmn_err(CE_WARN, "ROM board  num: 0x%08x\n", val);
 276                 else
 277                         cmn_err(CE_WARN, "Could not read board number\n");
 278                 if (rom_fast_read(adapter, 0x4010, &val) == 0)
 279                         cmn_err(CE_WARN, "ROM chip   num: 0x%08x\n", val);
 280                 else
 281                         cmn_err(CE_WARN, "Could not read chip number\n");
 282         }

 284         if (NX_IS_REVISION_P3(adapter->ahw.revision_id)) {
 285                 if (rom_fast_read(adapter, 0, &n) != 0 ||
 286                     (unsigned int)n != 0xcafecafe ||
 287                     rom_fast_read(adapter, 4, &n) != 0) {
 288                         cmn_err(CE_WARN, "%s: ERROR Reading crb_init area: "
 289                             "n: %08x\n", unm_nic_driver_name, n);
 290                         return (-1);
 291                 }

 293                 offset = n & 0xffffU;
 294                 n = (n >> 16) & 0xffffU;
 295         } else {
 296                 if (rom_fast_read(adapter, 0, &n) != 0 ||
 297                     !(n & 0x80000000)) {
 298                         cmn_err(CE_WARN, "%s: ERROR Reading crb_init area: "
 299                             "n: %08x\n", unm_nic_driver_name, n);
 300                         return (-1);
 301                 }
 302                 offset = 1;
 303                 n &= ~0x80000000;
 304         }

 306         if (n  >= 1024) {
 307                 cmn_err(CE_WARN, "%s: %s:n=0x%x Card flash not initialized\n",
 308                     unm_nic_driver_name, __FUNCTION__, n);
 309                 return (-1);
 310         }
```



```
 312         if (verbose)
 313                 cmn_err(CE_WARN, "%s: %d CRB init values found in ROM.\n",
 314                     unm_nic_driver_name, n);

 316         buf = kmem_zalloc(n * sizeof (struct crb_addr_pair), KM_SLEEP);
 317         if (buf == NULL) {
 318                 cmn_err(CE_WARN, "%s: pinit_from_rom: Unable to get memory\n",
 319                     unm_nic_driver_name);
 320                 return (-1);
 321         }

 318         for (i = 0; i < n; i++) {
 319                 if (rom_fast_read(adapter, 8*i + 4*offset, &val) != 0 ||
 320                     rom_fast_read(adapter, 8*i + 4*offset + 4, &addr) != 0) {
 321                         kmem_free(buf, n * sizeof (struct crb_addr_pair));
 322                         return (-1);
 323                 }

 325                 buf[i].addr = addr;
 326                 buf[i].data = val;

 328                 if (verbose)
 329                         cmn_err(CE_WARN, "%s: PCI:     0x%08x == 0x%08x\n",
 330                             unm_nic_driver_name,
 331                             (unsigned int)decode_crb_addr(
 332                             (unsigned long)addr), val);
 333         }

 335         for (i = 0; i < n; i++) {
 336                 off = decode_crb_addr((unsigned long)buf[i].addr) +
 337                     UNM_PCI_CRBSPACE;
 338                 /* skipping cold reboot MAGIC */
 339                 if (off == UNM_CAM_RAM(0x1fc)) {
 340                         continue;
 341                 }

 343                 if (NX_IS_REVISION_P3(adapter->ahw.revision_id)) {
 344                         /* do not reset PCI */
 345                         if (off == (ROMUSB_GLB + 0xbc)) {
 346                                 continue;
 347                         }
 348                         if (off == (ROMUSB_GLB + 0xc8)) /* core clock */
 349                                 continue;
 350                         if (off == (ROMUSB_GLB + 0x24)) /* MN clock */
 351                                 continue;
 352                         if (off == (ROMUSB_GLB + 0x1c)) /* MS clock */
 353                                 continue;
 354                         if (off == (UNM_CRB_PEG_NET_1 + 0x18)) {
 355                                 buf[i].data = 0x1020;
 356                         }
 357                         /* skip the function enable register */
 358                         if (off == UNM_PCIE_REG(PCIE_SETUP_FUNCTION)) {
 359                                 continue;
 360                         }
 361                         if (off == UNM_PCIE_REG(PCIE_SETUP_FUNCTION2)) {
 362                                 continue;
 363                         }

 365                         if ((off & 0x0ff00000) == UNM_CRB_SMB) {
 366                                 continue;
 367                         }

 369                 }

 371                 if (off == ADDR_ERROR) {
```

```
372                         cmn_err(CE_WARN, "%s: Err: Unknown addr: 0x%08lx\n",
373                             unm_nic_driver_name, buf[i].addr);
374                         continue;
375                 }

377                 /* After writing this register, HW needs time for CRB */
378                 /* to quiet down (else crb_window returns 0xffffffff) */
379                 if (off == UNM_ROMUSB_GLB_SW_RESET) {
380                         init_delay = 1;

382                         if (NX_IS_REVISION_P2(adapter->ahw.revision_id)) {
383                                 /* hold xdma in reset also */
384                                 buf[i].data = 0x8000ff;
385                         }
386                 }

388                 adapter->unm_nic_hw_write_wx(adapter, off, &buf[i].data, 4);

390                 if (init_delay == 1) {
391                         nx_msleep(1000);        /* Sleep 1000 msecs */
392                         init_delay = 0;
393                 }

395                 nx_msleep(1);                   /* Sleep 1 msec */
396         }

398         kmem_free(buf, n * sizeof (struct crb_addr_pair));

400         // disable_peg_cache_all
401         // unreset_net_cache
402         if (NX_IS_REVISION_P2(adapter->ahw.revision_id)) {
403                 val = UNM_CRB_READ_VAL_ADAPTER(UNM_ROMUSB_GLB_SW_RESET,
404                     adapter);
405                 UNM_CRB_WRITELIT_ADAPTER(UNM_ROMUSB_GLB_SW_RESET,
406                     (val & 0xffffff0f), adapter);
407         }

409         // p2dn replyCount
410         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_D+0xec, 0x1e, adapter);
411         // disable_peg_cache 0
412         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_D+0x4c, 8, adapter);
413         // disable_peg_cache 1
414         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_I+0x4c, 8, adapter);

416         // peg_clr_all
417         // peg_clr 0
418         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_0+0x8, 0, adapter);
419         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_0+0xc, 0, adapter);
420         // peg_clr 1
421         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_1+0x8, 0, adapter);
422         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_1+0xc, 0, adapter);
423         // peg_clr 2
424         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_2+0x8, 0, adapter);
425         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_2+0xc, 0, adapter);
426         // peg_clr 3
427         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_3+0x8, 0, adapter);
428         UNM_CRB_WRITELIT_ADAPTER(UNM_CRB_PEG_NET_3+0xc, 0, adapter);

430         return (0);
431 }
```

_____***unchanged_portion_omitted_***

****
32939 Thu Oct 23 11:04:58 2014
new/usr/src/uts/common/io/pciex/hotplug/pcie_hp.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
****
_____***unchanged_portion_omitted_***

```
 118 /*
 119  * routine to copy in a nvlist from userland
 120  */
 121 int
 122 pcie_copyin_nvlist(char *packed_buf, size_t packed_sz, nvlist_t **nvlp)
 123 {
 124         int             ret = DDI_SUCCESS;
 125         char            *packed;
 126         nvlist_t        *dest = NULL;

 128         if (packed_buf == NULL || packed_sz == 0)
 129                 return (DDI_EINVAL);

 131         /* copyin packed nvlist */
 132         packed = kmem_alloc(packed_sz, KM_SLEEP);
 132         if ((packed = kmem_alloc(packed_sz, KM_SLEEP)) == NULL)
 133                 return (DDI_ENOMEM);

 134         if (copyin(packed_buf, packed, packed_sz) != 0) {
 135                 cmn_err(CE_WARN, "pcie_copyin_nvlist: copyin failed.\n");
 136                 ret = DDI_FAILURE;
 137                 goto copyin_cleanup;
 138         }

 140         /* unpack packed nvlist */
 141         if ((ret = nvlist_unpack(packed, packed_sz, &dest, KM_SLEEP)) != 0) {
 142                 cmn_err(CE_WARN, "pcie_copyin_nvlist: nvlist_unpack "
 143                     "failed with err %d\n", ret);
 144                 switch (ret) {
 145                 case EINVAL:
 146                 case ENOTSUP:
 147                         ret = DDI_EINVAL;
 148                         goto copyin_cleanup;
 149                 case ENOMEM:
 150                         ret = DDI_ENOMEM;
 151                         goto copyin_cleanup;
 152                 default:
 153                         ret = DDI_FAILURE;
 154                         goto copyin_cleanup;
 155                 }
 156         }
 157         *nvlp = dest;
 158 copyin_cleanup:
 159         kmem_free(packed, packed_sz);
 160         return (ret);
 161 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   86645 Thu Oct 23 11:04:58 2014
new/usr/src/uts/common/io/rtw/rtw.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 786 /*
 787  * Returns -1 on failure.
 788  */
 789 static int
 790 rtw_srom_read(struct rtw_regs *regs, uint32_t flags, struct rtw_srom *sr,
 791     const char *dvname)
 792 {
 793         int rc;
 794         struct seeprom_descriptor sd;
 795         uint8_t ecr;

 797         (void) memset(&sd, 0, sizeof (sd));

 799         ecr = RTW_READ8(regs, RTW_9346CR);

 801         if ((flags & RTW_F_9356SROM) != 0) {
 802                 RTW_DPRINTF(RTW_DEBUG_ATTACH, "%s: 93c56 SROM\n", dvname);
 803                 sr->sr_size = 256;
 804                 sd.sd_chip = C56_66;
 805         } else {
 806                 RTW_DPRINTF(RTW_DEBUG_ATTACH, "%s: 93c46 SROM\n", dvname);
 807                 sr->sr_size = 128;
 808                 sd.sd_chip = C46;
 809         }

 811         ecr &= ~(RTW_9346CR_EEDI | RTW_9346CR_EEDO | RTW_9346CR_EESK |
 812             RTW_9346CR_EEM_MASK | RTW_9346CR_EECS);
 813         ecr |= RTW_9346CR_EEM_PROGRAM;

 815         RTW_WRITE8(regs, RTW_9346CR, ecr);

 817         sr->sr_content = kmem_zalloc(sr->sr_size, KM_SLEEP);

 819         if (sr->sr_content == NULL) {
 820                 cmn_err(CE_WARN, "%s: unable to allocate SROM buffer\n",
 821                     dvname);
 822                 return (ENOMEM);
 823         }

 819         (void) memset(sr->sr_content, 0, sr->sr_size);

 821         /*
 822          * RTL8180 has a single 8-bit register for controlling the
 823          * 93cx6 SROM.  There is no "ready" bit. The RTL8180
 824          * input/output sense is the reverse of read_seeprom's.
 825          */
 826         sd.sd_handle = regs->r_handle;
 827         sd.sd_base = regs->r_base;
 828         sd.sd_regsize = 1;
 829         sd.sd_control_offset = RTW_9346CR;
 830         sd.sd_status_offset = RTW_9346CR;
 831         sd.sd_dataout_offset = RTW_9346CR;
 832         sd.sd_CK = RTW_9346CR_EESK;
 833         sd.sd_CS = RTW_9346CR_EECS;
 834         sd.sd_DI = RTW_9346CR_EEDO;
 835         sd.sd_DO = RTW_9346CR_EEDI;
 836         /*
 837          * make read_seeprom enter EEPROM read/write mode
```



```
 838          */
 839         sd.sd_MS = ecr;
 840         sd.sd_RDY = 0;

 842         /*
 843          * TBD bus barriers
 844          */
 845         if (!read_seeprom(&sd, sr->sr_content, 0, sr->sr_size/2)) {
 846                 cmn_err(CE_WARN, "%s: could not read SROM\n", dvname);
 847                 kmem_free(sr->sr_content, sr->sr_size);
 848                 sr->sr_content = NULL;
 849                 return (-1);    /* XXX */
 850         }

 852         /*
 853          * end EEPROM read/write mode
 854          */
 855         RTW_WRITE8(regs, RTW_9346CR,
 856             (ecr & ~RTW_9346CR_EEM_MASK) | RTW_9346CR_EEM_NORMAL);
 857         RTW_WBRW(regs, RTW_9346CR, RTW_9346CR);

 859         if ((rc = rtw_recall_eeprom(regs, dvname)) != 0)
 860                 return (rc);

 862 #ifdef SROM_DEBUG
 863         {
 864                 int i;
 865                 RTW_DPRINTF(RTW_DEBUG_ATTACH,
 866                     "\n%s: serial ROM:\n\t", dvname);
 867                 for (i = 0; i < sr->sr_size/2; i++) {
 868                         RTW_DPRINTF(RTW_DEBUG_ATTACH,
 869                             "offset-0x%x: %04x", 2*i, sr->sr_content[i]);
 870                 }
 871         }
 872 #endif /* DEBUG */
 873         return (0);
 874 }
______unchanged_portion_omitted_
```

```
**********************************************************
   83079 Thu Oct 23 11:04:58 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_


1040 int
1041 mptsas_request_from_pool(mptsas_t *mpt, mptsas_cmd_t **cmd,
1042     struct scsi_pkt **pkt)
1043 {
1044         m_event_struct_t        *ioc_cmd = NULL;

1046         ioc_cmd = kmem_zalloc(M_EVENT_STRUCT_SIZE, KM_SLEEP);
1047         if (ioc_cmd == NULL) {
1048                 return (DDI_FAILURE);
1049         }
1047         ioc_cmd->m_event_linkp = NULL;
1048         mptsas_ioc_event_cmdq_add(mpt, ioc_cmd);
1049         *cmd = &(ioc_cmd->m_event_cmd);
1050         *pkt = &(ioc_cmd->m_event_pkt);

1052         return (DDI_SUCCESS);
1053 }
______unchanged_portion_omitted_
```

```
**********************************************************
   84832 Thu Oct 23 11:04:58 2014
new/usr/src/uts/common/io/scsi/adapters/pmcs/pmcs_attach.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1875 static int
1876 pmcs_add_more_chunks(pmcs_hw_t *pwp, unsigned long nsize)
1877 {
1878         pmcs_dmachunk_t *dc;
1879         unsigned long dl;
1880         pmcs_chunk_t    *pchunk = NULL;

1882         pwp->cip_dma_attr.dma_attr_align = sizeof (uint32_t);

1884         pchunk = kmem_zalloc(sizeof (pmcs_chunk_t), KM_SLEEP);
1885         if (pchunk == NULL) {
1886                 pmcs_prt(pwp, PMCS_PRT_DEBUG, NULL, NULL,
1887                     "Not enough memory for DMA chunks");
1888                 return (-1);
1889         }

1886         if (pmcs_dma_setup(pwp, &pwp->cip_dma_attr, &pchunk->acc_handle,
1887             &pchunk->dma_handle, nsize, (caddr_t *)&pchunk->addrp,
1888             &pchunk->dma_addr) == B_FALSE) {
1889                 pmcs_prt(pwp, PMCS_PRT_DEBUG, NULL, NULL,
1890                     "Failed to setup DMA for chunks");
1891                 kmem_free(pchunk, sizeof (pmcs_chunk_t));
1892                 return (-1);
1893         }

1895         if ((pmcs_check_acc_handle(pchunk->acc_handle) != DDI_SUCCESS) ||
1896             (pmcs_check_dma_handle(pchunk->dma_handle) != DDI_SUCCESS)) {
1897                 ddi_fm_service_impact(pwp->dip, DDI_SERVICE_UNAFFECTED);
1898                 return (-1);
1899         }

1901         bzero(pchunk->addrp, nsize);
1902         dc = NULL;
1903         for (dl = 0; dl < (nsize / PMCS_SGL_CHUNKSZ); dl++) {
1904                 pmcs_dmachunk_t *tmp;
1905                 tmp = kmem_alloc(sizeof (pmcs_dmachunk_t), KM_SLEEP);
1906                 tmp->nxt = dc;
1907                 dc = tmp;
1908         }
1909         mutex_enter(&pwp->dma_lock);
1910         pmcs_idma_chunks(pwp, dc, pchunk, nsize);
1911         pwp->nchunks++;
1912         mutex_exit(&pwp->dma_lock);
1913         return (0);
1914 }
______unchanged_portion_omitted_
```

```
**********************************************************
   31172 Thu Oct 23 11:04:58 2014
new/usr/src/uts/common/io/scsi/targets/ses_safte.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (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
 */
/*
 * Enclosure Services Devices, SAF-TE Enclosure Routines
 *
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

#include <sys/modctl.h>
#include <sys/file.h>
#include <sys/scsi/scsi.h>
#include <sys/stat.h>
#include <sys/scsi/targets/sesio.h>
#include <sys/scsi/targets/ses.h>


static int set_objstat_sel(ses_softc_t *, ses_objarg *, int);
static int wrbuf16(ses_softc_t *, uchar_t, uchar_t, uchar_t, uchar_t, int);
static void wrslot_stat(ses_softc_t *, int);
static int perf_slotop(ses_softc_t *, uchar_t, uchar_t, int);

#define ALL_ENC_STAT \
        (ENCSTAT_CRITICAL|ENCSTAT_UNRECOV|ENCSTAT_NONCRITICAL|ENCSTAT_INFO)

#define SCRATCH 64
#define NPSEUDO_THERM   1
#define NPSEUDO_ALARM   1
struct scfg {
        /*
         * Cached Configuration
         */
        uchar_t Nfans;          /* Number of Fans */
        uchar_t Npwr;           /* Number of Power Supplies */
        uchar_t Nslots;         /* Number of Device Slots */
        uchar_t DoorLock;       /* Door Lock Installed */
        uchar_t Ntherm;         /* Number of Temperature Sensors */
        uchar_t Nspkrs;         /* Number of Speakers */
        uchar_t  Nalarm;                /* Number of Alarms (at least one) */
```



```
        /*
         * Cached Flag Bytes for Global Status
         */
        uchar_t flag1;
        uchar_t flag2;
        /*
         * What object index ID is where various slots start.
         */
        uchar_t pwroff;
        uchar_t slotoff;
#define ALARM_OFFSET(cc)        (cc)->slotoff - 1
};
#define FLG1_ALARM      0x1
#define FLG1_GLOBFAIL   0x2
#define FLG1_GLOBWARN   0x4
#define FLG1_ENCPWROFF  0x8
#define FLG1_ENCFANFAIL 0x10
#define FLG1_ENCPWRFAIL 0x20
#define FLG1_ENCDRVFAIL 0x40
#define FLG1_ENCDRVWARN 0x80

#define FLG2_LOCKDOOR   0x4
#define SAFTE_PRIVATE   sizeof (struct scfg)

#if     !defined(lint)
_NOTE(MUTEX_PROTECTS_DATA(scsi_device::sd_mutex, scfg))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::Nfans))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::Npwr))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::Nslots))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::DoorLock))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::Ntherm))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::Nspkrs))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::Nalarm))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::flag1))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::flag2))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::pwroff))
_NOTE(DATA_READABLE_WITHOUT_LOCK(scfg::slotoff))
#endif

static int
safte_getconfig(ses_softc_t *ssc)
{
        struct scfg *cfg;
        int err;
        Uscmd local, *lp = &local;
        char rqbuf[SENSE_LENGTH], *sdata;
        static char cdb[CDB_GROUP1] =
            { SCMD_READ_BUFFER, 1, SAFTE_RD_RDCFG, 0, 0, 0, 0, 0, SCRATCH, 0 };

        cfg = ssc->ses_private;
        if (cfg == NULL)
                return (ENXIO);

        sdata = kmem_alloc(SCRATCH, KM_SLEEP);
        if (sdata == NULL)
                return (ENOMEM);

        lp->uscsi_flags = USCSI_READ|USCSI_RQENABLE;
        lp->uscsi_timeout = ses_io_time;
        lp->uscsi_cdb = cdb;
        lp->uscsi_bufaddr = sdata;
        lp->uscsi_buflen = SCRATCH;
        lp->uscsi_cdblen = sizeof (cdb);
        lp->uscsi_rqbuf = rqbuf;
        lp->uscsi_rqlen = sizeof (rqbuf);
```

```
123         err = ses_runcmd(ssc, lp);
124         if (err) {
125                 kmem_free(sdata, SCRATCH);
126                 return (err);
127         }

129         if ((lp->uscsi_buflen - lp->uscsi_resid) < 6) {
130                 SES_LOG(ssc, CE_NOTE, "Too little data (%ld) for configuration",
131                     lp->uscsi_buflen - lp->uscsi_resid);
132                 kmem_free(sdata, SCRATCH);
133                 return (EIO);
134         }
135         SES_LOG(ssc, SES_CE_DEBUG1, "Nfans %d Npwr %d Nslots %d Lck %d Ntherm "
136             "%d Nspkrs %d", sdata[0], sdata[1], sdata[2], sdata[3], sdata[4],
137             sdata[5]);

139         mutex_enter(&ssc->ses_devp->sd_mutex);
140         cfg->Nfans = sdata[0];
141         cfg->Npwr = sdata[1];
142         cfg->Nslots = sdata[2];
143         cfg->DoorLock = sdata[3];
144         cfg->Ntherm = sdata[4];
145         cfg->Nspkrs = sdata[5];
146         cfg->Nalarm = NPSEUDO_ALARM;
147         mutex_exit(&ssc->ses_devp->sd_mutex);
148         kmem_free(sdata, SCRATCH);
149         return (0);
150 }

152 int
153 safte_softc_init(ses_softc_t *ssc, int doinit)
154 {
155         int r, i;
156         struct scfg *cc;

158         if (doinit == 0) {
159                 mutex_enter(&ssc->ses_devp->sd_mutex);
160                 if (ssc->ses_nobjects) {
161                         if (ssc->ses_objmap) {
162                                 kmem_free(ssc->ses_objmap,
163                                     ssc->ses_nobjects * sizeof (encobj));
164                                 ssc->ses_objmap = NULL;
165                         }
166                         ssc->ses_nobjects = 0;
167                 }
168                 if (ssc->ses_private) {
169                         kmem_free(ssc->ses_private, SAFTE_PRIVATE);
170                         ssc->ses_private = NULL;
171                 }
172                 mutex_exit(&ssc->ses_devp->sd_mutex);
173                 return (0);
174         }

176         mutex_enter(&ssc->ses_devp->sd_mutex);
177         if (ssc->ses_private == NULL) {
178                 ssc->ses_private = kmem_zalloc(SAFTE_PRIVATE, KM_SLEEP);
183                 if (ssc->ses_private == NULL) {
184                         mutex_exit(&ssc->ses_devp->sd_mutex);
185                         return (ENOMEM);
186                 }
179         }

181         ssc->ses_nobjects = 0;
182         ssc->ses_encstat = 0;
183         mutex_exit(&ssc->ses_devp->sd_mutex);
```



```
185         if ((r = safte_getconfig(ssc)) != 0) {
186                 return (r);
187         }

189         /*
190          * The number of objects here, as well as that reported by the
191          * READ_BUFFER/GET_CONFIG call, are the over-temperature flags (15)
192          * that get reported during READ_BUFFER/READ_ENC_STATUS.
193          */
194         mutex_enter(&ssc->ses_devp->sd_mutex);
195         cc = ssc->ses_private;
196         ssc->ses_nobjects = cc->Nfans + cc->Npwr + cc->Nslots + cc->DoorLock +
197             cc->Ntherm + cc->Nspkrs + NPSEUDO_THERM + NPSEUDO_ALARM;
198         ssc->ses_objmap = (encobj *)
199             kmem_zalloc(ssc->ses_nobjects * sizeof (encobj), KM_SLEEP);
200         mutex_exit(&ssc->ses_devp->sd_mutex);
201         if (ssc->ses_objmap == NULL)
202                 return (ENOMEM);
203         r = 0;
204         /*
205          * Note that this is all arranged for the convenience
206          * in later fetches of status.
207          */
208         mutex_enter(&ssc->ses_devp->sd_mutex);
209         for (i = 0; i < cc->Nfans; i++)
210                 ssc->ses_objmap[r++].enctype = SESTYP_FAN;
211         cc->pwroff = (uchar_t)r;
212         for (i = 0; i < cc->Npwr; i++)
213                 ssc->ses_objmap[r++].enctype = SESTYP_POWER;
214         for (i = 0; i < cc->DoorLock; i++)
215                 ssc->ses_objmap[r++].enctype = SESTYP_DOORLOCK;
216         for (i = 0; i < cc->Nspkrs; i++)
217                 ssc->ses_objmap[r++].enctype = SESTYP_ALARM;
218         for (i = 0; i < cc->Ntherm; i++)
219                 ssc->ses_objmap[r++].enctype = SESTYP_THERM;
220         for (i = 0; i < NPSEUDO_THERM; i++)
221                 ssc->ses_objmap[r++].enctype = SESTYP_THERM;
222         ssc->ses_objmap[r++].enctype = SESTYP_ALARM;
223         cc->slotoff = (uchar_t)r;
224         for (i = 0; i < cc->Nslots; i++)
225                 ssc->ses_objmap[r++].enctype = SESTYP_DEVICE;
226         mutex_exit(&ssc->ses_devp->sd_mutex);
227         return (0);
228 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  140795 Thu Oct 23 11:04:59 2014
new/usr/src/uts/common/io/usb/hcd/uhci/uhciutil.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_


/*
 * uhci_alloc_memory_for_tds:
 *      - Allocates memory for the isoc/bulk td pools.
 */
static int
uhci_alloc_memory_for_tds(
        uhci_state_t            *uhcip,
        uint_t                  num_tds,
        uhci_bulk_isoc_xfer_t   *info)
{
        int                     result, i, j, err;
        size_t                  real_length;
        uint_t                  ccount, num;
        ddi_device_acc_attr_t   dev_attr;
        uhci_bulk_isoc_td_pool_t *td_pool_ptr1, *td_pool_ptr2;

        USB_DPRINTF_L4(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
            "uhci_alloc_memory_for_tds: num_tds: 0x%x info: 0x%p "
            "num_pools: %u", num_tds, (void *)info, info->num_pools);

        /* The host controller will be little endian */
        dev_attr.devacc_attr_version            = DDI_DEVICE_ATTR_V0;
        dev_attr.devacc_attr_endian_flags       = DDI_STRUCTURE_LE_ACC;
        dev_attr.devacc_attr_dataorder          = DDI_STRICTORDER_ACC;

        /* Allocate the TD pool structures */
        info->td_pools = kmem_zalloc((sizeof (uhci_bulk_isoc_td_pool_t) *
            info->num_pools), KM_SLEEP);
        if ((info->td_pools = kmem_zalloc(
            (sizeof (uhci_bulk_isoc_td_pool_t) * info->num_pools),
            KM_SLEEP)) == NULL) {
                USB_DPRINTF_L2(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
                    "uhci_alloc_memory_for_tds: alloc td_pools failed");

                return (USB_FAILURE);
        }

        for (i = 0; i < info->num_pools; i++) {
                if (info->num_pools == 1) {
                        num = num_tds;
                } else if (i < (info->num_pools - 1)) {
                        num = UHCI_MAX_TD_NUM_PER_POOL;
                } else {
                        num = (num_tds % UHCI_MAX_TD_NUM_PER_POOL);
                }

                td_pool_ptr1 = &info->td_pools[i];

                /* Allocate the bulk TD pool DMA handle */
                if (ddi_dma_alloc_handle(uhcip->uhci_dip,
                    &uhcip->uhci_dma_attr, DDI_DMA_SLEEP, 0,
                    &td_pool_ptr1->dma_handle) != DDI_SUCCESS) {

                        for (j = 0; j < i; j++) {
                                td_pool_ptr2 = &info->td_pools[j];
                                result = ddi_dma_unbind_handle(
                                    td_pool_ptr2->dma_handle);
```



```
                                ASSERT(result == DDI_SUCCESS);
                                ddi_dma_mem_free(&td_pool_ptr2->mem_handle);
                                ddi_dma_free_handle(&td_pool_ptr2->dma_handle);
                        }

                        kmem_free(info->td_pools,
                            (sizeof (uhci_bulk_isoc_td_pool_t) *
                            info->num_pools));

                        return (USB_FAILURE);
                }

                /* Allocate the memory for the bulk TD pool */
                if (ddi_dma_mem_alloc(td_pool_ptr1->dma_handle,
                    num * sizeof (uhci_td_t), &dev_attr,
                    DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, 0,
                    &td_pool_ptr1->pool_addr, &real_length,
                    &td_pool_ptr1->mem_handle) != DDI_SUCCESS) {

                        ddi_dma_free_handle(&td_pool_ptr1->dma_handle);

                        for (j = 0; j < i; j++) {
                                td_pool_ptr2 = &info->td_pools[j];
                                result = ddi_dma_unbind_handle(
                                    td_pool_ptr2->dma_handle);
                                ASSERT(result == DDI_SUCCESS);
                                ddi_dma_mem_free(&td_pool_ptr2->mem_handle);
                                ddi_dma_free_handle(&td_pool_ptr2->dma_handle);
                        }

                        kmem_free(info->td_pools,
                            (sizeof (uhci_bulk_isoc_td_pool_t) *
                            info->num_pools));

                        return (USB_FAILURE);
                }

                /* Map the bulk TD pool into the I/O address space */
                result = ddi_dma_addr_bind_handle(td_pool_ptr1->dma_handle,
                    NULL, (caddr_t)td_pool_ptr1->pool_addr, real_length,
                    DDI_DMA_RDWR | DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL,
                    &td_pool_ptr1->cookie, &ccount);

                /* Process the result */
                err = USB_SUCCESS;

                if (result != DDI_DMA_MAPPED) {
                        USB_DPRINTF_L2(PRINT_MASK_ATTA, uhcip->uhci_log_hdl,
                            "uhci_allocate_memory_for_tds: Result = %d",
                            result);
                        uhci_decode_ddi_dma_addr_bind_handle_result(uhcip,
                            result);

                        err = USB_FAILURE;
                }

                if ((result == DDI_DMA_MAPPED) && (ccount != 1)) {
                        /* The cookie count should be 1 */
                        USB_DPRINTF_L2(PRINT_MASK_ATTA,
                            uhcip->uhci_log_hdl,
                            "uhci_allocate_memory_for_tds: "
                            "More than 1 cookie");

                        result = ddi_dma_unbind_handle(
                            td_pool_ptr1->dma_handle);
                        ASSERT(result == DDI_SUCCESS);
```

```
3217                        err = USB_FAILURE;
3218                }

3220                if (err == USB_FAILURE) {

3222                        ddi_dma_mem_free(&td_pool_ptr1->mem_handle);
3223                        ddi_dma_free_handle(&td_pool_ptr1->dma_handle);

3225                        for (j = 0; j < i; j++) {
3226                                td_pool_ptr2 = &info->td_pools[j];
3227                                result = ddi_dma_unbind_handle(
3228                                    td_pool_ptr2->dma_handle);
3229                                ASSERT(result == DDI_SUCCESS);
3230                                ddi_dma_mem_free(&td_pool_ptr2->mem_handle);
3231                                ddi_dma_free_handle(&td_pool_ptr2->dma_handle);
3232                        }

3234                        kmem_free(info->td_pools,
3235                            (sizeof (uhci_bulk_isoc_td_pool_t) *
3236                            info->num_pools));

3238                        return (USB_FAILURE);
3239                }

3241                bzero((void *)td_pool_ptr1->pool_addr,
3242                    num * sizeof (uhci_td_t));
3243                td_pool_ptr1->num_tds = (ushort_t)num;
3244        }

3246        return (USB_SUCCESS);
3247 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   72910 Thu Oct 23 11:04:59 2014
new/usr/src/uts/common/io/usb/usba/whcdi.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

1418 /* Acknowledge WUSB Device Disconnect notification, refer to WUSB 1.0/7.6.2 */
1419 int
1420 wusb_hc_ack_disconn(wusb_hc_data_t *hc_data, uint8_t addr)
1421 {
1422         wusb_ie_dev_disconnect_t        *disconn_ie;
1423         uint8_t                         iehdl;
1424         int                             rval;

1426         ASSERT(mutex_owned(&hc_data->hc_mutex));

1428         /*
1429          * the scheme ensures each time only one device addr
1430          * is set each time
1431          */
1432         disconn_ie = kmem_zalloc(sizeof (wusb_ie_dev_disconnect_t), KM_SLEEP);
1433         if (!disconn_ie) {
1434                 return (USB_NO_RESOURCES);
1435         }

1434         disconn_ie->bIEIdentifier = WUSB_IE_DEV_DISCONNECT;
1435         disconn_ie->bDeviceAddress[0] = addr;
1436         /* padding, no active wusb device addr will be 1 */
1437         disconn_ie->bDeviceAddress[1] = 1;
1438         disconn_ie->bLength = 4;

1440         rval = wusb_hc_get_iehdl(hc_data, (wusb_ie_header_t *)disconn_ie,
1441             &iehdl);
1442         if (rval != USB_SUCCESS) {
1443                 USB_DPRINTF_L2(DPRINT_MASK_WHCDI, whcdi_log_handle,
1444                     "wusb_hc_ack_disconn: get ie handle fails");
1445                 kmem_free(disconn_ie, sizeof (wusb_ie_dev_disconnect_t));

1447                 return (rval);
1448         }

1450         rval = wusb_hc_add_mmc_ie(hc_data, 0, 0, iehdl,
1451             disconn_ie->bLength, (uint8_t *)disconn_ie);
1452         if (rval != USB_SUCCESS) {
1453                 USB_DPRINTF_L2(DPRINT_MASK_WHCDI, whcdi_log_handle,
1454                     "wusb_hc_ack_disconn: add dev disconnect ie fails");
1455                 wusb_hc_free_iehdl(hc_data, iehdl);
1456                 kmem_free(disconn_ie, sizeof (wusb_ie_dev_disconnect_t));

1458                 return (rval);
1459         }

1461         mutex_exit(&hc_data->hc_mutex);
1462         /*
1463          * WUSB 1.0/7.5.4 requires the IE to be transmitted at least
1464          * 100ms before ceasing, wait for 150ms here
1465          */
1466         delay(drv_usectohz(150000));
1467         mutex_enter(&hc_data->hc_mutex);

1469         /* cease transmitting the IE */
1470         (void) wusb_hc_remove_mmc_ie(hc_data, (uint8_t)iehdl);
1471         wusb_hc_free_iehdl(hc_data, iehdl);
1472         kmem_free(disconn_ie, sizeof (wusb_ie_dev_disconnect_t));
```



```
1474         return (USB_SUCCESS);
1475 }
______unchanged_portion_omitted_
```

```
**********************************************************
   69395 Thu Oct 23 11:04:59 2014
new/usr/src/uts/common/io/xge/drv/xgell.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

2492 static int
2493 xgell_stats_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2494 {
2495         xgelldev_t *lldev = (xgelldev_t *)cp;
2496         xge_hal_status_e status;
2497         int count = 0, retsize;
2498         char *buf;

2500         buf = kmem_alloc(XGELL_STATS_BUFSIZE, KM_SLEEP);
2501         if (buf == NULL) {
2502                 return (ENOSPC);
2503         }

2502         status = xge_hal_aux_stats_tmac_read(lldev->devh, XGELL_STATS_BUFSIZE,
2503             buf, &retsize);
2504         if (status != XGE_HAL_OK) {
2505                 kmem_free(buf, XGELL_STATS_BUFSIZE);
2506                 xge_debug_ll(XGE_ERR, "tmac_read(): status %d", status);
2507                 return (EINVAL);
2508         }
2509         count += retsize;

2511         status = xge_hal_aux_stats_rmac_read(lldev->devh,
2512             XGELL_STATS_BUFSIZE - count,
2513             buf+count, &retsize);
2514         if (status != XGE_HAL_OK) {
2515                 kmem_free(buf, XGELL_STATS_BUFSIZE);
2516                 xge_debug_ll(XGE_ERR, "rmac_read(): status %d", status);
2517                 return (EINVAL);
2518         }
2519         count += retsize;

2521         status = xge_hal_aux_stats_pci_read(lldev->devh,
2522             XGELL_STATS_BUFSIZE - count, buf + count, &retsize);
2523         if (status != XGE_HAL_OK) {
2524                 kmem_free(buf, XGELL_STATS_BUFSIZE);
2525                 xge_debug_ll(XGE_ERR, "pci_read(): status %d", status);
2526                 return (EINVAL);
2527         }
2528         count += retsize;

2530         status = xge_hal_aux_stats_sw_dev_read(lldev->devh,
2531             XGELL_STATS_BUFSIZE - count, buf + count, &retsize);
2532         if (status != XGE_HAL_OK) {
2533                 kmem_free(buf, XGELL_STATS_BUFSIZE);
2534                 xge_debug_ll(XGE_ERR, "sw_dev_read(): status %d", status);
2535                 return (EINVAL);
2536         }
2537         count += retsize;

2539         status = xge_hal_aux_stats_hal_read(lldev->devh,
2540             XGELL_STATS_BUFSIZE - count, buf + count, &retsize);
2541         if (status != XGE_HAL_OK) {
2542                 kmem_free(buf, XGELL_STATS_BUFSIZE);
2543                 xge_debug_ll(XGE_ERR, "pci_read(): status %d", status);
2544                 return (EINVAL);
2545         }
2546         count += retsize;
```



```
2548         *(buf + count - 1) = '\0'; /* remove last '\n' */
2549         (void) mi_mpprintf(mp, "%s", buf);
2550         kmem_free(buf, XGELL_STATS_BUFSIZE);

2552         return (0);
2553 }

2555 static int
2556 xgell_pciconf_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2557 {
2558         xgelldev_t *lldev = (xgelldev_t *)cp;
2559         xge_hal_status_e status;
2560         int retsize;
2561         char *buf;

2563         buf = kmem_alloc(XGELL_PCICONF_BUFSIZE, KM_SLEEP);
2567         if (buf == NULL) {
2568                 return (ENOSPC);
2569         }
2564         status = xge_hal_aux_pci_config_read(lldev->devh, XGELL_PCICONF_BUFSIZE,
2565             buf, &retsize);
2566         if (status != XGE_HAL_OK) {
2567                 kmem_free(buf, XGELL_PCICONF_BUFSIZE);
2568                 xge_debug_ll(XGE_ERR, "pci_config_read(): status %d", status);
2569                 return (EINVAL);
2570         }
2571         *(buf + retsize - 1) = '\0'; /* remove last '\n' */
2572         (void) mi_mpprintf(mp, "%s", buf);
2573         kmem_free(buf, XGELL_PCICONF_BUFSIZE);

2575         return (0);
2576 }

2578 static int
2579 xgell_about_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2580 {
2581         xgelldev_t *lldev = (xgelldev_t *)cp;
2582         xge_hal_status_e status;
2583         int retsize;
2584         char *buf;

2586         buf = kmem_alloc(XGELL_ABOUT_BUFSIZE, KM_SLEEP);
2593         if (buf == NULL) {
2594                 return (ENOSPC);
2595         }
2587         status = xge_hal_aux_about_read(lldev->devh, XGELL_ABOUT_BUFSIZE,
2588             buf, &retsize);
2589         if (status != XGE_HAL_OK) {
2590                 kmem_free(buf, XGELL_ABOUT_BUFSIZE);
2591                 xge_debug_ll(XGE_ERR, "about_read(): status %d", status);
2592                 return (EINVAL);
2593         }
2594         *(buf + retsize - 1) = '\0'; /* remove last '\n' */
2595         (void) mi_mpprintf(mp, "%s", buf);
2596         kmem_free(buf, XGELL_ABOUT_BUFSIZE);

2598         return (0);
2599 }

2601 static unsigned long bar0_offset = 0x110; /* adapter_control */

2603 static int
2604 xgell_bar0_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2605 {
2606         xgelldev_t *lldev = (xgelldev_t *)cp;
```

```
2607         xge_hal_status_e status;
2608         int retsize;
2609         char *buf;

2611         buf = kmem_alloc(XGELL_IOCTL_BUFSIZE, KM_SLEEP);
2621         if (buf == NULL) {
2622                 return (ENOSPC);
2623         }
2612         status = xge_hal_aux_bar0_read(lldev->devh, bar0_offset,
2613             XGELL_IOCTL_BUFSIZE, buf, &retsize);
2614         if (status != XGE_HAL_OK) {
2615                 kmem_free(buf, XGELL_IOCTL_BUFSIZE);
2616                 xge_debug_ll(XGE_ERR, "bar0_read(): status %d", status);
2617                 return (EINVAL);
2618         }
2619         *(buf + retsize - 1) = '\0'; /* remove last '\n' */
2620         (void) mi_mpprintf(mp, "%s", buf);
2621         kmem_free(buf, XGELL_IOCTL_BUFSIZE);

2623         return (0);
2624 }
______unchanged_portion_omitted_

2648 static int
2649 xgell_debug_level_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2650 {
2651         char *buf;

2653         buf = kmem_alloc(XGELL_IOCTL_BUFSIZE, KM_SLEEP);
2666         if (buf == NULL) {
2667                 return (ENOSPC);
2668         }
2654         (void) mi_mpprintf(mp, "debug_level %d", xge_hal_driver_debug_level());
2655         kmem_free(buf, XGELL_IOCTL_BUFSIZE);

2657         return (0);
2658 }
______unchanged_portion_omitted_

2677 static int
2678 xgell_debug_module_mask_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2679 {
2680         char *buf;

2682         buf = kmem_alloc(XGELL_IOCTL_BUFSIZE, KM_SLEEP);
2698         if (buf == NULL) {
2699                 return (ENOSPC);
2700         }
2683         (void) mi_mpprintf(mp, "debug_module_mask 0x%08x",
2684             xge_hal_driver_debug_module_mask());
2685         kmem_free(buf, XGELL_IOCTL_BUFSIZE);

2687         return (0);
2688 }
______unchanged_portion_omitted_

2712 static int
2713 xgell_devconfig_get(queue_t *q, mblk_t *mp, caddr_t cp, cred_t *credp)
2714 {
2715         xgelldev_t *lldev = (xgelldev_t *)(void *)cp;
2716         xge_hal_status_e status;
2717         int retsize;
2718         char *buf;

2720         buf = kmem_alloc(XGELL_DEVCONF_BUFSIZE, KM_SLEEP);
2739         if (buf == NULL) {
```



```
2740                 return (ENOSPC);
2741         }
2721         status = xge_hal_aux_device_config_read(lldev->devh,
2722             XGELL_DEVCONF_BUFSIZE, buf, &retsize);
2723         if (status != XGE_HAL_OK) {
2724                 kmem_free(buf, XGELL_DEVCONF_BUFSIZE);
2725                 xge_debug_ll(XGE_ERR, "device_config_read(): status %d",
2726                     status);
2727                 return (EINVAL);
2728         }
2729         *(buf + retsize - 1) = '\0'; /* remove last '\n' */
2730         (void) mi_mpprintf(mp, "%s", buf);
2731         kmem_free(buf, XGELL_DEVCONF_BUFSIZE);

2733         return (0);
2734 }
______unchanged_portion_omitted_
```

```
**********************************************************
   93566 Thu Oct 23 11:04:59 2014
new/usr/src/uts/common/io/yge/yge.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 989 static int
 990 yge_add_intr(yge_dev_t *dev, int intr_type)
 991 {
 992         dev_info_t              *dip;
 993         int                     count;
 994         int                     actual;
 995         int                     rv;
 996         int                     i, j;

 998         dip = dev->d_dip;

1000         rv = ddi_intr_get_nintrs(dip, intr_type, &count);
1001         if ((rv != DDI_SUCCESS) || (count == 0)) {
1002                 yge_error(dev, NULL,
1003                     "ddi_intr_get_nintrs failed, rv %d, count %d", rv, count);
1004                 return (DDI_FAILURE);
1005         }

1007         /*
1008          * Allocate the interrupt.  Note that we only bother with a single
1009          * interrupt.  One could argue that for MSI devices with dual ports,
1010          * it would be nice to have a separate interrupt per port.  But right
1011          * now I don't know how to configure that, so we'll just settle for
1012          * a single interrupt.
1013          */
1014         dev->d_intrcnt = 1;

1016         dev->d_intrsize = count * sizeof (ddi_intr_handle_t);
1017         dev->d_intrh = kmem_zalloc(dev->d_intrsize, KM_SLEEP);
1018         if (dev->d_intrh == NULL) {
1019                 yge_error(dev, NULL, "Unable to allocate interrupt handle");
1020                 return (DDI_FAILURE);
1021         }

1019         rv = ddi_intr_alloc(dip, dev->d_intrh, intr_type, 0, dev->d_intrcnt,
1020             &actual, DDI_INTR_ALLOC_STRICT);
1021         if ((rv != DDI_SUCCESS) || (actual == 0)) {
1022                 yge_error(dev, NULL,
1023                     "Unable to allocate interrupt, %d, count %d",
1024                     rv, actual);
1025                 kmem_free(dev->d_intrh, dev->d_intrsize);
1026                 return (DDI_FAILURE);
1027         }

1029         if ((rv = ddi_intr_get_pri(dev->d_intrh[0], &dev->d_intrpri)) !=
1030             DDI_SUCCESS) {
1031                 for (i = 0; i < dev->d_intrcnt; i++)
1032                         (void) ddi_intr_free(dev->d_intrh[i]);
1033                 yge_error(dev, NULL,
1034                     "Unable to get interrupt priority, %d", rv);
1035                 kmem_free(dev->d_intrh, dev->d_intrsize);
1036                 return (DDI_FAILURE);
1037         }

1039         if ((rv = ddi_intr_get_cap(dev->d_intrh[0], &dev->d_intrcap)) !=
1040             DDI_SUCCESS) {
1041                 yge_error(dev, NULL,
1042                     "Unable to get interrupt capabilities, %d", rv);
```



```
1043                 for (i = 0; i < dev->d_intrcnt; i++)
1044                         (void) ddi_intr_free(dev->d_intrh[i]);
1045                 kmem_free(dev->d_intrh, dev->d_intrsize);
1046                 return (DDI_FAILURE);
1047         }

1049         /* register interrupt handler to kernel */
1050         for (i = 0; i < dev->d_intrcnt; i++) {
1051                 if ((rv = ddi_intr_add_handler(dev->d_intrh[i], yge_intr,
1052                     dev, NULL)) != DDI_SUCCESS) {
1053                         yge_error(dev, NULL,
1054                             "Unable to add interrupt handler, %d", rv);
1055                         for (j = 0; j < i; j++)
1056                                 (void) ddi_intr_remove_handler(dev->d_intrh[j]);
1057                         for (i = 0; i < dev->d_intrcnt; i++)
1058                                 (void) ddi_intr_free(dev->d_intrh[i]);
1059                         kmem_free(dev->d_intrh, dev->d_intrsize);
1060                         return (DDI_FAILURE);
1061                 }
1062         }

1064         mutex_init(&dev->d_rxlock, NULL, MUTEX_DRIVER,
1065             DDI_INTR_PRI(dev->d_intrpri));
1066         mutex_init(&dev->d_txlock, NULL, MUTEX_DRIVER,
1067             DDI_INTR_PRI(dev->d_intrpri));
1068         mutex_init(&dev->d_phylock, NULL, MUTEX_DRIVER,
1069             DDI_INTR_PRI(dev->d_intrpri));
1070         mutex_init(&dev->d_task_mtx, NULL, MUTEX_DRIVER,
1071             DDI_INTR_PRI(dev->d_intrpri));

1073         return (DDI_SUCCESS);
1074 }
______unchanged_portion_omitted_
```

```
**********************************************************
  248730 Thu Oct 23 11:05:00 2014
new/usr/src/uts/common/os/sunddi.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

5510 /*
5511  * failing to remove a minor node is not of interest
5512  * therefore we do not generate an error message
5513  */
5514 static int
5515 i_log_devfs_minor_remove(dev_info_t *dip, char *minor_name)
5516 {
5517         char *pathname, *class_name;
5518         sysevent_t *ev;
5519         sysevent_id_t eid;
5520         sysevent_value_t se_val;
5521         sysevent_attr_list_t *ev_attr_list = NULL;

5523         /*
5524          * only log ddi_remove_minor_node() calls outside the scope
5525          * of attach/detach reconfigurations and when the dip is
5526          * still initialized.
5527          */
5528         if (DEVI_IS_ATTACHING(dip) || DEVI_IS_DETACHING(dip) ||
5529             (i_ddi_node_state(dip) < DS_INITIALIZED)) {
5530                 return (DDI_SUCCESS);
5531         }

5533         i_ddi_di_cache_invalidate();

5535         ev = sysevent_alloc(EC_DEVFS, ESC_DEVFS_MINOR_REMOVE, EP_DDI, SE_SLEEP);
5536         if (ev == NULL) {
5537                 return (DDI_SUCCESS);
5538         }

5540         pathname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
5541         if (pathname == NULL) {
5542                 sysevent_free(ev);
5543                 return (DDI_SUCCESS);
5544         }

5542         (void) ddi_pathname(dip, pathname);
5543         ASSERT(strlen(pathname));
5544         se_val.value_type = SE_DATA_TYPE_STRING;
5545         se_val.value.sv_string = pathname;
5546         if (sysevent_add_attr(&ev_attr_list, DEVFS_PATHNAME,
5547             &se_val, SE_SLEEP) != 0) {
5548                 kmem_free(pathname, MAXPATHLEN);
5549                 sysevent_free(ev);
5550                 return (DDI_SUCCESS);
5551         }

5553         kmem_free(pathname, MAXPATHLEN);

5555         /*
5556          * allow for NULL minor names
5557          */
5558         if (minor_name != NULL) {
5559                 se_val.value.sv_string = minor_name;
5560                 if (sysevent_add_attr(&ev_attr_list, DEVFS_MINOR_NAME,
5561                     &se_val, SE_SLEEP) != 0) {
5562                         sysevent_free_attr(ev_attr_list);
5563                         goto fail;
```



```
5564                 }
5565         }

5567         if ((class_name = i_ddi_devi_class(dip)) != NULL) {
5568                 /* add the device class, driver name and instance attributes */

5570                 se_val.value_type = SE_DATA_TYPE_STRING;
5571                 se_val.value.sv_string = class_name;
5572                 if (sysevent_add_attr(&ev_attr_list,
5573                     DEVFS_DEVI_CLASS, &se_val, SE_SLEEP) != 0) {
5574                         sysevent_free_attr(ev_attr_list);
5575                         goto fail;
5576                 }

5578                 se_val.value_type = SE_DATA_TYPE_STRING;
5579                 se_val.value.sv_string = (char *)ddi_driver_name(dip);
5580                 if (sysevent_add_attr(&ev_attr_list,
5581                     DEVFS_DRIVER_NAME, &se_val, SE_SLEEP) != 0) {
5582                         sysevent_free_attr(ev_attr_list);
5583                         goto fail;
5584                 }

5586                 se_val.value_type = SE_DATA_TYPE_INT32;
5587                 se_val.value.sv_int32 = ddi_get_instance(dip);
5588                 if (sysevent_add_attr(&ev_attr_list,
5589                     DEVFS_INSTANCE, &se_val, SE_SLEEP) != 0) {
5590                         sysevent_free_attr(ev_attr_list);
5591                         goto fail;
5592                 }

5594         }

5596         if (sysevent_attach_attributes(ev, ev_attr_list) != 0) {
5597                 sysevent_free_attr(ev_attr_list);
5598         } else {
5599                 (void) log_sysevent(ev, SE_SLEEP, &eid);
5600         }
5601 fail:
5602         sysevent_free(ev);
5603         return (DDI_SUCCESS);
5604 }
______unchanged_portion_omitted_

7753 /*
7754  * Allocate and initialize a device id.
7755  */
7756 int
7757 ddi_devid_init(
7758         dev_info_t      *dip,
7759         ushort_t        devid_type,
7760         ushort_t        nbytes,
7761         void            *id,
7762         ddi_devid_t     *ret_devid)
7763 {
7764         impl_devid_t    *i_devid;
7765         int             sz = sizeof (*i_devid) + nbytes - sizeof (char);
7766         int             driver_len;
7767         const char      *driver_name;

7769         switch (devid_type) {
7770         case DEVID_SCSI3_WWN:
7771                 /*FALLTHRU*/
7772         case DEVID_SCSI_SERIAL:
7773                 /*FALLTHRU*/
7774         case DEVID_ATA_SERIAL:
7775                 /*FALLTHRU*/
```

```
7776         case DEVID_ENCAP:
7777                 if (nbytes == 0)
7778                         return (DDI_FAILURE);
7779                 if (id == NULL)
7780                         return (DDI_FAILURE);
7781                 break;
7782         case DEVID_FAB:
7783                 if (nbytes != 0)
7784                         return (DDI_FAILURE);
7785                 if (id != NULL)
7786                         return (DDI_FAILURE);
7787                 nbytes = sizeof (int) +
7788                     sizeof (struct timeval32) + sizeof (short);
7789                 sz += nbytes;
7790                 break;
7791         default:
7792                 return (DDI_FAILURE);
7793         }

7795         i_devid = kmem_zalloc(sz, KM_SLEEP);
7799         if ((i_devid = kmem_zalloc(sz, KM_SLEEP)) == NULL)
7800                 return (DDI_FAILURE);

7797         i_devid->did_magic_hi = DEVID_MAGIC_MSB;
7798         i_devid->did_magic_lo = DEVID_MAGIC_LSB;
7799         i_devid->did_rev_hi = DEVID_REV_MSB;
7800         i_devid->did_rev_lo = DEVID_REV_LSB;
7801         DEVID_FORMTYPE(i_devid, devid_type);
7802         DEVID_FORMLEN(i_devid, nbytes);

7804         /* Fill in driver name hint */
7805         driver_name = ddi_driver_name(dip);
7806         driver_len = strlen(driver_name);
7807         if (driver_len > DEVID_HINT_SIZE) {
7808                 /* Pick up last four characters of driver name */
7809                 driver_name += driver_len - DEVID_HINT_SIZE;
7810                 driver_len = DEVID_HINT_SIZE;
7811         }

7813         bcopy(driver_name, i_devid->did_driver, driver_len);

7815         /* Fill in id field */
7816         if (devid_type == DEVID_FAB) {
7817                 char            *cp;
7818                 uint32_t        hostid;
7819                 struct timeval32 timestamp32;
7820                 int             i;
7821                 int             *ip;
7822                 short           gen;

7824                 /* increase the generation number */
7825                 mutex_enter(&devid_gen_mutex);
7826                 gen = devid_gen_number++;
7827                 mutex_exit(&devid_gen_mutex);

7829                 cp = i_devid->did_id;

7831                 /* Fill in host id (big-endian byte ordering) */
7832                 hostid = zone_get_hostid(NULL);
7833                 *cp++ = hibyte(hiword(hostid));
7834                 *cp++ = lobyte(hiword(hostid));
7835                 *cp++ = hibyte(loword(hostid));
7836                 *cp++ = lobyte(loword(hostid));

7838                 /*
7839                  * Fill in timestamp (big-endian byte ordering)
```



```
7840                  *
7841                  * (Note that the format may have to be changed
7842                  * before 2038 comes around, though it's arguably
7843                  * unique enough as it is..)
7844                  */
7845                 uniqtime32(&timestamp32);
7846                 ip = (int *)&timestamp32;
7847                 for (i = 0;
7848                     i < sizeof (timestamp32) / sizeof (int); i++, ip++) {
7849                         int     val;
7850                         val = *ip;
7851                         *cp++ = hibyte(hiword(val));
7852                         *cp++ = lobyte(hiword(val));
7853                         *cp++ = hibyte(loword(val));
7854                         *cp++ = lobyte(loword(val));
7855                 }

7857                 /* fill in the generation number */
7858                 *cp++ = hibyte(gen);
7859                 *cp++ = lobyte(gen);
7860         } else
7861                 bcopy(id, i_devid->did_id, nbytes);

7863         /* return device id */
7864         *ret_devid = (ddi_devid_t)i_devid;
7865         return (DDI_SUCCESS);
7866 }
```

_____***unchanged_portion_omitted_***

****
49471 Thu Oct 23 11:05:00 2014
new/usr/src/uts/common/rpc/sec_gss/svc_rpcsec_gss.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
****
_____***unchanged_portion_omitted_***

```
1840 /*
1841  * Add retransmit entry to the context cache entry for a new xid.
1842  * If there is already an entry, delete it before adding the new one.
1843  */
1844 static void retrans_add(client, xid, result)
1845         svc_rpc_gss_data *client;
1846         uint32_t        xid;
1847         rpc_gss_init_res *result;
1848 {
1849         retrans_entry   *rdata;

1851         if (client->retrans_data && client->retrans_data->xid == xid)
1852                 return;

1854         rdata = kmem_zalloc(sizeof (*rdata), KM_SLEEP);

1856         if (rdata == NULL)
1857                 return;

1856         rdata->xid = xid;
1857         rdata->result = *result;

1859         if (result->token.length != 0) {
1860                 GSS_DUP_BUFFER(rdata->result.token, result->token);
1861         }

1863         if (client->retrans_data)
1864                 retrans_del(client);

1866         client->retrans_data = rdata;
1867 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4481 Thu Oct 23 11:05:00 2014
new/usr/src/uts/i86pc/io/gfx_private/gfxp_devmap.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
```

```
/*
 * 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"

#include <sys/debug.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/buf.h>
#include <sys/errno.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/signal.h>
#include <vm/page.h>
#include <vm/as.h>
#include <vm/hat.h>
#include <vm/seg.h>
#include <vm/seg_dev.h>
#include <vm/hat_i86.h>
#include <sys/ddi.h>
#include <sys/devops.h>
#include <sys/sunddi.h>
#include <sys/ddi_impldefs.h>
#include <sys/fs/snode.h>
#include <sys/pci.h>
#include <sys/vmsystm.h>
#include <sys/int_fmtio.h>
#include "gfx_private.h"

#ifdef __xpv
#include <sys/hypervisor.h>
#endif

/*
 * Create a dummy ddi_umem_cookie given to gfxp_devmap_umem_setup().
 */
```



```
ddi_umem_cookie_t
gfxp_umem_cookie_init(caddr_t kva, size_t size)
{
        struct ddi_umem_cookie *umem_cookie;

        umem_cookie = kmem_zalloc(sizeof (struct ddi_umem_cookie), KM_SLEEP);

        if (umem_cookie == NULL)
                return (NULL);

        umem_cookie->cvaddr = kva;
        umem_cookie->type = KMEM_NON_PAGEABLE;
        umem_cookie->size = size;

        return ((ddi_umem_cookie_t *)umem_cookie);
}
______unchanged_portion_omitted_
```

```
**********************************************************
    8026 Thu Oct 23 11:05:00 2014
new/usr/src/uts/i86pc/io/gfx_private/gfxp_pci.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (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.
 */

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

#include <sys/debug.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/buf.h>
#include <sys/errno.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/signal.h>
#include <sys/file.h>
#include <sys/uio.h>
#include <sys/ioctl.h>
#include <sys/map.h>
#include <sys/proc.h>
#include <sys/user.h>
#include <sys/mman.h>
#include <sys/cred.h>
#include <sys/open.h>
#include <sys/stat.h>
#include <sys/utsname.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vnode.h>
#include <vm/page.h>
#include <vm/as.h>
#include <vm/hat.h>
#include <vm/seg.h>
#include <sys/ddi.h>
#include <sys/devops.h>
#include <sys/sunddi.h>
#include <sys/ddi_impldefs.h>
#include <sys/fs/snode.h>
```



```
#include <sys/pci.h>
#include <sys/modctl.h>
#include <sys/uio.h>
#include <sys/visual_io.h>
#include <sys/fbio.h>
#include <sys/ddidmareq.h>
#include <sys/tnf_probe.h>
#include <sys/kstat.h>
#include <sys/callb.h>
#include <sys/pci_cfgspace.h>
#include "gfx_private.h"

typedef struct gfxp_pci_bsf {
        uint16_t        vendor;
        uint16_t        device;
        uint8_t         bus;
        uint8_t         slot;
        uint8_t         function;
        uint8_t         found;
        dev_info_t      *dip;
} gfxp_pci_bsf_t;
______unchanged_portion_omitted_

gfxp_acc_handle_t
gfxp_pci_init_handle(uint8_t bus, uint8_t slot, uint8_t function,
        uint16_t *vendor, uint16_t *device)
{
        dev_info_t      *dip;
        gfxp_pci_bsf_t  *pci_bsf;

        /*
         * Find a PCI device based on its address, and return a unique handle
         * to be used in subsequent calls to read from or write to the config
         * space of this device.
         */

        pci_bsf = kmem_zalloc(sizeof (gfxp_pci_bsf_t), KM_SLEEP);
        if ((pci_bsf = kmem_zalloc(sizeof (gfxp_pci_bsf_t), KM_SLEEP))
                        == NULL) {
                return (NULL);
        }

        pci_bsf->bus = bus;
        pci_bsf->slot = slot;
        pci_bsf->function = function;

        ddi_walk_devs(ddi_root_node(), gfxp_pci_find_bsf, pci_bsf);

        if (pci_bsf->found) {
                dip = pci_bsf->dip;

                if (vendor) *vendor = pci_bsf->vendor;
                if (device) *device = pci_bsf->device;
        } else {
                dip = NULL;
                if (vendor) *vendor = 0x0000;
                if (device) *device = 0x0000;
        }

        kmem_free(pci_bsf, sizeof (gfxp_pci_bsf_t));

        return ((gfxp_acc_handle_t)dip);
}
______unchanged_portion_omitted_

int
```

```
gfxp_pci_device_present(uint16_t vendor, uint16_t device)
{
        gfxp_pci_bsf_t  *pci_bsf;
        int             rv;

        /*
         * Find a PCI device based on its device and vendor id.
         */

        pci_bsf = kmem_zalloc(sizeof (gfxp_pci_bsf_t), KM_SLEEP);
        if ((pci_bsf = kmem_zalloc(sizeof (gfxp_pci_bsf_t), KM_SLEEP)) == NULL)
            return (0);

        pci_bsf->vendor = vendor;
        pci_bsf->device = device;
        ddi_walk_devs(ddi_root_node(), gfxp_pci_find_vd, pci_bsf);

        if (pci_bsf->found) {
                rv = 1;
        } else {
                rv = 0;
        }

        kmem_free(pci_bsf, sizeof (gfxp_pci_bsf_t));

        return (rv);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   44789 Thu Oct 23 11:05:01 2014
new/usr/src/uts/intel/io/dktp/dcdev/dadk.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 716 /* ARGSUSED  */
 717 int
 718 dadk_ioctl(opaque_t objp, dev_t dev, int cmd, intptr_t arg, int flag,
 719         cred_t *cred_p, int *rval_p)
 720 {
 721         struct dadk *dadkp = (struct dadk *)objp;

 723         switch (cmd) {
 724         case DKIOCGETDEF:
 725                 {
 726                 struct buf      *bp;
 727                 int             err, head;
 728                 unsigned char   *secbuf;
 729                 STRUCT_DECL(defect_header, adh);

 731                 STRUCT_INIT(adh, flag & FMODELS);

 733                 /*
 734                  * copyin header ....
 735                  * yields head number and buffer address
 736                  */
 737                 if (ddi_copyin((caddr_t)arg, STRUCT_BUF(adh), STRUCT_SIZE(adh),
 738                     flag))
 739                         return (EFAULT);
 740                 head = STRUCT_FGET(adh, head);
 741                 if (head < 0 || head >= dadkp->dad_phyg.g_head)
 742                         return (ENXIO);
 743                 secbuf = kmem_zalloc(NBPSCTR, KM_SLEEP);

 744                 if (!secbuf)
 745                         return (ENOMEM);
 745                 bp = getrbuf(KM_SLEEP);
 747                 if (!bp) {
 748                         kmem_free(secbuf, NBPSCTR);
 749                         return (ENOMEM);
 750                 }

 747                 bp->b_edev = dev;
 748                 bp->b_dev  = cmpdev(dev);
 749                 bp->b_flags = B_BUSY;
 750                 bp->b_resid = 0;
 751                 bp->b_bcount = NBPSCTR;
 752                 bp->b_un.b_addr = (caddr_t)secbuf;
 753                 bp->b_blkno = head; /* I had to put it somwhere! */
 754                 bp->b_forw = (struct buf *)dadkp;
 755                 bp->b_back = (struct buf *)DCMD_GETDEF;

 757                 mutex_enter(&dadkp->dad_cmd_mutex);
 758                 dadkp->dad_cmd_count++;
 759                 mutex_exit(&dadkp->dad_cmd_mutex);
 760                 FLC_ENQUE(dadkp->dad_flcobjp, bp);
 761                 err = biowait(bp);
 762                 if (!err) {
 763                         if (ddi_copyout((caddr_t)secbuf,
 764                             STRUCT_FGETP(adh, buffer), NBPSCTR, flag))
 765                                 err = ENXIO;
 766                 }
 767                 kmem_free(secbuf, NBPSCTR);
```



```
 768                 freerbuf(bp);
 769                 return (err);
 770                 }
 771         case DIOCTL_RWCMD:
 772                 {
 773                 struct dadkio_rwcmd *rwcmdp;
 774                 int status, rw;

 776                 /*
 777                  * copied in by cmdk and, if necessary, converted to the
 778                  * correct datamodel
 779                  */
 780                 rwcmdp = (struct dadkio_rwcmd *)(intptr_t)arg;

 782                 /*
 783                  * handle the complex cases here; we pass these
 784                  * through to the driver, which will queue them and
 785                  * handle the requests asynchronously.  The simpler
 786                  * cases ,which can return immediately, fail here, and
 787                  * the request reverts to the dadk_ioctl routine, while
 788                  *  will reroute them directly to the ata driver.
 789                  */
 790                 switch (rwcmdp->cmd) {
 791                         case DADKIO_RWCMD_READ :
 792                                 /*FALLTHROUGH*/
 793                         case DADKIO_RWCMD_WRITE:
 794                                 rw = ((rwcmdp->cmd == DADKIO_RWCMD_WRITE) ?
 795                                     B_WRITE : B_READ);
 796                                 status = dadk_dk_buf_setup(dadkp,
 797                                     (opaque_t)rwcmdp, dev, ((flag &FKIOCTL) ?
 798                                     UIO_SYSSPACE : UIO_USERSPACE), rw);
 799                                 return (status);
 800                         default:
 801                                 return (EINVAL);
 802                         }
 803                 }
 804         case DKIOC_UPDATEFW:

 806                 /*
 807                  * Require PRIV_ALL privilege to invoke DKIOC_UPDATEFW
 808                  * to protect the firmware update from malicious use
 809                  */
 810                 if (PRIV_POLICY(cred_p, PRIV_ALL, B_FALSE, EPERM, NULL) != 0)
 811                         return (EPERM);
 812                 else
 813                         return (dadk_ctl_ioctl(dadkp, cmd, arg, flag));

 815         case DKIOCFLUSHWRITECACHE:
 816                 {
 817                         struct buf *bp;
 818                         int err = 0;
 819                         struct dk_callback *dkc = (struct dk_callback *)arg;
 820                         struct cmpkt *pktp;
 821                         int is_sync = 1;

 823                         mutex_enter(&dadkp->dad_mutex);
 824                         if (dadkp->dad_noflush || !  dadkp->dad_wce) {
 825                                 err = dadkp->dad_noflush ? ENOTSUP : 0;
 826                                 mutex_exit(&dadkp->dad_mutex);
 827                                 /*
 828                                  * If a callback was requested: a
 829                                  * callback will always be done if the
 830                                  * caller saw the DKIOCFLUSHWRITECACHE
 831                                  * ioctl return 0, and never done if the
 832                                  * caller saw the ioctl return an error.
 833                                  */
```

```
834                                 if ((flag & FKIOCTL) && dkc != NULL &&
835                                     dkc->dkc_callback != NULL) {
836                                         (*dkc->dkc_callback)(dkc->dkc_cookie,
837                                             err);
838                                         /*
839                                          * Did callback and reported error.
840                                          * Since we did a callback, ioctl
841                                          * should return 0.
842                                          */
843                                         err = 0;
844                                 }
845                                 return (err);
846                         }
847                         mutex_exit(&dadkp->dad_mutex);

849                         bp = getrbuf(KM_SLEEP);

851                         bp->b_edev = dev;
852                         bp->b_dev  = cmpdev(dev);
853                         bp->b_flags = B_BUSY;
854                         bp->b_resid = 0;
855                         bp->b_bcount = 0;
856                         SET_BP_SEC(bp, 0);

858                         if ((flag & FKIOCTL) && dkc != NULL &&
859                             dkc->dkc_callback != NULL) {
860                                 struct dk_callback *dkc2 =
861                                     (struct dk_callback *)kmem_zalloc(
862                                     sizeof (struct dk_callback), KM_SLEEP);

864                                 bcopy(dkc, dkc2, sizeof (*dkc2));
865                                 bp->b_private = dkc2;
866                                 bp->b_iodone = dadk_flushdone;
867                                 is_sync = 0;
868                         }

870                         /*
871                          * Setup command pkt
872                          * dadk_pktprep() can't fail since DDI_DMA_SLEEP set
873                          */
874                         pktp = dadk_pktprep(dadkp, NULL, bp,
875                             dadk_iodone, DDI_DMA_SLEEP, NULL);

877                         pktp->cp_time = DADK_FLUSH_CACHE_TIME;

879                         *((char *)(pktp->cp_cdbp)) = DCMD_FLUSH_CACHE;
880                         pktp->cp_byteleft = 0;
881                         pktp->cp_private = NULL;
882                         pktp->cp_secleft = 0;
883                         pktp->cp_srtsec = -1;
884                         pktp->cp_bytexfer = 0;

886                         CTL_IOSETUP(dadkp->dad_ctlobjp, pktp);

888                         mutex_enter(&dadkp->dad_cmd_mutex);
889                         dadkp->dad_cmd_count++;
890                         mutex_exit(&dadkp->dad_cmd_mutex);
891                         FLC_ENQUE(dadkp->dad_flcobjp, bp);

893                         if (is_sync) {
894                                 err = biowait(bp);
895                                 freerbuf(bp);
896                         }
897                         return (err);
898                 }
899         default:
```



```
900                 if (!dadkp->dad_rmb)
901                         return (dadk_ctl_ioctl(dadkp, cmd, arg, flag));
902         }

904         switch (cmd) {
905         case CDROMSTOP:
906                 return (dadk_rmb_ioctl(dadkp, DCMD_STOP_MOTOR, 0,
907                     0, DADK_SILENT));
908         case CDROMSTART:
909                 return (dadk_rmb_ioctl(dadkp, DCMD_START_MOTOR, 0,
910                     0, DADK_SILENT));
911         case DKIOCLOCK:
912                 return (dadk_rmb_ioctl(dadkp, DCMD_LOCK, 0, 0, DADK_SILENT));
913         case DKIOCUNLOCK:
914                 return (dadk_rmb_ioctl(dadkp, DCMD_UNLOCK, 0, 0, DADK_SILENT));
915         case DKIOCEJECT:
916         case CDROMEJECT:
917                 {
918                         int ret;

920                         if (ret = dadk_rmb_ioctl(dadkp, DCMD_UNLOCK, 0, 0,
921                             DADK_SILENT)) {
922                                 return (ret);
923                         }
924                         if (ret = dadk_rmb_ioctl(dadkp, DCMD_EJECT, 0, 0,
925                             DADK_SILENT)) {
926                                 return (ret);
927                         }
928                         mutex_enter(&dadkp->dad_mutex);
929                         dadkp->dad_iostate = DKIO_EJECTED;
930                         cv_broadcast(&dadkp->dad_state_cv);
931                         mutex_exit(&dadkp->dad_mutex);

933                         return (0);

935                 }
936         default:
937                 return (ENOTTY);
938         /*
939          * cdrom audio commands
940          */
941         case CDROMPAUSE:
942                 cmd = DCMD_PAUSE;
943                 break;
944         case CDROMRESUME:
945                 cmd = DCMD_RESUME;
946                 break;
947         case CDROMPLAYMSF:
948                 cmd = DCMD_PLAYMSF;
949                 break;
950         case CDROMPLAYTRKIND:
951                 cmd = DCMD_PLAYTRKIND;
952                 break;
953         case CDROMREADTOCHDR:
954                 cmd = DCMD_READTOCHDR;
955                 break;
956         case CDROMREADTOCENTRY:
957                 cmd = DCMD_READTOCENT;
958                 break;
959         case CDROMVOLCTRL:
960                 cmd = DCMD_VOLCTRL;
961                 break;
962         case CDROMSUBCHNL:
963                 cmd = DCMD_SUBCHNL;
964                 break;
965         case CDROMREADMODE2:
```

```
 966                 cmd = DCMD_READMODE2;
 967                 break;
 968         case CDROMREADMODE1:
 969                 cmd = DCMD_READMODE1;
 970                 break;
 971         case CDROMREADOFFSET:
 972                 cmd = DCMD_READOFFSET;
 973                 break;
 974         }
 975         return (dadk_rmb_ioctl(dadkp, cmd, arg, flag, 0));
 976 }
______unchanged_portion_omitted_

1681 static int
1682 dadk_rmb_ioctl(struct dadk *dadkp, int cmd, intptr_t arg, int flags, int silent)

1684 {
1685         struct buf *bp;
1686         int err;
1687         struct cmpkt *pktp;

1689         bp = getrbuf(KM_SLEEP);
1694         if ((bp = getrbuf(KM_SLEEP)) == NULL) {
1695                 return (ENOMEM);
1696         }
1690         pktp = dadk_pktprep(dadkp, NULL, bp, dadk_rmb_iodone, NULL, NULL);
1691         if (!pktp) {
1692                 freerbuf(bp);
1693                 return (ENOMEM);
1694         }
1695         bp->b_back  = (struct buf *)arg;
1696         bp->b_forw  = (struct buf *)dadkp->dad_flcobjp;
1697         pktp->cp_passthru = (opaque_t)(intptr_t)silent;

1699         err = dadk_ctl_ioctl(dadkp, cmd, (uintptr_t)pktp, flags);
1700         freerbuf(bp);
1701         gda_free(dadkp->dad_ctlobjp, pktp, NULL);
1702         return (err);


1705 }
______unchanged_portion_omitted_
```

```
**********************************************************
   79987 Thu Oct 23 11:05:01 2014
new/usr/src/uts/intel/io/drm/i915_gem.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

2444 static int
2445 i915_gem_object_get_page_list(struct drm_gem_object *obj)
2446 {
2447         struct drm_i915_gem_object *obj_priv = obj->driver_private;
2448         caddr_t va;
2449         long i;

2451         if (obj_priv->page_list)
2452                 return 0;
2453         pgcnt_t np = btop(obj->size);

2455         obj_priv->page_list = kmem_zalloc(np * sizeof(caddr_t), KM_SLEEP);
2456         if (obj_priv->page_list == NULL) {
2457                 DRM_ERROR("Faled to allocate page list\n");
2458                 return ENOMEM;
2459         }

2457         for (i = 0, va = obj->kaddr; i < np; i++, va += PAGESIZE) {
2458                 obj_priv->page_list[i] = va;
2459         }
2460         return 0;
2461 }
______unchanged_portion_omitted_
```

```
**********************************************************
   32818 Thu Oct 23 11:05:01 2014
new/usr/src/uts/intel/io/heci/heci_init.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 758 /*
 759  * allocate_me_clients_storage - allocate storage for me clients
 760  *
 761  * @dev: Device object for our driver
 762  *
 763  * @return 0 on success, <0 on failure.
 764  */
 765 static int
 766 allocate_me_clients_storage(struct iamt_heci_device *dev)
 767 {
 768         struct heci_me_client *clients;
 769         struct heci_me_client *client;
 770         uint8_t num, i, j;
 771         int err;

 773         if (dev->num_heci_me_clients == 0)
 774                 return (0);

 776         mutex_enter(&dev->device_lock);
 777         if (dev->me_clients) {
 778                 kmem_free(dev->me_clients, dev->num_heci_me_clients*
 779                     sizeof (struct heci_me_client));
 780                 dev->me_clients = NULL;
 781         }
 782         mutex_exit(&dev->device_lock);

 784         /* allocate storage for ME clients representation */
 785         clients = kmem_zalloc(dev->num_heci_me_clients*
 786             sizeof (struct heci_me_client), KM_SLEEP);
 787         if (!clients) {
 788                 DBG("memory allocation for ME clients failed.\n");
 789                 return (-ENOMEM);
 790         }

 788         mutex_enter(&dev->device_lock);
 789         dev->me_clients = clients;
 790         mutex_exit(&dev->device_lock);

 792         num = 0;
 793         for (i = 0; i < sizeof (dev->heci_me_clients); i++) {
 794                 for (j = 0; j < 8; j++) {
 795                         if ((dev->heci_me_clients[i] & (1 << j)) != 0) {
 796                                 client = &dev->me_clients[num];
 797                                 client->client_id = (i * 8) + j;
 798                                 client->flow_ctrl_creds = 0;
 799                                 err = host_client_properties(dev, client);
 800                                 if (err != 0) {
 801                                         mutex_enter(&dev->device_lock);
 802                                         kmem_free(dev->me_clients,
 803                                             dev->num_heci_me_clients*
 804                                             sizeof (struct heci_me_client));
 805                                         dev->me_clients = NULL;
 806                                         mutex_exit(&dev->device_lock);
 807                                         return (err);
 808                                 }
 809                                 num++;
 810                         }
 811                 }
```



```
 812         }

 814         return (0);
 815 }
______unchanged_portion_omitted_

1038 /*
1039  * heci_alloc_file_private - allocates a private file structure and set it up.
1040  * @file: the file structure
1041  *
1042  * @return  The allocated file or NULL on failure
1043  */
1044 struct heci_file_private *
1045 heci_alloc_file_private(struct heci_file *file)
1046 {
1047         struct heci_file_private *priv;

1049         priv = kmem_zalloc(sizeof (struct heci_file_private), KM_SLEEP);
1054         if (!priv)
1055                 return (NULL);

1051         heci_init_file_private(priv, file);

1053         return (priv);
1054 }
______unchanged_portion_omitted_

1071 /*
1072  * heci_disconnect_host_client  - send disconnect message  to fw from host
1073  * client.
1074  *
1075  * @dev: Device object for our driver
1076  * @file_ext: private data of the file object
1077  *
1078  * @return 0 on success, <0 on failure.
1079  */
1080 int
1081 heci_disconnect_host_client(struct iamt_heci_device *dev,
1082                 struct heci_file_private *file_ext)
1083 {
1084         int rets, err;
1085         long timeout = 15;      /* 15 seconds */
1086         struct heci_cb_private *priv_cb;
1087         clock_t delta = (clock_t)(timeout * HZ);

1089         if ((!dev) || (!file_ext))
1090                 return (-ENODEV);

1092         if (file_ext->state != HECI_FILE_DISCONNECTING)
1093                 return (0);

1095         priv_cb = kmem_zalloc(sizeof (struct heci_cb_private), KM_SLEEP);
1102         if (!priv_cb)
1103                 return (-ENOMEM);

1097         LIST_INIT_HEAD(&priv_cb->cb_list);
1098         priv_cb->file_private = file_ext;
1099         priv_cb->major_file_operations = HECI_CLOSE;
1100         mutex_enter(&dev->device_lock);
1101         if (dev->host_buffer_is_empty) {
1102                 dev->host_buffer_is_empty = 0;
1103                 if (heci_disconnect(dev, file_ext)) {
1104                         list_add_tail(&priv_cb->cb_list,
1105                             &dev->ctrl_rd_list.heci_cb.cb_list);
1106                 } else {
1107                         mutex_exit(&dev->device_lock);
```

```
1108                        rets = -ENODEV;
1109                        DBG("failed to call heci_disconnect.\n");
1110                        goto free;
1111                }
1112        } else {
1113                DBG("add disconnect cb to control write list\n");
1114                list_add_tail(&priv_cb->cb_list,
1115                    &dev->ctrl_wr_list.heci_cb.cb_list);
1116        }

1118        err = 0;
1119        while (err != -1 &&
1120            (HECI_FILE_DISCONNECTED != file_ext->state)) {

1122                err = cv_reltimedwait(&dev->wait_recvd_msg, &dev->device_lock,
1123                    delta, TR_CLOCK_TICK);
1124        }
1125        mutex_exit(&dev->device_lock);

1127        if (HECI_FILE_DISCONNECTED == file_ext->state) {
1128                rets = 0;
1129                DBG("successfully disconnected from fw client."
1130                    " me_client_id:%d, host_client_id:%d\n",
1131                    file_ext->me_client_id,
1132                    file_ext->host_client_id);
1133        } else {
1134                rets = -ENODEV;
1135                if (HECI_FILE_DISCONNECTED != file_ext->state)
1136                        DBG("wrong status client disconnect.\n");

1138                if (err)
1139                        DBG("wait failed disconnect err=%08x\n", err);

1141                DBG("failed to disconnect from fw client.\n"
1142                    " me_client_id:%d, host_client_id:%d\n",
1143                    file_ext->me_client_id,
1144                    file_ext->host_client_id);
1145        }

1147        mutex_enter(&dev->device_lock);
1148        heci_flush_list(&dev->ctrl_rd_list, file_ext);
1149        heci_flush_list(&dev->ctrl_wr_list, file_ext);
1150        mutex_exit(&dev->device_lock);
1151 free:
1152        heci_free_cb_private(priv_cb);
1153        return (rets);
1154 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   25971 Thu Oct 23 11:05:01 2014
new/usr/src/uts/intel/os/fmsmb.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

 631 /*
 632  * go throught the type 2 structure contained_ids looking for
 633  * the type 4 which  has strand_apicid == this strand_apicid
 634  */
 635 static int
 636 find_matching_proc(smbios_hdl_t *shp, uint_t strand_apicid,
 637     uint16_t bb_id, uint16_t proc_hdl, int is_proc)
 638 {
 639         int n;
 640         const smb_struct_t *sp;
 641         smbios_bboard_t bb;
 642         uint_t cont_count, cont_len;
 643         uint16_t cont_id;
 644         id_t *cont_hdl = NULL;
 645         int rc;


 648         (void) smbios_info_bboard(shp, bb_id, &bb);
 649         cont_count = (uint_t)bb.smbb_contn;
 650         if (cont_count == 0)
 651                 return (0);

 653         cont_len = sizeof (id_t);
 654         cont_hdl = kmem_zalloc(cont_count * cont_len, KM_SLEEP);
 655         if (cont_hdl == NULL)
 656                 return (0);

 656         rc = smbios_info_contains(shp, bb_id, cont_count, cont_hdl);
 657         if (rc > SMB_CONT_MAX) {
 658                 kmem_free(cont_hdl, cont_count * cont_len);
 659                 return (0);
 660         }
 661         cont_count = MIN(rc, cont_count);

 663         for (n = 0; n < cont_count; n++) {
 664                 cont_id = (uint16_t)cont_hdl[n];
 665                 sp = smb_lookup_id(shp, cont_id);
 666                 if (sp->smbst_hdr->smbh_type == SMB_TYPE_PROCESSOR) {
 667                         if (is_proc) {
 668                                 if (find_matching_apic(shp, cont_id,
 669                                     strand_apicid)) {
 670                                         kmem_free(cont_hdl,
 671                                             cont_count * cont_len);
 672                                         return (1);
 673                                 }
 674                         } else {
 675                                 if (cont_id == proc_hdl) {
 676                                         kmem_free(cont_hdl,
 677                                             cont_count * cont_len);
 678                                         return (1);
 679                                 }
 680                         }
 681                 }
 682         }
 683         if (cont_hdl != NULL)
 684                 kmem_free(cont_hdl, cont_count * cont_len);

 686         return (0);
```



```
 687 }
______unchanged_portion_omitted_
```

```
**********************************************************
  182715 Thu Oct 23 11:05:01 2014
new/usr/src/uts/sun4/io/pcicfg.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
______unchanged_portion_omitted_

5783 #ifdef PCICFG_INTERPRET_FCODE
5784 static int
5785 pcicfg_load_fcode(dev_info_t *dip, uint_t bus, uint_t device, uint_t func,
5786     uint16_t vendor_id, uint16_t device_id, uchar_t **fcode_addr,
5787     int *fcode_size, int rom_paddr, int rom_size)
5788 {
5789         pci_regspec_t           p;
5790         int                     pci_data;
5791         int                     start_of_fcode;
5792         int                     image_length;
5793         int                     code_type;
5794         ddi_acc_handle_t        h;
5795         ddi_device_acc_attr_t   acc;
5796         uint8_t                 *addr;
5797         int8_t                  image_not_found, indicator;
5798         uint16_t                vendor_id_img, device_id_img;
5799         int16_t                 rom_sig;
5800 #ifdef DEBUG
5801         int i;
5802 #endif

5804         DEBUG4("pcicfg_load_fcode() - "
5805             "bus %x device =%x func=%x rom_paddr=%lx\n",
5806             bus, device, func, rom_paddr);
5807         DEBUG2("pcicfg_load_fcode() - vendor_id=%x device_id=%x\n",
5808             vendor_id, device_id);

5810         *fcode_size = 0;
5811         *fcode_addr = NULL;

5813         acc.devacc_attr_version = DDI_DEVICE_ATTR_V0;
5814         acc.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
5815         acc.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

5817         p.pci_phys_hi = PCI_ADDR_MEM32 | PCICFG_MAKE_REG_HIGH(bus, device,
5818             func, PCI_CONF_ROM);

5820         p.pci_phys_mid = 0;
5821         p.pci_phys_low = 0;

5823         p.pci_size_low = rom_size;
5824         p.pci_size_hi = 0;

5826         if (pcicfg_map_phys(dip, &p, (caddr_t *)&addr, &acc, &h)) {
5827                 DEBUG1("Can Not map in ROM %x\n", p.pci_phys_low);
5828                 return (PCICFG_FAILURE);
5829         }

5831         /*
5832          * Walk the ROM to find the proper image for this device.
5833          */
5834         image_not_found = 1;
5835         while (image_not_found) {
5836                 DEBUG1("Expansion ROM maps to %lx\n", addr);

5838 #ifdef DEBUG
5839                 if (pcicfg_dump_fcode) {
5840                         for (i = 0; i < 100; i++)
```



```
5841                                 DEBUG2("ROM 0x%x --> 0x%x\n", i,
5842                                     ddi_get8(h, (uint8_t *)(addr + i)));
5843                 }
5844 #endif

5846                 /*
5847                  * Some device say they have an Expansion ROM, but do not, so
5848                  * for non-21554 devices use peek so we don't panic due to
5849                  * accessing non existent memory.
5850                  */
5851                 if (pcicfg_indirect_map(dip) == DDI_SUCCESS) {
5852                         rom_sig = ddi_get16(h,
5853                             (uint16_t *)(addr + PCI_ROM_SIGNATURE));
5854                 } else {
5855                         if (ddi_peek16(dip,
5856                             (int16_t *)(addr + PCI_ROM_SIGNATURE), &rom_sig)) {
5857                                 cmn_err(CE_WARN,
5858                                     "PCI Expansion ROM is not accessible");
5859                                 pcicfg_unmap_phys(&h, &p);
5860                                 return (PCICFG_FAILURE);
5861                         }
5862                 }

5864                 /*
5865                  * Validate the ROM Signature.
5866                  */
5867                 if ((uint16_t)rom_sig != 0xaa55) {
5868                         DEBUG1("Invalid ROM Signature %x\n", (uint16_t)rom_sig);
5869                         pcicfg_unmap_phys(&h, &p);
5870                         return (PCICFG_FAILURE);
5871                 }

5873                 DEBUG0("Valid ROM Signature Found\n");

5875                 start_of_fcode = ddi_get16(h, (uint16_t *)(addr + 2));

5877                 pci_data =  ddi_get16(h,
5878                     (uint16_t *)(addr + PCI_ROM_PCI_DATA_STRUCT_PTR));

5880                 DEBUG2("Pointer To PCI Data Structure %x %x\n", pci_data,
5881                     addr);

5883                 /*
5884                  * Validate the PCI Data Structure Signature.
5885                  * 0x52494350 = "PCIR"
5886                  */

5888                 if (ddi_get8(h, (uint8_t *)(addr + pci_data)) != 0x50) {
5889                         DEBUG0("Invalid PCI Data Structure Signature\n");
5890                         pcicfg_unmap_phys(&h, &p);
5891                         return (PCICFG_FAILURE);
5892                 }

5894                 if (ddi_get8(h, (uint8_t *)(addr + pci_data + 1)) != 0x43) {
5895                         DEBUG0("Invalid PCI Data Structure Signature\n");
5896                         pcicfg_unmap_phys(&h, &p);
5897                         return (PCICFG_FAILURE);
5898                 }
5899                 if (ddi_get8(h, (uint8_t *)(addr + pci_data + 2)) != 0x49) {
5900                         DEBUG0("Invalid PCI Data Structure Signature\n");
5901                         pcicfg_unmap_phys(&h, &p);
5902                         return (PCICFG_FAILURE);
5903                 }
5904                 if (ddi_get8(h, (uint8_t *)(addr + pci_data + 3)) != 0x52) {
5905                         DEBUG0("Invalid PCI Data Structure Signature\n");
5906                         pcicfg_unmap_phys(&h, &p);
```

```
5907                    return (PCICFG_FAILURE);
5908            }

5910            /*
5911             * Is this image for this device?
5912             */
5913            vendor_id_img = ddi_get16(h,
5914                (uint16_t *)(addr + pci_data + PCI_PDS_VENDOR_ID));
5915            device_id_img = ddi_get16(h,
5916                (uint16_t *)(addr + pci_data + PCI_PDS_DEVICE_ID));

5918            DEBUG2("This image is for vendor_id=%x device_id=%x\n",
5919                vendor_id_img, device_id_img);

5921            code_type = ddi_get8(h, addr + pci_data + PCI_PDS_CODE_TYPE);

5923            switch (code_type) {
5924            case PCI_PDS_CODE_TYPE_PCAT:
5925                    DEBUG0("ROM is of x86/PC-AT Type\n");
5926                    break;
5927            case PCI_PDS_CODE_TYPE_OPEN_FW:
5928                    DEBUG0("ROM is of Open Firmware Type\n");
5929                    break;
5930            default:
5931                    DEBUG1("ROM is of Unknown Type 0x%x\n", code_type);
5932                    break;
5933            }

5935            if ((vendor_id_img != vendor_id) ||
5936                (device_id_img != device_id) ||
5937                (code_type != PCI_PDS_CODE_TYPE_OPEN_FW)) {
5938                    DEBUG0("Firmware Image is not for this device..."
5939                        "goto next image\n");
5940                    /*
5941                     * Read indicator byte to see if there is another
5942                     * image in the ROM
5943                     */
5944                    indicator = ddi_get8(h,
5945                        (uint8_t *)(addr + pci_data + PCI_PDS_INDICATOR));

5947                    if (indicator != 1) {
5948                            /*
5949                             * There is another image in the ROM.
5950                             */
5951                            image_length = ddi_get16(h,  (uint16_t *)(addr +
5952                                pci_data + PCI_PDS_IMAGE_LENGTH)) * 512;

5954                            addr += image_length;
5955                    } else {
5956                            /*
5957                             * There are no more images.
5958                             */
5959                            DEBUG0("There are no more images in the ROM\n");
5960                            pcicfg_unmap_phys(&h, &p);

5962                            return (PCICFG_FAILURE);
5963                    }
5964            } else {
5965                    DEBUG0("Correct image was found\n");
5966                    image_not_found = 0;  /* Image was found */
5967            }
5968    }

5970    *fcode_size =  (ddi_get8(h, addr + start_of_fcode + 4) << 24) |
5971        (ddi_get8(h, addr + start_of_fcode + 5) << 16) |
5972        (ddi_get8(h, addr + start_of_fcode + 6) << 8) |
```



```
5973            (ddi_get8(h, addr + start_of_fcode + 7));

5975    DEBUG1("Fcode Size %x\n", *fcode_size);

5977    /*
5978     * Allocate page aligned buffer space
5979     */
5980    *fcode_addr = kmem_zalloc(ptob(btopr(*fcode_size)), KM_SLEEP);

5982    if (*fcode_addr == NULL) {
5983            DEBUG0("kmem_zalloc returned NULL\n");
5984            pcicfg_unmap_phys(&h, &p);
5985            return (PCICFG_FAILURE);
5986    }

5982    DEBUG1("Fcode Addr %lx\n", *fcode_addr);

5984    ddi_rep_get8(h, *fcode_addr, addr + start_of_fcode, *fcode_size,
5985        DDI_DEV_AUTOINCR);

5987    pcicfg_unmap_phys(&h, &p);

5989    return (PCICFG_SUCCESS);
5990 }
```

unchanged_portion_omitted_

```
**********************************************************
    9593 Thu Oct 23 11:05:02 2014
new/usr/src/uts/sun4/io/px/px_mmu.c
5253 kmem_alloc/kmem_zalloc won't fail with KM_SLEEP
5254 getrbuf won't fail with KM_SLEEP
**********************************************************
/*
 * 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 2012 Garrett D'Amore <garrett@damore.org>.  All rights reserved.
 */

/*
 * PX mmu initialization and configuration
 */
#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/async.h>
#include <sys/sysmacros.h>
#include <sys/sunddi.h>
#include <sys/ddi_impldefs.h>
#include <sys/vmem.h>
#include <sys/machsystm.h>      /* lddphys() */
#include <sys/iommutsb.h>
#include "px_obj.h"

int
px_mmu_attach(px_t *px_p)
{
        dev_info_t              *dip = px_p->px_dip;
        px_mmu_t                        *mmu_p;
        uint32_t                tsb_i = 0;
        char                    map_name[32];
        px_dvma_range_prop_t    *dvma_prop;
        int                     dvma_prop_len;
        uint32_t                cache_size, tsb_entries;

        /*
         * Allocate mmu state structure and link it to the
         * px state structure.
         */
        mmu_p = kmem_zalloc(sizeof (px_mmu_t), KM_SLEEP);
        if (mmu_p == NULL)
                return (DDI_FAILURE);

        px_p->px_mmu_p = mmu_p;
        mmu_p->mmu_px_p = px_p;
```



```
        mmu_p->mmu_inst = ddi_get_instance(dip);

        /*
         * Check for "virtual-dma" property that specifies
         * the DVMA range.
         */
        if (ddi_getlongprop(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
            "virtual-dma", (caddr_t)&dvma_prop, &dvma_prop_len) !=
            DDI_PROP_SUCCESS) {

                DBG(DBG_ATTACH, dip, "Getting virtual-dma failed\n");

                kmem_free(mmu_p, sizeof (px_mmu_t));
                px_p->px_mmu_p = NULL;

                return (DDI_FAILURE);
        }

        mmu_p->mmu_dvma_base = dvma_prop->dvma_base;
        mmu_p->mmu_dvma_end = dvma_prop->dvma_base +
            (dvma_prop->dvma_len - 1);
        tsb_entries = MMU_BTOP(dvma_prop->dvma_len);

        kmem_free(dvma_prop, dvma_prop_len);

        /*
         * Setup base and bounds for DVMA and bypass mappings.
         */
        mmu_p->mmu_dvma_cache_locks =
            kmem_zalloc(px_dvma_page_cache_entries, KM_SLEEP);

        mmu_p->dvma_base_pg = MMU_BTOP(mmu_p->mmu_dvma_base);
        mmu_p->mmu_dvma_reserve = tsb_entries >> 1;
        mmu_p->dvma_end_pg = MMU_BTOP(mmu_p->mmu_dvma_end);

        /*
         * Create a virtual memory map for dvma address space.
         * Reserve 'size' bytes of low dvma space for fast track cache.
         */
        (void) snprintf(map_name, sizeof (map_name), "%s%d_dvma",
            ddi_driver_name(dip), ddi_get_instance(dip));

        cache_size = MMU_PTOB(px_dvma_page_cache_entries *
            px_dvma_page_cache_clustsz);
        mmu_p->mmu_dvma_fast_end = mmu_p->mmu_dvma_base +
            cache_size - 1;

        mmu_p->mmu_dvma_map = vmem_create(map_name,
            (void *)(mmu_p->mmu_dvma_fast_end + 1),
            MMU_PTOB(tsb_entries) - cache_size, MMU_PAGE_SIZE,
            NULL, NULL, NULL, MMU_PAGE_SIZE, VM_SLEEP);

        mutex_init(&mmu_p->dvma_debug_lock, NULL, MUTEX_DRIVER, NULL);

        for (tsb_i = 0; tsb_i < tsb_entries; tsb_i++) {
                r_addr_t ra = 0;
                io_attributes_t attr;
                caddr_t va;

                if (px_lib_iommu_getmap(px_p->px_dip, PCI_TSBID(0, tsb_i),
                    &attr, &ra) != DDI_SUCCESS)
                        continue;

                va = (caddr_t)(MMU_PTOB(mmu_p->dvma_base_pg + tsb_i));

                if (va <= (caddr_t)mmu_p->mmu_dvma_fast_end) {
```

```
125                         uint32_t cache_i;

127                         /*
128                          * the va is within the *fast* dvma range; therefore,
129                          * lock its fast dvma page cache cluster in order to
130                          * both preserve the TTE and prevent the use of this
131                          * fast dvma page cache cluster by px_dvma_map_fast().
132                          * the lock value 0xFF comes from ldstub().
133                          */
134                         cache_i = tsb_i / px_dvma_page_cache_clustsz;
135                         ASSERT(cache_i < px_dvma_page_cache_entries);
136                         mmu_p->mmu_dvma_cache_locks[cache_i] = 0xFF;
137                 } else {
138                         (void) vmem_xalloc(mmu_p->mmu_dvma_map, MMU_PAGE_SIZE,
139                             MMU_PAGE_SIZE, 0, 0, va, va + MMU_PAGE_SIZE,
140                             VM_NOSLEEP | VM_BESTFIT | VM_PANIC);
141                 }
142         }

144         return (DDI_SUCCESS);
145 }
```

*unchanged_portion_omitted_*