```
**********************************************************
    1881 Mon May  5 14:29:41 2014
new/usr/src/man/man9f/ddi_get_time.9f
4776 man: don't lie about ddi_get_time(9f) uses
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
'\" te
.\" Copyright (c) 2000, Sun Microsystems, Inc.
.\" Copyright (c) 2014, Nexenta Systems, Inc.
#endif /* ! codereview */
.\"  All Rights Reserved
.\" The contents of this file are subject to the terms of the Common Development
.\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
.\" When distributing Covered Code, include this CDDL HEADER in each file and in
.TH DDI_GET_TIME 9F "Apr 15, 2014"
.TH DDI_GET_TIME 9F "Feb 18, 1998"
.SH NAME
ddi_get_time \- returns the current time in seconds
.SH SYNOPSIS
.LP
.nf
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>



\fBtime_t\fR \fBddi_get_time\fR(\fBvoid\fR);
.fi

.SH INTERFACE LEVEL
.sp
.LP
Solaris DDI specific (Solaris DDI).
.SH DESCRIPTION
.sp
.LP
\fBddi_get_time()\fR returns the current time in seconds since 00:00:00 UTC,
January 1, 1970. Changes in time of day clock may result in this value
changing. In other words, the value is not monotonically increasing and
therefore it must not be used to set wait or expiration intervals. For that,
instead use \fBddi_get_lbolt\fR(9F) or \fBgethrtime\fR(9F).
January 1, 1970. This value can be used to set of wait or expiration intervals.
.SH RETURN VALUES
.sp
.LP
\fBddi_get_time()\fR returns the time in seconds.
.SH CONTEXT
.sp
.LP
 This routine can be called from any context.
.SH SEE ALSO
.sp
.LP
\fBddi_get_lbolt\fR(9F), \fBdrv_getparm\fR(9F), \fBdrv_usectohz\fR(9F),
\fBgethrtime\fR(9F)
\fIddi_get_lbolt\fR(9F), \fIdrv_getparm\fR(9F), \fIdrv_usectohz\fR(9F)
.sp
.LP
\fIWriting Device Drivers\fR
.sp
.LP
 \fISTREAMS Programming Guide\fR
```

```
**********************************************************
   79084 Mon May  5 14:29:41 2014
new/usr/src/uts/common/inet/ipf/ip_fil_solaris.c
4787 ipf: remove rate_limit_message
**********************************************************
______unchanged_portion_omitted_

#include <sys/time.h>
#include <sys/varargs.h>

#ifndef _KERNEL
#include <stdio.h>
#endif

#define NULLADDR_RATE_LIMIT 10  /* 10 seconds */


/*
 * Print out warning message at rate-limited speed.
 */
static void rate_limit_message(ipf_stack_t *ifs,
                               int rate, const char *message, ...)
{
        static time_t last_time = 0;
        time_t now;
        va_list args;
        char msg_buf[256];
        int  need_printed = 0;

        now = ddi_get_time();

        /* make sure, no multiple entries */
        ASSERT(MUTEX_NOT_HELD(&(ifs->ifs_ipf_rw.ipf_lk)));
        MUTEX_ENTER(&ifs->ifs_ipf_rw);
        if (now - last_time >= rate) {
                need_printed = 1;
                last_time = now;
        }
        MUTEX_EXIT(&ifs->ifs_ipf_rw);

        if (need_printed) {
                va_start(args, message);
                (void)vsnprintf(msg_buf, 255, message, args);
                va_end(args);
#ifdef _KERNEL
                cmn_err(CE_WARN, msg_buf);
#else
                fprintf(std_err, msg_buf);
#endif
        }
}

/*
 * Return the first IP Address associated with an interface
 * For IPv6, we walk through the list of logical interfaces and return
 * the address of the first one that isn't a link-local interface.
 * We can't assume that it is :1 because another link-local address
 * may have been assigned there.
 */
/*ARGSUSED*/
int fr_ifpaddr(v, atype, ifptr, inp, inpmask, ifs)
int v, atype;
void *ifptr;
struct in_addr  *inp, *inpmask;
ipf_stack_t *ifs;
{
```



```
        struct sockaddr_in6 v6addr[2];
        struct sockaddr_in v4addr[2];
        net_ifaddr_t type[2];
        net_handle_t net_data;
        phy_if_t phyif;
        void *array;

        switch (v)
        {
        case 4:
                net_data = ifs->ifs_ipf_ipv4;
                array = v4addr;
                break;
        case 6:
                net_data = ifs->ifs_ipf_ipv6;
                array = v6addr;
                break;
        default:
                net_data = NULL;
                break;
        }

        if (net_data == NULL)
                return -1;

        phyif = (phy_if_t)ifptr;

        switch (atype)
        {
        case FRI_PEERADDR :
                type[0] = NA_PEER;
                break;

        case FRI_BROADCAST :
                type[0] = NA_BROADCAST;
                break;

        default :
                type[0] = NA_ADDRESS;
                break;
        }

        type[1] = NA_NETMASK;

        if (v == 6) {
                lif_if_t idx = 0;

                do {
                        idx = net_lifgetnext(net_data, phyif, idx);
                        if (net_getlifaddr(net_data, phyif, idx, 2, type,
                                           array) < 0)
                                return -1;
                        if (!IN6_IS_ADDR_LINKLOCAL(&v6addr[0].sin6_addr) &&
                            !IN6_IS_ADDR_MULTICAST(&v6addr[0].sin6_addr))
                                break;
                } while (idx != 0);

                if (idx == 0)
                        return -1;

                return fr_ifpfillv6addr(atype, &v6addr[0], &v6addr[1],
                                        inp, inpmask);
        }

        if (net_getlifaddr(net_data, phyif, 0, 2, type, array) < 0)
                return -1;
```

```
1376         return fr_ifpfillv4addr(atype, &v4addr[0], &v4addr[1], inp, inpmask);
1377 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   94820 Mon May  5 14:29:41 2014
new/usr/src/uts/common/io/comstar/port/iscsit/iscsit.c
4780 comstar iSCSI target shouldn't abuse ddi_get_time(9f)
Reviewed by: Eric Diven <eric.diven@delphix.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 * Copyright 2014 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 */

#include <sys/cpuvar.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/sysmacros.h>
#include <sys/socket.h>
#include <sys/strsubr.h>
#include <sys/nvpair.h>

#include <sys/stmf.h>
#include <sys/stmf_ioctl.h>
#include <sys/portif.h>
#include <sys/idm/idm.h>
#include <sys/idm/idm_conn_sm.h>

#include "iscsit_isns.h"
#include "iscsit.h"

#define ISCSIT_VERSION          BUILD_DATE "-1.18dev"
#define ISCSIT_NAME_VERSION     "COMSTAR ISCSIT v" ISCSIT_VERSION

/*
 * DDI entry points.
 */
static int iscsit_drv_attach(dev_info_t *, ddi_attach_cmd_t);
static int iscsit_drv_detach(dev_info_t *, ddi_detach_cmd_t);
static int iscsit_drv_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int iscsit_drv_open(dev_t *, int, int, cred_t *);
```



```
static int iscsit_drv_close(dev_t, int, int, cred_t *);
static boolean_t iscsit_drv_busy(void);
static int iscsit_drv_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

extern struct mod_ops mod_miscops;


static struct cb_ops iscsit_cb_ops = {
        iscsit_drv_open,        /* cb_open */
        iscsit_drv_close,       /* cb_close */
        nodev,                  /* cb_strategy */
        nodev,                  /* cb_print */
        nodev,                  /* cb_dump */
        nodev,                  /* cb_read */
        nodev,                  /* cb_write */
        iscsit_drv_ioctl,       /* cb_ioctl */
        nodev,                  /* cb_devmap */
        nodev,                  /* cb_mmap */
        nodev,                  /* cb_segmap */
        nochpoll,               /* cb_chpoll */
        ddi_prop_op,            /* cb_prop_op */
        NULL,                   /* cb_streamtab */
        D_MP,                   /* cb_flag */
        CB_REV,                 /* cb_rev */
        nodev,                  /* cb_aread */
        nodev,                  /* cb_awrite */
};
```

*unchanged_portion_omitted_*

```
/*
 * iscsit_check_cmdsn_and_queue
 *
 * Independent of the order in which the iSCSI target receives non-immediate
 * command PDU across the entire session and any multiple connections within
 * the session, the target must deliver the commands to the SCSI layer in
 * CmdSN order. So out-of-order non-immediate commands are queued up on a
 * session-wide wait queue. Duplicate commands are ignored.
 *
 */
static int
iscsit_check_cmdsn_and_queue(idm_pdu_t *rx_pdu)
{
        idm_conn_t              *ic = rx_pdu->isp_ic;
        iscsit_conn_t           *ict = ic->ic_handle;
        iscsit_sess_t           *ist = ict->ict_sess;
        iscsi_scsi_cmd_hdr_t    *hdr = (iscsi_scsi_cmd_hdr_t *)rx_pdu->isp_hdr;

        mutex_enter(&ist->ist_sn_mutex);
        if (hdr->opcode & ISCSI_OP_IMMEDIATE) {
                /* do not queue, handle it immediately */
                DTRACE_PROBE2(immediate__cmd, iscsit_sess_t *, ist,
                    idm_pdu_t *, rx_pdu);
                mutex_exit(&ist->ist_sn_mutex);
                return (ISCSIT_CMDSN_EQ_EXPCMDSN);
        }
        if (iscsit_sna_lt(ist->ist_expcmdsn, ntohl(hdr->cmdsn))) {
                /*
                 * Out-of-order commands (cmdSN higher than ExpCmdSN)
                 * are staged on a fixed-size circular buffer until
                 * the missing command is delivered to the SCSI layer.
                 * Irrespective of the order of insertion into the
                 * staging queue, the commands are processed out of the
                 * queue in cmdSN order only.
                 */
                rx_pdu->isp_queue_time = gethrtime();
                rx_pdu->isp_queue_time = ddi_get_time();
```

```
3129                 iscsit_add_pdu_to_queue(ist, rx_pdu);
3130                 mutex_exit(&ist->ist_sn_mutex);
3131                 return (ISCSIT_CMDSN_GT_EXPCMDSN);
3132         } else if (iscsit_sna_lt(ntohl(hdr->cmdsn), ist->ist_expcmdsn)) {
3133                 DTRACE_PROBE3(cmdsn__lt__expcmdsn, iscsit_sess_t *, ist,
3134                     iscsit_conn_t *, ict, idm_pdu_t *, rx_pdu);
3135                 mutex_exit(&ist->ist_sn_mutex);
3136                 return (ISCSIT_CMDSN_LT_EXPCMDSN);
3137         } else {
3138                 mutex_exit(&ist->ist_sn_mutex);
3139                 return (ISCSIT_CMDSN_EQ_EXPCMDSN);
3140         }
3141 }
______unchanged_portion_omitted_

3356 static void
3357 iscsit_rxpdu_queue_monitor_session(iscsit_sess_t *ist)
3358 {
3359         iscsit_cbuf_t   *cbuf   = ist->ist_rxpdu_queue;
3360         idm_pdu_t       *next_pdu = NULL;
3361         uint32_t        index, next_cmdsn, i;

3363         /*
3364          * Assume that all PDUs in the staging queue have a cmdsn >= expcmdsn.
3365          * Starting with the expcmdsn, iterate over the staged PDUs to find
3366          * the next PDU with a wait time greater than the threshold. If found
3367          * advance the staged PDU to the SCSI layer, skipping over the missing
3368          * PDU(s) to get past the hole in the command sequence. It is up to
3369          * the initiator to note that the target has not acknowledged a cmdsn
3370          * and take appropriate action.
3371          *
3372          * Since the PDU(s) arrive in any random order, it is possible that
3373          * that the actual wait time for a particular PDU is much longer than
3374          * the defined threshold. e.g. Consider a case where commands are sent
3375          * over 4 different connections, and cmdsn = 1004 arrives first, then
3376          * 1003, and 1002 and 1001 are lost due to a connection failure.
3377          * So now 1003 is waiting for 1002 to be delivered, and although the
3378          * wait time of 1004 > wait time of 1003, only 1003 will be considered
3379          * by the monitor thread. 1004 will be automatically processed by
3380          * iscsit_process_pdu_in_queue() once the scan is complete and the
3381          * expcmdsn becomes current.
3382          */
3383         mutex_enter(&ist->ist_sn_mutex);
3384         cbuf = ist->ist_rxpdu_queue;
3385         if (cbuf->cb_num_elems == 0) {
3386                 mutex_exit(&ist->ist_sn_mutex);
3387                 return;
3388         }
3389         for (next_pdu = NULL, i = 0; ; i++) {
3390                 next_cmdsn = ist->ist_expcmdsn + i; /* start at expcmdsn */
3391                 index = next_cmdsn % ISCSIT_RXPDU_QUEUE_LEN;
3392                 if ((next_pdu = cbuf->cb_buffer[index]) != NULL) {
3393                         /*
3394                          * If the PDU wait time has not exceeded threshold
3395                          * stop scanning the staging queue until the timer
3396                          * fires again
3397                          */
3398                         if ((gethrtime() - next_pdu->isp_queue_time)
3399                             < (rxpdu_queue_threshold * NANOSEC)) {
3398                         if ((ddi_get_time() - next_pdu->isp_queue_time)
3399                             < rxpdu_queue_threshold) {
3400                                 mutex_exit(&ist->ist_sn_mutex);
3401                                 return;
3402                         }
3403                         /*
3404                          * Remove the next PDU from the queue and post it
```



```
3405                          * to the SCSI layer, skipping over the missing
3406                          * PDU. Stop scanning the staging queue until
3407                          * the monitor timer fires again
3408                          */
3409                         (void) iscsit_remove_pdu_from_queue(ist, next_cmdsn);
3410                         mutex_exit(&ist->ist_sn_mutex);
3411                         DTRACE_PROBE3(advanced__to__blocked__cmdsn,
3412                             iscsit_sess_t *, ist, idm_pdu_t *, next_pdu,
3413                             uint32_t, next_cmdsn);
3414                         iscsit_post_staged_pdu(next_pdu);
3415                         /* Deliver any subsequent PDUs immediately */
3416                         iscsit_process_pdu_in_queue(ist);
3417                         return;
3418                 }
3419                 /*
3420                  * Skipping over i PDUs, e.g. a case where commands 1001 and
3421                  * 1002 are lost in the network, skip over both and post 1003
3422                  * expcmdsn then becomes 1004 at the end of the scan.
3423                  */
3424                 DTRACE_PROBE2(skipping__over__cmdsn, iscsit_sess_t *, ist,
3425                     uint32_t, next_cmdsn);
3426         }
3427         /*
3428          * following the assumption, staged cmdsn >= expcmdsn, this statement
3429          * is never reached.
3430          */
3431 }
______unchanged_portion_omitted_
```

```
**********************************************************
  267081 Mon May  5 14:29:41 2014
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_dhchap.c
4786 emlxs shouldn't abuse ddi_get_time(9f)
**********************************************************
```

```
/*
 * 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 Emulex.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */


#include <emlxs.h>

#ifdef DHCHAP_SUPPORT

#include <md5.h>
#include <sha1.h>
#ifdef S10
#include <sha1_consts.h>
#else
#include <sys/sha1_consts.h>
#endif /* S10 */
#include <bignum.h>
#include <sys/time.h>

#ifdef  S10
#define BIGNUM_CHUNK_32
#define BIG_CHUNK_TYPE                  uint32_t
#define CHARLEN2BIGNUMLEN(_val)         (_val/4)
#endif  /* S10 */

#define RAND

#ifndef ENABLE
#define ENABLE   1
#endif  /* ENABLE */

#ifndef DISABLE
#define DISABLE   0
#endif  /* DISABLE */
```



```
/* Required for EMLXS_CONTEXT in EMLXS_MSGF calls */
EMLXS_MSG_DEF(EMLXS_DHCHAP_C);

static char *emlxs_dhc_pstate_xlate(uint32_t state);
static char *emlxs_dhc_nstate_xlate(uint32_t state);
static uint32_t emlxs_check_dhgp(emlxs_port_t *port, NODELIST *ndlp,
        uint32_t *dh_id, uint16_t cnt, uint32_t *dhgp_id);
static void emlxs_dhc_set_reauth_time(emlxs_port_t *port,
        emlxs_node_t *ndlp, uint32_t status);

static void emlxs_auth_cfg_init(emlxs_hba_t *hba);
static void emlxs_auth_cfg_fini(emlxs_hba_t *hba);
static void emlxs_auth_cfg_read(emlxs_hba_t *hba);
static uint32_t emlxs_auth_cfg_parse(emlxs_hba_t *hba,
        emlxs_auth_cfg_t *config, char *prop_str);
static emlxs_auth_cfg_t *emlxs_auth_cfg_get(emlxs_hba_t *hba,
        uint8_t *lwwpn, uint8_t *rwwpn);
static emlxs_auth_cfg_t *emlxs_auth_cfg_create(emlxs_hba_t *hba,
        uint8_t *lwwpn, uint8_t *rwwpn);
static void emlxs_auth_cfg_destroy(emlxs_hba_t *hba,
        emlxs_auth_cfg_t *auth_cfg);
static void emlxs_auth_cfg_print(emlxs_hba_t *hba,
        emlxs_auth_cfg_t *auth_cfg);

static void emlxs_auth_key_init(emlxs_hba_t *hba);
static void emlxs_auth_key_fini(emlxs_hba_t *hba);
static void emlxs_auth_key_read(emlxs_hba_t *hba);
static uint32_t emlxs_auth_key_parse(emlxs_hba_t *hba,
        emlxs_auth_key_t *auth_key, char *prop_str);
static emlxs_auth_key_t *emlxs_auth_key_get(emlxs_hba_t *hba,
        uint8_t *lwwpn, uint8_t *rwwpn);
static emlxs_auth_key_t *emlxs_auth_key_create(emlxs_hba_t *hba,
        uint8_t *lwwpn, uint8_t *rwwpn);
static void emlxs_auth_key_destroy(emlxs_hba_t *hba,
        emlxs_auth_key_t *auth_key);
static void emlxs_auth_key_print(emlxs_hba_t *hba,
        emlxs_auth_key_t *auth_key);

static void emlxs_get_random_bytes(NODELIST *ndlp, uint8_t *rdn,
        uint32_t len);
static emlxs_auth_cfg_t *emlxs_auth_cfg_find(emlxs_port_t *port,
        uint8_t *rwwpn);
static emlxs_auth_key_t *emlxs_auth_key_find(emlxs_port_t *port,
        uint8_t *rwwpn);
static void emlxs_dhc_auth_complete(emlxs_port_t *port,
        emlxs_node_t *ndlp, uint32_t status);
static void emlxs_log_auth_event(emlxs_port_t *port, NODELIST *ndlp,
        char *subclass, char *info);
static int emlxs_issue_auth_negotiate(emlxs_port_t *port,
        emlxs_node_t *ndlp, uint8_t retry);
static void emlxs_cmpl_auth_negotiate_issue(fc_packet_t *pkt);
static uint32_t *emlxs_hash_rsp(emlxs_port_t *port,
        emlxs_port_dhc_t *port_dhc, NODELIST *ndlp, uint32_t tran_id,
        union challenge_val un_cval, uint8_t *dhval, uint32_t dhvallen);
static fc_packet_t *emlxs_prep_els_fc_pkt(emlxs_port_t *port,
        uint32_t d_id, uint32_t cmd_size, uint32_t rsp_size,
        uint32_t datalen, int32_t sleepflag);

static uint32_t *emlxs_hash_vrf(emlxs_port_t *port,
        emlxs_port_dhc_t *port_dhc, NODELIST *ndlp, uint32_t tran_id,
        union challenge_val un_cval);


static BIG_ERR_CODE
emlxs_interm_hash(emlxs_port_t *port, emlxs_port_dhc_t *port_dhc,
```

```
128         NODELIST *ndlp, void *hash_val, uint32_t tran_id,
129         union challenge_val un_cval, uint8_t *dhval, uint32_t *);

131 static BIG_ERR_CODE
132 emlxs_BIGNUM_get_pubkey(emlxs_port_t *port, emlxs_port_dhc_t *port_dhc,
133         NODELIST *ndlp, uint8_t *dhval, uint32_t *dhvallen,
134         uint32_t hash_size, uint32_t dhgp_id);
135 static BIG_ERR_CODE
136 emlxs_BIGNUM_get_dhval(emlxs_port_t *port, emlxs_port_dhc_t *port_dhc,
137         NODELIST *ndlp, uint8_t *dhval, uint32_t *dhval_len,
138         uint32_t dhgp_id, uint8_t *priv_key, uint32_t privkey_len);
139 static uint32_t *
140 emlxs_hash_verification(emlxs_port_t *port, emlxs_port_dhc_t *port_dhc,
141         NODELIST *ndlp, uint32_t tran_id, uint8_t *dhval,
142         uint32_t dhval_len, uint32_t flag, uint8_t *bi_cval);

144 static uint32_t *
145 emlxs_hash_get_R2(emlxs_port_t *port, emlxs_port_dhc_t *port_dhc,
146         NODELIST *ndlp, uint32_t tran_id, uint8_t *dhval,
147         uint32_t dhval_len, uint32_t flag, uint8_t *bi_cval);

149 static uint32_t emlxs_issue_auth_reject(emlxs_port_t *port,
150         NODELIST *ndlp, int retry, uint32_t *arg, uint8_t ReasonCode,
151         uint8_t ReasonCodeExplanation);

153 static uint32_t emlxs_disc_neverdev(emlxs_port_t *port, void *arg1,
154         void *arg2, void *arg3, void *arg4, uint32_t evt);
155 static uint32_t emlxs_rcv_auth_msg_unmapped_node(emlxs_port_t *port,
156         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
157 static uint32_t emlxs_rcv_auth_msg_npr_node(emlxs_port_t *port,
158         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
159 static uint32_t emlxs_cmpl_auth_msg_npr_node(emlxs_port_t *port,
160         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
161 static uint32_t emlxs_rcv_auth_msg_auth_negotiate_issue(emlxs_port_t *port,
162         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
163 static uint32_t emlxs_cmpl_auth_msg_auth_negotiate_issue(emlxs_port_t *port,
164         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
165 static uint32_t emlxs_rcv_auth_msg_auth_negotiate_rcv(emlxs_port_t *port,
166         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
167 static uint32_t emlxs_cmpl_auth_msg_auth_negotiate_rcv(emlxs_port_t *port,
168         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
169 static uint32_t
170 emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next(emlxs_port_t *port,
171         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
172 static uint32_t
173 emlxs_cmpl_auth_msg_auth_negotiate_cmpl_wait4next(emlxs_port_t *port,
174         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
175 static uint32_t
176 emlxs_rcv_auth_msg_dhchap_challenge_issue(emlxs_port_t *port, void *arg1,
177         void *arg2, void *arg3, void *arg4, uint32_t evt);
178 static uint32_t
179 emlxs_cmpl_auth_msg_dhchap_challenge_issue(emlxs_port_t *port, void *arg1,
180         void *arg2, void *arg3, void *arg4, uint32_t evt);
181 static uint32_t emlxs_rcv_auth_msg_dhchap_reply_issue(emlxs_port_t *port,
182         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
183 static uint32_t emlxs_cmpl_auth_msg_dhchap_reply_issue(emlxs_port_t *port,
184         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
185 static uint32_t
186 emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next(emlxs_port_t *port,
187         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
188 static uint32_t
189 emlxs_cmpl_auth_msg_dhchap_challenge_cmpl_wait4next(emlxs_port_t *port,
190         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
191 static uint32_t
192 emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next(emlxs_port_t *port,
193         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
```



```
194 static uint32_t
195 emlxs_cmpl_auth_msg_dhchap_reply_cmpl_wait4next(emlxs_port_t *port,
196         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
197 static uint32_t emlxs_rcv_auth_msg_dhchap_success_issue(emlxs_port_t *port,
198         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
199 static uint32_t
200 emlxs_cmpl_auth_msg_dhchap_success_issue(emlxs_port_t *port, void *arg1,
201         void *arg2, void *arg3, void *arg4, uint32_t evt);
202 static uint32_t
203 emlxs_rcv_auth_msg_dhchap_success_issue_wait4next(emlxs_port_t *port,
204         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
205 static uint32_t
206 emlxs_cmpl_auth_msg_dhchap_success_issue_wait4next(emlxs_port_t *port,
207         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
208 static uint32_t
209 emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next(emlxs_port_t *port,
210         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
211 static uint32_t
212 emlxs_cmpl_auth_msg_dhchap_success_cmpl_wait4next(emlxs_port_t *port,
213         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);


216 static uint32_t emlxs_device_recov_unmapped_node(emlxs_port_t *port,
217         void *arg1, void *arg2, void *arg3, void *arg4, uint32_t evt);
218 static uint32_t emlxs_device_rm_npr_node(emlxs_port_t *port, void *arg1,
219         void *arg2, void *arg3, void *arg4, uint32_t evt);
220 static uint32_t emlxs_device_recov_npr_node(emlxs_port_t *port, void *arg1,
221         void *arg2, void *arg3, void *arg4, uint32_t evt);
222 static uint32_t emlxs_device_rem_auth(emlxs_port_t *port, void *arg1,
223         void *arg2, void *arg3, void *arg4, uint32_t evt);
224 static uint32_t emlxs_device_recov_auth(emlxs_port_t *port, void *arg1,
225         void *arg2, void *arg3, void *arg4, uint32_t evt);

227 static uint8_t emlxs_null_wwn[8] =
228         {0, 0, 0, 0, 0, 0, 0, 0};
229 static uint8_t emlxs_fabric_wwn[8] =
230         {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

232 unsigned char dhgp1_pVal[] =
233 {0xEE, 0xAF, 0x0A, 0xB9, 0xAD, 0xB3, 0x8D, 0xD6, 0x9C, 0x33, 0xF8, 0x0A, 0xFA,
234 0x8F, 0xC5, 0xE8,
235 0x60, 0x72, 0x61, 0x87, 0x75, 0xFF, 0x3C, 0x0B, 0x9E, 0xA2, 0x31, 0x4C, 0x9C,
236 0x25, 0x65, 0x76,
237 0xD6, 0x74, 0xDF, 0x74, 0x96, 0xEA, 0x81, 0xD3, 0x38, 0x3B, 0x48, 0x13, 0xD6,
238 0x92, 0xC6, 0xE0,
239 0xE0, 0xD5, 0xD8, 0xE2, 0x50, 0xB9, 0x8B, 0xE4, 0x8E, 0x49, 0x5C, 0x1D, 0x60,
240 0x89, 0xDA, 0xD1,
241 0x5D, 0xC7, 0xD7, 0xB4, 0x61, 0x54, 0xD6, 0xB6, 0xCE, 0x8E, 0xF4, 0xAD, 0x69,
242 0xB1, 0x5D, 0x49,
243 0x82, 0x55, 0x9B, 0x29, 0x7B, 0xCF, 0x18, 0x85, 0xC5, 0x29, 0xF5, 0x66, 0x66,
244 0x0E, 0x57, 0xEC,
245 0x68, 0xED, 0xBC, 0x3C, 0x05, 0x72, 0x6C, 0xC0, 0x2F, 0xD4, 0xCB, 0xF4, 0x97,
246 0x6E, 0xAA, 0x9A,
247 0xFD, 0x51, 0x38, 0xFE, 0x83, 0x76, 0x43, 0x5B, 0x9F, 0xC6, 0x1D, 0x2F, 0xC0,
248 0xEB, 0x06, 0xE3,
249 };

251 unsigned char dhgp2_pVal[] =
252 {0xD7, 0x79, 0x46, 0x82, 0x6E, 0x81, 0x19, 0x14, 0xB3, 0x94, 0x01, 0xD5, 0x6A,
253 0x0A, 0x78, 0x43,
254 0xA8, 0xE7, 0x57, 0x5D, 0x73, 0x8C, 0x67, 0x2A, 0x09, 0x0A, 0xB1, 0x18, 0x7D,
255 0x69, 0x0D, 0xC4,
256 0x38, 0x72, 0xFC, 0x06, 0xA7, 0xB6, 0xA4, 0x3F, 0x3B, 0x95, 0xBE, 0xAE, 0xC7,
257 0xDF, 0x04, 0xB9,
258 0xD2, 0x42, 0xEB, 0xDC, 0x48, 0x11, 0x11, 0x28, 0x32, 0x16, 0xCE, 0x81, 0x6E,
259 0x00, 0x4B, 0x78,
```

```
0x6C, 0x5F, 0xCE, 0x85, 0x67, 0x80, 0xD4, 0x18, 0x37, 0xD9, 0x5A, 0xD7, 0x87,
0xA5, 0x0B, 0xBE,
0x90, 0xBD, 0x3A, 0x9C, 0x98, 0xAC, 0x0F, 0x5F, 0xC0, 0xDE, 0x74, 0x4B, 0x1C,
0xDE, 0x18, 0x91,
0x69, 0x08, 0x94, 0xBC, 0x1F, 0x65, 0xE0, 0x0D, 0xE1, 0x5B, 0x4B, 0x2A, 0xA6,
0xD8, 0x71, 0x00,
0xC9, 0xEC, 0xC2, 0x52, 0x7E, 0x45, 0xEB, 0x84, 0x9D, 0xEB, 0x14, 0xBB, 0x20,
0x49, 0xB1, 0x63,
0xEA, 0x04, 0x18, 0x7F, 0xD2, 0x7C, 0x1B, 0xD9, 0xC7, 0x95, 0x8C, 0xD4, 0x0C,
0xE7, 0x06, 0x7A,
0x9C, 0x02, 0x4F, 0x9B, 0x7C, 0x5A, 0x0B, 0x4F, 0x50, 0x03, 0x68, 0x61, 0x61,
0xF0, 0x60, 0x5B
};

unsigned char dhgp3_pVal[] =
{0x9D, 0xEF, 0x3C, 0xAF, 0xB9, 0x39, 0x27, 0x7A, 0xB1, 0xF1, 0x2A, 0x86, 0x17,
0xA4, 0x7B, 0xBB,
0xDB, 0xA5, 0x1D, 0xF4, 0x99, 0xAC, 0x4C, 0x80, 0xBE, 0xEE, 0xA9, 0x61, 0x4B,
0x19, 0xCC, 0x4D,
0x5F, 0x4F, 0x5F, 0x55, 0x6E, 0x27, 0xCB, 0xDE, 0x51, 0xC6, 0xA9, 0x4B, 0xE4,
0x60, 0x7A, 0x29,
0x15, 0x58, 0x90, 0x3B, 0xA0, 0xD0, 0xF8, 0x43, 0x80, 0xB6, 0x55, 0xBB, 0x9A,
0x22, 0xE8, 0xDC,
0xDF, 0x02, 0x8A, 0x7C, 0xEC, 0x67, 0xF0, 0xD0, 0x81, 0x34, 0xB1, 0xC8, 0xB9,
0x79, 0x89, 0x14,
0x9B, 0x60, 0x9E, 0x0B, 0xE3, 0xBA, 0xB6, 0x3D, 0x47, 0x54, 0x83, 0x81, 0xDB,
0xC5, 0xB1, 0xFC,
0x76, 0x4E, 0x3F, 0x4B, 0x53, 0xDD, 0x9D, 0xA1, 0x15, 0x8B, 0xFD, 0x3E, 0x2B,
0x9C, 0x8C, 0xF5,
0x6E, 0xDF, 0x01, 0x95, 0x39, 0x34, 0x96, 0x27, 0xDB, 0x2F, 0xD5, 0x3D, 0x24,
0xB7, 0xC4, 0x86,
0x65, 0x77, 0x2E, 0x43, 0x7D, 0x6C, 0x7F, 0x8C, 0xE4, 0x42, 0x73, 0x4A, 0xF7,
0xCC, 0xB7, 0xAE,
0x83, 0x7C, 0x26, 0x4A, 0xE3, 0xA9, 0xBE, 0xB8, 0x7F, 0x8A, 0x2F, 0xE9, 0xB8,
0xB5, 0x29, 0x2E,
0x5A, 0x02, 0x1F, 0xFF, 0x5E, 0x91, 0x47, 0x9E, 0x8C, 0xE7, 0xA2, 0x8C, 0x24,
0x42, 0xC6, 0xF3,
0x15, 0x18, 0x0F, 0x93, 0x49, 0x9A, 0x23, 0x4D, 0xCF, 0x76, 0xE3, 0xFE, 0xD1,
0x35, 0xF9, 0xBB
};

unsigned char dhgp4_pVal[] =
{0xAC, 0x6B, 0xDB, 0x41, 0x32, 0x4A, 0x9A, 0x9B, 0xF1, 0x66, 0xDE, 0x5E, 0x13,
0x89, 0x58, 0x2F,
0xAF, 0x72, 0xB6, 0x65, 0x19, 0x87, 0xEE, 0x07, 0xFC, 0x31, 0x92, 0x94, 0x3D,
0xB5, 0x60, 0x50,
0xA3, 0x73, 0x29, 0xCB, 0xB4, 0xA0, 0x99, 0xED, 0x81, 0x93, 0xE0, 0x75, 0x77,
0x67, 0xA1, 0x3D,
0xD5, 0x23, 0x12, 0xAB, 0x4B, 0x03, 0x31, 0x0D, 0xCD, 0x7F, 0x48, 0xA9, 0xDA,
0x04, 0xFD, 0x50,
0xE8, 0x08, 0x39, 0x69, 0xED, 0xB7, 0x67, 0xB0, 0xCF, 0x60, 0x95, 0x17, 0x9A,
0x16, 0x3A, 0xB3,
0x66, 0x1A, 0x05, 0xFB, 0xD5, 0xFA, 0xAA, 0xE8, 0x29, 0x18, 0xA9, 0x96, 0x2F,
0x0B, 0x93, 0xB8,
0x55, 0xF9, 0x79, 0x93, 0xEC, 0x97, 0x5E, 0xEA, 0xA8, 0x0D, 0x74, 0x0A, 0xDB,
0xF4, 0xFF, 0x74,
0x73, 0x59, 0xD0, 0x41, 0xD5, 0xC3, 0x3E, 0xA7, 0x1D, 0x28, 0x1E, 0x44, 0x6B,
0x14, 0x77, 0x3B,
0xCA, 0x97, 0xB4, 0x3A, 0x23, 0xFB, 0x80, 0x16, 0x76, 0xBD, 0x20, 0x7A, 0x43,
0x6C, 0x64, 0x81,
0xF1, 0xD2, 0xB9, 0x07, 0x87, 0x17, 0x46, 0x1A, 0x5B, 0x9D, 0x32, 0xE6, 0x88,
0xF8, 0x77, 0x48,
0x54, 0x45, 0x23, 0xB5, 0x24, 0xB0, 0xD5, 0x7D, 0x5E, 0xA7, 0x7A, 0x27, 0x75,
0xD2, 0xEC, 0xFA,
0x03, 0x2C, 0xFB, 0xDB, 0xF5, 0x2F, 0xB3, 0x78, 0x61, 0x60, 0x27, 0x90, 0x04,
0xE5, 0x7A, 0xE6,
```



```
0xAF, 0x87, 0x4E, 0x73, 0x03, 0xCE, 0x53, 0x29, 0x9C, 0xCC, 0x04, 0x1C, 0x7B,
0xC3, 0x08, 0xD8,
0x2A, 0x56, 0x98, 0xF3, 0xA8, 0xD0, 0xC3, 0x82, 0x71, 0xAE, 0x35, 0xF8, 0xE9,
0xDB, 0xFB, 0xB6,
0x94, 0xB5, 0xC8, 0x03, 0xD8, 0x9F, 0x7A, 0xE4, 0x35, 0xDE, 0x23, 0x6D, 0x52,
0x5F, 0x54, 0x75,
0x9B, 0x65, 0xE3, 0x72, 0xFC, 0xD6, 0x8E, 0xF2, 0x0F, 0xA7, 0x11, 0x1F, 0x9E,
0x4A, 0xFF, 0x73
};

/*
 * myrand is used for test only, eventually it should be replaced by the random
 * number. AND it is basically the private key.
 */
/* #define      MYRAND */
#ifdef MYRAND
unsigned char myrand[] =
{0x11, 0x11, 0x22, 0x22,
        0x33, 0x33, 0x44, 0x44,
        0x55, 0x55, 0x66, 0x66,
        0x77, 0x77, 0x88, 0x88,
0x99, 0x99, 0x00, 0x00};
#endif  /* MYRAND */




/* Node Events */
#define NODE_EVENT_DEVICE_RM    0x0 /* Auth response timeout & fail */
#define NODE_EVENT_DEVICE_RECOVERY 0x1 /* Auth response timeout & recovery */
#define NODE_EVENT_RCV_AUTH_MSG  0x2 /* Unsolicited Auth received */
#define NODE_EVENT_CMPL_AUTH_MSG 0x3
#define NODE_EVENT_MAX_EVENT     0x4

emlxs_table_t emlxs_event_table[] =
{
        {NODE_EVENT_DEVICE_RM, "DEVICE_REMOVE"},
        {NODE_EVENT_DEVICE_RECOVERY, "DEVICE_RECOVERY"},
        {NODE_EVENT_RCV_AUTH_MSG, "AUTH_MSG_RCVD"},
        {NODE_EVENT_CMPL_AUTH_MSG, "AUTH_MSG_CMPL"},

};      /* emlxs_event_table() */

emlxs_table_t emlxs_pstate_table[] =
{
        {ELX_FABRIC_STATE_UNKNOWN, "FABRIC_STATE_UNKNOWN"},
        {ELX_FABRIC_AUTH_DISABLED, "FABRIC_AUTH_DISABLED"},
        {ELX_FABRIC_AUTH_FAILED, "FABRIC_AUTH_FAILED"},
        {ELX_FABRIC_AUTH_SUCCESS, "FABRIC_AUTH_SUCCESS"},
        {ELX_FABRIC_IN_AUTH, "FABRIC_IN_AUTH"},
        {ELX_FABRIC_IN_REAUTH, "FABRIC_IN_REAUTH"},

};      /* emlxs_pstate_table() */

emlxs_table_t emlxs_nstate_table[] =
{
{NODE_STATE_UNKNOWN, "STATE_UNKNOWN"},
{NODE_STATE_AUTH_DISABLED, "AUTH_DISABLED"},
{NODE_STATE_AUTH_FAILED, "AUTH_FAILED"},
{NODE_STATE_AUTH_SUCCESS, "AUTH_SUCCESS"},
{NODE_STATE_AUTH_NEGOTIATE_ISSUE, "NEGOTIATE_ISSUE"},
{NODE_STATE_AUTH_NEGOTIATE_RCV, "NEGOTIATE_RCV"},
{NODE_STATE_AUTH_NEGOTIATE_CMPL_WAIT4NEXT, "NEGOTIATE_CMPL"},
{NODE_STATE_DHCHAP_CHALLENGE_ISSUE, "DHCHAP_CHALLENGE_ISSUE"},
{NODE_STATE_DHCHAP_REPLY_ISSUE, "DHCHAP_REPLY_ISSUE"},
{NODE_STATE_DHCHAP_CHALLENGE_CMPL_WAIT4NEXT, "DHCHAP_CHALLENGE_CMPL"},
```

```
392 {NODE_STATE_DHCHAP_REPLY_CMPL_WAIT4NEXT, "DHCHAP_REPLY_CMPL"},
393 {NODE_STATE_DHCHAP_SUCCESS_ISSUE, "DHCHAP_SUCCESS_ISSUE"},
394 {NODE_STATE_DHCHAP_SUCCESS_ISSUE_WAIT4NEXT, "DHCHAP_SUCCESS_ISSUE_WAIT"},
395 {NODE_STATE_DHCHAP_SUCCESS_CMPL_WAIT4NEXT, "DHCHAP_SUCCESS_CMPL"},
396 };      /* emlxs_nstate_table() */

398 extern char *
399 emlxs_dhc_event_xlate(uint32_t state)
400 {
401         static char buffer[32];
402         uint32_t i;
403         uint32_t count;

405         count = sizeof (emlxs_event_table) / sizeof (emlxs_table_t);
406         for (i = 0; i < count; i++) {
407                 if (state == emlxs_event_table[i].code) {
408                         return (emlxs_event_table[i].string);
409                 }
410         }

412         (void) sprintf(buffer, "event=0x%x", state);
413         return (buffer);

415 } /* emlxs_dhc_event_xlate() */


418 extern void
419 emlxs_dhc_state(emlxs_port_t *port, emlxs_node_t *ndlp, uint32_t state,
420         uint32_t reason, uint32_t explaination)
421 {
422         emlxs_hba_t *hba = HBA;
423         emlxs_port_dhc_t *port_dhc = &port->port_dhc;
424         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
425         uint32_t pstate;

427         if ((state != NODE_STATE_NOCHANGE) && (node_dhc->state != state)) {
428                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_state_msg,
429                     "Node:0x%x %s --> %s", ndlp->nlp_DID,
430                     emlxs_dhc_nstate_xlate(node_dhc->state),
431                     emlxs_dhc_nstate_xlate(state));

433                 node_dhc->prev_state = node_dhc->state;
434                 node_dhc->state = (uint16_t)state;

436                 /* Perform common functions based on state */
437                 switch (state) {
438                 case NODE_STATE_UNKNOWN:
439                 case NODE_STATE_AUTH_DISABLED:
440                         node_dhc->nlp_authrsp_tmo = 0;
441                         node_dhc->nlp_authrsp_tmocnt = 0;
442                         emlxs_dhc_set_reauth_time(port, ndlp, DISABLE);
443                         break;

445                 case NODE_STATE_AUTH_SUCCESS:
446                         /* Record auth time */
447                         if (ndlp->nlp_DID == FABRIC_DID) {
448                                 port_dhc->auth_time = DRV_TIME;
449                         } else if (node_dhc->parent_auth_cfg) {
450                                 node_dhc->parent_auth_cfg->auth_time = DRV_TIME;
451                         }
452                         hba->rdn_flag = 0;
453                         node_dhc->nlp_authrsp_tmo = 0;

455                         if (node_dhc->flag & NLP_SET_REAUTH_TIME) {
456                                 emlxs_dhc_set_reauth_time(port, ndlp, ENABLE);
457                         }
```



```
458                         break;

460                 default:
461                         break;
462                 }

464                 /* Check for switch port */
465                 if (ndlp->nlp_DID == FABRIC_DID) {
466                         switch (state) {
467                         case NODE_STATE_UNKNOWN:
468                                 pstate = ELX_FABRIC_STATE_UNKNOWN;
469                                 break;

471                         case NODE_STATE_AUTH_DISABLED:
472                                 pstate = ELX_FABRIC_AUTH_DISABLED;
473                                 break;

475                         case NODE_STATE_AUTH_FAILED:
476                                 pstate = ELX_FABRIC_AUTH_FAILED;
477                                 break;

479                         case NODE_STATE_AUTH_SUCCESS:
480                                 pstate = ELX_FABRIC_AUTH_SUCCESS;
481                                 break;

483                                 /* Auth active */
484                         default:
485                                 if (port_dhc->state ==
486                                     ELX_FABRIC_AUTH_SUCCESS) {
487                                         pstate = ELX_FABRIC_IN_REAUTH;
488                                 } else if (port_dhc->state !=
489                                     ELX_FABRIC_IN_REAUTH) {
490                                         pstate = ELX_FABRIC_IN_AUTH;
491                                 }
492                                 break;
493                         }

495                         if (port_dhc->state != pstate) {
496                                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_state_msg,
497                                     "Port: %s --> %s",
498                                     emlxs_dhc_pstate_xlate(port_dhc->state),
499                                     emlxs_dhc_pstate_xlate(pstate));

501                                 port_dhc->state = pstate;
502                         }
503                 }
504         }
505         /* Update auth status */
506         mutex_enter(&hba->auth_lock);
507         emlxs_dhc_status(port, ndlp, reason, explaination);
508         mutex_exit(&hba->auth_lock);

510         return;

512 } /* emlxs_dhc_state() */


515 /* auth_lock must be held when calling this */
516 extern void
517 emlxs_dhc_status(emlxs_port_t *port, emlxs_node_t *ndlp, uint32_t reason,
518         uint32_t explaination)
519 {
520         emlxs_port_dhc_t *port_dhc;
521         emlxs_node_dhc_t *node_dhc;
522         dfc_auth_status_t *auth_status;
523         uint32_t drv_time;
```

```
525         if (!ndlp || !ndlp->nlp_active || ndlp->node_dhc.state ==
526             NODE_STATE_UNKNOWN) {
527                 return;
528         }
529         port_dhc = &port->port_dhc;
530         node_dhc = &ndlp->node_dhc;
531
532         /* Get auth status object */
533         if (ndlp->nlp_DID == FABRIC_DID) {
534                 auth_status = &port_dhc->auth_status;
535         } else if (node_dhc->parent_auth_cfg) {
536                 auth_status = &node_dhc->parent_auth_cfg->auth_status;
537         } else {
538                 /* No auth status to be updated */
539                 return;
540         }
541
542         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_status_msg,
543             "Node:0x%x state=%s rsn=0x%x exp=0x%x (%x,%x)",
544             ndlp->nlp_DID, emlxs_dhc_nstate_xlate(node_dhc->state), reason,
545             explaination, auth_status->auth_state,
546             auth_status->auth_failReason);
547
548         /* Set state and auth_failReason */
549         switch (node_dhc->state) {
550         case NODE_STATE_UNKNOWN:        /* Connection */
551                 if (auth_status->auth_state != DFC_AUTH_STATE_FAILED) {
552                         auth_status->auth_state = DFC_AUTH_STATE_OFF;
553                         auth_status->auth_failReason = 0;
554                 }
555                 break;
556
557         case NODE_STATE_AUTH_DISABLED:
558                 auth_status->auth_state = DFC_AUTH_STATE_OFF;
559                 auth_status->auth_failReason = 0;
560                 break;
561
562         case NODE_STATE_AUTH_FAILED:
563                 /* Check failure reason and update if neccessary */
564                 switch (reason) {
565                 case AUTHRJT_FAILURE:   /* 0x01 */
566                 case AUTHRJT_LOGIC_ERR: /* 0x02 */
567                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
568                         auth_status->auth_failReason = DFC_AUTH_FAIL_REJECTED;
569                         break;
570
571                 case LSRJT_AUTH_REQUIRED:       /* 0x03 */
572                         switch (explaination) {
573                         case LSEXP_AUTH_REQUIRED:
574                                 auth_status->auth_state = DFC_AUTH_STATE_FAILED;
575                                 auth_status->auth_failReason =
576                                     DFC_AUTH_FAIL_LS_RJT;
577                                 break;
578                         default:
579                                 auth_status->auth_state = DFC_AUTH_STATE_FAILED;
580                                 auth_status->auth_failReason =
581                                     DFC_AUTH_FAIL_REJECTED;
582                         }
583                         break;
584
585                 case LSRJT_AUTH_LOGICAL_BSY:    /* 0x05 */
586                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
587                         auth_status->auth_failReason = DFC_AUTH_FAIL_BSY_LS_RJT;
588                         break;
```



```
590                 case LSRJT_AUTH_ELS_NOT_SUPPORTED:      /* 0x0B */
591                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
592                         auth_status->auth_failReason = DFC_AUTH_FAIL_LS_RJT;
593                         break;
594
595                 case LSRJT_AUTH_NOT_LOGGED_IN:  /* 0x09 */
596                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
597                         auth_status->auth_failReason = DFC_AUTH_FAIL_BSY_LS_RJT;
598                         break;
599                 }
600
601                 /* Make sure the state is set to failed at this point */
602                 if (auth_status->auth_state != DFC_AUTH_STATE_FAILED) {
603                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
604                         auth_status->auth_failReason = DFC_AUTH_FAIL_GENERIC;
605                 }
606                 break;
607
608         case NODE_STATE_AUTH_SUCCESS:
609                 auth_status->auth_state = DFC_AUTH_STATE_ON;
610                 auth_status->auth_failReason = 0;
611                 break;
612
613                 /* Authentication currently active */
614         default:
615                 /* Set defaults */
616                 auth_status->auth_state = DFC_AUTH_STATE_INP;
617                 auth_status->auth_failReason = 0;
618
619                 /* Check codes for exceptions */
620                 switch (reason) {
621                 case AUTHRJT_FAILURE:   /* 0x01 */
622                         switch (explaination) {
623                         case AUTHEXP_AUTH_FAILED:       /* 0x05 */
624                         case AUTHEXP_BAD_PAYLOAD:       /* 0x06 */
625                         case AUTHEXP_BAD_PROTOCOL:      /* 0x07 */
626                                 auth_status->auth_state = DFC_AUTH_STATE_FAILED;
627                                 auth_status->auth_failReason =
628                                     DFC_AUTH_FAIL_REJECTED;
629                                 break;
630                         }
631                         break;
632
633                 case AUTHRJT_LOGIC_ERR: /* 0x02 */
634                         switch (explaination) {
635                         case AUTHEXP_MECH_UNUSABLE:     /* 0x01 */
636                         case AUTHEXP_DHGROUP_UNUSABLE:  /* 0x02 */
637                         case AUTHEXP_HASHFUNC_UNUSABLE: /* 0x03 */
638                         case AUTHEXP_CONCAT_UNSUPP:     /* 0x09 */
639                         case AUTHEXP_BAD_PROTOVERS:     /* 0x0A */
640                                 auth_status->auth_state = DFC_AUTH_STATE_FAILED;
641                                 auth_status->auth_failReason =
642                                     DFC_AUTH_FAIL_REJECTED;
643                                 break;
644                         }
645                         break;
646
647                 case LSRJT_AUTH_REQUIRED:       /* 0x03 */
648                         switch (explaination) {
649                         case LSEXP_AUTH_REQUIRED:
650                                 auth_status->auth_state = DFC_AUTH_STATE_FAILED;
651                                 auth_status->auth_failReason =
652                                     DFC_AUTH_FAIL_LS_RJT;
653                                 break;
654                         }
655                         break;
```

```
657                 case LSRJT_AUTH_LOGICAL_BSY:    /* 0x05 */
658                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
659                         auth_status->auth_failReason = DFC_AUTH_FAIL_BSY_LS_RJT;
660                         break;
661
662                 case LSRJT_AUTH_ELS_NOT_SUPPORTED:      /* 0x0B */
663                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
664                         auth_status->auth_failReason = DFC_AUTH_FAIL_LS_RJT;
665                         break;
666
667                 case LSRJT_AUTH_NOT_LOGGED_IN:  /* 0x09 */
668                         auth_status->auth_state = DFC_AUTH_STATE_FAILED;
669                         auth_status->auth_failReason = DFC_AUTH_FAIL_BSY_LS_RJT;
670                         break;
671                 }
672                 break;
673         }
674
675         if (auth_status->auth_state != DFC_AUTH_STATE_ON) {
676                 auth_status->time_until_next_auth = 0;
677                 auth_status->localAuth = 0;
678                 auth_status->remoteAuth = 0;
679                 auth_status->group_priority = 0;
680                 auth_status->hash_priority = 0;
681                 auth_status->type_priority = 0;
682         } else {
683                 switch (node_dhc->nlp_reauth_status) {
684                 case NLP_HOST_REAUTH_ENABLED:
685                 case NLP_HOST_REAUTH_IN_PROGRESS:
686                         drv_time = DRV_TIME;
687
688                         if (node_dhc->nlp_reauth_tmo > drv_time) {
689                                 auth_status->time_until_next_auth =
690                                     node_dhc->nlp_reauth_tmo - drv_time;
691                         } else {
692                                 auth_status->time_until_next_auth = 0;
693                         }
694                         break;
695
696                 case NLP_HOST_REAUTH_DISABLED:
697                 default:
698                         auth_status->time_until_next_auth = 0;
699                         break;
700                 }
701
702                 if (node_dhc->flag & NLP_REMOTE_AUTH) {
703                         auth_status->localAuth = 0;
704                         auth_status->remoteAuth = 1;
705                 } else {
706                         auth_status->localAuth = 1;
707                         auth_status->remoteAuth = 0;
708                 }
709
710                 auth_status->type_priority = DFC_AUTH_TYPE_DHCHAP;
711
712                 switch (node_dhc->nlp_auth_dhgpid) {
713                 case GROUP_NULL:
714                         auth_status->group_priority = ELX_GROUP_NULL;
715                         break;
716
717                 case GROUP_1024:
718                         auth_status->group_priority = ELX_GROUP_1024;
719                         break;
720
721                 case GROUP_1280:
```



```
722                         auth_status->group_priority = ELX_GROUP_1280;
723                         break;
724
725                 case GROUP_1536:
726                         auth_status->group_priority = ELX_GROUP_1536;
727                         break;
728
729                 case GROUP_2048:
730                         auth_status->group_priority = ELX_GROUP_2048;
731                         break;
732                 }
733
734                 switch (node_dhc->nlp_auth_hashid) {
735                 case 0:
736                         auth_status->hash_priority = 0;
737                         break;
738
739                 case AUTH_SHA1:
740                         auth_status->hash_priority = ELX_SHA1;
741                         break;
742
743                 case AUTH_MD5:
744                         auth_status->hash_priority = ELX_MD5;
745                         break;
746                 }
747         }
748
749         return;
750
751 } /* emlxs_dhc_status()  */
752
753 static char *
754 emlxs_dhc_pstate_xlate(uint32_t state)
755 {
756         static char buffer[32];
757         uint32_t i;
758         uint32_t count;
759
760         count = sizeof (emlxs_pstate_table) / sizeof (emlxs_table_t);
761         for (i = 0; i < count; i++) {
762                 if (state == emlxs_pstate_table[i].code) {
763                         return (emlxs_pstate_table[i].string);
764                 }
765         }
766
767         (void) sprintf(buffer, "state=0x%x", state);
768         return (buffer);
769
770 } /* emlxs_dhc_pstate_xlate() */
771
772
773 static char *
774 emlxs_dhc_nstate_xlate(uint32_t state)
775 {
776         static char buffer[32];
777         uint32_t i;
778         uint32_t count;
779
780         count = sizeof (emlxs_nstate_table) / sizeof (emlxs_table_t);
781         for (i = 0; i < count; i++) {
782                 if (state == emlxs_nstate_table[i].code) {
783                         return (emlxs_nstate_table[i].string);
784                 }
785         }
786
787         (void) sprintf(buffer, "state=0x%x", state);
```

```
788         return (buffer);

790 } /* emlxs_dhc_nstate_xlate() */


793 static uint32_t
794 emlxs_check_dhgp(
795         emlxs_port_t *port,
796         NODELIST *ndlp,
797         uint32_t *dh_id,
798         uint16_t cnt,
799         uint32_t *dhgp_id)
800 {
801         uint32_t i, j, rc = 1;
802         uint32_t wnt;
803         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

805         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
806             "dhgp: 0x%x, id[0..4]=0x%x 0x%x 0x%x 0x%x 0x%x pri[1]=0x%x",
807             cnt, dh_id[0], dh_id[1], dh_id[2], dh_id[3], dh_id[4],
808             node_dhc->auth_cfg.dh_group_priority[1]);

810         /*
811          * Here are the rules, as the responder We always try to select ours
812          * highest setup
813          */

815         /* Check to see if there is any repeated dhgp in initiator's list */
816         /* If available, it is a invalid payload */
817         if (cnt >= 2) {
818                 for (i = 0; i <= cnt - 2; i++) {
819                         for (j = i + 1; j <= cnt - 1; j++) {
820                                 if (dh_id[i] == dh_id[j]) {
821                                         rc = 2;
822                                         EMLXS_MSGF(EMLXS_CONTEXT,
823                                             &emlxs_fcsp_detail_msg,
824                                             ":Rpt dhid[%x]=%x dhid[%x]=%x",
825                                             i, dh_id[i], j, dh_id[j]);
826                                         break;
827                                 }
828                         }

830                         if (rc == 2) {
831                                 break;
832                         }
833                 }

835                 if ((i == cnt - 1) && (j == cnt)) {
836                         rc = 1;
837                 }
838                 if (rc == 2) {
839                         /* duplicate invalid payload */
840                         return (rc);
841                 }
842         }
843         /* Check how many dhgps the responder specified */
844         wnt = 0;
845         while (node_dhc->auth_cfg.dh_group_priority[wnt] != 0xF) {
846                 wnt++;
847         }

849         /* Determine the most suitable dhgp the responder should use */
850         for (i = 0; i < wnt; i++) {
851                 for (j = 0; j < cnt; j++) {
852                         if (node_dhc->auth_cfg.dh_group_priority[i] ==
853                             dh_id[j]) {
```



```
854                                 rc = 0;
855                                 *dhgp_id =
856                                     node_dhc->auth_cfg.dh_group_priority[i];
857                                 break;
858                         }
859                 }

861                 if (rc == 0) {
862                         break;
863                 }
864         }

866         if (i == wnt) {
867                 /* no match */
868                 rc = 1;
869                 return (1);
870         }

872         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
873             "emlxs_check_dhgp: dhgp_id=0x%x", *dhgp_id);

875         return (rc);
876 } /* emlxs_check_dhgp */


879 static void
880 emlxs_get_random_bytes(
881         NODELIST *ndlp,
882         uint8_t *rdn,
883         uint32_t len)
884 {
885         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
886         hrtime_t now;
887         uint8_t sha1_digest[20];
888         SHA1_CTX sha1ctx;

890         now = gethrtime();

892         bzero(&sha1ctx, sizeof (SHA1_CTX));
893         SHA1Init(&sha1ctx);
894         SHA1Update(&sha1ctx, (void *) &node_dhc->auth_cfg.local_entity,
895             sizeof (NAME_TYPE));
896         SHA1Update(&sha1ctx, (void *) &now, sizeof (hrtime_t));
897         SHA1Final((void *) sha1_digest, &sha1ctx);
898         bcopy((void *) &sha1_digest[0], (void *) &rdn[0], len);

900         return;

902 } /* emlxs_get_random_bytes */


905 /* ************************** STATE MACHINE ************************ */

907 static void *emlxs_dhchap_action[] =
908 {
909         /* Action routine               Event */

911 /* NODE_STATE_UNKNOWN  0x00 */
912         (void *) emlxs_disc_neverdev,   /* DEVICE_RM */
913         (void *) emlxs_disc_neverdev,   /* DEVICE_RECOVERY */
914         (void *) emlxs_disc_neverdev,   /* RCV_AUTH_MSG */
915         (void *) emlxs_disc_neverdev,   /* CMPL_AUTH_MSG */

917 /* NODE_STATE_AUTH_DISABLED  0x01 */
918         (void *) emlxs_disc_neverdev,   /* DEVICE_RM */
919         (void *) emlxs_disc_neverdev,   /* DEVICE_RECOVERY */
```

```
920         (void *) emlxs_disc_neverdev,   /* RCV_AUTH_MSG */
921         (void *) emlxs_disc_neverdev,   /* CMPL_AUTH_MSG */

923 /* NODE_STATE_AUTH_FAILED  0x02 */
924         (void *) emlxs_device_rm_npr_node,      /* DEVICE_RM */
925         (void *) emlxs_device_recov_npr_node,   /* DEVICE_RECOVERY */
926         (void *) emlxs_rcv_auth_msg_npr_node,   /* RCV_AUTH_MSG */
927         (void *) emlxs_cmpl_auth_msg_npr_node,  /* CMPL_AUTH_MSG */

929 /* NODE_STATE_AUTH_SUCCESS  0x03 */
930         (void *) emlxs_disc_neverdev,                   /* DEVICE_RM */
931         (void *) emlxs_device_recov_unmapped_node,      /* DEVICE_RECOVERY */
932         (void *) emlxs_rcv_auth_msg_unmapped_node,      /* RCV_AUTH_MSG */
933         (void *) emlxs_disc_neverdev,                   /* CMPL_AUTH_MSG */

935 /* NODE_STATE_AUTH_NEGOTIATE_ISSUE  0x04 */
936         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
937         (void *) emlxs_device_recov_auth, /* DEVICE_RECOVERY */
938         (void *) emlxs_rcv_auth_msg_auth_negotiate_issue, /* RCV_AUTH_MSG  */
939         (void *) emlxs_cmpl_auth_msg_auth_negotiate_issue, /* CMPL_AUTH_MSG */

941 /* NODE_STATE_AUTH_NEGOTIATE_RCV  0x05 */
942         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
943         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
944         (void *) emlxs_rcv_auth_msg_auth_negotiate_rcv, /* RCV_AUTH_MSG */
945         (void *) emlxs_cmpl_auth_msg_auth_negotiate_rcv, /* CMPL_AUTH_MSG */

947 /* NODE_STATE_AUTH_NEGOTIATE_CMPL_WAIT4NEXT  0x06 */
948         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
949         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
950         (void *) emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next,
951                                         /* RCV_AUTH_MSG */
952         (void *) emlxs_cmpl_auth_msg_auth_negotiate_cmpl_wait4next,
953                                         /* CMPL_AUTH_MSG */

955 /* NODE_STATE_DHCHAP_CHALLENGE_ISSUE  0x07 */
956         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
957         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
958         (void *) emlxs_rcv_auth_msg_dhchap_challenge_issue, /* RCV_AUTH_MSG */
959         (void *) emlxs_cmpl_auth_msg_dhchap_challenge_issue, /* CMPL_AUTH_MSG */

961 /* NODE_STATE_DHCHAP_REPLY_ISSUE  0x08 */
962         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
963         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
964         (void *) emlxs_rcv_auth_msg_dhchap_reply_issue, /* RCV_AUTH_MSG */
965         (void *) emlxs_cmpl_auth_msg_dhchap_reply_issue, /* CMPL_AUTH_MSG */

967 /* NODE_STATE_DHCHAP_CHALLENGE_CMPL_WAIT4NEXT  0x09 */
968         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
969         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
970         (void *) emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next,
971                                         /* RCV_AUTH_MSG   */
972         (void *) emlxs_cmpl_auth_msg_dhchap_challenge_cmpl_wait4next,
973                                         /* CMPL_AUTH_MSG */

975 /* NODE_STATE_DHCHAP_REPLY_CMPL_WAIT4NEXT  0x0A */
976         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
977         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
978         (void *) emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next,
979                                         /* RCV_AUTH_MSG */
980         (void *) emlxs_cmpl_auth_msg_dhchap_reply_cmpl_wait4next,
981                                         /* CMPL_AUTH_MSG */

983 /* NODE_STATE_DHCHAP_SUCCESS_ISSUE  0x0B */
984         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
985         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
```



```
986         (void *) emlxs_rcv_auth_msg_dhchap_success_issue,
987                                         /* RCV_AUTH_MSG */
988         (void *) emlxs_cmpl_auth_msg_dhchap_success_issue,
989                                         /* CMPL_AUTH_MSG */

991 /* NODE_STATE_DHCHAP_SUCCESS_ISSUE_WAIT4NEXT  0x0C */
992         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
993         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
994         (void *) emlxs_rcv_auth_msg_dhchap_success_issue_wait4next,
995                                         /* RCV_AUTH_MSG */
996         (void *) emlxs_cmpl_auth_msg_dhchap_success_issue_wait4next,
997                                         /* CMPL_AUTH_MSG */

999 /* NODE_STATE_DHCHAP_SUCCESS_CMPL_WAIT4NEXT  0x0D */
1000         (void *) emlxs_device_rem_auth, /* DEVICE_RM */
1001         (void *) emlxs_device_recov_auth,       /* DEVICE_RECOVERY */
1002         (void *) emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next,
1003                                         /* RCV_AUTH_MSG */
1004         (void *) emlxs_cmpl_auth_msg_dhchap_success_cmpl_wait4next,
1005                                         /* CMPL_AUTH_MSG */

1007 }; /* emlxs_dhchap_action[] */


1010 extern int
1011 emlxs_dhchap_state_machine(emlxs_port_t *port, CHANNEL *cp,
1012                 IOCBQ *iocbq, MATCHMAP *mp,
1013                 NODELIST *ndlp, int evt)
1014 {
1015         emlxs_hba_t *hba = HBA;
1016         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
1017         uint32_t rc;
1018         uint32_t(*func) (emlxs_port_t *, CHANNEL *, IOCBQ *, MATCHMAP *,
1019             NODELIST *, uint32_t);

1021         mutex_enter(&hba->dhc_lock);

1023         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_event_msg,
1024             "%s: did=0x%x",
1025             emlxs_dhc_event_xlate(evt), ndlp->nlp_DID);

1027         node_dhc->disc_refcnt++;

1029         func = (uint32_t(*) (emlxs_port_t *, CHANNEL *, IOCBQ *, MATCHMAP *,
1030             NODELIST *, uint32_t))
1031             emlxs_dhchap_action[(node_dhc->state * NODE_EVENT_MAX_EVENT) + evt];

1033         rc = (func) (port, cp, iocbq, mp, ndlp, evt);

1035         node_dhc->disc_refcnt--;

1037         mutex_exit(&hba->dhc_lock);

1039         return (rc);

1041 } /* emlxs_dhchap_state_machine() */

1043 /* ARGSUSED */
1044 static uint32_t
1045 emlxs_disc_neverdev(
1046 emlxs_port_t *port,
1047 /* CHANNEL * rp, */ void *arg1,
1048 /* IOCBQ * iocbq, */ void *arg2,
1049 /* MATCHMAP * mp, */ void *arg3,
1050 /* NODELIST * ndlp */ void *arg4,
1051 uint32_t evt)
```

```
{
        NODELIST *ndlp = (NODELIST *) arg4;
        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
            "emlxs_disc_neverdev: did=0x%x.",
            ndlp->nlp_DID);

        emlxs_dhc_state(port, ndlp, NODE_STATE_UNKNOWN, 0, 0);

        return (node_dhc->state);

} /* emlxs_disc_neverdev() */


/*
 * ! emlxs_cmpl_dhchap_challenge_issue
 *
 * \pre \post \param   cmdiocb \param   rspiocb \return  void
 *
 * \b Description: iocb_cmpl callback function. when the ELS DHCHAP_Challenge
 * msg sent back got the ACC/RJT from initiator.
 *
 */
static void
emlxs_cmpl_dhchap_challenge_issue(fc_packet_t *pkt)
{
        emlxs_port_t *port = pkt->pkt_ulp_private;
        emlxs_buf_t *sbp;
        NODELIST *ndlp;
        uint32_t did;

        did = pkt->pkt_cmd_fhdr.d_id;
        sbp = (emlxs_buf_t *)pkt->pkt_fca_private;
        ndlp = sbp->node;

        if (!ndlp) {
                ndlp = emlxs_node_find_did(port, did);
        }
        if (pkt->pkt_state != FC_PKT_SUCCESS) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
                    "emlxs_cmpl_dhchap_challenge_issue: did=0x%x state=%x",
                    did, pkt->pkt_state);
        } else {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
                    "emlxs_cmpl_dhchap_challenge_issue: did=0x%x. Succcess.",
                    did);
        }

        if (ndlp) {
                if (pkt->pkt_state == FC_PKT_SUCCESS) {
                        (void) emlxs_dhchap_state_machine(port, NULL, NULL,
                            NULL, ndlp, NODE_EVENT_CMPL_AUTH_MSG);
                }
        }
        emlxs_pkt_free(pkt);

        return;

} /* emlxs_cmpl_dhchap_challenge_issue */




/*
 * ! emlxs_cmpl_dhchap_success_issue
```



```
 *
 * \pre \post \param   phba \param   cmdiocb \param   rspiocb \return  void
 *
 * \b Description: iocb_cmpl callback function.
 *
 */
static void
emlxs_cmpl_dhchap_success_issue(fc_packet_t *pkt)
{
        emlxs_port_t *port = pkt->pkt_ulp_private;
        NODELIST *ndlp;
        uint32_t did;
        emlxs_buf_t *sbp;

        did = pkt->pkt_cmd_fhdr.d_id;
        sbp = (emlxs_buf_t *)pkt->pkt_fca_private;
        ndlp = sbp->node;

        if (!ndlp) {
                ndlp = emlxs_node_find_did(port, did);
        }
        if (pkt->pkt_state != FC_PKT_SUCCESS) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
                    "emlxs_cmpl_dhchap_success_issue: 0x%x %x. No retry.",
                    did, pkt->pkt_state);
        } else {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
                    "emlxs_cmpl_dhchap_success_issue: did=0x%x. Succcess.",
                    did);
        }

        if (ndlp) {
                if (pkt->pkt_state == FC_PKT_SUCCESS) {
                        (void) emlxs_dhchap_state_machine(port, NULL, NULL,
                            NULL, ndlp, NODE_EVENT_CMPL_AUTH_MSG);
                }
        }
        emlxs_pkt_free(pkt);

        return;

} /* emlxs_cmpl_dhchap_success_issue */


/*
 * if rsp == NULL, this is only the DHCHAP_Success msg
 *
 * if rsp != NULL, DHCHAP_Success contains rsp to the challenge.
 */
/* ARGSUSED */
uint32_t
emlxs_issue_dhchap_success(
        emlxs_port_t *port,
        NODELIST *ndlp,
        int retry,
        uint8_t *rsp)
{
        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
        fc_packet_t *pkt;
        uint32_t cmd_size;
        uint32_t rsp_size;
        uint8_t *pCmd;
        uint16_t cmdsize;
        DHCHAP_SUCCESS_HDR *ap;
        uint8_t *tmp;
        uint32_t len;
```

```
1184         uint32_t ret;
1185
1186         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1187             "emlxs_issue_dhchap_success: did=0x%x", ndlp->nlp_DID);
1188
1189         if (ndlp->nlp_DID == FABRIC_DID) {
1190                 if (node_dhc->nlp_auth_hashid == AUTH_MD5)
1191                         len = MD5_LEN;
1192                 else
1193                         len = SHA1_LEN;
1194         } else {
1195                 len = (node_dhc->nlp_auth_hashid == AUTH_MD5) ?
1196                     MD5_LEN : SHA1_LEN;
1197         }
1198
1199         if (rsp == NULL) {
1200                 cmdsize = sizeof (DHCHAP_SUCCESS_HDR);
1201         } else {
1202
1203                 cmdsize = sizeof (DHCHAP_SUCCESS_HDR) + len;
1204         }
1205
1206         cmd_size = cmdsize;
1207         rsp_size = 4;
1208
1209         if ((pkt = emlxs_prep_els_fc_pkt(port, ndlp->nlp_DID, cmd_size,
1210             rsp_size, 0, KM_NOSLEEP)) == NULL) {
1211                 return (1);
1212         }
1213         pCmd = (uint8_t *)pkt->pkt_cmd;
1214
1215         ap = (DHCHAP_SUCCESS_HDR *)pCmd;
1216         tmp = (uint8_t *)pCmd;
1217
1218         ap->auth_els_code = ELS_CMD_AUTH_CODE;
1219         ap->auth_els_flags = 0x0;
1220         ap->auth_msg_code = DHCHAP_SUCCESS;
1221         ap->proto_version = 0x01;
1222
1223         /*
1224          * In case of rsp == NULL meaning that this is DHCHAP_Success issued
1225          * when Host is the initiator AND this DHCHAP_Success is issused in
1226          * response to the bi-directional authentication, meaning Host
1227          * authenticate another entity, therefore no more DHCHAP_Success
1228          * expected. OR this DHCHAP_Success is issued by host when host is
1229          * the responder BUT it is uni-directional auth, therefore no more
1230          * DHCHAP_Success expected.
1231          *
1232          * In case of rsp != NULL it indicates this DHCHAP_Success is issued
1233          * when host is the responder AND this DHCHAP_Success has reply
1234          * embedded therefore the host expects DHCHAP_Success from other
1235          * entity in transaction.
1236          */
1237         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1238             "emlxs_issue_dhchap_success: 0x%x 0x%x 0x%x 0x%x 0x%x %p",
1239             ndlp->nlp_DID, node_dhc->nlp_auth_hashid,
1240             node_dhc->nlp_auth_tranid_rsp,
1241             node_dhc->nlp_auth_tranid_ini, cmdsize, rsp);
1242
1243         if (rsp == NULL) {
1244                 ap->msg_len = LE_SWAP32(0x00000004);
1245                 ap->RspVal_len = 0x0;
1246
1247                 node_dhc->fc_dhchap_success_expected = 0;
1248         } else {
1249                 node_dhc->fc_dhchap_success_expected = 1;
```



```
1251                 ap->msg_len = LE_SWAP32(4 + len);
1252
1253                 tmp += sizeof (DHCHAP_SUCCESS_HDR) - sizeof (uint32_t);
1254                 *(uint32_t *)tmp = LE_SWAP32(len);
1255                 tmp += sizeof (uint32_t);
1256                 bcopy((void *)rsp, (void *)tmp, len);
1257         }
1258
1259         if (node_dhc->nlp_reauth_status == NLP_HOST_REAUTH_IN_PROGRESS) {
1260                 ap->tran_id = LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);
1261         } else {
1262                 if (node_dhc->nlp_auth_flag == 2) {
1263                         ap->tran_id =
1264                             LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);
1265                 } else if (node_dhc->nlp_auth_flag == 1) {
1266                         ap->tran_id =
1267                             LE_SWAP32(node_dhc->nlp_auth_tranid_ini);
1268                 } else {
1269                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
1270                             "emlxs_is_dhch_success: (1) 0x%x 0x%x 0x%x 0x%x",
1271                             ndlp->nlp_DID, node_dhc->nlp_auth_flag,
1272                             node_dhc->nlp_auth_tranid_rsp,
1273                             node_dhc->nlp_auth_tranid_ini);
1274
1275                         return (1);
1276                 }
1277         }
1278
1279         pkt->pkt_comp = emlxs_cmpl_dhchap_success_issue;
1280
1281         ret = emlxs_pkt_send(pkt, 1);
1282
1283         if (ret != FC_SUCCESS) {
1284                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1285                     "emlxs_issue_dhchap_success: Unable to send packet. 0x%x",
1286                     ret);
1287
1288                 emlxs_pkt_free(pkt);
1289
1290                 return (1);
1291         }
1292         return (0);
1293
1294 } /* emlxs_issue_dhchap_success */
1295
1296
1297 /*
1298  * ! emlxs_cmpl_auth_reject_issue
1299  *
1300  * \pre \post \param   phba \param   cmdiocb \param   rspiocb \return  void
1301  *
1302  * \b Description: iocb_cmpl callback function.
1303  *
1304  */
1305 static void
1306 emlxs_cmpl_auth_reject_issue(fc_packet_t *pkt)
1307 {
1308         emlxs_port_t *port = pkt->pkt_ulp_private;
1309         emlxs_buf_t *sbp;
1310         NODELIST *ndlp;
1311         uint32_t did;
1312
1313         did = pkt->pkt_cmd_fhdr.d_id;
1314         sbp = (emlxs_buf_t *)pkt->pkt_fca_private;
1315         ndlp = sbp->node;
```

```
1317         if (!ndlp) {
1318                 ndlp = emlxs_node_find_did(port, did);
1319         }
1320         if (pkt->pkt_state != FC_PKT_SUCCESS) {
1321                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1322                     "emlxs_cmpl_auth_reject_issue: 0x%x %x. No retry.",
1323                     did, pkt->pkt_state);
1324         } else {
1325                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1326                     "emlxs_cmpl_auth_reject_issue: did=0x%x. Succcess.",
1327                     did);
1328         }

1330         if (ndlp) {
1331                 /* setup the new state */
1332                 emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED, 0, 0);

1334                 if (pkt->pkt_state == FC_PKT_SUCCESS) {
1335                         (void) emlxs_dhchap_state_machine(port, NULL, NULL,
1336                             NULL, ndlp, NODE_EVENT_CMPL_AUTH_MSG);
1337                 }
1338         }
1339         emlxs_pkt_free(pkt);

1341         return;

1343 } /* emlxs_cmpl_auth_reject_issue */


1346 /*
1347  * If Logical Error and Reason Code Explanation is "Restart Authentication
1348  * Protocol" then the Transaction Identifier could be
1349  * any value.
1350  */
1351 /* ARGSUSED */
1352 static uint32_t
1353 emlxs_issue_auth_reject(
1354         emlxs_port_t *port,
1355         NODELIST *ndlp,
1356         int retry,
1357         uint32_t *arg,
1358         uint8_t ReasonCode,
1359         uint8_t ReasonCodeExplanation)
1360 {
1361         fc_packet_t *pkt;
1362         uint32_t cmd_size;
1363         uint32_t rsp_size;
1364         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
1365         uint16_t cmdsize;
1366         AUTH_RJT *ap;
1367         char info[64];

1369         if (node_dhc->nlp_authrsp_tmo) {
1370                 node_dhc->nlp_authrsp_tmo = 0;
1371         }
1372         cmdsize = sizeof (AUTH_RJT);
1373         cmd_size = cmdsize;
1374         rsp_size = 4;

1376         if ((pkt = emlxs_prep_els_fc_pkt(port, ndlp->nlp_DID, cmd_size,
1377             rsp_size, 0, KM_NOSLEEP)) == NULL) {
1378                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
1379                     "Auth reject failed: Unable to allocate pkt. 0x%x %x %x",
1380                     ndlp->nlp_DID, ReasonCode, ReasonCodeExplanation);
```



```
1382                 return (1);
1383         }
1384         ap = (AUTH_RJT *) pkt->pkt_cmd;
1385         ap->auth_els_code = ELS_CMD_AUTH_CODE;
1386         ap->auth_els_flags = 0x0;
1387         ap->auth_msg_code = AUTH_REJECT;
1388         ap->proto_version = 0x01;
1389         ap->msg_len = LE_SWAP32(4);

1391         if (node_dhc->nlp_auth_flag == 2) {
1392                 ap->tran_id = LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);
1393         } else if (node_dhc->nlp_auth_flag == 1) {
1394                 ap->tran_id = LE_SWAP32(node_dhc->nlp_auth_tranid_ini);
1395         } else {
1396                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
1397                     "Auth reject failed.Invalid flag=%d. 0x%x %x expl=%x",
1398                     ndlp->nlp_DID, node_dhc->nlp_auth_flag, ReasonCode,
1399                     ReasonCodeExplanation);

1401                 emlxs_pkt_free(pkt);

1403                 return (1);
1404         }

1406         ap->ReasonCode = ReasonCode;
1407         ap->ReasonCodeExplanation = ReasonCodeExplanation;

1409         pkt->pkt_comp = emlxs_cmpl_auth_reject_issue;

1411         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
1412             "Auth reject: did=0x%x reason=%x expl=%x",
1413             ndlp->nlp_DID, ReasonCode, ReasonCodeExplanation);

1415         if (emlxs_pkt_send(pkt, 1) != FC_SUCCESS) {
1416                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
1417                     "Auth reject failed. Unable to send pkt. 0x%x %x expl=%x",
1418                     ndlp->nlp_DID, node_dhc->nlp_auth_flag, ReasonCode,
1419                     ReasonCodeExplanation);

1421                 emlxs_pkt_free(pkt);

1423                 return (1);
1424         }
1425         (void) sprintf(info,
1426             "Auth-Reject: ReasonCode=0x%x, ReasonCodeExplanation=0x%x",
1427             ReasonCode, ReasonCodeExplanation);

1429         emlxs_log_auth_event(port, ndlp, ESC_EMLXS_28, info);

1431         return (0);

1433 } /* emlxs_issue_auth_reject */


1436 static fc_packet_t *
1437         emlxs_prep_els_fc_pkt(
1438         emlxs_port_t *port,
1439         uint32_t d_id,
1440         uint32_t cmd_size,
1441         uint32_t rsp_size,
1442         uint32_t datalen,
1443         int32_t sleepflag)
1444 {
1445         fc_packet_t *pkt;

1447         /* simulate the ULP stack's fc_packet send out */
```

```
1448         if (!(pkt = emlxs_pkt_alloc(port, cmd_size, rsp_size,
1449             datalen, sleepflag))) {
1450                 return (NULL);
1451         }
1452         pkt->pkt_tran_type = FC_PKT_EXCHANGE;
1453         pkt->pkt_timeout = 35;

1455         /* Build the fc header */
1456         pkt->pkt_cmd_fhdr.d_id = LE_SWAP24_LO(d_id);
1457         pkt->pkt_cmd_fhdr.r_ctl = R_CTL_ELS_REQ;
1458         pkt->pkt_cmd_fhdr.s_id = LE_SWAP24_LO(port->did);
1459         pkt->pkt_cmd_fhdr.type = FC_TYPE_EXTENDED_LS;
1460         pkt->pkt_cmd_fhdr.f_ctl =
1461             F_CTL_FIRST_SEQ | F_CTL_END_SEQ | F_CTL_SEQ_INITIATIVE;
1462         pkt->pkt_cmd_fhdr.seq_id = 0;
1463         pkt->pkt_cmd_fhdr.df_ctl = 0;
1464         pkt->pkt_cmd_fhdr.seq_cnt = 0;
1465         pkt->pkt_cmd_fhdr.ox_id = 0xFFFF;
1466         pkt->pkt_cmd_fhdr.rx_id = 0xFFFF;
1467         pkt->pkt_cmd_fhdr.ro = 0;

1469         return ((fc_packet_t *)pkt);

1471 } /* emlxs_prep_els_fc_pkt */


1474 /*
1475  * ! emlxs_issue_auth_negotiate
1476  *
1477  * \pre \post \param   port \param   ndlp \param   retry \param   flag \return
1478  * int
1479  *
1480  * \b Description:
1481  *
1482  * The routine is invoked when host as the authentication initiator which
1483  * issue the AUTH_ELS command AUTH_Negotiate to the other
1484  * entity ndlp. When this Auth_Negotiate command is completed, the iocb_cmpl
1485  * will get called as the solicited mbox cmd
1486  * callback. Some switch only support NULL dhchap in which case negotiate
1487  * should be modified to only have NULL DH specificed.
1488  *
1489  */
1490 /* ARGSUSED */
1491 static int
1492         emlxs_issue_auth_negotiate(
1493         emlxs_port_t *port,
1494         emlxs_node_t *ndlp,
1495         uint8_t retry)
1496 {
1497         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
1498         fc_packet_t *pkt;
1499         uint32_t cmd_size;
1500         uint32_t rsp_size;
1501         uint16_t cmdsize;
1502         AUTH_MSG_NEGOT_NULL_1 *null_ap1;
1503         AUTH_MSG_NEGOT_NULL_2 *null_ap2;
1504         uint32_t num_hs = 0;
1505         uint8_t flag;
1506         AUTH_MSG_NEGOT_1 *ap1;
1507         AUTH_MSG_NEGOT_2 *ap2;
1508         uint16_t para_len = 0;
1509         uint16_t hash_wcnt = 0;
1510         uint16_t dhgp_wcnt = 0;


1513         emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_NEGOTIATE_ISSUE, 0, 0);
```



```
1515         /* Full DH group support limit:2, only NULL group support limit:1 */
1516         flag = (node_dhc->nlp_auth_limit == 2) ? 1 : 0;

1518         /* first: determine the cmdsize based on the auth cfg parameters */
1519         if (flag == 1) {
1520                 /* May be Full DH group + 2 hash may not be */
1521                 cmdsize = sizeof (AUTH_MSG_NEGOT_NULL);

1523                 cmdsize += 2 + 2;       /* name tag: 2, name length: 2 */
1524                 cmdsize += 8;   /* WWN: 8 */
1525                 cmdsize += 4;   /* num of protocol: 4 */
1526                 cmdsize += 4;   /* protocol parms length: 4 */
1527                 cmdsize += 4;   /* protocol id: 4 */
1528                 para_len += 4;

1530                 cmdsize += 2 + 2;       /* hashlist: tag: 2, count:2 */
1531                 para_len += 4;

1533                 if (node_dhc->auth_cfg.hash_priority[1] == 0x00) {
1534                         /* only one hash func */
1535                         cmdsize += 4;
1536                         num_hs = 1;
1537                         para_len += 4;
1538                         hash_wcnt = 1;
1539                 } else {
1540                         /* two hash funcs */
1541                         cmdsize += 4 + 4;
1542                         num_hs = 2;
1543                         para_len += 4 + 4;
1544                         hash_wcnt = 2;
1545                 }

1547                 cmdsize += 2 + 2;
1548                 para_len += 4;
1549                 if (node_dhc->auth_cfg.dh_group_priority[1] == 0xf) {
1550                         /* only one dhgp specified: could be NULL or non-NULL */
1551                         cmdsize += 4;
1552                         para_len += 4;
1553                         dhgp_wcnt = 1;

1555                 } else if (node_dhc->auth_cfg.dh_group_priority[2] == 0xf) {
1556                         /* two dhgps specified */
1557                         cmdsize += 4 + 4;
1558                         para_len += 4 + 4;
1559                         dhgp_wcnt = 2;

1561                 } else if (node_dhc->auth_cfg.dh_group_priority[3] == 0xf) {
1562                         /* three dhgps specified */
1563                         cmdsize += 4 + 4 + 4;
1564                         para_len += 4 + 4 + 4;
1565                         dhgp_wcnt = 3;

1567                 } else if (node_dhc->auth_cfg.dh_group_priority[4] == 0xf) {
1568                         /* four dhgps specified */
1569                         cmdsize += 4 + 4 + 4 + 4;
1570                         para_len += 4 + 4 + 4 + 4;
1571                         dhgp_wcnt = 4;

1573                 } else if (node_dhc->auth_cfg.dh_group_priority[5] == 0xf) {
1574                         cmdsize += 4 + 4 + 4 + 4 + 4;
1575                         para_len += 4 + 4 + 4 + 4 + 4;
1576                         dhgp_wcnt = 5;

1578                 }
1579         } else {
```

```
1580                cmdsize = sizeof (AUTH_MSG_NEGOT_NULL);

1582                /*
1583                 * get the right payload size in byte: determined by config
1584                 * parameters
1585                 */
1586                cmdsize += 2 + 2 + 8;   /* name tag:2, name length:2, name */
1587                                        /* value content:8 */
1588                cmdsize += 4;   /* number of usable authentication */
1589                                /* protocols:4 */
1590                cmdsize += 4;   /* auth protocol params length: 4 */
1591                cmdsize += 4;   /* auth protocol identifier: 4 */

1593                /* hash list infor */
1594                cmdsize += 4;   /* hashlist: tag:2, count:2 */

1596                if (node_dhc->auth_cfg.hash_priority[1] == 0x00) {
1597                        cmdsize += 4;   /* only one hash function provided */
1598                        num_hs = 1;
1599                } else {
1600                        num_hs = 2;
1601                        cmdsize += 4 + 4;       /* sha1: 4, md5: 4 */
1602                }

1604                /* dhgp list info */
1605                /* since this is NULL DH group */
1606                cmdsize += 4;   /* dhgroup: tag:2, count:2 */
1607                cmdsize += 4;   /* set it to zero */
1608        }

1610        cmd_size = cmdsize;
1611        rsp_size = 4;

1613        if ((pkt = emlxs_prep_els_fc_pkt(port, ndlp->nlp_DID, cmd_size,
1614            rsp_size, 0, KM_NOSLEEP)) == NULL) {
1615                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
1616                    "issue_auth_negotiate: Unable to allocate pkt. 0x%x %d",
1617                    ndlp->nlp_DID, cmd_size);

1619                return (1);
1620        }
1621        /* Fill in AUTH_MSG_NEGOT payload */
1622        if (flag == 1) {
1623                if (hash_wcnt == 1) {
1624                        ap1 = (AUTH_MSG_NEGOT_1 *)pkt->pkt_cmd;
1625                        ap1->auth_els_code = ELS_CMD_AUTH_CODE;
1626                        ap1->auth_els_flags = 0x00;
1627                        ap1->auth_msg_code = AUTH_NEGOTIATE;
1628                        ap1->proto_version = 0x01;
1629                        ap1->msg_len = LE_SWAP32(cmdsize -
1630                            sizeof (AUTH_MSG_NEGOT_NULL));
1631                } else {
1632                        ap2 = (AUTH_MSG_NEGOT_2 *)pkt->pkt_cmd;
1633                        ap2->auth_els_code = ELS_CMD_AUTH_CODE;
1634                        ap2->auth_els_flags = 0x00;
1635                        ap2->auth_msg_code = AUTH_NEGOTIATE;
1636                        ap2->proto_version = 0x01;
1637                        ap2->msg_len = LE_SWAP32(cmdsize -
1638                            sizeof (AUTH_MSG_NEGOT_NULL));
1639                }
1640        } else {
1641                if (node_dhc->auth_cfg.hash_priority[1] == 0x00) {
1642                        null_ap1 = (AUTH_MSG_NEGOT_NULL_1 *)pkt->pkt_cmd;
1643                        null_ap1->auth_els_code = ELS_CMD_AUTH_CODE;
1644                        null_ap1->auth_els_flags = 0x0;
1645                        null_ap1->auth_msg_code = AUTH_NEGOTIATE;
```



```
1646                        null_ap1->proto_version = 0x01;
1647                        null_ap1->msg_len = LE_SWAP32(cmdsize -
1648                            sizeof (AUTH_MSG_NEGOT_NULL));

1650                } else {
1651                        null_ap2 = (AUTH_MSG_NEGOT_NULL_2 *)pkt->pkt_cmd;
1652                        null_ap2->auth_els_code = ELS_CMD_AUTH_CODE;
1653                        null_ap2->auth_els_flags = 0x0;
1654                        null_ap2->auth_msg_code = AUTH_NEGOTIATE;
1655                        null_ap2->proto_version = 0x01;
1656                        null_ap2->msg_len = LE_SWAP32(cmdsize -
1657                            sizeof (AUTH_MSG_NEGOT_NULL));
1658                }
1659        }

1661        /*
1662         * For host reauthentication heart beat, the tran_id is incremented
1663         * by one for each heart beat being fired and round back to 1 when
1664         * 0xffffffff is reached. tran_id 0 is reserved as the initial linkup
1665         * authentication transaction id.
1666         */

1668        /* responder flag:2, initiator flag:1 */
1669        node_dhc->nlp_auth_flag = 2;    /* ndlp is the always the auth */
1670                                        /* responder */

1672        if (node_dhc->nlp_reauth_status == NLP_HOST_REAUTH_IN_PROGRESS) {
1673                if (node_dhc->nlp_auth_tranid_rsp == 0xffffffff) {
1674                        node_dhc->nlp_auth_tranid_rsp = 1;
1675                } else {
1676                        node_dhc->nlp_auth_tranid_rsp++;
1677                }
1678        } else {        /* !NLP_HOST_REAUTH_IN_PROGRESS */
1679                node_dhc->nlp_auth_tranid_rsp = 0;
1680        }

1682        if (flag == 1) {
1683                if (hash_wcnt == 1) {
1684                        ap1->tran_id =
1685                            LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);

1687                        ap1->params.name_tag = AUTH_NAME_ID;
1688                        ap1->params.name_len = AUTH_NAME_LEN;
1689                        bcopy((void *)&port->wwpn,
1690                            (void *) &ap1->params.nodeName, sizeof (NAME_TYPE));
1691                        ap1->params.proto_num = AUTH_PROTO_NUM;
1692                        ap1->params.para_len = LE_SWAP32(para_len);
1693                        ap1->params.proto_id = AUTH_DHCHAP;
1694                        ap1->params.HashList_tag = HASH_LIST_TAG;
1695                        ap1->params.HashList_wcnt = LE_SWAP16(hash_wcnt);
1696                        ap1->params.HashList_value1 =
1697                            node_dhc->auth_cfg.hash_priority[0];
1698                        ap1->params.DHgIDList_tag = DHGID_LIST_TAG;
1699                        ap1->params.DHgIDList_wnt = LE_SWAP16(dhgp_wcnt);

1701                        switch (dhgp_wcnt) {
1702                        case 5:
1703                                ap1->params.DHgIDList_g4 =
1704                                    (node_dhc->auth_cfg.dh_group_priority[4]);
1705                                ap1->params.DHgIDList_g3 =
1706                                    (node_dhc->auth_cfg.dh_group_priority[3]);
1707                                ap1->params.DHgIDList_g2 =
1708                                    (node_dhc->auth_cfg.dh_group_priority[2]);
1709                                ap1->params.DHgIDList_g1 =
1710                                    (node_dhc->auth_cfg.dh_group_priority[1]);
1711                                ap1->params.DHgIDList_g0 =
```

```
1712                                    (node_dhc->auth_cfg.dh_group_priority[0]);
1713                            break;
1714                    case 4:
1715                            ap1->params.DHgIDList_g3 =
1716                                (node_dhc->auth_cfg.dh_group_priority[3]);
1717                            ap1->params.DHgIDList_g2 =
1718                                (node_dhc->auth_cfg.dh_group_priority[2]);
1719                            ap1->params.DHgIDList_g1 =
1720                                (node_dhc->auth_cfg.dh_group_priority[1]);
1721                            ap1->params.DHgIDList_g0 =
1722                                (node_dhc->auth_cfg.dh_group_priority[0]);
1723                            break;
1724                    case 3:
1725                            ap1->params.DHgIDList_g2 =
1726                                (node_dhc->auth_cfg.dh_group_priority[2]);
1727                            ap1->params.DHgIDList_g1 =
1728                                (node_dhc->auth_cfg.dh_group_priority[1]);
1729                            ap1->params.DHgIDList_g0 =
1730                                (node_dhc->auth_cfg.dh_group_priority[0]);
1731                            break;
1732                    case 2:
1733                            ap1->params.DHgIDList_g1 =
1734                                (node_dhc->auth_cfg.dh_group_priority[1]);
1735                            ap1->params.DHgIDList_g0 =
1736                                (node_dhc->auth_cfg.dh_group_priority[0]);
1737                            break;
1738                    case 1:
1739                            ap1->params.DHgIDList_g0 =
1740                                (node_dhc->auth_cfg.dh_group_priority[0]);
1741                            break;
1742                    }
1743            } else {
1744                    ap2->tran_id =
1745                        LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);

1747                    ap2->params.name_tag = AUTH_NAME_ID;
1748                    ap2->params.name_len = AUTH_NAME_LEN;
1749                    bcopy((void *) &port->wwpn,
1750                        (void *) &ap2->params.nodeName, sizeof (NAME_TYPE));
1751                    ap2->params.proto_num = AUTH_PROTO_NUM;
1752                    ap2->params.para_len = LE_SWAP32(para_len);
1753                    ap2->params.proto_id = AUTH_DHCHAP;
1754                    ap2->params.HashList_tag = HASH_LIST_TAG;
1755                    ap2->params.HashList_wcnt = LE_SWAP16(hash_wcnt);
1756                    ap2->params.HashList_value1 =
1757                        (node_dhc->auth_cfg.hash_priority[0]);
1758                    ap2->params.HashList_value2 =
1759                        (node_dhc->auth_cfg.hash_priority[1]);

1761                    ap2->params.DHgIDList_tag = DHGID_LIST_TAG;
1762                    ap2->params.DHgIDList_wnt = LE_SWAP16(dhgp_wcnt);

1764                    switch (dhgp_wcnt) {
1765                    case 5:
1766                            ap2->params.DHgIDList_g4 =
1767                                (node_dhc->auth_cfg.dh_group_priority[4]);
1768                            ap2->params.DHgIDList_g3 =
1769                                (node_dhc->auth_cfg.dh_group_priority[3]);
1770                            ap2->params.DHgIDList_g2 =
1771                                (node_dhc->auth_cfg.dh_group_priority[2]);
1772                            ap2->params.DHgIDList_g1 =
1773                                (node_dhc->auth_cfg.dh_group_priority[1]);
1774                            ap2->params.DHgIDList_g0 =
1775                                (node_dhc->auth_cfg.dh_group_priority[0]);
1776                            break;
1777                    case 4:
```



```
1778                            ap2->params.DHgIDList_g3 =
1779                                (node_dhc->auth_cfg.dh_group_priority[3]);
1780                            ap2->params.DHgIDList_g2 =
1781                                (node_dhc->auth_cfg.dh_group_priority[2]);
1782                            ap2->params.DHgIDList_g1 =
1783                                (node_dhc->auth_cfg.dh_group_priority[1]);
1784                            ap2->params.DHgIDList_g0 =
1785                                (node_dhc->auth_cfg.dh_group_priority[0]);
1786                            break;
1787                    case 3:
1788                            ap2->params.DHgIDList_g2 =
1789                                (node_dhc->auth_cfg.dh_group_priority[2]);
1790                            ap2->params.DHgIDList_g1 =
1791                                (node_dhc->auth_cfg.dh_group_priority[1]);
1792                            ap2->params.DHgIDList_g0 =
1793                                (node_dhc->auth_cfg.dh_group_priority[0]);
1794                            break;
1795                    case 2:
1796                            ap2->params.DHgIDList_g1 =
1797                                (node_dhc->auth_cfg.dh_group_priority[1]);
1798                            ap2->params.DHgIDList_g0 =
1799                                (node_dhc->auth_cfg.dh_group_priority[0]);
1800                            break;
1801                    case 1:
1802                            ap2->params.DHgIDList_g0 =
1803                                (node_dhc->auth_cfg.dh_group_priority[0]);
1804                            break;
1805                    }
1806            }
1807    } else {
1808            if (num_hs == 1) {
1809                    null_ap1->tran_id =
1810                        LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);

1812                    null_ap1->params.name_tag = AUTH_NAME_ID;
1813                    null_ap1->params.name_len = AUTH_NAME_LEN;
1814                    bcopy((void *) &port->wwpn,
1815                        (void *) &null_ap1->params.nodeName,
1816                        sizeof (NAME_TYPE));
1817                    null_ap1->params.proto_num = AUTH_PROTO_NUM;
1818                    null_ap1->params.para_len = LE_SWAP32(0x00000014);
1819                    null_ap1->params.proto_id = AUTH_DHCHAP;
1820                    null_ap1->params.HashList_tag = HASH_LIST_TAG;
1821                    null_ap1->params.HashList_wcnt = LE_SWAP16(0x0001);
1822                    null_ap1->params.HashList_value1 =
1823                        (node_dhc->auth_cfg.hash_priority[0]);
1824                    null_ap1->params.DHgIDList_tag = DHGID_LIST_TAG;
1825                    null_ap1->params.DHgIDList_wnt = LE_SWAP16(0x0001);
1826                    null_ap1->params.DHgIDList_g0 = 0x0;
1827            } else {
1828                    null_ap2->tran_id =
1829                        LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);

1831                    null_ap2->params.name_tag = AUTH_NAME_ID;
1832                    null_ap2->params.name_len = AUTH_NAME_LEN;
1833                    bcopy((void *) &port->wwpn,
1834                        (void *) &null_ap2->params.nodeName,
1835                        sizeof (NAME_TYPE));
1836                    null_ap2->params.proto_num = AUTH_PROTO_NUM;
1837                    null_ap2->params.para_len = LE_SWAP32(0x00000018);
1838                    null_ap2->params.proto_id = AUTH_DHCHAP;

1840                    null_ap2->params.HashList_tag = HASH_LIST_TAG;
1841                    null_ap2->params.HashList_wcnt = LE_SWAP16(0x0002);
1842                    null_ap2->params.HashList_value1 =
1843                        (node_dhc->auth_cfg.hash_priority[0]);
```

```
1844                        null_ap2->params.HashList_value2 =
1845                            (node_dhc->auth_cfg.hash_priority[1]);
1847                        null_ap2->params.DHgIDList_tag = DHGID_LIST_TAG;
1848                        null_ap2->params.DHgIDList_wnt = LE_SWAP16(0x0001);
1849                        null_ap2->params.DHgIDList_g0 = 0x0;
1850                }
1851        }
1853        pkt->pkt_comp = emlxs_cmpl_auth_negotiate_issue;
1855        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
1856            "issue_auth_negotiate: %x flag=%d size=%d hash=%x,%x tid=%x,%x",
1857            ndlp->nlp_DID, flag, cmd_size,
1858            node_dhc->auth_cfg.hash_priority[0],
1859            node_dhc->auth_cfg.hash_priority[1],
1860            node_dhc->nlp_auth_tranid_rsp, node_dhc->nlp_auth_tranid_ini);
1862        if (emlxs_pkt_send(pkt, 1) != FC_SUCCESS) {
1863                emlxs_pkt_free(pkt);
1865                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
1866                    "issue_auth_negotiate: Unable to send pkt. did=0x%x",
1867                    ndlp->nlp_DID);
1869                return (1);
1870        }
1871        return (0);
1873 } /* emlxs_issue_auth_negotiate() */
1877 /*
1878  * ! emlxs_cmpl_auth_negotiate_issue
1879  *
1880  * \pre \post \param   phba \param   cmdiocb \param   rspiocb \return  void
1881  *
1882  * \b Description: iocb_cmpl callback function.
1883  *
1884  */
1885 static void
1886 emlxs_cmpl_auth_negotiate_issue(fc_packet_t *pkt)
1887 {
1888        emlxs_port_t *port = pkt->pkt_ulp_private;
1889        emlxs_buf_t *sbp;
1890        NODELIST *ndlp;
1891        emlxs_node_dhc_t *node_dhc;
1892        uint32_t did;
1894        did = pkt->pkt_cmd_fhdr.d_id;
1895        sbp = (emlxs_buf_t *)pkt->pkt_fca_private;
1896        ndlp = sbp->node;
1897        node_dhc = &ndlp->node_dhc;
1899        if (!ndlp) {
1900                ndlp = emlxs_node_find_did(port, did);
1901        }
1902        if (pkt->pkt_state != FC_PKT_SUCCESS) {
1903                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1904                    "emlxs_cmpl_dhchap_negotiate_issue: 0x%x %x. Noretry.",
1905                    did, pkt->pkt_state);
1906        } else {
1907                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1908                    "emlxs_cmpl_dhchap_negotiate_issue: did=0x%x. Succcess.",
1909                    did);
```



```
1910        }
1912        if (ndlp) {
1913                if (pkt->pkt_state == FC_PKT_SUCCESS) {
1914                        (void) emlxs_dhchap_state_machine(port, NULL, NULL,
1915                            NULL, ndlp, NODE_EVENT_CMPL_AUTH_MSG);
1916                } else {
1917                        emlxs_dhc_set_reauth_time(port, ndlp, DISABLE);
1919                        emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED,
1920                            0, 0);
1922                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
1923                            "Reauth disabled. did=0x%x state=%x",
1924                            ndlp->nlp_DID, node_dhc->state);
1926                        emlxs_dhc_auth_complete(port, ndlp, 1);
1927                }
1928        }
1929        emlxs_pkt_free(pkt);
1931        return;
1933 } /* emlxs_cmpl_auth_negotiate_issue */
1936 /*
1937  * ! emlxs_cmpl_auth_msg_auth_negotiate_issue
1938  *
1939  * \pre \post \param   port \param   CHANNEL * rp \param   arg \param   evt
1940  * \return  uint32_t \b Description:
1941  *
1942  * This routine is invoked when the host receive the solicited ACC/RJT ELS
1943  * cmd from an NxPort or FxPort that has received the ELS
1944  * AUTH Negotiate msg from the host. in case of RJT, Auth_Negotiate should
1945  * be retried in emlxs_cmpl_auth_negotiate_issue
1946  * call. in case of ACC, the host must be the initiator because its current
1947  * state could be "AUTH_NEGOTIATE_RCV" if it is the
1948  * responder. Then the next stat = AUTH_NEGOTIATE_CMPL_WAIT4NEXT
1949  */
1950 /* ARGSUSED */
1951 static uint32_t
1952 emlxs_cmpl_auth_msg_auth_negotiate_issue(
1953 emlxs_port_t *port,
1954 /* CHANNEL * rp, */ void *arg1,
1955 /* IOCBQ * iocbq, */ void *arg2,
1956 /* MATCHMAP * mp, */ void *arg3,
1957 /* NODELIST * ndlp, */ void *arg4,
1958 uint32_t evt)
1959 {
1960        NODELIST *ndlp = (NODELIST *)arg4;
1961        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
1963        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
1964            "emlxs_cmpl_auth_msg_auth_negotiate_issue: did=0x%x",
1965            ndlp->nlp_DID);
1967        /* start the emlxs_dhc_authrsp_timeout timer */
1968        if (node_dhc->nlp_authrsp_tmo == 0) {
1969                node_dhc->nlp_authrsp_tmo = DRV_TIME +
1970                    node_dhc->auth_cfg.authentication_timeout;
1971        }
1972        /*
1973         * The next state should be
1974         * emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next
1975         */
```

```
1976         emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_NEGOTIATE_CMPL_WAIT4NEXT,
1977             0, 0);

1979         return (node_dhc->state);

1981 } /* emlxs_cmpl_auth_msg_auth_negotiate_issue */



1985 /*
1986  * ! emlxs_rcv_auth_msg_auth_negotiate_issue
1987  *
1988  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
1989  * uint32_t \b Description:
1990  *
1991  * This routine is supported for HBA in either auth initiator mode or
1992  * responder mode.
1993  *
1994  * This routine is invoked when the host receive an unsolicited ELS AUTH Msg
1995  * from an NxPort or FxPort to which the host has just
1996  * sent out an ELS AUTH negotiate msg. and the NxPort or FxPort also LS_ACC
1997  * to the host's AUTH_Negotiate msg.
1998  *
1999  * If this unsolicited ELS auth msg is from the FxPort or a NxPort with a
2000  * numerically lower WWPN, the host will be the winner in
2001  * this authentication transaction initiation phase, the host as the
2002  * initiator will send back ACC and then Auth_Reject message
2003  * with the Reason Code 'Logical Error' and Reason Code Explanation'
2004  * Authentication Transaction Already Started' and with the
2005  * current state unchanged and mark itself as auth_initiator.
2006  *
2007  * Otherwise, the host will be the responder that will reply to the received
2008  * AUTH_Negotiate message will ACC (or RJT?) and abort
2009  * its own transaction upon receipt of the AUTH_Reject message. The new state
2010  * will be "AUTH_NEGOTIATE_RCV" and mark the host as
2011  * auth_responder.
2012  */
2013 /* ARGSUSED */
2014 static uint32_t
2015 emlxs_rcv_auth_msg_auth_negotiate_issue(
2016 emlxs_port_t *port,
2017 /* CHANNEL * rp, */ void *arg1,
2018 /* IOCBQ * iocbq, */ void *arg2,
2019 /* MATCHMAP * mp, */ void *arg3,
2020 /* NODELIST * ndlp */ void *arg4,
2021 uint32_t evt)
2022 {
2023         NODELIST *ndlp = (NODELIST *)arg4;
2024         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
2025         IOCBQ *iocbq = (IOCBQ *) arg2;
2026         uint8_t ReasonCode;
2027         uint8_t ReasonCodeExplanation;

2029         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
2030             "emlxs_rcv_auth_msg_auth_negotiate_issue: did=0x%x",
2031             ndlp->nlp_DID);

2033         /* Anyway we accept it first and then send auth_reject */
2034         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

2036         /* host is always the initiator and it should win */
2037         ReasonCode = AUTHRJT_LOGIC_ERR;
2038         ReasonCodeExplanation = AUTHEXP_AUTHTRAN_STARTED;

2040         emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_NEGOTIATE_ISSUE,
2041             ReasonCode, ReasonCodeExplanation);
```



```
2042         (void) emlxs_issue_auth_reject(port, ndlp, 0, 0, ReasonCode,
2043             ReasonCodeExplanation);

2045         return (node_dhc->state);

2047 } /* emlxs_rcv_auth_msg_auth_negotiate_issue */



2050 /*
2051  * ! emlxs_cmpl_dhchap_reply_issue
2052  *
2053  * \pre \post \param   phba \param   cmdiocb \param   rspiocb \return  void
2054  *
2055  * \b Description: iocb_cmpl callback function.
2056  *
2057  */
2058 static void
2059 emlxs_cmpl_dhchap_reply_issue(fc_packet_t *pkt)
2060 {
2061         emlxs_port_t *port = pkt->pkt_ulp_private;
2062         emlxs_buf_t *sbp;
2063         NODELIST *ndlp;
2064         uint32_t did;

2066         did = pkt->pkt_cmd_fhdr.d_id;
2067         sbp = (emlxs_buf_t *)pkt->pkt_fca_private;
2068         ndlp = sbp->node;

2070         if (!ndlp) {
2071                 ndlp = emlxs_node_find_did(port, did);
2072         }
2073         if (pkt->pkt_state != FC_PKT_SUCCESS) {
2074                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
2075                     "emlxs_cmpl_dhchap_reply_issue: 0x%x %x. No retry.",
2076                     did, pkt->pkt_state);
2077         } else {
2078                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
2079                     "emlxs_cmpl_dhchap_reply_issue: did=0x%x. Succcess.",
2080                     did);
2081         }

2083         if (ndlp) {
2084                 if (pkt->pkt_state == FC_PKT_SUCCESS) {
2085                         (void) emlxs_dhchap_state_machine(port, NULL, NULL,
2086                             NULL, ndlp, NODE_EVENT_CMPL_AUTH_MSG);
2087                 }
2088         }
2089         emlxs_pkt_free(pkt);

2091         return;

2093 } /* emlxs_cmpl_dhchap_reply_issue */



2096 /*
2097  * arg: the AUTH_Negotiate payload from the initiator. payload_len: the
2098  * payload length
2099  *
2100  * We always send out the challenge parameter based on our preference
2101  * order configured on the host side no matter what perference
2102  * order looks like from auth_negotiate . In other words, if the host issue
2103  * the challenge the host will make the decision as to
2104  * what hash function, what dhgp_id is to be used.
2105  *
2106  * This challenge value should not be confused with the challenge value for
2107  * bi-dir as part of reply when host is the initiator.
```

```
 */
/* ARGSUSED */
uint32_t
emlxs_issue_dhchap_challenge(
        emlxs_port_t *port,
        NODELIST *ndlp,
        int retry,
        void *arg,
        uint32_t payload_len,
        uint32_t hash_id,
        uint32_t dhgp_id)
{
        emlxs_hba_t *hba = HBA;
        fc_packet_t *pkt;
        uint32_t cmd_size;
        uint32_t rsp_size;
        uint16_t cmdsize = 0;
        uint8_t *pCmd;
        emlxs_port_dhc_t *port_dhc = &port->port_dhc;
        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
        DHCHAP_CHALL *chal;
        uint8_t *tmp;
        uint8_t random_number[20];
        uint8_t dhval[256];
        uint32_t dhval_len;
        uint32_t tran_id;
        BIG_ERR_CODE err = BIG_OK;

        /*
         * we assume the HBAnyware should configure the driver the right
         * parameters for challenge. for now, we create our own challenge.
         */
        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
            "emlxs_issue_dhchap_challenge: did=0x%x hashlist=[%x,%x,%x,%x]",
            ndlp->nlp_DID, node_dhc->auth_cfg.hash_priority[0],
            node_dhc->auth_cfg.hash_priority[1],
            node_dhc->auth_cfg.hash_priority[2],
            node_dhc->auth_cfg.hash_priority[3]);

        /*
         * Here is my own challenge structure:
         *
         * 1: AUTH_MSG_HDR (12 bytes + 4 bytes + 8 bytes) 2: hasd_id (4
         * bytes) 3: dhgp_id (4 bytes) 4: cval_len (4 bytes) 5: cval
         * (20 bytes or 16 bytes: cval_len bytes) 6: dhval_len (4 bytes)
         * 7: dhval (dhval_len bytes) all these information should be stored
         * in port_dhc struct
         */
        if (hash_id == AUTH_SHA1) {
                cmdsize = (12 + 4 + 8) + (4 + 4 + 4) + 20 + 4;
        } else if (hash_id == AUTH_MD5) {
                cmdsize = (12 + 4 + 8) + (4 + 4 + 4) + 16 + 4;
        } else {
                return (1);
        }


        switch (dhgp_id) {
        case GROUP_NULL:
                break;

        case GROUP_1024:
                cmdsize += 128;
                break;

        case GROUP_1280:
```



```
                cmdsize += 160;
                break;

        case GROUP_1536:
                cmdsize += 192;
                break;

        case GROUP_2048:
                cmdsize += 256;
                break;

        default:
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_issue_dhchap_challenge: Invalid dhgp_id=0x%x",
                    dhgp_id);
                return (1);
        }

        cmd_size = cmdsize;
        rsp_size = 4;

        if ((pkt = emlxs_prep_els_fc_pkt(port, ndlp->nlp_DID, cmd_size,
            rsp_size,
            0, KM_NOSLEEP)) == NULL) {
                return (1);
        }
        pCmd = (uint8_t *)pkt->pkt_cmd;

        tmp = (uint8_t *)arg;
        tmp += 8;
        /* collect tran_id: this tran_id is set by the initiator */
        tran_id = *(uint32_t *)tmp;

        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
            "emlxs_issue_dhchap_challenge: 0x%x 0x%x 0x%x 0x%x %d 0x%x 0x%x 0x%x",
            ndlp->nlp_DID, node_dhc->nlp_auth_tranid_ini,
            node_dhc->nlp_auth_tranid_rsp,
            cmdsize, tran_id, hash_id, dhgp_id);

        /* store the tran_id : ndlp is the initiator */
        node_dhc->nlp_auth_tranid_ini = LE_SWAP32(tran_id);

        tmp += sizeof (uint32_t);

        chal = (DHCHAP_CHALL *)pCmd;
        chal->cnul.msg_hdr.auth_els_code = ELS_CMD_AUTH_CODE;
        chal->cnul.msg_hdr.auth_els_flags = 0x0;
        chal->cnul.msg_hdr.auth_msg_code = DHCHAP_CHALLENGE;
        chal->cnul.msg_hdr.proto_version = 0x01;
        chal->cnul.msg_hdr.msg_len = LE_SWAP32(cmdsize - 12);
        chal->cnul.msg_hdr.tran_id = tran_id;
        chal->cnul.msg_hdr.name_tag = (AUTH_NAME_ID);
        chal->cnul.msg_hdr.name_len = (AUTH_NAME_LEN);

        bcopy((void *) &port->wwpn,
            (void *) &chal->cnul.msg_hdr.nodeName, sizeof (NAME_TYPE));

        chal->cnul.hash_id = hash_id;
        chal->cnul.dhgp_id = dhgp_id;

        chal->cnul.cval_len = ((chal->cnul.hash_id == AUTH_SHA1) ?
            LE_SWAP32(SHA1_LEN) : LE_SWAP32(MD5_LEN));

        tmp = (uint8_t *)pCmd;
        tmp += sizeof (DHCHAP_CHALL_NULL);
```

```
#ifdef RAND
        /* generate a random number as the challenge */
        bzero(random_number, LE_SWAP32(chal->cnul.cval_len));

        if (hba->rdn_flag == 1) {
                emlxs_get_random_bytes(ndlp, random_number, 20);
        } else {
                (void) random_get_pseudo_bytes(random_number,
                    LE_SWAP32(chal->cnul.cval_len));
        }

        /*
         * the host should store the challenge for later usage when later on
         * host get the reply msg, host needs to verify it by using its old
         * challenge, its private key as the input to the hash function. the
         * challenge as the random_number should be stored in
         * node_dhc->hrsp_cval[]
         */
        if (ndlp->nlp_DID == FABRIC_DID) {
                bcopy((void *) &random_number[0],
                    (void *) &node_dhc->hrsp_cval[0],
                    LE_SWAP32(chal->cnul.cval_len));
                /* save another copy in partner's ndlp */
                bcopy((void *) &random_number[0],
                    (void *) &node_dhc->nlp_auth_misc.hrsp_cval[0],
                    LE_SWAP32(chal->cnul.cval_len));
        } else {
                bcopy((void *) &random_number[0],
                    (void *) &node_dhc->nlp_auth_misc.hrsp_cval[0],
                    LE_SWAP32(chal->cnul.cval_len));
        }
        bcopy((void *) &random_number[0], (void *) tmp,
            LE_SWAP32(chal->cnul.cval_len));

#endif  /* RAND */

        /* for test only hardcode the challenge value */
#ifdef MYRAND
        if (ndlp->nlp_DID == FABRIC_DID) {
                bcopy((void *) myrand, (void *) &node_dhc->hrsp_cval[0],
                    LE_SWAP32(chal->cnul.cval_len));
                /* save another copy in partner's ndlp */
                bcopy((void *) myrand,
                    (void *) &node_dhc->nlp_auth_misc.hrsp_cval[0],
                    LE_SWAP32(chal->cnul.cval_len));
        } else {
                bcopy((void *) myrand,
                    (void *) &node_dhc->nlp_auth_misc.hrsp_cval[0],
                    LE_SWAP32(chal->cnul.cval_len));
        }
        bcopy((void *) myrand, (void *) tmp,
            LE_SWAP32(chal->cnul.cval_len));

#endif  /* MYRAND */

        if (ndlp->nlp_DID == FABRIC_DID) {
                node_dhc->hrsp_cval_len = LE_SWAP32(chal->cnul.cval_len);
                node_dhc->nlp_auth_misc.hrsp_cval_len =
                    LE_SWAP32(chal->cnul.cval_len);
        } else {
                node_dhc->nlp_auth_misc.hrsp_cval_len =
                    LE_SWAP32(chal->cnul.cval_len);
        }

        tmp += LE_SWAP32(chal->cnul.cval_len);
```



```
        /*
         * we need another random number as the private key x which will be
         * used to compute the public key i.e. g^x mod p we intentionally set
         * the length of private key as the same length of challenge. we have
         * to store the private key in node_dhc->hrsp_priv_key[20].
         */
#ifdef RAND

        if (dhgp_id != GROUP_NULL) {

                bzero(random_number, LE_SWAP32(chal->cnul.cval_len));

                if (hba->rdn_flag == 1) {
                        emlxs_get_random_bytes(ndlp, random_number, 20);
                } else {
                        (void) random_get_pseudo_bytes(random_number,
                            LE_SWAP32(chal->cnul.cval_len));
                }

                if (ndlp->nlp_DID == FABRIC_DID) {
                        bcopy((void *) &random_number[0],
                            (void *) node_dhc->hrsp_priv_key,
                            LE_SWAP32(chal->cnul.cval_len));
                        bcopy((void *) &random_number[0],
                            (void *) node_dhc->nlp_auth_misc.hrsp_priv_key,
                            LE_SWAP32(chal->cnul.cval_len));
                } else {
                        bcopy((void *) &random_number[0],
                            (void *) node_dhc->nlp_auth_misc.hrsp_priv_key,
                            LE_SWAP32(chal->cnul.cval_len));
                }
        }
#endif  /* RAND */

#ifdef MYRAND
        if (dhgp_id != GROUP_NULL) {
                /* For test only we hardcode the priv_key here */
                bcopy((void *) myrand, (void *) node_dhc->hrsp_priv_key,
                    LE_SWAP32(chal->cnul.cval_len));

                if (ndlp->nlp_DID == FABRIC_DID) {
                        bcopy((void *) myrand,
                            (void *) node_dhc->hrsp_priv_key,
                            LE_SWAP32(chal->cnul.cval_len));
                        bcopy((void *) myrand,
                            (void *) node_dhc->nlp_auth_misc.hrsp_priv_key,
                            LE_SWAP32(chal->cnul.cval_len));
                } else {
                        bcopy((void *) myrand,
                            (void *) node_dhc->nlp_auth_misc.hrsp_priv_key,
                            LE_SWAP32(chal->cnul.cval_len));
                }
        }
#endif  /* MYRAND */

        /* also store the hash function and dhgp_id being used in challenge. */
        /* These information could be configurable through HBAnyware */
        node_dhc->nlp_auth_hashid = hash_id;
        node_dhc->nlp_auth_dhgpid = dhgp_id;

        /*
         * generate the DH value DH value is g^x mod p  and it is also called
         * public key in which g is 2, x is the random number ontained above.
         * p is the dhgp3_pVal
         */
```

```
#ifdef MYRAND

        /* to get (g^x mod p) with x private key */
        if (dhgp_id != GROUP_NULL) {

                err = emlxs_BIGNUM_get_dhval(port, port_dhc, ndlp, dhval,
                    &dhval_len, chal->cnul.dhgp_id,
                    myrand, LE_SWAP32(chal->cnul.cval_len));

                if (err != BIG_OK) {
                        emlxs_pkt_free(pkt);

                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                            "emlxs_issue_dhchap_challenge: error. 0x%x",
                            err);

                        return (1);
                }
                /* we are not going to use dhval and dhval_len */

                /* *(uint32_t *)tmp = dhval_len; */
                if (ndlp->nlp_DID == FABRIC_DID) {
                        *(uint32_t *)tmp =
                            LE_SWAP32(node_dhc->hrsp_pubkey_len);
                } else {
                        *(uint32_t *)tmp =
                            LE_SWAP32(
                            node_dhc->nlp_auth_misc.hrsp_pubkey_len);
                }

                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
                    "emlxs_issue_dhchap_challenge: 0x%x: 0x%x 0x%x",
                    ndlp->nlp_DID, *(uint32_t *)tmp, dhval_len);

                tmp += sizeof (uint32_t);

                if (ndlp->nlp_DID == FABRIC_DID) {
                        bcopy((void *) node_dhc->hrsp_pub_key, (void *)tmp,
                            node_dhc->hrsp_pubkey_len);
                } else {
                        bcopy((void *) node_dhc->nlp_auth_misc.hrsp_pub_key,
                            (void *)tmp,
                            node_dhc->nlp_auth_misc.hrsp_pubkey_len);
                }
        } else {
                /* NULL DHCHAP */
                *(uint32_t *)tmp = 0;
        }

#endif  /* MYRAND */

#ifdef RAND

        /* to get (g^x mod p) with x private key */
        if (dhgp_id != GROUP_NULL) {

                err = emlxs_BIGNUM_get_dhval(port, port_dhc, ndlp, dhval,
                    &dhval_len, chal->cnul.dhgp_id,
                    random_number, LE_SWAP32(chal->cnul.cval_len));

                if (err != BIG_OK) {
                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                            "emlxs_issue_dhchap_challenge: error. 0x%x",
                            err);

                        emlxs_pkt_free(pkt);
```



```
                        return (1);
                }
                /* we are not going to use dhval and dhval_len */

                /* *(uint32_t *)tmp = dhval_len; */
                if (ndlp->nlp_DID == FABRIC_DID) {
                        *(uint32_t *)tmp =
                            LE_SWAP32(node_dhc->hrsp_pubkey_len);
                } else {
                        *(uint32_t *)tmp =
                            LE_SWAP32(
                            node_dhc->nlp_auth_misc.hrsp_pubkey_len);
                }

                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
                    "emlxs_issue_dhchap_challenge: did=0x%x: pubkey_len=0x%x",
                    ndlp->nlp_DID, *(uint32_t *)tmp);

                tmp += sizeof (uint32_t);

                if (ndlp->nlp_DID == FABRIC_DID) {
                        bcopy((void *) node_dhc->hrsp_pub_key, (void *)tmp,
                            node_dhc->hrsp_pubkey_len);
                } else {
                        bcopy((void *) node_dhc->nlp_auth_misc.hrsp_pub_key,
                            (void *)tmp,
                            node_dhc->nlp_auth_misc.hrsp_pubkey_len);
                }
        } else {
                /* NULL DHCHAP */
                *(uint32_t *)tmp = 0;
        }

#endif  /* RAND */

        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
            "emlxs_issue_dhchap_challenge: 0x%x 0x%x 0x%x 0x%x 0x%x",
            ndlp->nlp_DID, node_dhc->nlp_auth_tranid_ini,
            node_dhc->nlp_auth_tranid_rsp,
            chal->cnul.hash_id, chal->cnul.dhgp_id);

        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
            "emlxs_issue_dhchap_challenge: 0x%x 0x%x 0x%x 0x%x",
            ndlp->nlp_DID, tran_id, node_dhc->nlp_auth_hashid,
            node_dhc->nlp_auth_dhgpid);

        pkt->pkt_comp = emlxs_cmpl_dhchap_challenge_issue;

        if (emlxs_pkt_send(pkt, 1) != FC_SUCCESS) {
                emlxs_pkt_free(pkt);

                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_issue_dhchap_challenge: Unable to send fc packet.");

                return (1);
        }
        return (0);

} /* emlxs_issue_dhchap_challenge */


/*
 * DHCHAP_Reply msg
 */
/* ARGSUSED */
uint32_t
```

```
2504 emlxs_issue_dhchap_reply(
2505         emlxs_port_t *port,
2506         NODELIST *ndlp,
2507         int retry,
2508         uint32_t *arg1, /* response */
2509         uint8_t *dhval,
2510         uint32_t dhval_len,
2511         uint8_t *arg2,  /* random number */
2512         uint32_t arg2_len)
2513 {
2514         fc_packet_t *pkt;
2515         uint32_t cmd_size;
2516         uint32_t rsp_size;
2517         uint16_t cmdsize = 0;
2518         DHCHAP_REPLY_HDR *ap;
2519         uint8_t *pCmd;
2520         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

2522         /* Header size */
2523         cmdsize = sizeof (DHCHAP_REPLY_HDR);

2525         /* Rsp value len size (4) + Response value size */
2526         if (ndlp->nlp_DID == FABRIC_DID) {
2527                 if (node_dhc->hash_id == AUTH_MD5) {
2528                         cmdsize += 4 + MD5_LEN;
2529                 }
2530                 if (node_dhc->hash_id == AUTH_SHA1) {
2531                         cmdsize += 4 + SHA1_LEN;
2532                 }
2533         } else {
2534                 if (node_dhc->nlp_auth_hashid == AUTH_MD5) {
2535                         cmdsize += 4 + MD5_LEN;
2536                 }
2537                 if (node_dhc->nlp_auth_hashid == AUTH_SHA1) {
2538                         cmdsize += 4 + SHA1_LEN;
2539                 }
2540         }

2542         /* DH value len size (4) + DH value size */
2543         if (ndlp->nlp_DID == FABRIC_DID) {
2544                 switch (node_dhc->dhgp_id) {
2545                 case GROUP_NULL:

2547                         break;

2549                 case GROUP_1024:
2550                 case GROUP_1280:
2551                 case GROUP_1536:
2552                 case GROUP_2048:
2553                 default:
2554                         break;
2555                 }
2556         }

2558         cmdsize += 4 + dhval_len;

2560         /* Challenge value len size (4) + Challenge value size */
2561         if (node_dhc->auth_cfg.bidirectional == 0) {
2562                 cmdsize += 4;
2563         } else {
2564                 if (ndlp->nlp_DID == FABRIC_DID) {
2565                         cmdsize += 4 + ((node_dhc->hash_id == AUTH_MD5) ?
2566                             MD5_LEN : SHA1_LEN);
2567                 } else {
2568                         cmdsize += 4 +
2569                             ((node_dhc->nlp_auth_hashid == AUTH_MD5) ? MD5_LEN :
```



```
2570                             SHA1_LEN);
2571                 }
2572         }

2574         cmd_size = cmdsize;
2575         rsp_size = 4;

2577         if ((pkt = emlxs_prep_els_fc_pkt(port, ndlp->nlp_DID, cmd_size,
2578             rsp_size, 0, KM_NOSLEEP)) == NULL) {
2579                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2580                     "emlxs_issue_dhchap_reply failed: did=0x%x size=%x,%x",
2581                     ndlp->nlp_DID, cmd_size, rsp_size);

2583                 return (1);
2584         }
2585         pCmd = (uint8_t *)pkt->pkt_cmd;

2587         ap = (DHCHAP_REPLY_HDR *)pCmd;
2588         ap->auth_els_code = ELS_CMD_AUTH_CODE;
2589         ap->auth_els_flags = 0x0;
2590         ap->auth_msg_code = DHCHAP_REPLY;
2591         ap->proto_version = 0x01;
2592         ap->msg_len = LE_SWAP32(cmdsize - sizeof (DHCHAP_REPLY_HDR));
2593         ap->tran_id = LE_SWAP32(node_dhc->nlp_auth_tranid_rsp);

2595         pCmd = (uint8_t *)(pCmd + sizeof (DHCHAP_REPLY_HDR));

2597         if (ndlp->nlp_DID == FABRIC_DID) {
2598                 if (node_dhc->hash_id == AUTH_MD5) {
2599                         *(uint32_t *)pCmd = LE_SWAP32(MD5_LEN);
2600                 } else {
2601                         *(uint32_t *)pCmd = LE_SWAP32(SHA1_LEN);
2602                 }
2603         } else {
2604                 if (node_dhc->nlp_auth_hashid == AUTH_MD5) {
2605                         *(uint32_t *)pCmd = LE_SWAP32(MD5_LEN);
2606                 } else {
2607                         *(uint32_t *)pCmd = LE_SWAP32(SHA1_LEN);
2608                 }
2609         }

2611         pCmd = (uint8_t *)(pCmd + 4);

2613         if (ndlp->nlp_DID == FABRIC_DID) {
2614                 if (node_dhc->hash_id == AUTH_MD5) {
2615                         bcopy((void *)arg1, pCmd, MD5_LEN);
2616                         pCmd = (uint8_t *)(pCmd + MD5_LEN);
2617                 } else {
2618                         bcopy((void *)arg1, (void *)pCmd, SHA1_LEN);

2620                         pCmd = (uint8_t *)(pCmd + SHA1_LEN);
2621                 }
2622         } else {
2623                 if (node_dhc->nlp_auth_hashid == AUTH_MD5) {
2624                         bcopy((void *)arg1, pCmd, MD5_LEN);
2625                         pCmd = (uint8_t *)(pCmd + MD5_LEN);
2626                 } else {
2627                         bcopy((void *)arg1, (void *)pCmd, SHA1_LEN);
2628                         pCmd = (uint8_t *)(pCmd + SHA1_LEN);
2629                 }
2630         }

2632         *(uint32_t *)pCmd = LE_SWAP32(dhval_len);

2634         if (dhval_len != 0) {
2635                 pCmd = (uint8_t *)(pCmd + 4);
```

```
2637                switch (node_dhc->dhgp_id) {
2638                case GROUP_NULL:

2640                        break;

2642                case GROUP_1024:
2643                case GROUP_1280:
2644                case GROUP_1536:
2645                case GROUP_2048:
2646                default:
2647                        break;
2648                }
2649                /* elx_bcopy((void *)dhval, (void *)pCmd, dhval_len); */
2650                /*
2651                 * The new DH parameter (g^y mod p) is stored in
2652                 * node_dhc->pub_key
2653                 */
2654                /* pubkey_len should be equal to dhval_len */

2656                if (ndlp->nlp_DID == FABRIC_DID) {
2657                        bcopy((void *) node_dhc->pub_key, (void *)pCmd,
2658                            node_dhc->pubkey_len);
2659                } else {
2660                        bcopy((void *) node_dhc->nlp_auth_misc.pub_key,
2661                            (void *)pCmd,
2662                            node_dhc->nlp_auth_misc.pubkey_len);
2663                }
2664                pCmd = (uint8_t *)(pCmd + dhval_len);
2665        } else
2666                pCmd = (uint8_t *)(pCmd + 4);

2668        if (node_dhc->auth_cfg.bidirectional == 0) {
2669                *(uint32_t *)pCmd = 0x0;
2670        } else {
2671                if (ndlp->nlp_DID == FABRIC_DID) {
2672                        if (node_dhc->hash_id == AUTH_MD5) {
2673                                *(uint32_t *)pCmd = LE_SWAP32(MD5_LEN);
2674                                pCmd = (uint8_t *)(pCmd + 4);
2675                                bcopy((void *)arg2, (void *)pCmd, arg2_len);
2676                        } else if (node_dhc->hash_id == AUTH_SHA1) {
2677                                *(uint32_t *)pCmd = LE_SWAP32(SHA1_LEN);
2678                                pCmd = (uint8_t *)(pCmd + 4);
2679                                /* store the challenge */
2680                                bcopy((void *)arg2, (void *)pCmd, arg2_len);
2681                        }
2682                } else {
2683                        if (node_dhc->nlp_auth_hashid == AUTH_MD5) {
2684                                *(uint32_t *)pCmd = LE_SWAP32(MD5_LEN);
2685                                pCmd = (uint8_t *)(pCmd + 4);
2686                                bcopy((void *)arg2, (void *)pCmd, arg2_len);
2687                        } else if (node_dhc->nlp_auth_hashid == AUTH_SHA1) {
2688                                *(uint32_t *)pCmd = LE_SWAP32(SHA1_LEN);
2689                                pCmd = (uint8_t *)(pCmd + 4);
2690                                bcopy((void *)arg2, (void *)pCmd, arg2_len);
2691                        }
2692                }
2693        }

2695        pkt->pkt_comp = emlxs_cmpl_dhchap_reply_issue;

2697        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
2698            "emlxs_issue_dhchap_reply: did=0x%x  (%x,%x,%x,%x,%x,%x)",
2699            ndlp->nlp_DID, dhval_len, arg2_len, cmdsize,
2700            node_dhc->hash_id, node_dhc->nlp_auth_hashid,
2701            LE_SWAP32(ap->tran_id));
```



```
2703        if (emlxs_pkt_send(pkt, 1) != FC_SUCCESS) {
2704                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2705                    "emlxs_issue_dhchap_reply failed: Unable to send packet.");

2707                emlxs_pkt_free(pkt);

2709                return (1);
2710        }
2711        return (0);

2713 } /* emlxs_issue_dhchap_reply */



2717 /*
2718  * ! emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next
2719  *
2720  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
2721  * uint32_t \b Description:
2722  *
2723  * This routine is invoked when the host received an unsolicted ELS AUTH MSG
2724  * from an NxPort or FxPort which already replied (ACC)
2725  * the ELS AUTH_Negotiate msg from the host. if msg is DHCHAP_Chellenge,
2726  * based on the msg content (DHCHAP computation etc.,)
2727  * the host send back ACC and 1. send back AUTH_Reject and set next state =
2728  * NPR_NODE or 2. send back DHCHAP_Reply msg and set
2729  * next state = DHCHAP_REPLY_ISSUE for bi-directional, the DHCHAP_Reply
2730  * includes challenge from host. for uni-directional, no
2731  * more challenge. if msg is AUTH_Reject or anything else, host send back
2732  * ACC and set next state = NPR_NODE. And based on the
2733  * reject code, host may need to retry negotiate with NULL DH only
2734  *
2735  * If the msg is AUTH_ELS cmd, cancel the nlp_authrsp_timeout timer immediately.
2736  *
2737  */
2738 /* ARGSUSED */
2739 static uint32_t
2740 emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next(
2741 emlxs_port_t *port,
2742 /* CHANNEL * rp, */ void *arg1,
2743 /* IOCBQ * iocbq, */ void *arg2,
2744 /* MATCHMAP * mp, */ void *arg3,
2745 /* NODELIST * ndlp */ void *arg4,
2746 uint32_t evt)
2747 {
2748        emlxs_hba_t *hba = HBA;
2749        emlxs_port_dhc_t *port_dhc = &port->port_dhc;
2750        IOCBQ *iocbq = (IOCBQ *)arg2;
2751        MATCHMAP *mp = (MATCHMAP *)arg3;
2752        NODELIST *ndlp = (NODELIST *)arg4;
2753        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
2754        uint8_t *bp;
2755        uint32_t *lp;
2756        DHCHAP_CHALL_NULL *ncval;
2757        uint16_t namelen;
2758        uint32_t dhvallen;
2759        uint8_t *tmp;
2760        uint8_t ReasonCode;
2761        uint8_t ReasonCodeExplanation;

2763        union challenge_val un_cval;

2765        uint8_t *dhval = NULL;
2766        uint8_t random_number[20];      /* for both SHA1 and MD5 */
2767        uint32_t *arg5 = NULL;  /* response */
```

```
2768         uint32_t tran_id;       /* Transaction Identifier */
2769         uint32_t arg2len = 0;   /* len of new challenge for bidir auth */

2771         AUTH_RJT *rjt;

2773         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
2774             "emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next: did=0x%x",
2775             ndlp->nlp_DID);

2777         emlxs_dhc_state(port, ndlp, NODE_STATE_DHCHAP_REPLY_ISSUE, 0, 0);

2779         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

2781         bp = mp->virt;
2782         lp = (uint32_t *)bp;

2784         /*
2785          * 1. we process the DHCHAP_Challenge 2. ACC it first 3. based on the
2786          * result of 1 we DHCHAP_Reply or AUTH_Reject
2787          */
2788         ncval = (DHCHAP_CHALL_NULL *)((uint8_t *)lp);

2790         if (ncval->msg_hdr.auth_els_code != ELS_CMD_AUTH_CODE) {
2791                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2792                     "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x %x",
2793                     ndlp->nlp_DID, ncval->msg_hdr.auth_els_code);

2795                 /* need to setup reason code/reason explanation code  */
2796                 ReasonCode = AUTHRJT_FAILURE;
2797                 ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;
2798                 goto AUTH_Reject;
2799         }
2800         if (ncval->msg_hdr.auth_msg_code == AUTH_REJECT) {
2801                 rjt = (AUTH_RJT *)((uint8_t *)lp);
2802                 ReasonCode = rjt->ReasonCode;
2803                 ReasonCodeExplanation = rjt->ReasonCodeExplanation;

2805                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
2806                     "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x.%x,%x",
2807                     ndlp->nlp_DID, ReasonCode, ReasonCodeExplanation);

2809                 switch (ReasonCode) {
2810                 case AUTHRJT_LOGIC_ERR:
2811                         switch (ReasonCodeExplanation) {
2812                         case AUTHEXP_MECH_UNUSABLE:
2813                         case AUTHEXP_DHGROUP_UNUSABLE:
2814                         case AUTHEXP_HASHFUNC_UNUSABLE:
2815                                 ReasonCode = AUTHRJT_LOGIC_ERR;
2816                                 ReasonCodeExplanation = AUTHEXP_RESTART_AUTH;
2817                                 break;

2819                         case AUTHEXP_RESTART_AUTH:
2820                                 /*
2821                                  * Cancel the rsp timer if not cancelled yet.
2822                                  * and restart auth tran now.
2823                                  */
2824                                 if (node_dhc->nlp_authrsp_tmo != 0) {
2825                                         node_dhc->nlp_authrsp_tmo = 0;
2826                                         node_dhc->nlp_authrsp_tmocnt = 0;
2827                                 }
2828                                 if (emlxs_dhc_auth_start(port, ndlp, NULL,
2829                                     NULL) != 0) {
2830                                         EMLXS_MSGF(EMLXS_CONTEXT,
2831                                             &emlxs_fcsp_debug_msg,
2832                                             "Reauth timeout. failed. 0x%x %x",
2833                                             ndlp->nlp_DID, node_dhc->state);
```



```
2834                                 }
2835                                 return (node_dhc->state);

2837                         default:
2838                                 ReasonCode = AUTHRJT_FAILURE;
2839                                 ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
2840                                 break;
2841                         }
2842                         break;

2844                 case AUTHRJT_FAILURE:
2845                 default:
2846                         ReasonCode = AUTHRJT_FAILURE;
2847                         ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
2848                         break;
2849                 }

2851                 goto AUTH_Reject;
2852         }
2853         if (ncval->msg_hdr.auth_msg_code != DHCHAP_CHALLENGE) {
2854                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2855                     "emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x.%x",
2856                     ndlp->nlp_DID, ncval->msg_hdr.auth_msg_code);

2858                 ReasonCode = AUTHRJT_FAILURE;
2859                 ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;
2860                 goto AUTH_Reject;
2861         }
2862         tran_id = ncval->msg_hdr.tran_id;

2864         if (LE_SWAP32(tran_id) != node_dhc->nlp_auth_tranid_rsp) {
2865                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2866                     "rcv_auth_msg_auth_negotiate_cmpl_wait4next:0x%x %x!=%x",
2867                     ndlp->nlp_DID, LE_SWAP32(tran_id),
2868                     node_dhc->nlp_auth_tranid_rsp);

2870                 ReasonCode = AUTHRJT_FAILURE;
2871                 ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
2872                 goto AUTH_Reject;
2873         }
2874         node_dhc->nlp_authrsp_tmo = 0;

2876         namelen = ncval->msg_hdr.name_len;

2878         if (namelen == AUTH_NAME_LEN) {
2879                 /*
2880                  * store another copy of wwn of fabric/or nport used in
2881                  * AUTH_ELS cmd
2882                  */
2883                 bcopy((void *)&ncval->msg_hdr.nodeName,
2884                     (void *)&node_dhc->nlp_auth_wwn, sizeof (NAME_TYPE));
2885         }
2886         /* Collect the challenge value */
2887         tmp = (uint8_t *)((uint8_t *)lp + sizeof (DHCHAP_CHALL_NULL));

2889         if (ncval->hash_id == AUTH_MD5) {
2890                 if (ncval->cval_len != LE_SWAP32(MD5_LEN)) {
2891                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2892                     "rcv_auth_msg_auth_negotiate_cmpl_wait4next:0x%x.%x!=%x",
2893                     ndlp->nlp_DID, ncval->cval_len, LE_SWAP32(MD5_LEN));

2895                         ReasonCode = AUTHRJT_FAILURE;
2896                         ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
2897                         goto AUTH_Reject;
2898                 }
2899                 bzero(un_cval.md5.val, sizeof (MD5_CVAL));
```

```
2900                bcopy((void *)tmp, (void *)un_cval.md5.val,
2901                    sizeof (MD5_CVAL));
2902                tmp += sizeof (MD5_CVAL);

2904                arg2len = MD5_LEN;

2906        } else if (ncval->hash_id == AUTH_SHA1) {
2907                if (ncval->cval_len != LE_SWAP32(SHA1_LEN)) {
2908                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2909                    "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x %x!=%x",
2910                    ndlp->nlp_DID, ncval->cval_len, LE_SWAP32(MD5_LEN));

2912                        ReasonCode = AUTHRJT_FAILURE;
2913                        ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
2914                        goto AUTH_Reject;
2915                }
2916                bzero(un_cval.sha1.val, sizeof (SHA1_CVAL));
2917                bcopy((void *)tmp, (void *)un_cval.sha1.val,
2918                    sizeof (SHA1_CVAL));
2919                tmp += sizeof (SHA1_CVAL);

2921                arg2len = SHA1_LEN;

2923        } else {
2924        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2925            "emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x %x",
2926            ndlp->nlp_DID, ncval->hash_id);

2928                ReasonCode = AUTHRJT_FAILURE;
2929                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
2930                goto AUTH_Reject;
2931        }

2933        /*
2934         * store hash_id for later usage : hash_id is set by responder in its
2935         * dhchap_challenge
2936         */
2937        node_dhc->hash_id = ncval->hash_id;

2939        /* always use this */
2940        /* store another copy of the hash_id */
2941        node_dhc->nlp_auth_hashid = ncval->hash_id;

2943        /* store dhgp_id for later usage */
2944        node_dhc->dhgp_id = ncval->dhgp_id;

2946        /* store another copy of dhgp_id */
2947        /* always use this */
2948        node_dhc->nlp_auth_dhgpid = ncval->dhgp_id;

2950        /*
2951         * ndlp->nlp_auth_hashid, nlp_auth_dhgpid store the hashid and dhgpid
2952         * when this very ndlp is the auth transaction responder (in other
2953         * words, responder means that this ndlp is send the host the
2954         * challenge. ndlp could be fffffe or another initiator or target
2955         * nport.
2956         */

2958        dhvallen = *((uint32_t *)(tmp));

2960        switch (ncval->dhgp_id) {
2961        case GROUP_NULL:
2962                /* null DHCHAP only */
2963                if (LE_SWAP32(dhvallen) != 0) {
2964                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2965                    "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x %x %x",
```



```
2966                    ndlp->nlp_DID, ncval->dhgp_id, LE_SWAP32(dhvallen));

2968                        ReasonCode = AUTHRJT_FAILURE;
2969                        ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
2970                        goto AUTH_Reject;
2971                }
2972                break;

2974        case GROUP_1024:
2975        case GROUP_1280:
2976        case GROUP_1536:
2977        case GROUP_2048:
2978                /* Collect the DH Value */
2979                tmp += sizeof (uint32_t);

2981                dhval = (uint8_t *)kmem_zalloc(LE_SWAP32(dhvallen),
2982                    KM_NOSLEEP);
2983                if (dhval == NULL) {
2984                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2985                    "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x %x %x",
2986                    ndlp->nlp_DID, ncval->dhgp_id, dhval);

2988                        ReasonCode = AUTHRJT_LOGIC_ERR;
2989                        ReasonCodeExplanation = AUTHEXP_RESTART_AUTH;
2990                        goto AUTH_Reject;
2991                }
2992                bcopy((void *)tmp, (void *)dhval, LE_SWAP32(dhvallen));
2993                break;

2995        default:
2996                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
2997                    "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x %x.",
2998                    ndlp->nlp_DID, ncval->dhgp_id);

3000                ReasonCode = AUTHRJT_FAILURE;
3001                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
3002                goto AUTH_Reject;
3003        }

3005        /*
3006         * Calculate the hash value, hash function, DH group, secret etc.
3007         * could be stored in port_dhc.
3008         */

3010        /* arg5 has the response with NULL or Full DH group support */
3011        arg5 = (uint32_t *)emlxs_hash_rsp(port, port_dhc,
3012            ndlp, tran_id, un_cval, dhval, LE_SWAP32(dhvallen));

3014        /* Or should check ndlp->auth_cfg..... */
3015        if (node_dhc->auth_cfg.bidirectional == 1) {
3016                /* get arg2 here */
3017                /*
3018                 * arg2 is the new challenge C2 from initiator if bi-dir auth
3019                 * is supported
3020                 */
3021                bzero(&random_number, sizeof (random_number));

3023                if (hba->rdn_flag == 1) {
3024                        emlxs_get_random_bytes(ndlp, random_number, 20);
3025                } else {
3026                        (void) random_get_pseudo_bytes(random_number, arg2len);
3027                }

3029                /* cache it for later verification usage */
3030                if (ndlp->nlp_DID == FABRIC_DID) {
3031                        bcopy((void *)&random_number[0],
```

```
3032                            (void *)&node_dhc->bi_cval[0], arg2len);
3033                        node_dhc->bi_cval_len = arg2len;

3035                        /* save another copy in our partner's ndlp */
3036                        bcopy((void *)&random_number[0],
3037                            (void *)&node_dhc->nlp_auth_misc.bi_cval[0],
3038                            arg2len);
3039                        node_dhc->nlp_auth_misc.bi_cval_len = arg2len;
3040                } else {
3041                        bcopy((void *)&random_number[0],
3042                            (void *)&node_dhc->nlp_auth_misc.bi_cval[0],
3043                            arg2len);
3044                        node_dhc->nlp_auth_misc.bi_cval_len = arg2len;
3045                }
3046        }
3047        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3048            "rcv_auth_msg_auth_negotiate_cmpl_wait4next:0x%x(%x,%x,%x,%x,%x)",
3049            ndlp->nlp_DID, node_dhc->nlp_auth_tranid_rsp,
3050            node_dhc->nlp_auth_tranid_ini,
3051            ncval->hash_id, ncval->dhgp_id, dhvallen);

3053        /* Issue ELS DHCHAP_Reply */
3054        /*
3055         * arg1 has the response, arg2 has the new challenge if needed (g^y
3056         * mod p) is the pubkey: all are ready and to go
3057         */

3059        /* return 0 success, otherwise failure */
3060        if (emlxs_issue_dhchap_reply(port, ndlp, 0, arg5, dhval,
3061            LE_SWAP32(dhvallen),
3062            random_number, arg2len)) {
3063        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3064            "rcv_auth_msg_auth_negotiate_cmpl_wait4next: 0x%x.failed.",
3065            ndlp->nlp_DID);

3067                kmem_free(dhval, LE_SWAP32(dhvallen));
3068                ReasonCode = AUTHRJT_LOGIC_ERR;
3069                ReasonCodeExplanation = AUTHEXP_RESTART_AUTH;
3070                goto AUTH_Reject;
3071        }
3072        return (node_dhc->state);

3074 AUTH_Reject:

3076        emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED, ReasonCode,
3077            ReasonCodeExplanation);
3078        (void) emlxs_issue_auth_reject(port, ndlp, 0, 0, ReasonCode,
3079            ReasonCodeExplanation);
3080        emlxs_dhc_auth_complete(port, ndlp, 1);

3082        return (node_dhc->state);

3084 } /* emlxs_rcv_auth_msg_auth_negotiate_cmpl_wait4next */


3087 /*
3088  * This routine should be set to emlxs_disc_neverdev
3089  *
3090  */
3091 /* ARGSUSED */
3092 static uint32_t
3093 emlxs_cmpl_auth_msg_auth_negotiate_cmpl_wait4next(
3094 emlxs_port_t    *port,
3095 /* CHANNEL * rp, */ void *arg1,
3096 /* IOCBQ * iocbq, */ void *arg2,
3097 /* MATCHMAP * mp, */ void *arg3,
```



```
3098 /* NODELIST * ndlp */ void *arg4,
3099 uint32_t evt)
3100 {
3101        NODELIST *ndlp = (NODELIST *)arg4;

3103        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3104            "cmpl_auth_msg_auth_negotiate_cmpl_wait4next.0x%x. Not iplted.",
3105            ndlp->nlp_DID);

3107        return (0);
3108 } /* emlxs_cmpl_auth_msg_auth_negotiate_cmpl_wait4next() */


3111 /*
3112  * ! emlxs_rcv_auth_msg_dhchap_reply_issue
3113  *
3114  * This routine is invoked when the host received an unsolicited ELS AUTH
3115  * msg from an NxPort or FxPort into which the host has
3116  * sent an ELS DHCHAP_Reply msg. since the host is the initiator and the
3117  * AUTH transaction is in progress between host and the
3118  * NxPort or FxPort, as a result, the host will send back ACC and AUTH_Reject
3119  * and set the next state = NPR_NODE.
3120  *
3121  */
3122 /* ARGSUSED */
3123 static uint32_t
3124 emlxs_rcv_auth_msg_dhchap_reply_issue(
3125 emlxs_port_t *port,
3126 /* CHANNEL * rp, */ void *arg1,
3127 /* IOCBQ   * iocbq, */ void *arg2,
3128 /* MATCHMAP * mp, */ void *arg3,
3129 /* NODELIST * ndlp */ void *arg4,
3130 uint32_t evt)
3131 {
3132        NODELIST *ndlp = (NODELIST *)arg4;

3134        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3135            "rcv_auth_msg_dhchap_reply_issue called. 0x%x. Not implemented.",
3136            ndlp->nlp_DID);

3138        return (0);

3140 } /* emlxs_rcv_auth_msg_dhchap_reply_issue */



3144 /*
3145  * ! emlxs_cmpl_auth_msg_dhchap_reply_issue
3146  *
3147  * This routine is invoked when
3148  * the host received a solicited ACC/RJT from ELS command from an NxPort
3149  * or FxPort that already received the ELS DHCHAP_Reply
3150  * msg from the host. in case of ACC, next state = DHCHAP_REPLY_CMPL_WAIT4NEXT
3151  * in case of RJT, next state = NPR_NODE
3152  */
3153 /* ARGSUSED */
3154 static uint32_t
3155 emlxs_cmpl_auth_msg_dhchap_reply_issue(
3156 emlxs_port_t *port,
3157 /* CHANNEL  * rp, */ void *arg1,
3158 /* IOCBQ * iocbq, */ void *arg2,
3159 /* MATCHMAP * mp, */ void *arg3,
3160 /* NODELIST * ndlp */ void *arg4,
3161 uint32_t evt)
3162 {
3163        NODELIST *ndlp = (NODELIST *) arg4;
```

```
3164         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

3166         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3167             "emlxs_cmpl_auth_msg_dhchap_reply_issue: did=0x%x",
3168             ndlp->nlp_DID);

3170         /* start the emlxs_dhc_authrsp_timeout timer now */
3171         if (node_dhc->nlp_authrsp_tmo == 0) {
3172                 node_dhc->nlp_authrsp_tmo = DRV_TIME +
3173                     node_dhc->auth_cfg.authentication_timeout;
3174         }
3175         /*
3176          * The next state should be
3177          * emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next
3178          */
3179         emlxs_dhc_state(port, ndlp,
3180             NODE_STATE_DHCHAP_REPLY_CMPL_WAIT4NEXT, 0, 0);

3182         return (node_dhc->state);

3184 } /* emlxs_cmpl_auth_msg_dhchap_reply_issue */



3188 /*
3189  * ! emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next
3190  *
3191  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
3192  * uint32_t \b Description: This rountine is invoked
3193  * when the host received an unsolicited ELS AUTH Msg from the NxPort or
3194  * FxPort that already sent ACC back to the host after
3195  * receipt of DHCHAP_Reply msg. In normal case, this unsolicited msg could
3196  * be DHCHAP_Success msg.
3197  *
3198  * if msg is ELS DHCHAP_Success, based on the payload, host send back ACC and 1.
3199  * for uni-directional, and set next state =
3200  * REG_LOGIN. 2. for bi-directional,  and host do some computations
3201  * (hash etc) and send back either DHCHAP_Success Msg and set
3202  * next state = DHCHAP_SUCCESS_ISSUE_WAIT4NEXT or AUTH_Reject and set next
3203  * state = NPR_NODE. if msg is ELS AUTH_Reject, then
3204  * send back ACC and set next state = NPR_NODE if msg is anything else, then
3205  * RJT and set next state = NPR_NODE
3206  */
3207 /* ARGSUSED */
3208 static uint32_t
3209 emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next(
3210 emlxs_port_t *port,
3211 /* CHANNEL * rp, */ void *arg1,
3212 /* IOCBQ * iocbq, */ void *arg2,
3213 /* MATCHMAP * mp, */ void *arg3,
3214 /* NODELIST * ndlp */ void *arg4,
3215 uint32_t evt)
3216 {
3217         emlxs_port_dhc_t *port_dhc = &port->port_dhc;
3218         IOCBQ *iocbq = (IOCBQ *)arg2;
3219         MATCHMAP *mp = (MATCHMAP *)arg3;
3220         NODELIST *ndlp = (NODELIST *)arg4;
3221         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
3222         uint8_t *bp;
3223         uint32_t *lp;
3224         DHCHAP_SUCCESS_HDR *dh_success;
3225         uint8_t *tmp;
3226         uint8_t rsp_size;
3227         AUTH_RJT *auth_rjt;
3228         uint32_t tran_id;
3229         uint32_t *hash_val;
```



```
3230         union challenge_val un_cval;
3231         uint8_t ReasonCode;
3232         uint8_t ReasonCodeExplanation;
3233         char info[64];

3235         bp = mp->virt;
3236         lp = (uint32_t *)bp;

3238         /*
3239          * 1. we process the DHCHAP_Success or AUTH_Reject 2. ACC it first 3.
3240          * based on the result of 1 we goto the next stage SCR etc.
3241          */

3243         /* sp = (SERV_PARM *)((uint8_t *)lp + sizeof(uint32_t)); */
3244         dh_success = (DHCHAP_SUCCESS_HDR *)((uint8_t *)lp);

3246         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3247             "rcv_auth_msg_dhchap_reply_cmpl_wait4next: 0x%x 0x%x 0x%x",
3248             ndlp->nlp_DID, dh_success->auth_els_code,
3249             dh_success->auth_msg_code);

3251         node_dhc->nlp_authrsp_tmo = 0;

3253         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

3255         if (dh_success->auth_msg_code == AUTH_REJECT) {
3256                 /* ACC it and retry etc.  */
3257                 auth_rjt = (AUTH_RJT *) dh_success;
3258                 ReasonCode = auth_rjt->ReasonCode;
3259                 ReasonCodeExplanation = auth_rjt->ReasonCodeExplanation;

3261         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3262             "emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next: 0x%x.(%x,%x)",
3263             ndlp->nlp_DID, ReasonCode, ReasonCodeExplanation);

3265                 switch (ReasonCode) {
3266                 case AUTHRJT_LOGIC_ERR:
3267                         switch (ReasonCodeExplanation) {
3268                         case AUTHEXP_MECH_UNUSABLE:
3269                         case AUTHEXP_DHGROUP_UNUSABLE:
3270                         case AUTHEXP_HASHFUNC_UNUSABLE:
3271                                 ReasonCode = AUTHRJT_LOGIC_ERR;
3272                                 ReasonCodeExplanation = AUTHEXP_RESTART_AUTH;
3273                                 break;

3275                         case AUTHEXP_RESTART_AUTH:
3276                                 /*
3277                                  * Cancel the rsp timer if not cancelled yet.
3278                                  * and restart auth tran now.
3279                                  */
3280                                 if (node_dhc->nlp_authrsp_tmo != 0) {
3281                                         node_dhc->nlp_authrsp_tmo = 0;
3282                                         node_dhc->nlp_authrsp_tmocnt = 0;
3283                                 }
3284                                 if (emlxs_dhc_auth_start(port, ndlp,
3285                                     NULL, NULL) != 0) {
3286                                         EMLXS_MSGF(EMLXS_CONTEXT,
3287                                             &emlxs_fcsp_debug_msg,
3288                                             "Reauth timeout.failed. 0x%x %x",
3289                                             ndlp->nlp_DID, node_dhc->state);
3290                                 }
3291                                 return (node_dhc->state);

3293                         default:
3294                                 ReasonCode = AUTHRJT_FAILURE;
3295                                 ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
```

```
3296                                    break;
3297                            }
3298                            break;

3300                    case AUTHRJT_FAILURE:
3301                    default:
3302                            ReasonCode = AUTHRJT_FAILURE;
3303                            ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3304                            emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED,
3305                                ReasonCode, ReasonCodeExplanation);
3306                            goto out;
3307                    }

3309                    goto AUTH_Reject;
3310            }
3311            if (dh_success->auth_msg_code == DHCHAP_SUCCESS) {

3313                    /* Verify the tran_id */
3314                    tran_id = dh_success->tran_id;

3316            EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3317                "rcv_auth_msg_dhchap_reply_cmpl_wait4next: 0x%x 0x%x 0x%x 0x%x 0x%x",
3318                ndlp->nlp_DID, LE_SWAP32(tran_id),
3319                node_dhc->nlp_auth_tranid_rsp,
3320                node_dhc->nlp_auth_tranid_ini);

3322                    if (LE_SWAP32(tran_id) != node_dhc->nlp_auth_tranid_rsp) {
3323                    EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3324                        "rcv_auth_msg_dhchap_reply_cmpl_wait4next:0x%x %x!=%x",
3325                        ndlp->nlp_DID, LE_SWAP32(tran_id),
3326                        node_dhc->nlp_auth_tranid_rsp);

3328                            ReasonCode = AUTHRJT_FAILURE;
3329                            ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;
3330                            goto AUTH_Reject;
3331                    }
3332                    if (node_dhc->auth_cfg.bidirectional == 0) {
3333                            node_dhc->flag |=
3334                                (NLP_REMOTE_AUTH | NLP_SET_REAUTH_TIME);

3336                            emlxs_dhc_state(port, ndlp,
3337                                NODE_STATE_AUTH_SUCCESS, 0, 0);
3338                            emlxs_log_auth_event(port, ndlp, ESC_EMLXS_20,
3339                                "Host-initiated-unidir-auth-success");
3340                            emlxs_dhc_auth_complete(port, ndlp, 0);
3341                    } else {
3342                            /* bidir auth needed */
3343                            /* if (LE_SWAP32(dh_success->msg_len) > 4) { */

3345                            tmp = (uint8_t *)((uint8_t *)lp);
3346                            tmp += 8;
3347                            tran_id = *(uint32_t *)tmp;
3348                            tmp += 4;
3349                            rsp_size = *(uint32_t *)tmp;
3350                            tmp += 4;

3352                            /* tmp has the response from responder */

3354                            /*
3355                             * node_dhc->bi_cval has the bidir challenge value
3356                             * from initiator
3357                             */

3359                            if (LE_SWAP32(rsp_size) == 16) {
3360                                    bzero(un_cval.md5.val, LE_SWAP32(rsp_size));
3361                                    if (ndlp->nlp_DID == FABRIC_DID)
```



```
3362                                            bcopy((void *)node_dhc->bi_cval,
3363                                                (void *)un_cval.md5.val,
3364                                                LE_SWAP32(rsp_size));
3365                                    else
3366                                    bcopy(
3367                                        (void *)node_dhc->nlp_auth_misc.bi_cval,
3368                                        (void *)un_cval.md5.val,
3369                                        LE_SWAP32(rsp_size));

3371                            } else if (LE_SWAP32(rsp_size) == 20) {

3373                                    bzero(un_cval.sha1.val, LE_SWAP32(rsp_size));
3374                                    if (ndlp->nlp_DID == FABRIC_DID)
3375                                            bcopy((void *)node_dhc->bi_cval,
3376                                                (void *)un_cval.sha1.val,
3377                                                LE_SWAP32(rsp_size));
3378                                    else
3379                                    bcopy(
3380                                        (void *)node_dhc->nlp_auth_misc.bi_cval,
3381                                        (void *)un_cval.sha1.val,
3382                                        LE_SWAP32(rsp_size));
3383                            }
3384                            /* verify the response */
3385                            /* NULL DHCHAP works for now */
3386                            /* for DH group as well */

3388                            /*
3389                             * Cai2 = H (C2 || ((g^x mod p)^y mod p) ) = H (C2 ||
3390                             * (g^xy mod p) )
3391                             *
3392                             * R = H (Ti || Km || Cai2) R ?= R2
3393                             */
3394                            hash_val = emlxs_hash_vrf(port, port_dhc, ndlp,
3395                                tran_id, un_cval);

3397                            if (bcmp((void *)tmp, (void *)hash_val,
3398                                LE_SWAP32(rsp_size))) {
3399                                    if (hash_val != NULL) {
3400                                            /* not identical */
3401                    EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3402                        "emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next: 0x%x.failed. %x",
3403                        ndlp->nlp_DID, *(uint32_t *)hash_val);
3404                                    }
3405                                    ReasonCode = AUTHRJT_FAILURE;
3406                                    ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3407                                    goto AUTH_Reject;
3408                            }
3409                            emlxs_dhc_state(port, ndlp,
3410                                NODE_STATE_DHCHAP_SUCCESS_ISSUE_WAIT4NEXT, 0, 0);

3412                            /* send out DHCHAP_SUCCESS */
3413                            (void) emlxs_issue_dhchap_success(port, ndlp, 0, 0);
3414                    }
3415            }
3416            return (node_dhc->state);

3418    AUTH_Reject:

3420            emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED,
3421                ReasonCode, ReasonCodeExplanation);
3422            (void) emlxs_issue_auth_reject(port, ndlp, 0, 0, ReasonCode,
3423                ReasonCodeExplanation);
3424            emlxs_dhc_auth_complete(port, ndlp, 1);

3426            return (node_dhc->state);
3427    out:
```

```
3428         (void) sprintf(info,
3429             "Auth Failed: ReasonCode=0x%x, ReasonCodeExplanation=0x%x",
3430             ReasonCode, ReasonCodeExplanation);

3432         emlxs_log_auth_event(port, ndlp, ESC_EMLXS_20, info);
3433         emlxs_dhc_auth_complete(port, ndlp, 1);

3435         return (node_dhc->state);

3437 } /* emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next */



3441 /*
3442  * This routine should be set to emlxs_disc_neverdev as it shouldnot happen.
3443  *
3444  */
3445 /* ARGSUSED */
3446 static uint32_t
3447 emlxs_cmpl_auth_msg_dhchap_reply_cmpl_wait4next(
3448 emlxs_port_t *port,
3449 /* CHANNEL * rp, */ void *arg1,
3450 /* IOCBQ  * iocbq, */ void *arg2,
3451 /* MATCHMAP * mp, */ void *arg3,
3452 /* NODELIST * ndlp */ void *arg4,
3453 uint32_t evt)
3454 {
3455         NODELIST *ndlp = (NODELIST *)arg4;

3457         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3458             "cmpl_auth_msg_dhchap_reply_cmpl_wait4next. 0x%x.Not ipleted.",
3459             ndlp->nlp_DID);

3461         return (0);

3463 } /* emlxs_cmpl_auth_msg_dhchap_reply_cmpl_wait4next */



3466 /*
3467  * emlxs_rcv_auth_msg_dhchap_success_issue_wait4next
3468  *
3469  * This routine is supported
3470  * for HBA in either auth initiator mode or responder mode.
3471  *
3472  * This routine is invoked when the host as the auth responder received
3473  * an unsolicited ELS AUTH msg from the NxPort as the auth
3474  * initiator that already received the ELS DHCHAP_Success.
3475  *
3476  * If the host is the auth initiator and since the AUTH transction is
3477  * already in progress, therefore, any auth els msg should not
3478  * happen and if happened, RJT and move to NPR_NODE.
3479  *
3480  * If the host is the auth reponder, this unsolicited els auth msg should
3481  * be DHCHAP_Success for this bi-directional auth
3482  * transaction. In which case, the host should send ACC back and move state
3483  * to REG_LOGIN. If this unsolicited els auth msg is
3484  * DHCHAP_Reject, which could mean that the auth failed, then host should
3485  * send back ACC and set the next state to NPR_NODE.
3486  *
3487  */
3488 /* ARGSUSED */
3489 static uint32_t
3490 emlxs_rcv_auth_msg_dhchap_success_issue_wait4next(
3491 emlxs_port_t *port,
3492 /* CHANNEL * rp, */ void *arg1,
3493 /* IOCBQ * iocbq, */ void *arg2,
```



```
3494 /* MATCHMAP * mp, */ void *arg3,
3495 /* NODELIST * ndlp */ void *arg4,
3496 uint32_t evt)
3497 {
3498         NODELIST *ndlp = (NODELIST *) arg4;

3500         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3501             "rcv_auth_msg_dhchap_success_issue_wait4next. 0x%x. Not iplted.",
3502             ndlp->nlp_DID);

3504         return (0);

3506 } /* emlxs_rcv_auth_msg_dhchap_success_issue_wait4next */



3510 /*
3511  * ! emlxs_cmpl_auth_msg_dhchap_success_issue_wait4next
3512  *
3513  * This routine is invoked when
3514  * the host as the auth initiator received an solicited ACC/RJT from the
3515  * NxPort or FxPort that already received DHCHAP_Success
3516  * Msg the host sent before. in case of ACC, set next state = REG_LOGIN.
3517  * in case of RJT, set next state = NPR_NODE.
3518  *
3519  */
3520 /* ARGSUSED */
3521 static uint32_t
3522 emlxs_cmpl_auth_msg_dhchap_success_issue_wait4next(
3523 emlxs_port_t *port,
3524 /* CHANNEL * rp, */ void *arg1,
3525 /* IOCBQ * iocbq, */ void *arg2,
3526 /* MATCHMAP * mp, */ void *arg3,
3527 /* NODELIST * ndlp */ void *arg4,
3528 uint32_t evt)
3529 {
3530         NODELIST *ndlp = (NODELIST *)arg4;
3531         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

3533         /*
3534          * Either host is the initiator and auth or (reauth bi-direct) is
3535          * done, so start host reauth heartbeat timer now if host side reauth
3536          * heart beat never get started. Or host is the responder and the
3537          * other entity is done with its reauth heart beat with
3538          * uni-directional auth. Anyway we start host side reauth heart beat
3539          * timer now.
3540          */

3542         node_dhc->flag &= ~NLP_REMOTE_AUTH;
3543         node_dhc->flag |= NLP_SET_REAUTH_TIME;

3545         emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_SUCCESS, 0, 0);
3546         emlxs_log_auth_event(port, ndlp, ESC_EMLXS_25,
3547             "Host-initiated-bidir-auth-success");
3548         emlxs_dhc_auth_complete(port, ndlp, 0);

3550         return (node_dhc->state);

3552 } /* emlxs_cmpl_auth_msg_dhchap_success_issue_wait4next */



3555 /*
3556  * ! emlxs_cmpl_auth_msg_auth_negotiate_rcv
3557  *
3558  * This routine is invoked when
3559  * the host received the solicited ACC/RJT ELS cmd from an FxPort or an
```

```
3560  * NxPort that has received the ELS DHCHAP_Challenge.
3561  * The host is the auth responder and the auth transaction is still in
3562  * progress.
3563  *
3564  */
3565 /* ARGSUSED */
3566 static uint32_t
3567 emlxs_cmpl_auth_msg_auth_negotiate_rcv(
3568 emlxs_port_t *port,
3569 /* CHANNEL * rp, */ void *arg1,
3570 /* IOCBQ * iocbq, */ void *arg2,
3571 /* MATCHMAP * mp, */ void *arg3,
3572 /* NODELIST * ndlp */ void *arg4,
3573 uint32_t evt)
3574 {
3575         NODELIST *ndlp = (NODELIST *)arg4;

3577         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3578             "cmpl_auth_msg_auth_negotiate_rcv called. 0x%x. Not implemented.",
3579             ndlp->nlp_DID);

3581         return (0);

3583 } /* emlxs_cmpl_auth_msg_auth_negotiate_rcv */



3587 /*
3588  * ! emlxs_rcv_auth_msg_dhchap_challenge_issue
3589  *
3590  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
3591  * uint32_t \b Description: This routine should be
3592  * emlxs_disc_neverdev. The host is the auth responder and the auth
3593  * transaction is still in progress, any unsolicited els auth
3594  * msg is unexpected and should not happen in normal case.
3595  *
3596  * If DHCHAP_Reject, ACC and next state = NPR_NODE. anything else, RJT and
3597  * next state = NPR_NODE.
3598  */
3599 /* ARGSUSED */
3600 static uint32_t
3601 emlxs_rcv_auth_msg_dhchap_challenge_issue(
3602 emlxs_port_t *port,
3603 /* CHANNEL * rp, */ void *arg1,
3604 /* IOCBQ * iocbq, */ void *arg2,
3605 /* MATCHMAP * mp, */ void *arg3,
3606 /* NODELIST * ndlp */ void *arg4,
3607 uint32_t evt)
3608 {
3609         NODELIST *ndlp = (NODELIST *)arg4;

3611         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3612             "rcv_auth_msg_dhchap_challenge_issue called. 0x%x. Not iplted.",
3613             ndlp->nlp_DID);

3615         return (0);

3617 } /* emlxs_rcv_auth_msg_dhchap_challenge_issue */



3621 /*
3622  * ! emlxs_cmpl_auth_msg_dhchap_challenge_issue
3623  *
3624  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
3625  * uint32_t \b Description: This routine is invoked when
```



```
3626  * the host as the responder received the solicited response (ACC or RJT)
3627  * from initiator to the DHCHAP_Challenge msg sent from
3628  * host. In case of ACC, the next state = DHCHAP_CHALLENGE_CMPL_WAIT4NEXT
3629  * In case of RJT, the next state = NPR_NODE.
3630  *
3631  */
3632 /* ARGSUSED */
3633 static uint32_t
3634 emlxs_cmpl_auth_msg_dhchap_challenge_issue(
3635 emlxs_port_t *port,
3636 /* CHANNEL * rp, */ void *arg1,
3637 /* IOCBQ * iocbq, */ void *arg2,
3638 /* MATCHMAP * mp, */ void *arg3,
3639 /* NODELIST * ndlp */ void *arg4,
3640 uint32_t evt)
3641 {
3642         NODELIST *ndlp = (NODELIST *)arg4;
3643         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

3645         /*
3646          * The next state should be
3647          * emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next
3648          */
3649         emlxs_dhc_state(port, ndlp,
3650             NODE_STATE_DHCHAP_CHALLENGE_CMPL_WAIT4NEXT, 0, 0);

3652         /* Start the fc_authrsp_timeout timer */
3653         if (node_dhc->nlp_authrsp_tmo == 0) {
3654         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3655             "cmpl_auth_msg_dhchap_challenge_issue: Starting authrsp timer.");

3657                 node_dhc->nlp_authrsp_tmo = DRV_TIME +
3658                     node_dhc->auth_cfg.authentication_timeout;
3659         }
3660         return (node_dhc->state);

3662 } /* emlxs_cmpl_auth_msg_dhchap_challenge_issue */


3665 /*
3666  * ! emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next
3667  *
3668  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
3669  * uint32_t \b Description: This routine is invoked when
3670  * the host as the auth responder received an unsolicited auth msg from the
3671  * FxPort or NxPort that already sent ACC to the DHCH_
3672  * Challenge it received. In normal case this unsolicited auth msg should
3673  * be DHCHAP_Reply msg from the initiator.
3674  *
3675  * For DHCHAP_Reply msg, the host send back ACC and then do verification
3676  * (hash?) and send back DHCHAP_Success and next state as
3677  * DHCHAP_SUCCESS_ISSUE or DHCHAP_Reject and next state as NPR_NODE based on
3678  * the verification result.
3679  *
3680  * For bi-directional auth transaction, Reply msg should have the new
3681  * challenge value from the initiator. thus the Success msg
3682  * sent out should have the corresponding Reply from the responder.
3683  *
3684  * For uni-directional, Reply msg received does not contains the new
3685  * challenge and therefore the Success msg does not include the
3686  * Reply msg.
3687  *
3688  * For DHCHAP_Reject, send ACC and moved to the next state NPR_NODE. For
3689  * anything else, send RJT and moved to NPR_NODE.
3690  *
3691  */
```

```
3692 /* ARGSUSED */
3693 static uint32_t
3694 emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next(
3695 emlxs_port_t *port,
3696 /* CHANNEL * rp, */ void *arg1,
3697 /* IOCBQ * iocbq, */ void *arg2,
3698 /* MATCHMAP * mp, */ void *arg3,
3699 /* NODELIST * ndlp */ void *arg4,
3700 uint32_t evt)
3701 {
3702         emlxs_port_dhc_t *port_dhc = &port->port_dhc;
3703         IOCBQ *iocbq = (IOCBQ *)arg2;
3704         MATCHMAP *mp = (MATCHMAP *)arg3;
3705         NODELIST *ndlp = (NODELIST *)arg4;
3706         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
3707         uint8_t *bp;
3708         uint32_t *lp;
3709         DHCHAP_REPLY_HDR *dh_reply;
3710         uint8_t *tmp;
3711         uint32_t rsp_len;
3712         uint8_t rsp[20];        /* should cover SHA-1 and MD5's rsp */
3713         uint32_t dhval_len;
3714         uint8_t dhval[512];
3715         uint32_t cval_len;
3716         uint8_t cval[20];
3717         uint32_t tran_id;
3718         uint32_t *hash_val = NULL;
3719         uint8_t ReasonCode;
3720         uint8_t ReasonCodeExplanation;
3721         AUTH_RJT *rjt;

3723         /* ACC the ELS DHCHAP_Reply msg first */

3725         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

3727         bp = mp->virt;
3728         lp = (uint32_t *)bp;

3730         /*
3731          * send back ELS AUTH_Reject or DHCHAP_Success msg based on the
3732          * verification result. i.e., hash computation etc.
3733          */
3734         dh_reply = (DHCHAP_REPLY_HDR *)((uint8_t *)lp);
3735         tmp = (uint8_t *)((uint8_t *)lp);

3737         tran_id = dh_reply->tran_id;

3739         if (LE_SWAP32(tran_id) != node_dhc->nlp_auth_tranid_ini) {

3741         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3742             "rcv_auth_msg_dhchap_challenge_cmpl_wait4next:0x%x 0x%x 0x%x",
3743             ndlp->nlp_DID, tran_id, node_dhc->nlp_auth_tranid_ini);

3745                 ReasonCode = AUTHRJT_FAILURE;
3746                 ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;
3747                 goto Reject;
3748         }

3750         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3751             "rcv_a_m_dhch_chll_cmpl_wait4next:0x%x 0x%x 0x%x 0x%x 0x%x",
3752             ndlp->nlp_DID, tran_id, node_dhc->nlp_auth_tranid_ini,
3753             node_dhc->nlp_auth_tranid_rsp, dh_reply->auth_msg_code);

3755         /* cancel the nlp_authrsp_timeout timer and send out Auth_Reject */
3756         if (node_dhc->nlp_authrsp_tmo) {
3757                 node_dhc->nlp_authrsp_tmo = 0;
```



```
3758         }
3759         if (dh_reply->auth_msg_code == AUTH_REJECT) {

3761                 rjt = (AUTH_RJT *)((uint8_t *)lp);
3762                 ReasonCode = rjt->ReasonCode;
3763                 ReasonCodeExplanation = rjt->ReasonCodeExplanation;

3765         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3766             "rcv_a_msg_dhch_chall_cmpl_wait4next:RJT rcved:0x%x 0x%x",
3767             ReasonCode, ReasonCodeExplanation);

3769                 switch (ReasonCode) {
3770                 case AUTHRJT_LOGIC_ERR:
3771                         switch (ReasonCodeExplanation) {
3772                         case AUTHEXP_MECH_UNUSABLE:
3773                         case AUTHEXP_DHGROUP_UNUSABLE:
3774                         case AUTHEXP_HASHFUNC_UNUSABLE:
3775                                 ReasonCode = AUTHRJT_LOGIC_ERR;
3776                                 ReasonCodeExplanation = AUTHEXP_RESTART_AUTH;
3777                                 break;

3779                         case AUTHEXP_RESTART_AUTH:
3780                                 /*
3781                                  * Cancel the rsp timer if not cancelled yet.
3782                                  * and restart auth tran now.
3783                                  */
3784                                 if (node_dhc->nlp_authrsp_tmo != 0) {
3785                                         node_dhc->nlp_authrsp_tmo = 0;
3786                                         node_dhc->nlp_authrsp_tmocnt = 0;
3787                                 }
3788                                 if (emlxs_dhc_auth_start(port, ndlp,
3789                                     NULL, NULL) != 0) {
3790                                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
3791                                     "Reauth timeout.Auth initfailed. 0x%x %x",
3792                                     ndlp->nlp_DID, node_dhc->state);
3793                                 }
3794                                 return (node_dhc->state);

3796                         default:
3797                                 ReasonCode = AUTHRJT_FAILURE;
3798                                 ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3799                                 break;
3800                         }
3801                         break;

3803                 case AUTHRJT_FAILURE:
3804                 default:
3805                         ReasonCode = AUTHRJT_FAILURE;
3806                         ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3807                         break;
3808                 }

3810                 goto Reject;

3812         }
3813         if (dh_reply->auth_msg_code == DHCHAP_REPLY) {

3815                 /* We must send out DHCHAP_Success msg and wait for ACC */
3816                 /* _AND_ if bi-dir auth, we have to wait for next */

3818                 /*
3819                  * Send back DHCHAP_Success or AUTH_Reject based on the
3820                  * verification result
3821                  */
3822                 tmp += sizeof (DHCHAP_REPLY_HDR);
3823                 rsp_len = LE_SWAP32(*(uint32_t *)tmp);
```

```
3824                tmp += sizeof (uint32_t);

3826                /* collect the response data */
3827                bcopy((void *)tmp, (void *)rsp, rsp_len);

3829                tmp += rsp_len;
3830                dhval_len = LE_SWAP32(*(uint32_t *)tmp);

3832                tmp += sizeof (uint32_t);




3836                if (dhval_len != 0) {
3837                        /* collect the DH value */
3838                        bcopy((void *)tmp, (void *)dhval, dhval_len);
3839                        tmp += dhval_len;
3840                }
3841                /*
3842                 * Check to see if there is any challenge for bi-dir auth in
3843                 * the reply msg
3844                 */
3845                cval_len = LE_SWAP32(*(uint32_t *)tmp);
3846                if (cval_len != 0) {
3847                        /* collect challenge value */
3848                        tmp += sizeof (uint32_t);
3849                        bcopy((void *)tmp, (void *)cval, cval_len);

3851                        if (ndlp->nlp_DID == FABRIC_DID) {
3852                                node_dhc->nlp_auth_bidir = 1;
3853                        } else {
3854                                node_dhc->nlp_auth_bidir = 1;
3855                        }
3856                } else {
3857                        node_dhc->nlp_auth_bidir = 0;
3858                }

3860        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
3861            "rcv_a_m_dhchap_challenge_cmpl_wait4next:Reply:%x %lx %x %x %x\n",
3862            ndlp->nlp_DID, *(uint32_t *)rsp, rsp_len, dhval_len, cval_len);

3864                /* Verify the response based on the hash func, dhgp_id etc. */
3865                /*
3866                 * all the information needed are stored in
3867                 * node_dhc->hrsp_xxx or ndlp->nlp_auth_misc.
3868                 */
3869                /*
3870                 * Basically compare the rsp value with the computed hash
3871                 * value
3872                 */

3874                /* allocate hash_val first as rsp_len bytes */
3875                /*
3876                 * we set bi-cval pointer as NULL because we are using
3877                 * node_dhc->hrsp_cval[]
3878                 */
3879                hash_val = emlxs_hash_verification(port, port_dhc, ndlp,
3880                    (tran_id), dhval, (dhval_len), 1, 0);

3882                if (hash_val == NULL) {
3883                        ReasonCode = AUTHRJT_FAILURE;
3884                        ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3885                        goto Reject;
3886                }
3887                if (bcmp((void *) rsp, (void *)hash_val, rsp_len)) {
3888                        /* not identical */
3889        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
```



```
3890            "rcv_auth_msg_dhchap_challenge_cmpl_wait4next: Not authted(1).");

3892                        ReasonCode = AUTHRJT_FAILURE;
3893                        ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3894                        goto Reject;
3895                }
3896                kmem_free(hash_val, rsp_len);
3897                hash_val = NULL;

3899                /* generate the reply based on the challenge received if any */
3900                if ((cval_len) != 0) {
3901                        /*
3902                         * Cal R2 = H (Ti || Km || Ca2) Ca2 = H (C2 || ((g^y
3903                         * mod p)^x mod p) ) = H (C2 || (g^(x*y) mod p)) = H
3904                         * (C2 || seskey) Km is the password associated with
3905                         * responder. Here cval: C2 dhval: (g^y mod p)
3906                         */
3907                        hash_val = emlxs_hash_get_R2(port, port_dhc,
3908                            ndlp, (tran_id), dhval,
3909                            (dhval_len), 1, cval);

3911                        if (hash_val == NULL) {
3912                                ReasonCode = AUTHRJT_FAILURE;
3913                                ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3914                                goto Reject;
3915                        }
3916                }
3917                emlxs_dhc_state(port, ndlp,
3918                    NODE_STATE_DHCHAP_SUCCESS_ISSUE, 0, 0);

3920                if (emlxs_issue_dhchap_success(port, ndlp, 0,
3921                    (uint8_t *)hash_val)) {
3922                        ReasonCode = AUTHRJT_FAILURE;
3923                        ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
3924                        goto Reject;
3925                }
3926        }
3927        return (node_dhc->state);

3929 Reject:

3931        emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED,
3932            ReasonCode, ReasonCodeExplanation);
3933        (void) emlxs_issue_auth_reject(port, ndlp, 0, 0, ReasonCode,
3934            ReasonCodeExplanation);
3935        emlxs_dhc_auth_complete(port, ndlp, 1);

3937 out:

3939        return (node_dhc->state);

3941 } /* emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next */



3945 /*
3946  * This routine should be emlxs_disc_neverdev.
3947  *
3948  */
3949 /* ARGSUSED */
3950 static uint32_t
3951 emlxs_cmpl_auth_msg_dhchap_challenge_cmpl_wait4next(
3952 emlxs_port_t *port,
3953 /* CHANNEL * rp, */ void *arg1,
3954 /* IOCBQ * iocbq, */ void *arg2,
3955 /* MATCHMAP * mp, */ void *arg3,
```

```
3956 /* NODELIST * ndlp */ void *arg4,
3957 uint32_t evt)
3958 {
3959         NODELIST *ndlp = (NODELIST *)arg4;

3961         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3962             "cmpl_a_m_dhch_chall_cmpl_wait4next.0x%x. Not implemented.",
3963             ndlp->nlp_DID);

3965         return (0);

3967 } /* emlxs_cmpl_auth_msg_dhchap_challenge_cmpl_wait4next */


3970 /*
3971  * ! emlxs_rcv_auth_msg_dhchap_success_issue
3972  *
3973  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
3974  * uint32_t \b Description:
3975  *
3976  * The host is the auth responder and the auth transaction is still in
3977  * progress, any unsolicited els auth msg is unexpected and
3978  * should not happen. If DHCHAP_Reject received, ACC back and move to next
3979  * state NPR_NODE. anything else, RJT and move to
3980  * NPR_NODE.
3981  */
3982 /* ARGSUSED */
3983 static uint32_t
3984 emlxs_rcv_auth_msg_dhchap_success_issue(
3985 emlxs_port_t *port,
3986 /* CHANNEL * rp, */ void *arg1,
3987 /* IOCBQ * iocbq, */ void *arg2,
3988 /* MATCHMAP * mp, */ void *arg3,
3989 /* NODELIST * ndlp */ void *arg4,
3990 uint32_t evt)
3991 {
3992         NODELIST *ndlp = (NODELIST *)arg4;

3994         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
3995             "rcv_a_m_dhch_success_issue called. did=0x%x. Not implemented.",
3996             ndlp->nlp_DID);

3998         return (0);

4000 } /* emlxs_rcv_auth_msg_dhchap_success_issue */



4004 /*
4005  * emlxs_cmpl_auth_msg_dhchap_success_issue
4006  *
4007  * This routine is invoked when
4008  * host as the auth responder received the solicited response (ACC or RJT)
4009  * from the initiator that received DHCHAP_ Success.
4010  *
4011  * For uni-dirctional authentication, we are done so the next state =
4012  * REG_LOGIN for bi-directional authentication, we will expect
4013  * DHCHAP_Success msg. so the next state = DHCHAP_SUCCESS_CMPL_WAIT4NEXT
4014  * and start the emlxs_dhc_authrsp_timeout timer
4015  */
4016 /* ARGSUSED */
4017 static uint32_t
4018 emlxs_cmpl_auth_msg_dhchap_success_issue(
4019 emlxs_port_t *port,
4020 /* CHANNEL * rp, */ void *arg1,
4021 /* IOCBQ * iocbq, */ void *arg2,
```



```
4022 /* MATCHMAP * mp, */ void *arg3,
4023 /* NODELIST * ndlp */ void *arg4,
4024 uint32_t evt)
4025 {
4026         NODELIST *ndlp = (NODELIST *)arg4;
4027         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

4029         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4030             "cmpl_a_m_dhch_success_issue: did=0x%x auth_bidir=0x%x",
4031             ndlp->nlp_DID, node_dhc->nlp_auth_bidir);

4033         if (node_dhc->nlp_auth_bidir == 1) {
4034                 /* we would expect the bi-dir authentication result */

4036                 /*
4037                  * the next state should be
4038                  * emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next
4039                  */
4040                 emlxs_dhc_state(port, ndlp,
4041                     NODE_STATE_DHCHAP_SUCCESS_CMPL_WAIT4NEXT, 0, 0);

4043                 /* start the emlxs_dhc_authrsp_timeout timer */
4044                 node_dhc->nlp_authrsp_tmo = DRV_TIME +
4045                     node_dhc->auth_cfg.authentication_timeout;
4046         } else {
4047                 node_dhc->flag &= ~NLP_REMOTE_AUTH;

4049                 emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_SUCCESS, 0, 0);
4050                 emlxs_log_auth_event(port, ndlp, ESC_EMLXS_22,
4051                     "Node-initiated-unidir-reauth-success");
4052                 emlxs_dhc_auth_complete(port, ndlp, 0);
4053         }

4055         return (node_dhc->state);

4057 } /* emlxs_cmpl_auth_msg_dhchap_success_issue */


4060 /* ARGSUSED */
4061 static uint32_t
4062 emlxs_device_recov_unmapped_node(
4063         emlxs_port_t *port,
4064         void *arg1,
4065         void *arg2,
4066         void *arg3,
4067         void *arg4,
4068         uint32_t evt)
4069 {
4070         NODELIST *ndlp = (NODELIST *)arg4;

4072         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4073             "emlxs_device_recov_unmapped_node called. 0x%x. Not implemented.",
4074             ndlp->nlp_DID);

4076         return (0);

4078 } /* emlxs_device_recov_unmapped_node */



4082 /* ARGSUSED */
4083 static uint32_t
4084 emlxs_device_rm_npr_node(
4085         emlxs_port_t *port,
4086         void *arg1,
4087         void *arg2,
```

```
4088         void *arg3,
4089         void *arg4,
4090         uint32_t evt)
4091 {
4092         NODELIST *ndlp = (NODELIST *)arg4;

4094         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4095             "emlxs_device_rm_npr_node called. 0x%x. Not implemented.",
4096             ndlp->nlp_DID);

4098         return (0);

4100 } /* emlxs_device_rm_npr_node */


4103 /* ARGSUSED */
4104 static uint32_t
4105 emlxs_device_recov_npr_node(
4106         emlxs_port_t *port,
4107         void *arg1,
4108         void *arg2,
4109         void *arg3,
4110         void *arg4,
4111         uint32_t evt)
4112 {
4113         NODELIST *ndlp = (NODELIST *)arg4;

4115         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4116             "emlxs_device_recov_npr_node called. 0x%x. Not implemented.",
4117             ndlp->nlp_DID);

4119         return (0);

4121 } /* emlxs_device_recov_npr_node */


4124 /* ARGSUSED */
4125 static uint32_t
4126 emlxs_device_rem_auth(
4127 emlxs_port_t *port,
4128 /* CHANNEL * rp, */ void *arg1,
4129 /* IOCBQ * iocbq, */ void *arg2,
4130 /* MATCHMAP * mp, */ void *arg3,
4131 /* NODELIST * ndlp */ void *arg4,
4132 uint32_t evt)
4133 {
4134         NODELIST *ndlp = (NODELIST *)arg4;
4135         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

4137         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4138             "emlxs_device_rem_auth: 0x%x.",
4139             ndlp->nlp_DID);

4141         emlxs_dhc_state(port, ndlp, NODE_STATE_UNKNOWN, 0, 0);

4143         return (node_dhc->state);

4145 } /* emlxs_device_rem_auth */


4148 /*
4149  * This routine is invoked when linkdown event happens during authentication
4150  */
4151 /* ARGSUSED */
4152 static uint32_t
4153 emlxs_device_recov_auth(
```



```
4154 emlxs_port_t *port,
4155 /* CHANNEL * rp, */ void *arg1,
4156 /* IOCBQ * iocbq, */ void *arg2,
4157 /* MATCHMAP * mp, */ void *arg3,
4158 /* NODELIST * ndlp */ void *arg4,
4159 uint32_t evt)
4160 {
4161         NODELIST *ndlp = (NODELIST *)arg4;
4162         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

4164         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4165             "emlxs_device_recov_auth: 0x%x.",
4166             ndlp->nlp_DID);

4168         node_dhc->nlp_authrsp_tmo = 0;

4170         emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED, 0, 0);

4172         return (node_dhc->state);

4174 } /* emlxs_device_recov_auth */



4178 /*
4179  * This routine is invoked when the host as the responder sent out the
4180  * ELS DHCHAP_Success to the initiator, the initiator ACC
4181  * it. AND then the host received an unsolicited auth msg from the initiator,
4182  * this msg is supposed to be the ELS DHCHAP_Success
4183  * msg for the bi-directional authentication.
4184  *
4185  * next state should be REG_LOGIN
4186  */
4187 /* ARGSUSED */
4188 static uint32_t
4189 emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next(
4190 emlxs_port_t *port,
4191 /* CHANNEL * rp, */ void *arg1,
4192 /* IOCBQ * iocbq, */ void *arg2,
4193 /* MATCHMAP * mp, */ void *arg3,
4194 /* NODELIST * ndlp */ void *arg4,
4195 uint32_t evt)
4196 {
4197         IOCBQ *iocbq = (IOCBQ *)arg2;
4198         MATCHMAP *mp = (MATCHMAP *)arg3;
4199         NODELIST *ndlp = (NODELIST *)arg4;
4200         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
4201         uint8_t *bp;
4202         uint32_t *lp;
4203         DHCHAP_SUCCESS_HDR *dh_success;
4204         AUTH_RJT *auth_rjt;
4205         uint8_t ReasonCode;
4206         uint8_t ReasonCodeExplanation;

4208         bp = mp->virt;
4209         lp = (uint32_t *)bp;

4211         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4212             "emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next: did=0x%x",
4213             ndlp->nlp_DID);

4215         dh_success = (DHCHAP_SUCCESS_HDR *)((uint8_t *)lp);

4217         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

4219         if (dh_success->auth_msg_code == AUTH_REJECT) {
```

```
4220                /* ACC it and retry etc.  */
4221                auth_rjt = (AUTH_RJT *)dh_success;
4222                ReasonCode = auth_rjt->ReasonCode;
4223                ReasonCodeExplanation = auth_rjt->ReasonCodeExplanation;

4225        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4226            "rcv_a_m_dhch_success_cmpl_wait4next:REJECT rvd. 0x%x 0x%x 0x%x",
4227            ndlp->nlp_DID, ReasonCode, ReasonCodeExplanation);

4229                switch (ReasonCode) {
4230                case AUTHRJT_LOGIC_ERR:
4231                        switch (ReasonCodeExplanation) {
4232                        case AUTHEXP_MECH_UNUSABLE:
4233                        case AUTHEXP_DHGROUP_UNUSABLE:
4234                        case AUTHEXP_HASHFUNC_UNUSABLE:
4235                                ReasonCode = AUTHRJT_LOGIC_ERR;
4236                                ReasonCodeExplanation = AUTHEXP_RESTART_AUTH;
4237                                break;

4239                        case AUTHEXP_RESTART_AUTH:
4240                                /*
4241                                 * Cancel the rsp timer if not cancelled yet.
4242                                 * and restart auth tran now.
4243                                 */
4244                                if (node_dhc->nlp_authrsp_tmo != 0) {
4245                                        node_dhc->nlp_authrsp_tmo = 0;
4246                                        node_dhc->nlp_authrsp_tmocnt = 0;
4247                                }
4248                                if (emlxs_dhc_auth_start(port, ndlp,
4249                                    NULL, NULL) != 0) {
4250                                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
4251                                    "Reauth timeout. Auth initfailed. 0x%x %x",
4252                                    ndlp->nlp_DID, node_dhc->state);
4253                                }
4254                                return (node_dhc->state);

4256                        default:
4257                                ReasonCode = AUTHRJT_FAILURE;
4258                                ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
4259                                break;

4261                        }
4262                        break;

4264                case AUTHRJT_FAILURE:
4265                default:
4266                        ReasonCode = AUTHRJT_FAILURE;
4267                        ReasonCodeExplanation = AUTHEXP_AUTH_FAILED;
4268                        break;

4270                }

4272                goto Reject;

4274        } else if (dh_success->auth_msg_code == DHCHAP_SUCCESS) {
4275                if (LE_SWAP32(dh_success->tran_id) !=
4276                    node_dhc->nlp_auth_tranid_ini) {
4277        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4278            "rcv_a_m_dhch_success_cmpl_wait4next: 0x%x 0x%lx, 0x%lx",
4279            ndlp->nlp_DID, dh_success->tran_id, node_dhc->nlp_auth_tranid_ini);

4281                        ReasonCode = AUTHRJT_FAILURE;
4282                        ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;
4283                        goto Reject;
4284                }
4285                node_dhc->flag |= NLP_REMOTE_AUTH;
```



```
4287                emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_SUCCESS, 0, 0);
4288                emlxs_log_auth_event(port, ndlp, ESC_EMLXS_26,
4289                    "Node-initiated-bidir-reauth-success");
4290                emlxs_dhc_auth_complete(port, ndlp, 0);
4291        } else {
4292                ReasonCode = AUTHRJT_FAILURE;
4293                ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;
4294                goto Reject;
4295        }

4297        return (node_dhc->state);

4299 Reject:

4301        emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED,
4302            ReasonCode, ReasonCodeExplanation);
4303        (void) emlxs_issue_auth_reject(port, ndlp, 0, 0, ReasonCode,
4304            ReasonCodeExplanation);
4305        emlxs_dhc_auth_complete(port, ndlp, 1);

4307 out:

4309        return (node_dhc->state);

4311 } /* emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next */


4314 /* ARGSUSED */
4315 static uint32_t
4316 emlxs_cmpl_auth_msg_dhchap_success_cmpl_wait4next(
4317 emlxs_port_t *port,
4318 /* CHANNEL * rp, */ void *arg1,
4319 /* IOCBQ * iocbq, */ void *arg2,
4320 /* MATCHMAP * mp, */ void *arg3,
4321 /* NODELIST * ndlp */ void *arg4,
4322 uint32_t evt)
4323 {

4325        return (0);

4327 } /* emlxs_cmpl_auth_msg_dhchap_success_cmpl_wait4next */


4330 /* ARGSUSED */
4331 static uint32_t
4332 emlxs_rcv_auth_msg_auth_negotiate_rcv(
4333 emlxs_port_t *port,
4334 /* CHANNEL * rp, */ void *arg1,
4335 /* IOCBQ * iocbq, */ void *arg2,
4336 /* MATCHMAP * mp, */ void *arg3,
4337 /* NODELIST * ndlp */ void *arg4,
4338 uint32_t evt)
4339 {
4340        NODELIST *ndlp = (NODELIST *)arg4;

4342        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4343            "rcv_a_m_auth_negotiate_rcv called. did=0x%x. Not implemented.",
4344            ndlp->nlp_DID);

4346        return (0);

4348 } /* emlxs_rcv_auth_msg_auth_negotiate_rcv */


4351 /* ARGSUSED */
```

```
4352 static uint32_t
4353 emlxs_rcv_auth_msg_npr_node(
4354 emlxs_port_t *port,
4355 /* CHANNEL  * rp, */ void *arg1,
4356 /* IOCBQ * iocbq, */ void *arg2,
4357 /* MATCHMAP * mp, */ void *arg3,
4358 /* NODELIST * ndlp */ void *arg4,
4359         uint32_t evt)
4360 {
4361         IOCBQ *iocbq = (IOCBQ *)arg2;
4362         MATCHMAP *mp = (MATCHMAP *)arg3;
4363         NODELIST *ndlp = (NODELIST *)arg4;
4364         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
4365         uint8_t *bp;

4367         uint32_t *lp;
4368         uint32_t msglen;
4369         uint8_t *tmp;

4371         AUTH_MSG_HDR *msg;

4373         uint8_t *temp;
4374         uint32_t rc, i, hs_id[2], dh_id[5];
4375                                         /* from initiator */
4376         uint32_t hash_id, dhgp_id;      /* to be used by responder */
4377         uint16_t num_hs = 0;
4378         uint16_t num_dh = 0;

4380         bp = mp->virt;
4381         lp = (uint32_t *)bp;

4383         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4384             "emlxs_rcv_auth_msg_npr_node:");

4386         /*
4387          * 1. process the auth msg, should acc first no matter what. 2.
4388          * return DHCHAP_Challenge for AUTH_Negotiate auth msg, AUTH_Reject
4389          * for anything else.
4390          */
4391         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

4393         msg = (AUTH_MSG_HDR *)((uint8_t *)lp);
4394         msglen = msg->msg_len;
4395         tmp = ((uint8_t *)lp);

4397         /* temp is used for error checking */
4398         temp = (uint8_t *)((uint8_t *)lp);
4399         /* Check the auth_els_code */
4400         if (((*(uint32_t *)temp) & 0xFFFFFFFF) != LE_SWAP32(0x90000B01)) {
4401                 /* ReasonCode = AUTHRJT_FAILURE; */
4402                 /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */

4404                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4405                     "emlxs_rcv_auth_msg_npr_node: payload(1)=0x%x",
4406                     (*(uint32_t *)temp));

4408                 goto AUTH_Reject;
4409         }
4410         temp += 3 * sizeof (uint32_t);
4411         /* Check name tag and name length */
4412         if (((*(uint32_t *)temp) & 0xFFFFFFFF) != LE_SWAP32(0x00010008)) {
4413                 /* ReasonCode = AUTHRJT_FAILURE; */
4414                 /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */

4416                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4417                     "emlxs_rcv_auth_msg_npr_node: payload(2)=0x%x",
```



```
4418                     (*(uint32_t *)temp));

4420                 goto AUTH_Reject;
4421         }
4422         temp += sizeof (uint32_t) + 8;
4423         /* Check proto_num */
4424         if (((*(uint32_t *)temp) & 0xFFFFFFFF) != LE_SWAP32(0x00000001)) {
4425                 /* ReasonCode = AUTHRJT_FAILURE; */
4426                 /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */

4428                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4429                     "emlxs_rcv_auth_msg_npr_node: payload(3)=0x%x",
4430                     (*(uint32_t *)temp));

4432                 goto AUTH_Reject;
4433         }
4434         temp += sizeof (uint32_t);
4435         /* Get para_len */
4436         /* para_len = LE_SWAP32(*(uint32_t *)temp); */

4438         temp += sizeof (uint32_t);
4439         /* Check proto_id */
4440         if (((*(uint32_t *)temp) & 0xFFFFFFFF) != AUTH_DHCHAP) {
4441                 /* ReasonCode = AUTHRJT_FAILURE; */
4442                 /* ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL; */

4444                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4445                     "emlxs_rcv_auth_msg_npr_node: payload(4)=0x%x",
4446                     (*(uint32_t *)temp));

4448                 goto AUTH_Reject;
4449         }
4450         temp += sizeof (uint32_t);
4451         /* Check hashlist tag */
4452         if ((LE_SWAP32(*(uint32_t *)temp) & 0xFFFF0000) >> 16 !=
4453             LE_SWAP16(HASH_LIST_TAG)) {
4454                 /* ReasonCode = AUTHRJT_FAILURE; */
4455                 /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */

4457                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4458                     "emlxs_rcv_auth_msg_npr_node: payload(5)=0x%x",
4459                     (LE_SWAP32(*(uint32_t *)temp) & 0xFFFF0000) >> 16);

4461                 goto AUTH_Reject;
4462         }
4463         /* Get num_hs  */
4464         num_hs = LE_SWAP32(*(uint32_t *)temp) & 0x0000FFFF;

4466         temp += sizeof (uint32_t);
4467         /* Check HashList_value1 */
4468         hs_id[0] = *(uint32_t *)temp;

4470         if ((hs_id[0] != AUTH_MD5) && (hs_id[0] != AUTH_SHA1)) {
4471                 /* ReasonCode = AUTHRJT_LOGIC_ERR; */
4472                 /* ReasonCodeExplanation = AUTHEXP_HASHFUNC_UNUSABLE; */

4474                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4475                     "emlxs_rcv_auth_msg_npr_node: payload(6)=0x%x",
4476                     (*(uint32_t *)temp));

4478                 goto AUTH_Reject;
4479         }
4480         if (num_hs == 1) {
4481                 hs_id[1] = 0;
4482         } else if (num_hs == 2) {
4483                 temp += sizeof (uint32_t);
```

```
4484                hs_id[1] = *(uint32_t *)temp;
4485
4486                if ((hs_id[1] != AUTH_MD5) && (hs_id[1] != AUTH_SHA1)) {
4487                        /* ReasonCode = AUTHRJT_LOGIC_ERR; */
4488                        /* ReasonCodeExplanation = AUTHEXP_HASHFUNC_UNUSABLE; */
4489
4490                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4491                            "emlxs_rcv_auth_msg_npr_node: payload(7)=0x%x",
4492                            (*(uint32_t *)temp));
4493
4494                        goto AUTH_Reject;
4495                }
4496                if (hs_id[0] == hs_id[1]) {
4497                        /* ReasonCode = AUTHRJT_FAILURE; */
4498                        /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */
4499
4500                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4501                            "emlxs_rcv_auth_msg_npr_node: payload(8)=0x%x",
4502                            (*(uint32_t *)temp));
4503
4504                        goto AUTH_Reject;
4505                }
4506        } else {
4507                /* ReasonCode = AUTHRJT_FAILURE; */
4508                /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */
4509
4510                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4511                    "emlxs_rcv_auth_msg_npr_node: payload(9)=0x%x",
4512                    (*(uint32_t *)(temp - sizeof (uint32_t))));
4513
4514                goto AUTH_Reject;
4515        }
4516
4517        /* Which hash_id should we use */
4518        if (num_hs == 1) {
4519                /*
4520                 * We always use the highest priority specified by us if we
4521                 * match initiator's , Otherwise, we use the next higher we
4522                 * both have. CR 26238
4523                 */
4524                if (node_dhc->auth_cfg.hash_priority[0] == hs_id[0]) {
4525                        hash_id = node_dhc->auth_cfg.hash_priority[0];
4526                } else if (node_dhc->auth_cfg.hash_priority[1] == hs_id[0]) {
4527                        hash_id = node_dhc->auth_cfg.hash_priority[1];
4528                } else {
4529                        /* ReasonCode = AUTHRJT_LOGIC_ERR; */
4530                        /* ReasonCodeExplanation = AUTHEXP_HASHFUNC_UNUSABLE; */
4531
4532                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4533                            "emlxs_rcv_auth_msg_npr_node: payload(10)=0x%lx",
4534                            (*(uint32_t *)temp));
4535
4536                        goto AUTH_Reject;
4537                }
4538        } else {
4539                /*
4540                 * Since the initiator specified two hashs, we always select
4541                 * our first one.
4542                 */
4543                hash_id = node_dhc->auth_cfg.hash_priority[0];
4544        }
4545
4546        temp += sizeof (uint32_t);
4547        /* Check DHgIDList_tag */
4548        if ((LE_SWAP32(*(uint32_t *)temp) & 0xFFFF0000) >> 16 !=
4549            LE_SWAP16(DHGID_LIST_TAG)) {
```



```
4550                /* ReasonCode = AUTHRJT_FAILURE; */
4551                /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */
4552
4553                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4554                    "emlxs_rcv_auth_msg_npr_node: payload(11)=0x%lx",
4555                    (*(uint32_t *)temp));
4556
4557                goto AUTH_Reject;
4558        }
4559        /* Get num_dh */
4560        num_dh = LE_SWAP32(*(uint32_t *)temp) & 0x0000FFFF;
4561
4562        if (num_dh == 0) {
4563                /* ReasonCode = AUTHRJT_FAILURE; */
4564                /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */
4565
4566                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4567                    "emlxs_rcv_auth_msg_npr_node: payload(12)=0x%lx",
4568                    (*(uint32_t *)temp));
4569
4570                goto AUTH_Reject;
4571        }
4572        for (i = 0; i < num_dh; i++) {
4573                temp += sizeof (uint32_t);
4574                /* Check DHgIDList_g0 */
4575                dh_id[i] = (*(uint32_t *)temp);
4576        }
4577
4578        rc = emlxs_check_dhgp(port, ndlp, dh_id, num_dh, &dhgp_id);
4579
4580        if (rc == 1) {
4581                /* ReasonCode = AUTHRJT_LOGIC_ERR; */
4582                /* ReasonCodeExplanation = AUTHEXP_DHGROUP_UNUSABLE; */
4583
4584                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4585                    "emlxs_rcv_auth_msg_npr_node: payload(13)=0x%lx",
4586                    (*(uint32_t *)temp));
4587
4588                goto AUTH_Reject;
4589        } else if (rc == 2) {
4590                /* ReasonCode = AUTHRJT_FAILURE; */
4591                /* ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD; */
4592
4593                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
4594                    "emlxs_rcv_auth_msg_npr_node: payload(14)=0x%lx",
4595                    (*(uint32_t *)temp));
4596
4597                goto AUTH_Reject;
4598        }
4599        /* We should update the tran_id */
4600        node_dhc->nlp_auth_tranid_ini = msg->tran_id;
4601
4602        if (msg->auth_msg_code == AUTH_NEGOTIATE) {
4603                node_dhc->nlp_auth_flag = 1;    /* ndlp is the initiator */
4604
4605                /* Send back the DHCHAP_Challenge with the proper paramaters */
4606                if (emlxs_issue_dhchap_challenge(port, ndlp, 0, tmp,
4607                    LE_SWAP32(msglen),
4608                    hash_id, dhgp_id)) {
4609                        goto AUTH_Reject;
4610                }
4611                emlxs_dhc_state(port, ndlp,
4612                    NODE_STATE_DHCHAP_CHALLENGE_ISSUE, 0, 0);
4613
4614        } else {
4615                goto AUTH_Reject;
```

```
4616         }

4618         return (node_dhc->state);

4620 AUTH_Reject:

4622         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4623             "emlxs_rcv_auth_msg_npr_node: AUTH_Reject it.");

4625         return (node_dhc->state);

4627 } /* emlxs_rcv_auth_msg_npr_node */


4630 /* ARGSUSED */
4631 static uint32_t
4632 emlxs_cmpl_auth_msg_npr_node(
4633 emlxs_port_t *port,
4634 /* CHANNEL * rp, */ void *arg1,
4635 /* IOCBQ * iocbq, */ void *arg2,
4636 /* MATCHMAP * mp, */ void *arg3,
4637 /* NODELIST * ndlp */ void *arg4,
4638 uint32_t evt)
4639 {
4640         NODELIST *ndlp = (NODELIST *)arg4;
4641         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

4643         /*
4644          * we donot cancel the nodev timeout here because we donot know if we
4645          * can get the authentication restarted from other side once we got
4646          * the new auth transaction kicked off we cancel nodev tmo
4647          * immediately.
4648          */
4649         /* we goto change the hba state back to where it used to be */
4650         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4651             "emlxs_cmpl_auth_msg_npr_node: 0x%x 0x%x prev_state=0x%x\n",
4652             ndlp->nlp_DID, node_dhc->state, node_dhc->prev_state);

4654         return (node_dhc->state);

4656 } /* emlxs_cmpl_auth_msg_npr_node */


4659 /*
4660  * ! emlxs_rcv_auth_msg_unmapped_node
4661  *
4662  * \pre \post \param   phba \param   ndlp \param   arg \param   evt \return
4663  * uint32_t
4664  *
4665  * \b Description: This routine is invoked when the host received an
4666  * unsolicited els authentication msg from the Fx_Port which is
4667  * wellknown port 0xFFFFFE in unmapped state, or from Nx_Port which is
4668  * in the unmapped state meaning that it is either a target
4669  * which there is no scsi id associated with it or it could be another
4670  * initiator. (end-to-end)
4671  *
4672  * For the Fabric F_Port (FFFFFE) we mark the port to the state in re_auth
4673  * state without disruppting the traffic. Then the fabric
4674  * will go through the authentication processes until it is done.
4675  *
4676  * most of the cases, the fabric should send us AUTH_Negotiate ELS msg. Once
4677  * host received this auth_negotiate els msg, host
4678  * should sent back ACC first and then send random challenge, plus DH value
4679  * (i.e., host's publick key)
4680  *
4681  * Host side needs to store the challenge value and public key for later
```



```
4682  * verification usage. (i.e., to verify the response from
4683  * initiator)
4684  *
4685  * If two FC_Ports start the reauthentication transaction at the same time,
4686  * one of the two authentication transactions shall be
4687  * aborted. In case of Host and Fabric the Nx_Port shall remain the
4688  * authentication initiator, while the Fx_Port shall become
4689  * the authentication responder.
4690  *
4691  */
4692 /* ARGSUSED */
4693 static uint32_t
4694 emlxs_rcv_auth_msg_unmapped_node(
4695 emlxs_port_t *port,
4696 /* CHANNEL * rp, */ void *arg1,
4697 /* IOCBQ * iocbq, */ void *arg2,
4698 /* MATCHMAP * mp, */ void *arg3,
4699 /* NODELIST * ndlp */ void *arg4,
4700         uint32_t evt)
4701 {
4702         IOCBQ *iocbq = (IOCBQ *)arg2;
4703         MATCHMAP *mp = (MATCHMAP *)arg3;
4704         NODELIST *ndlp = (NODELIST *)arg4;
4705         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
4706         uint8_t *bp;
4707         uint32_t *lp;
4708         uint32_t msglen;
4709         uint8_t *tmp;

4711         uint8_t ReasonCode;
4712         uint8_t ReasonCodeExplanation;
4713         AUTH_MSG_HDR *msg;
4714         uint8_t *temp;
4715         uint32_t rc, i, hs_id[2], dh_id[5];
4716                                         /* from initiator */
4717         uint32_t hash_id, dhgp_id;      /* to be used by responder */
4718         uint16_t num_hs = 0;
4719         uint16_t num_dh = 0;

4721         /*
4722          * 1. process the auth msg, should acc first no matter what. 2.
4723          * return DHCHAP_Challenge for AUTH_Negotiate auth msg, AUTH_Reject
4724          * for anything else.
4725          */
4726         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4727             "emlxs_rcv_auth_msg_unmapped_node: Sending ACC: did=0x%x",
4728             ndlp->nlp_DID);

4730         (void) emlxs_els_reply(port, iocbq, ELS_CMD_ACC, ELS_CMD_AUTH, 0, 0);

4732         bp = mp->virt;
4733         lp = (uint32_t *)bp;

4735         msg = (AUTH_MSG_HDR *)((uint8_t *)lp);
4736         msglen = msg->msg_len;

4738         tmp = ((uint8_t *)lp);

4740         /* temp is used for error checking */
4741         temp = (uint8_t *)((uint8_t *)lp);
4742         /* Check the auth_els_code */
4743         if (((*(uint32_t *)temp) & 0xFFFFFFFF) != LE_SWAP32(0x90000B01)) {
4744                 ReasonCode = AUTHRJT_FAILURE;
4745                 ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4747                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
```

```
4748                    "emlxs_rcv_auth_msg_unmapped_node: payload(1)=0x%x",
4749                    (*(uint32_t *)temp));

4751                goto AUTH_Reject;
4752        }
4753        temp += 3 * sizeof (uint32_t);
4754        /* Check name tag and name length */
4755        if (((*(uint32_t *)temp) & 0xFFFFFFFF) != LE_SWAP32(0x00010008)) {
4756                ReasonCode = AUTHRJT_FAILURE;
4757                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4759                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4760                    "emlxs_rcv_auth_msg_unmapped_node: payload(2)=0x%x",
4761                    (*(uint32_t *)temp));

4763                goto AUTH_Reject;
4764        }
4765        temp += sizeof (uint32_t) + 8;
4766        /* Check proto_num */
4767        if (((*(uint32_t *)temp) & 0xFFFFFFFF) != LE_SWAP32(0x00000001)) {
4768                ReasonCode = AUTHRJT_FAILURE;
4769                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4771                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4772                    "emlxs_rcv_auth_msg_unmapped_node: payload(3)=0x%x",
4773                    (*(uint32_t *)temp));

4775                goto AUTH_Reject;
4776        }
4777        temp += sizeof (uint32_t);

4779        /* Get para_len */
4780        /* para_len = *(uint32_t *)temp; */
4781        temp += sizeof (uint32_t);

4783        /* Check proto_id */
4784        if (((*(uint32_t *)temp) & 0xFFFFFFFF) != AUTH_DHCHAP) {
4785                ReasonCode = AUTHRJT_FAILURE;
4786                ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;

4788                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4789                    "emlxs_rcv_auth_msg_unmapped_node: payload(4)=0x%x",
4790                    (*(uint32_t *)temp));

4792                goto AUTH_Reject;
4793        }
4794        temp += sizeof (uint32_t);
4795        /* Check hashlist tag */
4796        if ((LE_SWAP32(*(uint32_t *)temp) & 0xFFFF0000) >> 16 !=
4797            LE_SWAP16(HASH_LIST_TAG)) {
4798                ReasonCode = AUTHRJT_FAILURE;
4799                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4801                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4802                    "emlxs_rcv_auth_msg_unmapped_node: payload(5)=0x%x",
4803                    (LE_SWAP32(*(uint32_t *)temp) & 0xFFFF0000) >> 16);

4805                goto AUTH_Reject;
4806        }
4807        /* Get num_hs  */
4808        num_hs = LE_SWAP32(*(uint32_t *)temp) & 0x0000FFFF;

4810        temp += sizeof (uint32_t);
4811        /* Check HashList_value1 */
4812        hs_id[0] = *(uint32_t *)temp;
```



```
4814        if ((hs_id[0] != AUTH_MD5) && (hs_id[0] != AUTH_SHA1)) {
4815                ReasonCode = AUTHRJT_LOGIC_ERR;
4816                ReasonCodeExplanation = AUTHEXP_HASHFUNC_UNUSABLE;

4818                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4819                    "emlxs_rcv_auth_msg_unmapped_node: payload(6)=0x%x",
4820                    (*(uint32_t *)temp));

4822                goto AUTH_Reject;
4823        }
4824        if (num_hs == 1) {
4825                hs_id[1] = 0;
4826        } else if (num_hs == 2) {
4827                temp += sizeof (uint32_t);
4828                hs_id[1] = *(uint32_t *)temp;

4830                if ((hs_id[1] != AUTH_MD5) && (hs_id[1] != AUTH_SHA1)) {
4831                        ReasonCode = AUTHRJT_LOGIC_ERR;
4832                        ReasonCodeExplanation = AUTHEXP_HASHFUNC_UNUSABLE;

4834                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4835                            "emlxs_rcv_auth_msg_unmapped_node: payload(7)=0x%x",
4836                            (*(uint32_t *)temp));

4838                        goto AUTH_Reject;
4839                }
4840                if (hs_id[0] == hs_id[1]) {
4841                        ReasonCode = AUTHRJT_FAILURE;
4842                        ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4844                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4845                            "emlxs_rcv_auth_msg_unmapped_node: payload(8)=0x%x",
4846                            (*(uint32_t *)temp));

4848                        goto AUTH_Reject;
4849                }
4850        } else {
4851                ReasonCode = AUTHRJT_FAILURE;
4852                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4854                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4855                    "emlxs_rcv_auth_msg_unmapped_node: payload(9)=0x%x",
4856                    (*(uint32_t *)(temp - sizeof (uint32_t))));

4858                goto AUTH_Reject;
4859        }

4861        /* Which hash_id should we use */
4862        if (num_hs == 1) {
4863                /*
4864                 * We always use the highest priority specified by us if we
4865                 * match initiator's , Otherwise, we use the next higher we
4866                 * both have. CR 26238
4867                 */
4868                if (node_dhc->auth_cfg.hash_priority[0] == hs_id[0]) {
4869                        hash_id = node_dhc->auth_cfg.hash_priority[0];
4870                } else if (node_dhc->auth_cfg.hash_priority[1] == hs_id[0]) {
4871                        hash_id = node_dhc->auth_cfg.hash_priority[1];
4872                } else {
4873                        ReasonCode = AUTHRJT_LOGIC_ERR;
4874                        ReasonCodeExplanation = AUTHEXP_HASHFUNC_UNUSABLE;

4876                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4877                            "emlxs_rcv_auth_msg_unmapped_node: pload(10)=0x%x",
4878                            (*(uint32_t *)temp));
```

```
4880                        goto AUTH_Reject;
4881                }
4882        } else {
4883                /*
4884                 * Since the initiator specified two hashs, we always select
4885                 * our first one.
4886                 */
4887                hash_id = node_dhc->auth_cfg.hash_priority[0];
4888        }

4890        temp += sizeof (uint32_t);
4891        /* Check DHgIDList_tag */
4892        if ((LE_SWAP32(*(uint32_t *)temp) & 0xFFFF0000) >> 16 !=
4893            LE_SWAP16(DHGID_LIST_TAG)) {
4894                ReasonCode = AUTHRJT_FAILURE;
4895                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4897                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4898                    "emlxs_rcv_auth_msg_unmapped_node: payload(11)=0x%x",
4899                    (*(uint32_t *)temp));

4901                goto AUTH_Reject;
4902        }
4903        /* Get num_dh */
4904        num_dh = LE_SWAP32(*(uint32_t *)temp) & 0x0000FFFF;

4906        if (num_dh == 0) {
4907                ReasonCode = AUTHRJT_FAILURE;
4908                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4910                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4911                    "emlxs_rcv_auth_msg_unmapped_node: payload(12)=0x%x",
4912                    (*(uint32_t *)temp));

4914                goto AUTH_Reject;
4915        }
4916        for (i = 0; i < num_dh; i++) {
4917                temp += sizeof (uint32_t);
4918                /* Check DHgIDList_g0 */
4919                dh_id[i] = (*(uint32_t *)temp);
4920        }

4922        rc = emlxs_check_dhgp(port, ndlp, dh_id, num_dh, &dhgp_id);

4924        if (rc == 1) {
4925                ReasonCode = AUTHRJT_LOGIC_ERR;
4926                ReasonCodeExplanation = AUTHEXP_DHGROUP_UNUSABLE;

4928                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4929                    "emlxs_rcv_auth_msg_unmapped_node: payload(13)=0x%x",
4930                    (*(uint32_t *)temp));

4932                goto AUTH_Reject;
4933        } else if (rc == 2) {
4934                ReasonCode = AUTHRJT_FAILURE;
4935                ReasonCodeExplanation = AUTHEXP_BAD_PAYLOAD;

4937                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4938                    "emlxs_rcv_auth_msg_unmapped_node: payload(14)=0x%x",
4939                    (*(uint32_t *)temp));

4941                goto AUTH_Reject;
4942        }
4943        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4944            "emlxs_rcv_auth_msg_unmapped_node: 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x",
4945            hash_id, dhgp_id, msg->auth_msg_code, msglen, msg->tran_id);
```



```
4947        /*
4948         * since ndlp is the initiator, tran_id is store in
4949         * nlp_auth_tranid_ini
4950         */
4951        node_dhc->nlp_auth_tranid_ini = LE_SWAP32(msg->tran_id);

4953        if (msg->auth_msg_code == AUTH_NEGOTIATE) {

4955                /*
4956                 * at this point, we know for sure we received the
4957                 * auth-negotiate msg from another entity, so cancel the
4958                 * auth-rsp timeout timer if we are expecting it. should
4959                 * never happen?
4960                 */
4961                node_dhc->nlp_auth_flag = 1;

4963                if (node_dhc->nlp_authrsp_tmo) {
4964                        node_dhc->nlp_authrsp_tmo = 0;
4965                }
4966                /*
4967                 * If at this point, the host is doing reauthentication
4968                 * (reauth heart beat) to this ndlp, then Host should remain
4969                 * as the auth initiator, host should reply to the received
4970                 * AUTH_Negotiate message with an AUTH_Reject message with
4971                 * Reason Code 'Logical Error' and Reason Code Explanation
4972                 * 'Authentication Transaction Already Started'.
4973                 */
4974                if (node_dhc->nlp_reauth_status ==
4975                    NLP_HOST_REAUTH_IN_PROGRESS) {
4976                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
4977                    "emlxs_rcv_auth_msg_unmapped_node: Ht reauth inprgress.");

4979                        ReasonCode = AUTHRJT_LOGIC_ERR;
4980                        ReasonCodeExplanation = AUTHEXP_AUTHTRAN_STARTED;

4982                        goto AUTH_Reject;
4983                }
4984                /* Send back the DHCHAP_Challenge with the proper paramaters */
4985                if (emlxs_issue_dhchap_challenge(port, ndlp, 0, tmp,
4986                    LE_SWAP32(msglen),
4987                    hash_id, dhgp_id)) {

4989                        goto AUTH_Reject;
4990                }
4991                /* setup the proper state */
4992                emlxs_dhc_state(port, ndlp,
4993                    NODE_STATE_DHCHAP_CHALLENGE_ISSUE, 0, 0);

4995        } else {
4996                ReasonCode = AUTHRJT_FAILURE;
4997                ReasonCodeExplanation = AUTHEXP_BAD_PROTOCOL;

4999                goto AUTH_Reject;
5000        }

5002        return (node_dhc->state);

5004 AUTH_Reject:

5006        emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED,
5007            ReasonCode, ReasonCodeExplanation);
5008        (void) emlxs_issue_auth_reject(port, ndlp, 0, 0, ReasonCode,
5009            ReasonCodeExplanation);
5010        emlxs_dhc_auth_complete(port, ndlp, 1);
```

```
5012         return (node_dhc->state);

5014 } /* emlxs_rcv_auth_msg_unmapped_node */




5019 /*
5020  * emlxs_hash_vrf for verification only the host is the initiator in
5021  * the routine.
5022  */
5023 /* ARGSUSED */
5024 static uint32_t *
5025 emlxs_hash_vrf(
5026         emlxs_port_t *port,
5027         emlxs_port_dhc_t *port_dhc,
5028         NODELIST *ndlp,
5029         uint32_t tran_id,
5030         union challenge_val un_cval)
5031 {
5032         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
5033         uint32_t dhgp_id;
5034         uint32_t hash_id;
5035         uint32_t *hash_val;
5036         uint32_t hash_size;
5037         MD5_CTX mdctx;
5038         SHA1_CTX sha1ctx;
5039         uint8_t sha1_digest[20];
5040         uint8_t md5_digest[16];
5041         uint8_t mytran_id = 0x00;

5043         char *remote_key;

5045         tran_id = (AUTH_TRAN_ID_MASK & tran_id);
5046         mytran_id = (uint8_t)(LE_SWAP32(tran_id));


5049         if (ndlp->nlp_DID == FABRIC_DID) {
5050                 remote_key = (char *)node_dhc->auth_key.remote_password;
5051                 hash_id = node_dhc->hash_id;
5052                 dhgp_id = node_dhc->dhgp_id;
5053         } else {
5054                 remote_key = (char *)node_dhc->auth_key.remote_password;
5055                 hash_id = node_dhc->nlp_auth_hashid;
5056                 dhgp_id = node_dhc->nlp_auth_dhgpid;
5057         }

5059         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
5060             "emlxs_hash_vrf: 0x%x 0x%x 0x%x tran_id=0x%x",
5061             ndlp->nlp_DID, hash_id, dhgp_id, mytran_id);

5063         if (dhgp_id == 0) {
5064                 /* NULL DHCHAP */
5065                 if (hash_id == AUTH_MD5) {
5066                         bzero(&mdctx, sizeof (MD5_CTX));

5068                         hash_size = MD5_LEN;

5070                         MD5Init(&mdctx);

5072                         /* Transaction Identifier T */
5073                         MD5Update(&mdctx, (unsigned char *) &mytran_id, 1);

5075                         MD5Update(&mdctx, (unsigned char *) remote_key,
5076                             node_dhc->auth_key.remote_password_length);
```



```
5078                         /* Augmented challenge: NULL DHCHAP i.e., Challenge */
5079                         MD5Update(&mdctx,
5080                             (unsigned char *)&(un_cval.md5.val[0]), MD5_LEN);

5082                         MD5Final((uint8_t *)md5_digest, &mdctx);

5084                         hash_val = (uint32_t *)kmem_alloc(hash_size,
5085                             KM_NOSLEEP);
5086                         if (hash_val == NULL) {
5087                                 return (NULL);
5088                         } else {
5089                                 bcopy((void *)&md5_digest,
5090                                     (void *)hash_val, MD5_LEN);
5091                         }
5092                         /*
5093                          * emlxs_md5_digest_to_hex((uint8_t *)hash_val,
5094                          * output);
5095                          */
5096                 }
5097                 if (hash_id == AUTH_SHA1) {
5098                         bzero(&sha1ctx, sizeof (SHA1_CTX));
5099                         hash_size = SHA1_LEN;
5100                         SHA1Init(&sha1ctx);

5102                         SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

5104                         SHA1Update(&sha1ctx, (void *)remote_key,
5105                             node_dhc->auth_key.remote_password_length);

5107                         SHA1Update(&sha1ctx,
5108                             (void *)&(un_cval.sha1.val[0]), SHA1_LEN);

5110                         SHA1Final((void *)sha1_digest, &sha1ctx);

5112                         /*
5113                          * emlxs_sha1_digest_to_hex((uint8_t *)hash_val,
5114                          * output);
5115                          */

5117                         hash_val = (uint32_t *)kmem_alloc(hash_size,
5118                             KM_NOSLEEP);
5119                         if (hash_val == NULL) {
5120                                 return (NULL);
5121                         } else {
5122                                 bcopy((void *)&sha1_digest,
5123                                     (void *)hash_val, SHA1_LEN);
5124                         }
5125                 }
5126                 return ((uint32_t *)hash_val);
5127         } else {
5128                 /* Verification of bi-dir auth for DH-CHAP group */
5129                 /* original challenge is node_dhc->bi_cval[] */
5130                 /* session key is node_dhc->ses_key[] */
5131                 /* That's IT */
5132                 /*
5133                  * H(bi_cval || ses_key) = C H(Ti || Km || C)  = hash_val
5134                  */
5135                 if (hash_id == AUTH_MD5) {
5136                         bzero(&mdctx, sizeof (MD5_CTX));
5137                         hash_size = MD5_LEN;

5139                         MD5Init(&mdctx);

5141                         MD5Update(&mdctx,
5142                             (void *)&(un_cval.md5.val[0]), MD5_LEN);
```

```
5144                        if (ndlp->nlp_DID == FABRIC_DID) {
5145                                MD5Update(&mdctx,
5146                                    (void *)&node_dhc->ses_key[0],
5147                                    node_dhc->seskey_len);
5148                        } else {
5149                                /* ses_key is obtained in emlxs_hash_rsp */
5150                                MD5Update(&mdctx,
5151                                    (void *)&node_dhc->nlp_auth_misc.ses_key[0],
5152                                    node_dhc->nlp_auth_misc.seskey_len);
5153                        }

5155                        MD5Final((void *)md5_digest, &mdctx);

5157                        MD5Init(&mdctx);

5159                        MD5Update(&mdctx, (void *)&mytran_id, 1);

5161                        MD5Update(&mdctx, (void *)remote_key,
5162                            node_dhc->auth_key.remote_password_length);

5164                        MD5Update(&mdctx, (void *)md5_digest, MD5_LEN);

5166                        MD5Final((void *)md5_digest, &mdctx);

5168                        hash_val = (uint32_t *)kmem_alloc(hash_size,
5169                            KM_NOSLEEP);
5170                        if (hash_val == NULL) {
5171                                return (NULL);
5172                        } else {
5173                                bcopy((void *)&md5_digest,
5174                                    (void *)hash_val, MD5_LEN);
5175                        }
5176                }
5177                if (hash_id == AUTH_SHA1) {
5178                        bzero(&sha1ctx, sizeof (SHA1_CTX));
5179                        hash_size = SHA1_LEN;

5181                        SHA1Init(&sha1ctx);

5183                        SHA1Update(&sha1ctx,
5184                            (void *)&(un_cval.sha1.val[0]), SHA1_LEN);

5186                        if (ndlp->nlp_DID == FABRIC_DID) {
5187                                SHA1Update(&sha1ctx,
5188                                    (void *)&node_dhc->ses_key[0],
5189                                    node_dhc->seskey_len);
5190                        } else {
5191                                /* ses_key was obtained in emlxs_hash_rsp */
5192                                SHA1Update(&sha1ctx,
5193                                    (void *)&node_dhc->nlp_auth_misc.ses_key[0],
5194                                    node_dhc->nlp_auth_misc.seskey_len);
5195                        }

5197                        SHA1Final((void *)sha1_digest, &sha1ctx);

5199                        SHA1Init(&sha1ctx);

5201                        SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

5203                        SHA1Update(&sha1ctx, (void *)remote_key,
5204                            node_dhc->auth_key.remote_password_length);

5206                        SHA1Update(&sha1ctx, (void *)sha1_digest, SHA1_LEN);

5208                        SHA1Final((void *)sha1_digest, &sha1ctx);
```



```
5210                        hash_val = (uint32_t *)kmem_alloc(hash_size,
5211                            KM_NOSLEEP);
5212                        if (hash_val == NULL) {
5213                                return (NULL);
5214                        } else {
5215                                bcopy((void *)&sha1_digest,
5216                                    (void *)hash_val, SHA1_LEN);
5217                        }
5218                }
5219                return ((uint32_t *)hash_val);
5220        }

5222 } /* emlxs_hash_vrf */


5225 /*
5226  * If dhval == NULL, NULL DHCHAP else, DHCHAP group.
5227  *
5228  * This routine is used by the auth transaction initiator (Who does the
5229  * auth-negotiate) to calculate the R1 (response) based on
5230  * the dh value it received, its own random private key, the challenge it
5231  * received, and Transaction id, as well as the password
5232  * associated with this very initiator in the auth pair.
5233  */
5234 uint32_t *
5235 emlxs_hash_rsp(
5236 emlxs_port_t *port,
5237 emlxs_port_dhc_t *port_dhc,
5238 NODELIST *ndlp,
5239 uint32_t tran_id,
5240 union challenge_val un_cval,
5241 uint8_t *dhval,
5242 uint32_t dhvallen)
5243 {
5244        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
5245        uint32_t dhgp_id;
5246        uint32_t hash_id;
5247        uint32_t *hash_val;
5248        uint32_t hash_size;
5249        MD5_CTX mdctx;
5250        SHA1_CTX sha1ctx;
5251        uint8_t sha1_digest[20];
5252        uint8_t md5_digest[16];
5253        uint8_t Cai[20];
5254        uint8_t mytran_id = 0x00;
5255        char *mykey;
5256        BIG_ERR_CODE err = BIG_OK;

5258        if (ndlp->nlp_DID == FABRIC_DID) {
5259                hash_id = node_dhc->hash_id;
5260                dhgp_id = node_dhc->dhgp_id;
5261        } else {
5262                hash_id = node_dhc->nlp_auth_hashid;
5263                dhgp_id = node_dhc->nlp_auth_dhgpid;
5264        }

5266        tran_id = (AUTH_TRAN_ID_MASK & tran_id);
5267        mytran_id = (uint8_t)(LE_SWAP32(tran_id));

5269        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
5270            "emlxs_hash_rsp: 0x%x 0x%x 0x%x 0x%x dhvallen=0x%x",
5271            ndlp->nlp_DID, hash_id, dhgp_id, mytran_id, dhvallen);

5273        if (ndlp->nlp_DID == FABRIC_DID) {
5274                mykey = (char *)node_dhc->auth_key.local_password;
```

```
5276         } else {
5277                 mykey = (char *)node_dhc->auth_key.local_password;
5278         }

5280         if (dhval == NULL) {
5281                 /* NULL DHCHAP */
5282                 if (hash_id == AUTH_MD5) {
5283                         bzero(&mdctx, sizeof (MD5_CTX));
5284                         hash_size = MD5_LEN;

5286                         MD5Init(&mdctx);

5288                         MD5Update(&mdctx, (unsigned char *)&mytran_id, 1);

5290                         MD5Update(&mdctx, (unsigned char *)mykey,
5291                             node_dhc->auth_key.local_password_length);

5293                         MD5Update(&mdctx,
5294                             (unsigned char *)&(un_cval.md5.val[0]),
5295                             MD5_LEN);

5297                         MD5Final((uint8_t *)md5_digest, &mdctx);

5299                         hash_val = (uint32_t *)kmem_alloc(hash_size,
5300                             KM_NOSLEEP);
5301                         if (hash_val == NULL) {
5302                                 return (NULL);
5303                         } else {
5304                                 bcopy((void *)&md5_digest,
5305                                     (void *)hash_val, MD5_LEN);
5306                         }

5308                         /*
5309                          * emlxs_md5_digest_to_hex((uint8_t *)hash_val,
5310                          * output);
5311                          */

5313                 }
5314                 if (hash_id == AUTH_SHA1) {
5315                         bzero(&sha1ctx, sizeof (SHA1_CTX));
5316                         hash_size = SHA1_LEN;
5317                         SHA1Init(&sha1ctx);

5319                         SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

5321                         SHA1Update(&sha1ctx, (void *)mykey,
5322                             node_dhc->auth_key.local_password_length);

5324                         SHA1Update(&sha1ctx,
5325                             (void *)&(un_cval.sha1.val[0]), SHA1_LEN);

5327                         SHA1Final((void *)sha1_digest, &sha1ctx);

5329                         /*
5330                          * emlxs_sha1_digest_to_hex((uint8_t *)hash_val,
5331                          * output);
5332                          */

5334                         hash_val = (uint32_t *)kmem_alloc(hash_size,
5335                             KM_NOSLEEP);
5336                         if (hash_val == NULL) {
5337                                 return (NULL);
5338                         } else {
5339                                 bcopy((void *)&sha1_digest,
5340                                     (void *)hash_val, SHA1_LEN);
5341                         }
```



```
5342                 }
5343                 return ((uint32_t *)hash_val);
5344         } else {

5346                 /* process DH grops */
5347                 /*
5348                  * calculate interm hash value Ca1 Ca1 = H(C1 || (g^x mod
5349                  * p)^y mod p) in which C1 is the challenge received. g^x mod
5350                  * p is the dhval received y is the random number in 16 bytes
5351                  * for MD5, 20 bytes for SHA1 p is hardcoded value based on
5352                  * different DH groups.
5353                  *
5354                  * To calculate hash value R1 R1 = H (Ti || Kn || Cai) in which
5355                  * Ti is the transaction identifier Kn is the shared secret.
5356                  * Cai is the result from interm hash.
5357                  *
5358                  * g^y mod p is reserved in port_dhc as pubkey (public key).for
5359                  * bi-dir challenge is another random number. y is prikey
5360                  * (private key). ((g^x mod p)^y mod p) is sekey (session
5361                  * key)
5362                  */
5363                 err = emlxs_interm_hash(port, port_dhc, ndlp,
5364                     (void *)&Cai, tran_id,
5365                     un_cval, dhval, &dhvallen);

5367                 if (err != BIG_OK) {
5368                         return (NULL);
5369                 }
5370                 if (hash_id == AUTH_MD5) {
5371                         bzero(&mdctx, sizeof (MD5_CTX));
5372                         hash_size = MD5_LEN;

5374                         MD5Init(&mdctx);

5376                         MD5Update(&mdctx, (unsigned char *)&mytran_id, 1);

5378                         MD5Update(&mdctx, (unsigned char *)mykey,
5379                             node_dhc->auth_key.local_password_length);

5381                         MD5Update(&mdctx, (unsigned char *)Cai, MD5_LEN);

5383                         MD5Final((uint8_t *)md5_digest, &mdctx);

5385                         hash_val = (uint32_t *)kmem_alloc(hash_size,
5386                             KM_NOSLEEP);
5387                         if (hash_val == NULL) {
5388                                 return (NULL);
5389                         } else {
5390                                 bcopy((void *)&md5_digest,
5391                                     (void *)hash_val, MD5_LEN);
5392                         }
5393                 }
5394                 if (hash_id == AUTH_SHA1) {
5395                         bzero(&sha1ctx, sizeof (SHA1_CTX));
5396                         hash_size = SHA1_LEN;

5398                         SHA1Init(&sha1ctx);

5400                         SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

5402                         SHA1Update(&sha1ctx, (void *)mykey,
5403                             node_dhc->auth_key.local_password_length);

5405                         SHA1Update(&sha1ctx, (void *)&Cai[0], SHA1_LEN);

5407                         SHA1Final((void *)sha1_digest, &sha1ctx);
```

```
5409                        hash_val = (uint32_t *)kmem_alloc(hash_size,
5410                            KM_NOSLEEP);
5411                        if (hash_val == NULL) {
5412                                return (NULL);
5413                        } else {
5414                                bcopy((void *)&sha1_digest,
5415                                    (void *)hash_val, SHA1_LEN);
5416                        }
5417                }
5418                return ((uint32_t *)hash_val);
5419        }

5421 } /* emlxs_hash_rsp */


5424 /*
5425  * To get the augmented challenge Cai Stored in hash_val
5426  *
5427  * Cai = Hash (C1 || ((g^x mod p)^y mod p)) = Hash (C1 || (g^(x*y) mod p)
5428  *
5429  * C1:challenge received from the remote entity (g^x mod p): dh val
5430  * received from the remote entity (remote entity's pubkey) y:
5431  * random private key from the local entity Hash: hash function used in
5432  * agreement. (g^(x*y) mod p): shared session key (aka
5433  * shared secret) (g^y mod p): local entity's pubkey
5434  */
5435 /* ARGSUSED */
5436 BIG_ERR_CODE
5437 emlxs_interm_hash(
5438 emlxs_port_t *port,
5439 emlxs_port_dhc_t *port_dhc,
5440 NODELIST *ndlp,
5441 void *hash_val,
5442 uint32_t tran_id,
5443 union challenge_val un_cval,
5444 uint8_t *dhval,
5445 uint32_t *dhvallen)
5446 {
5447         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
5448         uint32_t dhgp_id;
5449         uint32_t hash_id;
5450         MD5_CTX mdctx;
5451         SHA1_CTX sha1ctx;
5452         uint8_t sha1_digest[20];
5453         uint8_t md5_digest[16];
5454         uint32_t hash_size;
5455         BIG_ERR_CODE err = BIG_OK;

5457         if (ndlp->nlp_DID == FABRIC_DID) {
5458                 hash_id = node_dhc->hash_id;
5459                 dhgp_id = node_dhc->dhgp_id;
5460         } else {
5461                 hash_id = node_dhc->nlp_auth_hashid;
5462                 dhgp_id = node_dhc->nlp_auth_dhgpid;
5463         }

5465         if (hash_id == AUTH_MD5) {
5466                 bzero(&mdctx, sizeof (MD5_CTX));
5467                 hash_size = MD5_LEN;
5468                 MD5Init(&mdctx);
5469                 MD5Update(&mdctx,
5470                     (unsigned char *)&(un_cval.md5.val[0]), MD5_LEN);

5472                 /*
5473                  * get the pub key (g^y mod p) and session key (g^(x*y) mod
```



```
5474                  * p) and stored them in the partner's ndlp structure
5475                  */
5476                 err = emlxs_BIGNUM_get_pubkey(port, port_dhc, ndlp,
5477                     dhval, dhvallen, hash_size, dhgp_id);

5479                 if (err != BIG_OK) {
5480                         return (err);
5481                 }
5482                 if (ndlp->nlp_DID == FABRIC_DID) {
5483                         MD5Update(&mdctx,
5484                             (unsigned char *)&node_dhc->ses_key[0],
5485                             node_dhc->seskey_len);
5486                 } else {
5487                 MD5Update(&mdctx,
5488                     (unsigned char *)&node_dhc->nlp_auth_misc.ses_key[0],
5489                     node_dhc->nlp_auth_misc.seskey_len);
5490                 }

5492                 MD5Final((uint8_t *)md5_digest, &mdctx);

5494                 bcopy((void *)&md5_digest, (void *)hash_val, MD5_LEN);
5495         }
5496         if (hash_id == AUTH_SHA1) {
5497                 bzero(&sha1ctx, sizeof (SHA1_CTX));

5499                 hash_size = SHA1_LEN;

5501                 SHA1Init(&sha1ctx);

5503                 SHA1Update(&sha1ctx, (void *)&(un_cval.sha1.val[0]), SHA1_LEN);

5505                 /* get the pub key and session key */
5506                 err = emlxs_BIGNUM_get_pubkey(port, port_dhc, ndlp,
5507                     dhval, dhvallen, hash_size, dhgp_id);

5509                 if (err != BIG_OK) {
5510                         return (err);
5511                 }
5512                 if (ndlp->nlp_DID == FABRIC_DID) {
5513                         SHA1Update(&sha1ctx, (void *)&node_dhc->ses_key[0],
5514                             node_dhc->seskey_len);
5515                 } else {
5516                         SHA1Update(&sha1ctx,
5517                             (void *)&node_dhc->nlp_auth_misc.ses_key[0],
5518                             node_dhc->nlp_auth_misc.seskey_len);
5519                 }

5521                 SHA1Final((void *)sha1_digest, &sha1ctx);

5523                 bcopy((void *)&sha1_digest, (void *)hash_val, SHA1_LEN);
5524         }
5525         return (err);

5527 } /* emlxs_interm_hash */

5529 /*
5530  * This routine get the pubkey and session key. these pubkey and session
5531  * key are stored in the partner's ndlp structure.
5532  */
5533 /* ARGSUSED */
5534 BIG_ERR_CODE
5535 emlxs_BIGNUM_get_pubkey(
5536                         emlxs_port_t *port,
5537                         emlxs_port_dhc_t *port_dhc,
5538                         NODELIST *ndlp,
5539                         uint8_t *dhval,
```

```
5540                        uint32_t *dhvallen,
5541                        uint32_t hash_size,
5542                        uint32_t dhgp_id)
5543 {
5544         emlxs_hba_t *hba = HBA;

5546         BIGNUM a, e, n, result;
5547         uint32_t plen;
5548         uint8_t random_number[20];
5549         unsigned char *tmp = NULL;
5550         BIGNUM g, result1;

5552 #ifdef BIGNUM_CHUNK_32
5553         uint8_t gen[] = {0x00, 0x00, 0x00, 0x02};
5554 #else
5555         uint8_t gen[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02};
5556 #endif /* BIGNUM_CHUNK_32 */

5558         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
5559         BIG_ERR_CODE err = BIG_OK;

5561         /*
5562          * compute a^e mod n assume a < n, n odd, result->value at least as
5563          * long as n->value.
5564          *
5565          * a is the public key received from responder. e is the private key
5566          * generated by me. n is the wellknown modulus.
5567          */

5569         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
5570             "emlxs_BIGNUM_get_pubkey: 0x%x 0x%x 0x%x 0x%x",
5571             ndlp->nlp_DID, *dhvallen, hash_size, dhgp_id);

5573         /* size should be in the unit of (BIG_CHUNK_TYPE) words */
5574         if (big_init(&a, CHARLEN2BIGNUMLEN(*dhvallen))  != BIG_OK) {
5575                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5576                     "emlxs_BIGNUM_get_pubkey: big_init failed. a size=%d",
5577                     CHARLEN2BIGNUMLEN(*dhvallen));

5579                 err = BIG_NO_MEM;
5580                 return (err);
5581         }
5582         /* a: (g^x mod p) */
5583         /*
5584          * dhval is in big-endian format. This call converts from
5585          * byte-big-endian format to big number format (words in little
5586          * endian order, but bytes within the words big endian)
5587          */
5588         bytestring2bignum(&a, (unsigned char *)dhval, *dhvallen);

5590         if (big_init(&e, CHARLEN2BIGNUMLEN(hash_size)) != BIG_OK) {
5591                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5592                     "emlxs_BIGNUM_get_pubkey: big_init failed. e size=%d",
5593                     CHARLEN2BIGNUMLEN(hash_size));

5595                 err = BIG_NO_MEM;
5596                 goto ret1;
5597         }
5598 #ifdef RAND

5600         bzero(&random_number, hash_size);

5602         /* to get random private key: y */
5603         /* remember y is short lived private key */
5604         if (hba->rdn_flag == 1) {
5605                 emlxs_get_random_bytes(ndlp, random_number, 20);
```



```
5606         } else {
5607                 (void) random_get_pseudo_bytes(random_number, hash_size);
5608         }

5610         /* e: y */
5611         bytestring2bignum(&e, (unsigned char *)random_number, hash_size);

5613 #endif  /* RAND */

5615 #ifdef MYRAND
5616         bytestring2bignum(&e, (unsigned char *)myrand, hash_size);

5618         printf("myrand random_number as Y ================\n");
5619         for (i = 0; i < 5; i++) {
5620                 for (j = 0; j < 4; j++) {
5621                         printf("%x", myrand[(i * 4) + j]);
5622                 }
5623                 printf("\n");
5624         }
5625 #endif  /* MYRAND */

5627         switch (dhgp_id) {
5628         case GROUP_1024:
5629                 plen = 128;
5630                 tmp = dhgp1_pVal;
5631                 break;

5633         case GROUP_1280:
5634                 plen = 160;
5635                 tmp = dhgp2_pVal;
5636                 break;

5638         case GROUP_1536:
5639                 plen = 192;
5640                 tmp = dhgp3_pVal;
5641                 break;

5643         case GROUP_2048:
5644                 plen = 256;
5645                 tmp = dhgp4_pVal;
5646                 break;
5647         }

5649         if (big_init(&n, CHARLEN2BIGNUMLEN(plen)) != BIG_OK) {
5650                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5651                     "emlxs_BIGNUM_get_pubkey: big_init failed. n size=%d",
5652                     CHARLEN2BIGNUMLEN(plen));
5653                 err = BIG_NO_MEM;
5654                 goto ret2;
5655         }
5656         bytestring2bignum(&n, (unsigned char *)tmp, plen);

5658         if (big_init(&result, CHARLEN2BIGNUMLEN(512)) != BIG_OK) {
5659                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5660                     "emlxs_BIGNUM_get_pubkey: big_init failed. result size=%d",
5661                     CHARLEN2BIGNUMLEN(512));

5663                 err = BIG_NO_MEM;
5664                 goto ret3;
5665         }
5666         if (big_cmp_abs(&a, &n) > 0) {
5667                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5668                     "emlxs_BIGNUM_get_pubkey: big_cmp_abs error.");
5669                 err = BIG_GENERAL_ERR;
5670                 goto ret4;
5671         }
```

```
5672         /* perform computation on big numbers to get seskey  */
5673         /* a^e mod n */
5674         /* i.e., (g^x mod p)^y mod p  */

5676         if (big_modexp(&result, &a, &e, &n, NULL) != BIG_OK) {
5677                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5678                     "emlxs_BIGNUM_get_pubkey: big_modexp result error");
5679                 err = BIG_NO_MEM;
5680                 goto ret4;
5681         }
5682         /* convert big number ses_key to bytestring */
5683         if (ndlp->nlp_DID == FABRIC_DID) {
5684                 /*
5685                  * This call converts from big number format to
5686                  * byte-big-endian format. big number format is words in
5687                  * little endian order, but bytes within words in native byte
5688                  * order
5689                  */
5690                 bignum2bytestring(node_dhc->ses_key, &result,
5691                     sizeof (BIG_CHUNK_TYPE) * (result.len));
5692                 node_dhc->seskey_len = sizeof (BIG_CHUNK_TYPE) * (result.len);

5694                 /* we can store another copy in ndlp */
5695                 bignum2bytestring(node_dhc->nlp_auth_misc.ses_key, &result,
5696                     sizeof (BIG_CHUNK_TYPE) * (result.len));
5697                 node_dhc->nlp_auth_misc.seskey_len =
5698                     sizeof (BIG_CHUNK_TYPE) * (result.len);
5699         } else {
5700                 /* for end-to-end auth */
5701                 bignum2bytestring(node_dhc->nlp_auth_misc.ses_key, &result,
5702                     sizeof (BIG_CHUNK_TYPE) * (result.len));
5703                 node_dhc->nlp_auth_misc.seskey_len =
5704                     sizeof (BIG_CHUNK_TYPE) * (result.len);
5705         }

5707         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
5708             "emlxs_BIGNUM_get_pubkey: after seskey cal: 0x%x 0x%x 0x%x",
5709             node_dhc->nlp_auth_misc.seskey_len, result.size, result.len);


5712         /* to get pub_key: g^y mod p, g is 2 */

5714         if (big_init(&g, 1) != BIG_OK) {
5715                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5716                     "emlxs_BIGNUM_get_pubkey: big_init failed. g size=1");

5718                 err = BIG_NO_MEM;
5719                 goto ret4;
5720         }
5721         if (big_init(&result1, CHARLEN2BIGNUMLEN(512)) != BIG_OK) {
5722                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5723                     "emlxs_BIGNUM_get_pubkey: big_init failed. result1 size=%d",
5724                     CHARLEN2BIGNUMLEN(512));
5725                 err = BIG_NO_MEM;
5726                 goto ret5;
5727         }

5729         bytestring2bignum(&g,
5730             (unsigned char *)&gen, sizeof (BIG_CHUNK_TYPE));

5732         if (big_modexp(&result1, &g, &e, &n, NULL) != BIG_OK) {
5733                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5734                     "emlxs_BIGNUM_get_pubkey: big_modexp result1 error");
5735                 err = BIG_NO_MEM;
5736                 goto ret6;
5737         }
```



```
5738         /* convert big number pub_key to bytestring */
5739         if (ndlp->nlp_DID == FABRIC_DID) {

5741                 bignum2bytestring(node_dhc->pub_key, &result1,
5742                     sizeof (BIG_CHUNK_TYPE) * (result1.len));
5743                 node_dhc->pubkey_len = (result1.len) * sizeof (BIG_CHUNK_TYPE);

5745                 /* save another copy in ndlp */
5746                 bignum2bytestring(node_dhc->nlp_auth_misc.pub_key, &result1,
5747                     sizeof (BIG_CHUNK_TYPE) * (result1.len));
5748                 node_dhc->nlp_auth_misc.pubkey_len =
5749                     (result1.len) * sizeof (BIG_CHUNK_TYPE);

5751         } else {
5752                 /* for end-to-end auth */
5753                 bignum2bytestring(node_dhc->nlp_auth_misc.pub_key, &result1,
5754                     sizeof (BIG_CHUNK_TYPE) * (result1.len));
5755                 node_dhc->nlp_auth_misc.pubkey_len =
5756                     (result1.len) * sizeof (BIG_CHUNK_TYPE);
5757         }

5759         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
5760             "emlxs_BIGNUM_get_pubkey: after pubkey cal: 0x%x 0x%x 0x%x",
5761             node_dhc->nlp_auth_misc.pubkey_len, result1.size, result1.len);


5764 ret6:
5765         big_finish(&result1);
5766 ret5:
5767         big_finish(&g);
5768 ret4:
5769         big_finish(&result);
5770 ret3:
5771         big_finish(&n);
5772 ret2:
5773         big_finish(&e);
5774 ret1:
5775         big_finish(&a);

5777         return (err);

5779 } /* emlxs_BIGNUM_get_pubkey */


5782 /*
5783  * g^x mod p x is the priv_key g and p are wellknow based on dhgp_id
5784  */
5785 /* ARGSUSED */
5786 static BIG_ERR_CODE
5787 emlxs_BIGNUM_get_dhval(
5788 emlxs_port_t *port,
5789 emlxs_port_dhc_t *port_dhc,
5790 NODELIST *ndlp,
5791 uint8_t *dhval,
5792 uint32_t *dhval_len,
5793 uint32_t dhgp_id,
5794 uint8_t *priv_key,
5795 uint32_t privkey_len)
5796 {
5797         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
5798         BIGNUM g, e, n, result1;
5799         uint32_t plen;
5800         unsigned char *tmp = NULL;

5802 #ifdef BIGNUM_CHUNK_32
5803         uint8_t gen[] = {0x00, 0x00, 0x00, 0x02};
```

```
#else
        uint8_t gen[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02};
#endif /* BIGNUM_CHUNK_32 */

        BIG_ERR_CODE err = BIG_OK;

        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
            "emlxs_BIGNUM_get_dhval: did=0x%x privkey_len=0x%x dhgp_id=0x%x",
            ndlp->nlp_DID, privkey_len, dhgp_id);

        if (big_init(&result1, CHARLEN2BIGNUMLEN(512)) != BIG_OK) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_BIGNUM_get_dhval: big_init failed. result1 size=%d",
                    CHARLEN2BIGNUMLEN(512));

                err = BIG_NO_MEM;
                return (err);
        }
        if (big_init(&g, 1) != BIG_OK) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_BIGNUM_get_dhval: big_init failed. g size=1");

                err = BIG_NO_MEM;
                goto ret1;
        }
        /* get g */
        bytestring2bignum(&g, (unsigned char *)gen, sizeof (BIG_CHUNK_TYPE));

        if (big_init(&e, CHARLEN2BIGNUMLEN(privkey_len)) != BIG_OK) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_BIGNUM_get_dhval: big_init failed. e size=%d",
                    CHARLEN2BIGNUMLEN(privkey_len));

                err = BIG_NO_MEM;
                goto ret2;
        }
        /* get x */
        bytestring2bignum(&e, (unsigned char *)priv_key, privkey_len);

        switch (dhgp_id) {
        case GROUP_1024:
                plen = 128;
                tmp = dhgp1_pVal;
                break;

        case GROUP_1280:
                plen = 160;
                tmp = dhgp2_pVal;
                break;

        case GROUP_1536:
                plen = 192;
                tmp = dhgp3_pVal;
                break;

        case GROUP_2048:
                plen = 256;
                tmp = dhgp4_pVal;
                break;
        }

        if (big_init(&n, CHARLEN2BIGNUMLEN(plen)) != BIG_OK) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_BIGNUM_get_dhval: big_init failed. n size=%d",
                    CHARLEN2BIGNUMLEN(plen));
```



```
                err = BIG_NO_MEM;
                goto ret3;
        }
        /* get p */
        bytestring2bignum(&n, (unsigned char *)tmp, plen);

        /* to cal: (g^x mod p) */
        if (big_modexp(&result1, &g, &e, &n, NULL) != BIG_OK) {
                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
                    "emlxs_BIGNUM_get_dhval: big_modexp result1 error");

                err = BIG_GENERAL_ERR;
                goto ret4;
        }
        /* convert big number pub_key to bytestring */
        if (ndlp->nlp_DID == FABRIC_DID) {
                bignum2bytestring(node_dhc->hrsp_pub_key, &result1,
                    sizeof (BIG_CHUNK_TYPE) * (result1.len));
                node_dhc->hrsp_pubkey_len =
                    (result1.len) * sizeof (BIG_CHUNK_TYPE);

                /* save another copy in partner's ndlp */
                bignum2bytestring(node_dhc->nlp_auth_misc.hrsp_pub_key,
                    &result1,
                    sizeof (BIG_CHUNK_TYPE) * (result1.len));

                node_dhc->nlp_auth_misc.hrsp_pubkey_len =
                    (result1.len) * sizeof (BIG_CHUNK_TYPE);
        } else {
                bignum2bytestring(node_dhc->nlp_auth_misc.hrsp_pub_key,
                    &result1,
                    sizeof (BIG_CHUNK_TYPE) * (result1.len));
                node_dhc->nlp_auth_misc.hrsp_pubkey_len =
                    (result1.len) * sizeof (BIG_CHUNK_TYPE);
        }


        if (ndlp->nlp_DID == FABRIC_DID) {
                bcopy((void *)node_dhc->hrsp_pub_key, (void *)dhval,
                    node_dhc->hrsp_pubkey_len);
        } else {
                bcopy((void *)node_dhc->nlp_auth_misc.hrsp_pub_key,
                    (void *)dhval,
                    node_dhc->nlp_auth_misc.hrsp_pubkey_len);
        }

        *(uint32_t *)dhval_len = (result1.len) * sizeof (BIG_CHUNK_TYPE);


ret4:
        big_finish(&result1);
ret3:
        big_finish(&e);
ret2:
        big_finish(&n);
ret1:
        big_finish(&g);

        return (err);

} /* emlxs_BIGNUM_get_dhval */


/*
 * to get ((g^y mod p)^x mod p) a^e mod n
 */
```

```
5936 BIG_ERR_CODE
5937 emlxs_BIGNUM_pubkey(
5938                     emlxs_port_t *port,
5939                     void *pubkey,
5940                     uint8_t *dhval,     /* g^y mod p */
5941                     uint32_t dhvallen,
5942                     uint8_t *key,       /* x */
5943                     uint32_t key_size,
5944                     uint32_t dhgp_id,
5945                     uint32_t *pubkeylen)
5946 {
5947         BIGNUM a, e, n, result;
5948         uint32_t plen;
5949         unsigned char *tmp = NULL;
5950         BIG_ERR_CODE err = BIG_OK;

5952         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
5953             "emlxs_BIGNUM_pubkey: dhvallen=0x%x dhgp_id=0x%x",
5954             dhvallen, dhgp_id);

5956         if (big_init(&a, CHARLEN2BIGNUMLEN(dhvallen)) != BIG_OK) {
5957                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5958                     "emlxs_BIGNUM_pubkey: big_init failed. a size=%d",
5959                     CHARLEN2BIGNUMLEN(dhvallen));

5961                 err = BIG_NO_MEM;
5962                 return (err);
5963         }
5964         /* get g^y mod p */
5965         bytestring2bignum(&a, (unsigned char *)dhval, dhvallen);

5967         if (big_init(&e, CHARLEN2BIGNUMLEN(key_size)) != BIG_OK) {
5968                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
5969                     "emlxs_BIGNUM_pubkey: big_init failed. e size=%d",
5970                     CHARLEN2BIGNUMLEN(key_size));

5972                 err = BIG_NO_MEM;
5973                 goto ret1;
5974         }
5975         /* get x */
5976         bytestring2bignum(&e, (unsigned char *)key, key_size);

5978         switch (dhgp_id) {
5979         case GROUP_1024:
5980                 plen = 128;
5981                 tmp = dhgp1_pVal;
5982                 break;

5984         case GROUP_1280:
5985                 plen = 160;
5986                 tmp = dhgp2_pVal;
5987                 break;

5989         case GROUP_1536:
5990                 plen = 192;
5991                 tmp = dhgp3_pVal;
5992                 break;

5994         case GROUP_2048:
5995                 plen = 256;
5996                 tmp = dhgp4_pVal;
5997                 break;
5998         }

6000         if (big_init(&n, CHARLEN2BIGNUMLEN(plen)) != BIG_OK) {
6001                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
```



```
6002                     "emlxs_BIGNUM_pubkey: big_init failed. n size=%d",
6003                     CHARLEN2BIGNUMLEN(plen));

6005                 err = BIG_NO_MEM;
6006                 goto ret2;
6007         }
6008         bytestring2bignum(&n, (unsigned char *)tmp, plen);

6010         if (big_init(&result, CHARLEN2BIGNUMLEN(512)) != BIG_OK) {
6011                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6012                     "emlxs_BIGNUM_pubkey: big_init failed. result size=%d",
6013                     CHARLEN2BIGNUMLEN(512));

6015                 err = BIG_NO_MEM;
6016                 goto ret3;
6017         }
6018         if (big_cmp_abs(&a, &n) > 0) {
6019                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6020                     "emlxs_BIGNUM_pubkey: big_cmp_abs error");

6022                 err = BIG_GENERAL_ERR;
6023                 goto ret4;
6024         }
6025         if (big_modexp(&result, &a, &e, &n, NULL) != BIG_OK) {
6026                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6027                     "emlxs_BIGNUM_pubkey: big_modexp result error");

6029                 err = BIG_NO_MEM;
6030                 goto ret4;
6031         }
6032         bignum2bytestring(pubkey, &result,
6033             sizeof (BIG_CHUNK_TYPE) * (result.len));
6034         *pubkeylen = sizeof (BIG_CHUNK_TYPE) * (result.len);

6036         /* This pubkey is actually session key */

6038 ret4:
6039         big_finish(&result);
6040 ret3:
6041         big_finish(&n);
6042 ret2:
6043         big_finish(&e);
6044 ret1:
6045         big_finish(&a);

6047         return (err);

6049 } /* emlxs_BIGNUM_pubkey */


6052 /*
6053  * key: x dhval: (g^y mod p) tran_id: Ti bi_cval: C2 hash_id: H dhgp_id: p/g
6054  *
6055  * Cai = H (C2 || ((g^y mod p)^x mod p) )
6056  *
6057  */
6058 /* ARGSUSED */
6059 BIG_ERR_CODE
6060 emlxs_hash_Cai(
6061         emlxs_port_t *port,
6062         emlxs_port_dhc_t *port_dhc,
6063         NODELIST *ndlp,
6064         void *Cai,
6065         uint32_t hash_id,
6066         uint32_t dhgp_id,
6067         uint32_t tran_id,
```

```
6068         uint8_t *cval,
6069         uint32_t cval_len,
6070         uint8_t *key,
6071         uint8_t *dhval,
6072         uint32_t dhvallen)
6073 {
6074         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
6075         MD5_CTX mdctx;
6076         SHA1_CTX sha1ctx;
6077         uint8_t sha1_digest[20];
6078         uint8_t md5_digest[16];
6079         uint8_t pubkey[512];
6080         uint32_t pubkey_len = 0;
6081         uint32_t key_size;
6082         BIG_ERR_CODE err = BIG_OK;

6084         key_size = cval_len;
6085         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
6086             "emlxs_hash_Cai: 0x%x 0x%x 0x%x 0x%x 0x%x",
6087             ndlp->nlp_DID, hash_id, dhgp_id, tran_id, dhvallen);

6089         if (hash_id == AUTH_MD5) {
6090                 bzero(&mdctx, sizeof (MD5_CTX));
6091                 MD5Init(&mdctx);
6092                 MD5Update(&mdctx, (unsigned char *)cval, cval_len);

6094                 /* this pubkey obtained is actually the session key */
6095                 /*
6096                  * pubkey: ((g^y mod p)^x mod p)
6097                  */
6098                 err = emlxs_BIGNUM_pubkey(port, pubkey, dhval, dhvallen,
6099                     key, key_size, dhgp_id, &pubkey_len);

6101                 if (err != BIG_OK) {
6102                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6103                             "emlxs_hash_Cai: MD5 BIGNUM_pubkey error: 0x%x",
6104                             err);

6106                         err = BIG_GENERAL_ERR;
6107                         return (err);
6108                 }
6109                 if (pubkey_len == 0) {
6110                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6111                             "emlxs_hash_Cai: MD5 BIGNUM_pubkey error: len=0");

6113                         err = BIG_GENERAL_ERR;
6114                         return (err);
6115                 }
6116                 if (ndlp->nlp_DID == FABRIC_DID) {
6117                         bcopy((void *)pubkey,
6118                             (void *)node_dhc->hrsp_ses_key, pubkey_len);
6119                         node_dhc->hrsp_seskey_len = pubkey_len;

6121                         /* store extra copy */
6122                         bcopy((void *)pubkey,
6123                             (void *)node_dhc->nlp_auth_misc.hrsp_ses_key,
6124                             pubkey_len);
6125                         node_dhc->nlp_auth_misc.hrsp_seskey_len = pubkey_len;

6127                 } else {
6128                         bcopy((void *)pubkey,
6129                             (void *)node_dhc->nlp_auth_misc.hrsp_ses_key,
6130                             pubkey_len);
6131                         node_dhc->nlp_auth_misc.hrsp_seskey_len = pubkey_len;
6132                 }
```



```
6134                 MD5Update(&mdctx, (unsigned char *)pubkey, pubkey_len);
6135                 MD5Final((uint8_t *)md5_digest, &mdctx);
6136                 bcopy((void *)&md5_digest, (void *)Cai, MD5_LEN);
6137         }
6138         if (hash_id == AUTH_SHA1) {
6139                 bzero(&sha1ctx, sizeof (SHA1_CTX));
6140                 SHA1Init(&sha1ctx);

6142                 SHA1Update(&sha1ctx, (void *)cval, cval_len);

6144                 err = emlxs_BIGNUM_pubkey(port, pubkey, dhval, dhvallen,
6145                     key, key_size, dhgp_id, &pubkey_len);

6147                 if (err != BIG_OK) {
6148                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6149                             "emlxs_hash_Cai: SHA1 BIGNUM_pubkey error: 0x%x",
6150                             err);

6152                         err = BIG_GENERAL_ERR;
6153                         return (err);
6154                 }
6155                 if (pubkey_len == 0) {
6156                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6157                             "emlxs_hash_Cai: SA1 BUM_pubkey error: key_len=0");

6159                         err = BIG_GENERAL_ERR;
6160                         return (err);
6161                 }
6162                 if (ndlp->nlp_DID == FABRIC_DID) {
6163                         bcopy((void *)pubkey,
6164                             (void *)node_dhc->hrsp_ses_key,
6165                             pubkey_len);
6166                         node_dhc->hrsp_seskey_len = pubkey_len;

6168                         /* store extra copy */
6169                         bcopy((void *)pubkey,
6170                             (void *)node_dhc->nlp_auth_misc.hrsp_ses_key,
6171                             pubkey_len);
6172                         node_dhc->nlp_auth_misc.hrsp_seskey_len = pubkey_len;

6174                 } else {
6175                         bcopy((void *)pubkey,
6176                             (void *)node_dhc->nlp_auth_misc.hrsp_ses_key,
6177                             pubkey_len);
6178                         node_dhc->nlp_auth_misc.hrsp_seskey_len = pubkey_len;
6179                 }

6181                 SHA1Update(&sha1ctx, (void *)pubkey, pubkey_len);
6182                 SHA1Final((void *)sha1_digest, &sha1ctx);
6183                 bcopy((void *)&sha1_digest, (void *)Cai, SHA1_LEN);
6184         }
6185         return (err);

6187 } /* emlxs_hash_Cai */


6190 /*
6191  * This routine is to verify the DHCHAP_Reply from initiator by the host
6192  * as the responder.
6193  *
6194  * flag: 1: if host is the responder 0: if host is the initiator
6195  *
6196  * if bi_cval != NULL, this routine is used to calculate the response based
6197  * on the challenge from initiator as part of
6198  * DHCHAP_Reply for bi-dirctional authentication.
6199  *
```

```
6200  */
6201 /* ARGSUSED */
6202 static uint32_t *
6203 emlxs_hash_verification(
6204         emlxs_port_t *port,
6205         emlxs_port_dhc_t *port_dhc,
6206         NODELIST *ndlp,
6207         uint32_t tran_id,
6208         uint8_t *dhval,
6209         uint32_t dhval_len,
6210         uint32_t flag,  /* always 1 for now */
6211         uint8_t *bi_cval)
6212 {                       /* always 0 for now */
6213         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
6214         uint32_t dhgp_id;
6215         uint32_t hash_id;
6216         uint32_t *hash_val = NULL;
6217         uint32_t hash_size;
6218         MD5_CTX mdctx;
6219         SHA1_CTX sha1ctx;
6220         uint8_t sha1_digest[20];
6221         uint8_t md5_digest[16];
6222         uint8_t Cai[20];
6223         /* union challenge_val un_cval; */
6224         uint8_t key[20];
6225         uint8_t cval[20];
6226         uint32_t cval_len;
6227         uint8_t mytran_id = 0x00;
6228         char *remote_key;
6229         BIG_ERR_CODE err = BIG_OK;

6231         tran_id = (AUTH_TRAN_ID_MASK & tran_id);
6232         mytran_id = (uint8_t)(LE_SWAP32(tran_id));

6234         if (ndlp->nlp_DID == FABRIC_DID) {
6235                 remote_key = (char *)node_dhc->auth_key.remote_password;
6236         } else {
6237                 /*
6238                  * in case of end-to-end auth, this remote password should be
6239                  * the password associated with the remote entity. (i.e.,)
6240                  * for now it is actually local_password.
6241                  */
6242                 remote_key = (char *)node_dhc->auth_key.remote_password;
6243         }

6245         if (flag == 0) {
6246                 dhgp_id = node_dhc->dhgp_id;
6247                 hash_id = node_dhc->hash_id;
6248         } else {
6249                 dhgp_id = node_dhc->nlp_auth_dhgpid;
6250                 hash_id = node_dhc->nlp_auth_hashid;
6251         }

6253         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
6254             "emlxs_hash_verification: 0x%x 0x%x hash_id=0x%x dhgp_id=0x%x",
6255             ndlp->nlp_DID, mytran_id, hash_id, dhgp_id);

6257         if (dhval_len == 0) {
6258                 /* NULL DHCHAP group */
6259                 if (hash_id == AUTH_MD5) {
6260                         bzero(&mdctx, sizeof (MD5_CTX));
6261                         hash_size = MD5_LEN;
6262                         MD5Init(&mdctx);

6264                         MD5Update(&mdctx, (unsigned char *)&mytran_id, 1);
```



```
6266                         if (ndlp->nlp_DID == FABRIC_DID) {
6267                                 MD5Update(&mdctx,
6268                                     (unsigned char *)remote_key,
6269                                     node_dhc->auth_key.remote_password_length);
6270                         } else {
6271                                 MD5Update(&mdctx,
6272                                     (unsigned char *)remote_key,
6273                                     node_dhc->auth_key.remote_password_length);
6274                         }

6276                         if (ndlp->nlp_DID == FABRIC_DID) {
6277                                 MD5Update(&mdctx,
6278                                     (unsigned char *)&node_dhc->hrsp_cval[0],
6279                                     MD5_LEN);
6280                         } else {
6281                 MD5Update(&mdctx,
6282                     (unsigned char *)&node_dhc->nlp_auth_misc.hrsp_cval[0],
6283                     MD5_LEN);
6284                         }

6286                         MD5Final((uint8_t *)md5_digest, &mdctx);

6288                         hash_val = (uint32_t *)kmem_alloc(hash_size,
6289                             KM_NOSLEEP);
6290                         if (hash_val == NULL) {
6291                                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6292                                     "emlxs_hash_verification: alloc failed");

6294                                 return (NULL);
6295                         } else {
6296                                 bcopy((void *)md5_digest,
6297                                     (void *)hash_val, MD5_LEN);
6298                         }
6299                 }
6300                 if (hash_id == AUTH_SHA1) {
6301                         bzero(&sha1ctx, sizeof (SHA1_CTX));
6302                         hash_size = SHA1_LEN;
6303                         SHA1Init(&sha1ctx);
6304                         SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

6306                         if (ndlp->nlp_DID == FABRIC_DID) {
6307                                 SHA1Update(&sha1ctx, (void *)remote_key,
6308                                     node_dhc->auth_key.remote_password_length);
6309                         } else {
6310                                 SHA1Update(&sha1ctx, (void *)remote_key,
6311                                     node_dhc->auth_key.remote_password_length);
6312                         }

6314                         if (ndlp->nlp_DID == FABRIC_DID) {
6315                                 SHA1Update(&sha1ctx,
6316                                     (void *)&node_dhc->hrsp_cval[0],
6317                                     SHA1_LEN);
6318                         } else {
6319                         SHA1Update(&sha1ctx,
6320                             (void *)&node_dhc->nlp_auth_misc.hrsp_cval[0],
6321                             SHA1_LEN);
6322                         }

6324                         SHA1Final((void *)sha1_digest, &sha1ctx);
6325                         hash_val = (uint32_t *)kmem_zalloc(hash_size,
6326                             KM_NOSLEEP);
6327                         if (hash_val == NULL) {
6328                                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6329                                     "emlxs_hash_verification: alloc failed");

6331                                 return (NULL);
```

```
6332                        } else {
6333                                bcopy((void *)sha1_digest,
6334                                    (void *)hash_val, SHA1_LEN);
6335                        }
6336                }
6337                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
6338                    "emlxs_hash_verification: hash_val=0x%x",
6339                    *(uint32_t *)hash_val);

6341                return ((uint32_t *)hash_val);
6342        } else {

6344                /* DHCHAP group 1,2,3,4 */
6345                /*
6346                 * host received (g^x mod p) as dhval host has its own
6347                 * private key y as node_dhc->hrsp_priv_key[] host has its
6348                 * original challenge c as node_dhc->hrsp_cval[]
6349                 *
6350                 * H(c || (g^x mod p)^y mod p) = Cai H(Ti || Km || Cai) =
6351                 * hash_val returned. Ti : tran_id, Km : shared secret, Cai:
6352                 * obtained above.
6353                 */
6354                if (hash_id == AUTH_MD5) {
6355                        if (ndlp->nlp_DID == FABRIC_DID) {
6356                                bcopy((void *)node_dhc->hrsp_priv_key,
6357                                    (void *)key, MD5_LEN);
6358                        } else {
6359                        bcopy(
6360                            (void *)node_dhc->nlp_auth_misc.hrsp_priv_key,
6361                            (void *)key, MD5_LEN);
6362                        }
6363                }
6364                if (hash_id == AUTH_SHA1) {
6365                        if (ndlp->nlp_DID == FABRIC_DID) {
6366                                bcopy((void *)node_dhc->hrsp_priv_key,
6367                                    (void *)key, SHA1_LEN);
6368                        } else {
6369                        bcopy(
6370                            (void *)node_dhc->nlp_auth_misc.hrsp_priv_key,
6371                            (void *)key, SHA1_LEN);
6372                        }
6373                }
6374                if (ndlp->nlp_DID == FABRIC_DID) {
6375                        bcopy((void *)node_dhc->hrsp_cval,
6376                            (void *)cval, node_dhc->hrsp_cval_len);
6377                        cval_len = node_dhc->hrsp_cval_len;
6378                } else {
6379                        bcopy((void *)node_dhc->nlp_auth_misc.hrsp_cval,
6380                            (void *)cval,
6381                            node_dhc->nlp_auth_misc.hrsp_cval_len);
6382                        cval_len = node_dhc->nlp_auth_misc.hrsp_cval_len;
6383                }

6385                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
6386                    "emlxs_hash_verification: N-Null gp. 0x%x 0x%x",
6387                    ndlp->nlp_DID, cval_len);

6389                err = emlxs_hash_Cai(port, port_dhc, ndlp, (void *)Cai,
6390                    hash_id, dhgp_id,
6391                    tran_id, cval, cval_len,
6392                    key, dhval, dhval_len);

6394                if (err != BIG_OK) {
6395                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6396                            "emlxs_hash_verification: Cai error. ret=0x%x",
6397                            err);
```



```
6399                        return (NULL);
6400                }
6401                if (hash_id == AUTH_MD5) {
6402                        bzero(&mdctx, sizeof (MD5_CTX));
6403                        hash_size = MD5_LEN;

6405                        MD5Init(&mdctx);
6406                        MD5Update(&mdctx, (unsigned char *)&mytran_id, 1);

6408                        if (ndlp->nlp_DID == FABRIC_DID) {
6409                                MD5Update(&mdctx,
6410                                    (unsigned char *)remote_key,
6411                                    node_dhc->auth_key.remote_password_length);
6412                        } else {
6413                                MD5Update(&mdctx,
6414                                    (unsigned char *)remote_key,
6415                                    node_dhc->auth_key.remote_password_length);
6416                        }

6418                        MD5Update(&mdctx, (unsigned char *)Cai, MD5_LEN);
6419                        MD5Final((uint8_t *)md5_digest, &mdctx);

6421                        hash_val = (uint32_t *)kmem_zalloc(hash_size,
6422                            KM_NOSLEEP);
6423                        if (hash_val == NULL) {
6424                                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6425                                    "emlxs_hash_vf: alloc failed(Non-NULL dh)");

6427                                return (NULL);
6428                        } else {
6429                                bcopy((void *)&md5_digest,
6430                                    (void *)hash_val, MD5_LEN);
6431                        }
6432                }
6433                if (hash_id == AUTH_SHA1) {
6434                        bzero(&sha1ctx, sizeof (SHA1_CTX));
6435                        hash_size = SHA1_LEN;

6437                        SHA1Init(&sha1ctx);
6438                        SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

6440                        if (ndlp->nlp_DID == FABRIC_DID) {
6441                                SHA1Update(&sha1ctx, (void *)remote_key,
6442                                    node_dhc->auth_key.remote_password_length);
6443                        } else {
6444                                SHA1Update(&sha1ctx, (void *)remote_key,
6445                                    node_dhc->auth_key.remote_password_length);
6446                        }

6448                        SHA1Update(&sha1ctx, (void *)Cai, SHA1_LEN);
6449                        SHA1Final((void *)sha1_digest, &sha1ctx);

6451                        hash_val = (uint32_t *)kmem_zalloc(hash_size,
6452                            KM_NOSLEEP);
6453                        if (hash_val == NULL) {
6454                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6455                            "emlxs_hash_vf: val alloc failed (Non-NULL dh)");

6457                                return (NULL);
6458                        } else {
6459                                bcopy((void *)&sha1_digest,
6460                                    (void *)hash_val, SHA1_LEN);
6461                        }
6462                }
6463                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
```

```
6464                    "emlxs_hash_verification: hash_val=0x%x",
6465                    *(uint32_t *)hash_val);

6467                return ((uint32_t *)hash_val);
6468        }

6470 } /* emlxs_hash_verification */



6474 /*
6475  * When DHCHAP_Success msg was sent from responder to the initiator,
6476  * with bi-directional authentication requested, the
6477  * DHCHAP_Success contains the response R2 to the challenge C2 received.
6478  *
6479  * DHCHAP response R2: The value of R2 is computed using the hash function
6480  * H() selected by the HashID parameter of the
6481  * DHCHAP_Challenge msg, and the augmented challenge Ca2.
6482  *
6483  * NULL DH group: Ca2 = C2 Non NULL DH group: Ca2 = H(C2 ||
6484  * (g^y mod p)^x mod p)) x is selected by the authentication responder
6485  * which is the node_dhc->hrsp_priv_key[] (g^y mod p) is dhval received
6486  * from authentication initiator.
6487  *
6488  * R2 = H(Ti || Km || Ca2) Ti is the least significant byte of the
6489  * transaction id. Km is the secret associated with the
6490  * authentication responder.
6491  *
6492  * emlxs_hash_get_R2 and emlxs_hash_verification could be mergerd into one
6493  * function later.
6494  *
6495  */
6496 static uint32_t *
6497 emlxs_hash_get_R2(
6498         emlxs_port_t *port,
6499         emlxs_port_dhc_t *port_dhc,
6500         NODELIST *ndlp,
6501         uint32_t tran_id,
6502         uint8_t *dhval,
6503         uint32_t dhval_len,
6504         uint32_t flag,  /* flag 1 rsponder or 0 initiator */
6505         uint8_t *bi_cval)
6506 {
6507         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

6509         uint32_t dhgp_id;
6510         uint32_t hash_id;
6511         uint32_t *hash_val = NULL;
6512         uint32_t hash_size;
6513         MD5_CTX mdctx;
6514         SHA1_CTX sha1ctx;
6515         uint8_t sha1_digest[20];
6516         uint8_t md5_digest[16];
6517         uint8_t Cai[20];
6518         /* union challenge_val un_cval; */
6519         uint8_t key[20];
6520         uint32_t cval_len;
6521         uint8_t mytran_id = 0x00;

6523         char *mykey;
6524         BIG_ERR_CODE err = BIG_OK;

6526         if (ndlp->nlp_DID == FABRIC_DID) {
6527                 dhgp_id = node_dhc->nlp_auth_dhgpid;
6528                 hash_id = node_dhc->nlp_auth_hashid;
6529         } else {
```



```
6530                 if (flag == 0) {
6531                         dhgp_id = node_dhc->dhgp_id;
6532                         hash_id = node_dhc->hash_id;
6533                 } else {
6534                         dhgp_id = node_dhc->nlp_auth_dhgpid;
6535                         hash_id = node_dhc->nlp_auth_hashid;
6536                 }
6537         }

6539         tran_id = (AUTH_TRAN_ID_MASK & tran_id);
6540         mytran_id = (uint8_t)(LE_SWAP32(tran_id));

6542         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_detail_msg,
6543             "emlxs_hash_get_R2:0x%x 0x%x dhgp_id=0x%x mytran_id=0x%x",
6544             ndlp->nlp_DID, hash_id, dhgp_id, mytran_id);

6546         if (ndlp->nlp_DID == FABRIC_DID) {
6547                 mykey = (char *)node_dhc->auth_key.local_password;

6549         } else {
6550                 /* in case of end-to-end mykey should be remote_password */
6551                 mykey = (char *)node_dhc->auth_key.remote_password;
6552         }

6554         if (dhval_len == 0) {
6555                 /* NULL DHCHAP group */
6556                 if (hash_id == AUTH_MD5) {
6557                         bzero(&mdctx, sizeof (MD5_CTX));
6558                         hash_size = MD5_LEN;
6559                         MD5Init(&mdctx);

6561                         MD5Update(&mdctx, (unsigned char *)&mytran_id, 1);

6563                         if (ndlp->nlp_DID == FABRIC_DID) {
6564                                 MD5Update(&mdctx, (unsigned char *)mykey,
6565                                     node_dhc->auth_key.local_password_length);
6566                         } else {
6567                                 MD5Update(&mdctx, (unsigned char *)mykey,
6568                                     node_dhc->auth_key.remote_password_length);
6569                         }

6571                         MD5Update(&mdctx, (unsigned char *)bi_cval, MD5_LEN);

6573                         MD5Final((uint8_t *)md5_digest, &mdctx);

6575                         hash_val = (uint32_t *)kmem_alloc(hash_size,
6576                             KM_NOSLEEP);
6577                         if (hash_val == NULL) {
6578                                 return (NULL);
6579                         } else {
6580                                 bcopy((void *)md5_digest,
6581                                     (void *)hash_val, MD5_LEN);
6582                         }
6583                 }
6584                 if (hash_id == AUTH_SHA1) {
6585                         bzero(&sha1ctx, sizeof (SHA1_CTX));
6586                         hash_size = SHA1_LEN;
6587                         SHA1Init(&sha1ctx);
6588                         SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

6590                         if (ndlp->nlp_DID == FABRIC_DID) {
6591                                 SHA1Update(&sha1ctx, (void *)mykey,
6592                                     node_dhc->auth_key.local_password_length);
6593                         } else {
6594                                 SHA1Update(&sha1ctx, (void *)mykey,
6595                                     node_dhc->auth_key.remote_password_length);
```

```
6596                        }

6598                        SHA1Update(&sha1ctx, (void *)bi_cval, SHA1_LEN);
6599                        SHA1Final((void *)sha1_digest, &sha1ctx);
6600                        hash_val = (uint32_t *)kmem_alloc(hash_size,
6601                            KM_NOSLEEP);
6602                        if (hash_val == NULL) {
6603                                return (NULL);
6604                        } else {
6605                                bcopy((void *)sha1_digest,
6606                                    (void *)hash_val, SHA1_LEN);
6607                        }
6608                }
6609        } else {
6610                /* NON-NULL DHCHAP */
6611                if (ndlp->nlp_DID == FABRIC_DID) {
6612                        if (hash_id == AUTH_MD5) {
6613                                bcopy((void *)node_dhc->hrsp_priv_key,
6614                                    (void *)key, MD5_LEN);
6615                        }
6616                        if (hash_id == AUTH_SHA1) {
6617                                bcopy((void *)node_dhc->hrsp_priv_key,
6618                                    (void *)key, SHA1_LEN);
6619                        }
6620                        cval_len = node_dhc->hrsp_cval_len;
6621                } else {
6622                        if (hash_id == AUTH_MD5) {
6623                        bcopy(
6624                            (void *)node_dhc->nlp_auth_misc.hrsp_priv_key,
6625                            (void *)key, MD5_LEN);
6626                        }
6627                        if (hash_id == AUTH_SHA1) {
6628                        bcopy(
6629                            (void *)node_dhc->nlp_auth_misc.hrsp_priv_key,
6630                            (void *)key, SHA1_LEN);
6631                        }
6632                        cval_len = node_dhc->nlp_auth_misc.hrsp_cval_len;
6633                }

6635                /* use bi_cval here */
6636                /*
6637                 * key: x dhval: (g^y mod p) tran_id: Ti bi_cval: C2 hash_id:
6638                 * H dhgp_id: p/g
6639                 *
6640                 * Cai = H (C2 || ((g^y mod p)^x mod p) )
6641                 *
6642                 * R2 = H (Ti || Km || Cai)
6643                 */
6644                err = emlxs_hash_Cai(port, port_dhc, ndlp, (void *)Cai,
6645                    hash_id, dhgp_id, tran_id, bi_cval, cval_len,
6646                    key, dhval, dhval_len);

6648                if (err != BIG_OK) {
6649                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6650                            "emlxs_hash_get_R2: emlxs_hash_Cai error. ret=0x%x",
6651                            err);

6653                        return (NULL);
6654                }
6655                if (hash_id == AUTH_MD5) {
6656                        bzero(&mdctx, sizeof (MD5_CTX));
6657                        hash_size = MD5_LEN;

6659                        MD5Init(&mdctx);
6660                        MD5Update(&mdctx, (unsigned char *) &mytran_id, 1);
```



```
6662                        /*
6663                         * Here we use the same key: mykey, note: this mykey
6664                         * should be the key associated with the
6665                         * authentication responder i.e. the remote key.
6666                         */
6667                        if (ndlp->nlp_DID == FABRIC_DID)
6668                                MD5Update(&mdctx, (unsigned char *)mykey,
6669                                    node_dhc->auth_key.local_password_length);
6670                        else
6671                                MD5Update(&mdctx, (unsigned char *)mykey,
6672                                    node_dhc->auth_key.remote_password_length);

6674                        MD5Update(&mdctx, (unsigned char *)Cai, MD5_LEN);
6675                        MD5Final((uint8_t *)md5_digest, &mdctx);

6677                        hash_val = (uint32_t *)kmem_alloc(hash_size,
6678                            KM_NOSLEEP);
6679                        if (hash_val == NULL) {
6680                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6681                            "emlxs_hash_get_R2: hash_val MD5 alloc failed.");

6683                                return (NULL);
6684                        } else {
6685                                bcopy((void *)md5_digest,
6686                                    (void *)hash_val, MD5_LEN);
6687                        }
6688                }
6689                if (hash_id == AUTH_SHA1) {
6690                        bzero(&sha1ctx, sizeof (SHA1_CTX));
6691                        hash_size = SHA1_LEN;

6693                        SHA1Init(&sha1ctx);
6694                        SHA1Update(&sha1ctx, (void *)&mytran_id, 1);

6696                        if (ndlp->nlp_DID == FABRIC_DID) {
6697                                SHA1Update(&sha1ctx, (void *)mykey,
6698                                    node_dhc->auth_key.local_password_length);
6699                        } else {
6700                                SHA1Update(&sha1ctx, (void *)mykey,
6701                                    node_dhc->auth_key.remote_password_length);
6702                        }

6704                        SHA1Update(&sha1ctx, (void *)Cai, SHA1_LEN);
6705                        SHA1Final((void *)sha1_digest, &sha1ctx);

6707                        hash_val = (uint32_t *)kmem_alloc(hash_size,
6708                            KM_NOSLEEP);
6709                        if (hash_val == NULL) {
6710                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_error_msg,
6711                            "emlxs_hash_get_R2: hash_val SHA1 alloc failed.");

6713                                return (NULL);
6714                        } else {
6715                                bcopy((void *)sha1_digest,
6716                                    (void *)hash_val, SHA1_LEN);
6717                        }
6718                }
6719        }

6721        return ((uint32_t *)hash_val);

6723 } /* emlxs_hash_get_R2 */



6727 /*
```

```
 */
static void
emlxs_log_auth_event(
        emlxs_port_t *port,
        NODELIST *ndlp,
        char *subclass,
        char *info)
{
        emlxs_hba_t *hba = HBA;
        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
        nvlist_t *attr_list = NULL;
        dev_info_t *dip = hba->dip;
        emlxs_auth_cfg_t *auth_cfg;
        char *tmp = "No_more_logging_information_available";

        uint8_t lwwn[8];
        uint8_t rwwn[8];
        char *lwwn_tmp = NULL;
        char *rwwn_tmp = NULL;
        char *mytmp_lwwn, *mytmp_rwwn;
        int i;

        auth_cfg = &(node_dhc->auth_cfg);

        if (info == NULL) {
                info = tmp;
        }
        bcopy((void *) &auth_cfg->local_entity, (void *)lwwn, 8);
        lwwn_tmp = (char *)kmem_zalloc(32, KM_NOSLEEP);
        if (lwwn_tmp == NULL) {
                return;
        }
        mytmp_lwwn = lwwn_tmp;

        for (i = 0; i < 8; i++) {
                lwwn_tmp = (char *)sprintf((char *)lwwn_tmp, "%02X", lwwn[i]);
                lwwn_tmp += 2;
        }
        mytmp_lwwn[16] = '\0';

        bcopy((void *)&auth_cfg->remote_entity, (void *)rwwn, 8);
        rwwn_tmp = (char *)kmem_zalloc(32, KM_NOSLEEP);

        mytmp_rwwn = rwwn_tmp;

        if (rwwn_tmp == NULL) {
                kmem_free(mytmp_lwwn, 32);
                return;
        }
        for (i = 0; i < 8; i++) {
                rwwn_tmp = (char *)sprintf((char *)rwwn_tmp, "%02X", rwwn[i]);
                rwwn_tmp += 2;
        }
        mytmp_rwwn[16] = '\0';

        if (nvlist_alloc(&attr_list, NV_UNIQUE_NAME_TYPE, KM_NOSLEEP)
            == DDI_SUCCESS) {
                if ((nvlist_add_uint32(attr_list, "instance",
                    ddi_get_instance(dip)) == DDI_SUCCESS) &&
                    (nvlist_add_string(attr_list, "lwwn",
                    (char *)mytmp_lwwn) == DDI_SUCCESS) &&
                    (nvlist_add_string(attr_list, "rwwn",
                    (char *)mytmp_rwwn) == DDI_SUCCESS) &&
                    (nvlist_add_string(attr_list, "Info",
                    info) == DDI_SUCCESS) &&
                    (nvlist_add_string(attr_list, "Class",
```



```
                    "EC_emlx") == DDI_SUCCESS) &&
                    (nvlist_add_string(attr_list, "SubClass",
                    subclass) == DDI_SUCCESS)) {

                        (void) ddi_log_sysevent(dip,
                            DDI_VENDOR_EMLX,
                            EC_EMLXS,
                            subclass,
                            attr_list,
                            NULL,
                            DDI_NOSLEEP);
                }
                nvlist_free(attr_list);
                attr_list = NULL;
        }
        kmem_free(mytmp_lwwn, 32);
        kmem_free(mytmp_rwwn, 32);

        return;

} /* emlxs_log_auth_event() */


/* ************************** AUTH DHC INTERFACE ********************** */

extern int
emlxs_dhc_auth_start(
        emlxs_port_t *port,
        emlxs_node_t *ndlp,
        uint8_t *deferred_sbp,
        uint8_t *deferred_ubp)
{
        emlxs_hba_t *hba = HBA;
        emlxs_config_t *cfg = &CFG;
        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
        emlxs_auth_cfg_t *auth_cfg;
        emlxs_auth_key_t *auth_key;
        uint32_t i;
        uint32_t fabric;
        uint32_t fabric_switch;

        /* The ubp represents an unsolicted PLOGI */
        /* The sbp represents a solicted PLOGI    */

        fabric = ((ndlp->nlp_DID & FABRIC_DID_MASK) == FABRIC_DID_MASK) ? 1 : 0;
        fabric_switch = ((ndlp->nlp_DID == FABRIC_DID) ? 1 : 0);

        /* Return is authentication is not enabled */
        if (cfg[CFG_AUTH_ENABLE].current == 0) {
                EMLXS_MSGF(EMLXS_CONTEXT,
                    &emlxs_fcsp_start_msg,
                    "Not started. Auth disabled. did=0x%x", ndlp->nlp_DID);

                emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_DISABLED, 0, 0);

                return (1);
        }
        if (port->vpi != 0 && cfg[CFG_AUTH_NPIV].current == 0) {
                EMLXS_MSGF(EMLXS_CONTEXT,
                    &emlxs_fcsp_start_msg,
                    "Not started. NPIV auth disabled. did=0x%x", ndlp->nlp_DID);

                emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_DISABLED, 0, 0);

                return (1);
        }
```

```
6860         if (!fabric_switch && fabric) {
6861                 EMLXS_MSGF(EMLXS_CONTEXT,
6862                     &emlxs_fcsp_start_msg,
6863                     "Not started. FS auth disabled. did=0x%x", ndlp->nlp_DID);

6865                 emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_DISABLED, 0, 0);

6867                 return (1);
6868         }
6869         /* Return if fcsp support to this node is not enabled */
6870         if (!fabric_switch && cfg[CFG_AUTH_E2E].current == 0) {
6871                 EMLXS_MSGF(EMLXS_CONTEXT,
6872                     &emlxs_fcsp_start_msg,
6873                     "Not started. E2E auth disabled. did=0x%x", ndlp->nlp_DID);

6875                 emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_DISABLED, 0, 0);

6877                 return (1);
6878         }
6879         if ((deferred_sbp && node_dhc->deferred_sbp) ||
6880             (deferred_ubp && node_dhc->deferred_ubp)) {
6881                 /* Clear previous authentication */
6882                 emlxs_dhc_auth_stop(port, ndlp);
6883         }
6884         mutex_enter(&hba->auth_lock);

6886         /* Intialize node */
6887         node_dhc->parent_auth_cfg = NULL;
6888         node_dhc->parent_auth_key = NULL;

6890         /* Acquire auth configuration */
6891         if (fabric_switch) {
6892                 auth_cfg = emlxs_auth_cfg_find(port,
6893                     (uint8_t *)emlxs_fabric_wwn);
6894                 auth_key = emlxs_auth_key_find(port,
6895                     (uint8_t *)emlxs_fabric_wwn);
6896         } else {
6897                 auth_cfg = emlxs_auth_cfg_find(port,
6898                     (uint8_t *)&ndlp->nlp_portname);
6899                 auth_key = emlxs_auth_key_find(port,
6900                     (uint8_t *)&ndlp->nlp_portname);
6901         }

6903         if (!auth_cfg) {
6904                 mutex_exit(&hba->auth_lock);

6906                 EMLXS_MSGF(EMLXS_CONTEXT,
6907                     &emlxs_fcsp_start_msg,
6908                     "Not started. No auth cfg entry found. did=0x%x",
6909                     ndlp->nlp_DID);

6911                 emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_DISABLED, 0, 0);

6913                 return (1);
6914         }
6915         if (fabric_switch) {
6916                 auth_cfg->node = NULL;
6917         } else {
6918                 node_dhc->parent_auth_cfg = auth_cfg;
6919                 auth_cfg->node = ndlp;
6920         }

6922         if (!auth_key) {
6923                 mutex_exit(&hba->auth_lock);

6925                 EMLXS_MSGF(EMLXS_CONTEXT,
```



```
6926                     &emlxs_fcsp_start_msg,
6927                     "Not started. No auth key entry found. did=0x%x",
6928                     ndlp->nlp_DID);

6930                 emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_DISABLED, 0, 0);

6932                 return (1);
6933         }
6934         if (fabric_switch) {
6935                 auth_key->node = NULL;
6936         } else {
6937                 node_dhc->parent_auth_key = auth_key;
6938                 auth_key->node = ndlp;
6939         }

6941         /* Remote port does not support fcsp */
6942         if (ndlp->sparm.cmn.fcsp_support == 0) {
6943                 switch (auth_cfg->authentication_mode) {
6944                 case AUTH_MODE_PASSIVE:
6945                         mutex_exit(&hba->auth_lock);

6947                         EMLXS_MSGF(EMLXS_CONTEXT,
6948                             &emlxs_fcsp_start_msg,
6949                             "Not started. Auth unsupported. did=0x%x",
6950                             ndlp->nlp_DID);

6952                         emlxs_dhc_state(port, ndlp,
6953                             NODE_STATE_AUTH_DISABLED, 0, 0);
6954                         return (1);

6956                 case AUTH_MODE_ACTIVE:
6957                         mutex_exit(&hba->auth_lock);

6959                         EMLXS_MSGF(EMLXS_CONTEXT,
6960                             &emlxs_fcsp_start_msg,
6961                             "Failed. Auth unsupported. did=0x%x",
6962                             ndlp->nlp_DID);

6964                         /*
6965                          * Save packet for deferred completion until
6966                          * authentication is complete
6967                          */
6968                         ndlp->node_dhc.deferred_sbp = deferred_sbp;
6969                         ndlp->node_dhc.deferred_ubp = deferred_ubp;

6971                         goto failed;

6973                 case AUTH_MODE_DISABLED:
6974                 default:
6975                         mutex_exit(&hba->auth_lock);

6977                         EMLXS_MSGF(EMLXS_CONTEXT,
6978                             &emlxs_fcsp_start_msg,
6979                             "Not started. Auth mode=disabled. did=0x%x",
6980                             ndlp->nlp_DID);

6982                         emlxs_dhc_state(port, ndlp,
6983                             NODE_STATE_AUTH_DISABLED, 0, 0);
6984                         return (1);
6985                 }
6986         } else {        /* Remote port supports fcsp */
6987                 switch (auth_cfg->authentication_mode) {
6988                 case AUTH_MODE_PASSIVE:
6989                 case AUTH_MODE_ACTIVE:
6990                         /* start auth */
6991                         break;
```

```
6993                case AUTH_MODE_DISABLED:
6994                default:
6995                        mutex_exit(&hba->auth_lock);

6997                        EMLXS_MSGF(EMLXS_CONTEXT,
6998                            &emlxs_fcsp_start_msg,
6999                            "Failed. Auth mode=disabled. did=0x%x",
7000                            ndlp->nlp_DID);

7002                        /*
7003                         * Save packet for deferred completion until
7004                         * authentication is complete
7005                         */
7006                        ndlp->node_dhc.deferred_sbp = deferred_sbp;
7007                        ndlp->node_dhc.deferred_ubp = deferred_ubp;

7009                        goto failed;
7010                }
7011        }

7013        /* We have a GO for authentication */

7015        /*
7016         * Save pointers for deferred completion until authentication is
7017         * complete
7018         */
7019        node_dhc->deferred_sbp = deferred_sbp;
7020        node_dhc->deferred_ubp = deferred_ubp;

7022        bzero(&node_dhc->auth_cfg, sizeof (node_dhc->auth_cfg));
7023        bzero(&node_dhc->auth_key, sizeof (node_dhc->auth_key));

7025        /* Program node's auth cfg */
7026        bcopy((uint8_t *)&port->wwpn,
7027            (uint8_t *)&node_dhc->auth_cfg.local_entity, 8);
7028        bcopy((uint8_t *)&ndlp->nlp_portname,
7029            (uint8_t *)&node_dhc->auth_cfg.remote_entity, 8);

7031        node_dhc->auth_cfg.authentication_timeout =
7032            auth_cfg->authentication_timeout;
7033        node_dhc->auth_cfg.authentication_mode =
7034            auth_cfg->authentication_mode;

7036        /*
7037         * If remote password type is "ignore", then only unidirectional auth
7038         * is allowed
7039         */
7040        if (auth_key->remote_password_type == 3) {
7041                node_dhc->auth_cfg.bidirectional = 0;
7042        } else {
7043                node_dhc->auth_cfg.bidirectional = auth_cfg->bidirectional;
7044        }

7046        node_dhc->auth_cfg.reauthenticate_time_interval =
7047            auth_cfg->reauthenticate_time_interval;

7049        for (i = 0; i < 4; i++) {
7050                switch (auth_cfg->authentication_type_priority[i]) {
7051                case ELX_DHCHAP:
7052                        node_dhc->auth_cfg.authentication_type_priority[i] =
7053                            AUTH_DHCHAP;
7054                        break;

7056                case ELX_FCAP:
7057                        node_dhc->auth_cfg.authentication_type_priority[i] =
```



```
7058                            AUTH_FCAP;
7059                        break;

7061                case ELX_FCPAP:
7062                        node_dhc->auth_cfg.authentication_type_priority[i] =
7063                            AUTH_FCPAP;
7064                        break;

7066                case ELX_KERBEROS:
7067                        node_dhc->auth_cfg.authentication_type_priority[i] =
7068                            AUTH_KERBEROS;
7069                        break;

7071                default:
7072                        node_dhc->auth_cfg.authentication_type_priority[i] =
7073                            0;
7074                        break;
7075                }

7077                switch (auth_cfg->hash_priority[i]) {
7078                case ELX_SHA1:
7079                        node_dhc->auth_cfg.hash_priority[i] = AUTH_SHA1;
7080                        break;

7082                case ELX_MD5:
7083                        node_dhc->auth_cfg.hash_priority[i] = AUTH_MD5;
7084                        break;

7086                default:
7087                        node_dhc->auth_cfg.hash_priority[i] = 0;
7088                        break;
7089                }
7090        }

7092        for (i = 0; i < 8; i++) {
7093                switch (auth_cfg->dh_group_priority[i]) {
7094                case ELX_GROUP_NULL:
7095                        node_dhc->auth_cfg.dh_group_priority[i] = GROUP_NULL;
7096                        break;

7098                case ELX_GROUP_1024:
7099                        node_dhc->auth_cfg.dh_group_priority[i] = GROUP_1024;
7100                        break;

7102                case ELX_GROUP_1280:
7103                        node_dhc->auth_cfg.dh_group_priority[i] = GROUP_1280;
7104                        break;

7106                case ELX_GROUP_1536:
7107                        node_dhc->auth_cfg.dh_group_priority[i] = GROUP_1536;
7108                        break;

7110                case ELX_GROUP_2048:
7111                        node_dhc->auth_cfg.dh_group_priority[i] = GROUP_2048;
7112                        break;

7114                default:
7115                        node_dhc->auth_cfg.dh_group_priority[i] = 0xF;
7116                        break;
7117                }
7118        }

7120        /* Program the node's key */
7121        if (auth_key) {
7122                bcopy((uint8_t *)auth_key,
7123                    (uint8_t *)&node_dhc->auth_key,
```

```
7124                    sizeof (emlxs_auth_key_t));
7125                node_dhc->auth_key.next = NULL;
7126                node_dhc->auth_key.prev = NULL;

7128                bcopy((uint8_t *)&port->wwpn,
7129                    (uint8_t *)&node_dhc->auth_key.local_entity, 8);
7130                bcopy((uint8_t *)&ndlp->nlp_portname,
7131                    (uint8_t *)&node_dhc->auth_key.remote_entity,
7132                    8);
7133        }
7134        mutex_exit(&hba->auth_lock);

7136        node_dhc->nlp_auth_limit = 2;
7137        node_dhc->nlp_fb_vendor = 1;

7139        node_dhc->nlp_authrsp_tmocnt = 0;
7140        node_dhc->nlp_authrsp_tmo = 0;

7142        if (deferred_ubp) {
7143                /* Acknowledge the unsolicited PLOGI */
7144                /* This should trigger the other port to start authentication */
7145                if (emlxs_ub_send_login_acc(port,
7146                    (fc_unsol_buf_t *)deferred_ubp) != FC_SUCCESS) {
7147                        EMLXS_MSGF(EMLXS_CONTEXT,
7148                            &emlxs_fcsp_start_msg,
7149                            "Not started. Unable to send PLOGI ACC. did=0x%x",
7150                            ndlp->nlp_DID);

7152                        goto failed;
7153                }
7154                /* Start the auth rsp timer */
7155                node_dhc->nlp_authrsp_tmo = DRV_TIME +
7156                    node_dhc->auth_cfg.authentication_timeout;

7158                EMLXS_MSGF(EMLXS_CONTEXT,
7159                    &emlxs_fcsp_start_msg,
7160                    "Authrsp timer activated. did=0x%x",
7161                    ndlp->nlp_DID);

7163                /* The next state should be emlxs_rcv_auth_msg_unmapped_node */
7164                emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_SUCCESS, 0, 0);
7165        } else {
7166                node_dhc->nlp_auth_flag = 1;    /* host is the initiator */

7168                EMLXS_MSGF(EMLXS_CONTEXT,
7169                    &emlxs_fcsp_start_msg,
7170                    "Auth initiated. did=0x%x limit=%d sbp=%p",
7171                    ndlp->nlp_DID, node_dhc->nlp_auth_limit, deferred_sbp);

7173                if (emlxs_issue_auth_negotiate(port, ndlp, 0)) {
7174                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_start_msg,
7175                            "Failed. Auth initiation failed. did=0x%x",
7176                            ndlp->nlp_DID);

7178                        goto failed;
7179                }
7180        }

7182        return (0);

7184 failed:

7186        emlxs_dhc_state(port, ndlp, NODE_STATE_AUTH_FAILED, 0, 0);

7188        /* Complete authentication with failed status */
7189        emlxs_dhc_auth_complete(port, ndlp, 1);
```



```
7191        return (0);

7193 } /* emlxs_dhc_auth_start() */



7197 /* This is called to indicate the driver has lost connection with this node */
7198 extern void
7199 emlxs_dhc_auth_stop(
7200        emlxs_port_t *port,
7201        emlxs_node_t *ndlp)
7202 {
7203        emlxs_port_dhc_t *port_dhc = &port->port_dhc;
7204        emlxs_node_dhc_t *node_dhc;
7205        uint32_t i;

7207        if (port_dhc->state == ELX_FABRIC_STATE_UNKNOWN) {
7208                /* Nothing to stop */
7209                return;
7210        }
7211        if (ndlp) {
7212                node_dhc = &ndlp->node_dhc;

7214                if (node_dhc->state == NODE_STATE_UNKNOWN) {
7215                        /* Nothing to stop */
7216                        return;
7217                }
7218                if (ndlp->nlp_DID != FABRIC_DID) {
7219                        emlxs_dhc_state(port, ndlp, NODE_STATE_UNKNOWN, 0, 0);
7220                }
7221                emlxs_dhc_auth_complete(port, ndlp, 2);
7222        } else {        /* Lost connection to all nodes for this port */
7223                rw_enter(&port->node_rwlock, RW_READER);
7224                for (i = 0; i < EMLXS_NUM_HASH_QUES; i++) {
7225                        ndlp = port->node_table[i];

7227                        if (!ndlp) {
7228                                continue;
7229                        }
7230                        node_dhc = &ndlp->node_dhc;

7232                        if (node_dhc->state == NODE_STATE_UNKNOWN) {
7233                                continue;
7234                        }
7235                        if (ndlp->nlp_DID != FABRIC_DID) {
7236                                emlxs_dhc_state(port, ndlp,
7237                                    NODE_STATE_UNKNOWN, 0, 0);
7238                        }
7239                        emlxs_dhc_auth_complete(port, ndlp, 2);
7240                }
7241                rw_exit(&port->node_rwlock);
7242        }

7244        return;

7246 } /* emlxs_dhc_auth_stop */


7249 /* state = 0   - Successful completion. Continue connection to node */
7250 /* state = 1   - Failed completion. Do not continue with connection to node */
7251 /* state = 2   - Stopped completion. Do not continue with connection to node */

7253 static void
7254 emlxs_dhc_auth_complete(
7255                        emlxs_port_t *port,
```

```
7256                        emlxs_node_t *ndlp,
7257                        uint32_t status)
7258 {
7259         emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
7260         uint32_t fabric;
7261         uint32_t fabric_switch;

7263         fabric = ((ndlp->nlp_DID & FABRIC_DID_MASK) == FABRIC_DID_MASK) ? 1 : 0;
7264         fabric_switch = ((ndlp->nlp_DID == FABRIC_DID) ? 1 : 0);

7266         EMLXS_MSGF(EMLXS_CONTEXT,
7267             &emlxs_fcsp_complete_msg,
7268             "did=0x%x status=%d sbp=%p ubp=%p",
7269             ndlp->nlp_DID, status, node_dhc->deferred_sbp,
7270             node_dhc->deferred_ubp);

7272         if (status == 1) {
7273                 if (fabric_switch) {
7274                         /* Virtual link down */
7275                         (void) emlxs_port_offline(port, 0xffffffff);
7276                 } else if (!fabric) {
7277                         /* Port offline */
7278                         (void) emlxs_port_offline(port, ndlp->nlp_DID);
7279                 }
7280         }
7281         /* Send a LOGO if authentication was not successful */
7282         if (status == 1) {
7283                 EMLXS_MSGF(EMLXS_CONTEXT,
7284                     &emlxs_fcsp_complete_msg,
7285                     "Sending LOGO to did=0x%x...",
7286                     ndlp->nlp_DID);
7287                 emlxs_send_logo(port, ndlp->nlp_DID);
7288         }

7290         /* Process deferred cmpl now */
7291         emlxs_mb_deferred_cmpl(port, status,
7292             (emlxs_buf_t *)node_dhc->deferred_sbp,
7293             (fc_unsol_buf_t *)node_dhc->deferred_ubp, 0);

7295         node_dhc->deferred_sbp = 0;
7296         node_dhc->deferred_ubp = 0;

7298         return;

7300 } /* emlxs_dhc_auth_complete */


7303 extern void
7304 emlxs_dhc_attach(emlxs_hba_t *hba)
7305 {
7306         char buf[32];

7308         (void) sprintf(buf, "%s_auth_lock mutex", DRIVER_NAME);
7309         mutex_init(&hba->auth_lock, buf, MUTEX_DRIVER, NULL);

7311         (void) sprintf(buf, "%s_dhc_lock mutex", DRIVER_NAME);
7312         mutex_init(&hba->dhc_lock, buf, MUTEX_DRIVER, NULL);

7314         emlxs_auth_cfg_init(hba);

7316         emlxs_auth_key_init(hba);

7318         hba->rdn_flag = 1;

7320         return;
```



```
7322 } /* emlxs_dhc_attach() */


7325 extern void
7326 emlxs_dhc_detach(emlxs_hba_t *hba)
7327 {
7328         emlxs_auth_cfg_fini(hba);

7330         emlxs_auth_key_fini(hba);

7332         mutex_destroy(&hba->dhc_lock);
7333         mutex_destroy(&hba->auth_lock);

7335         return;

7337 } /* emlxs_dhc_detach() */


7340 extern void
7341 emlxs_dhc_init_sp(emlxs_port_t *port, uint32_t did, SERV_PARM *sp, char **msg)
7342 {
7343         emlxs_hba_t *hba = HBA;
7344         emlxs_config_t *cfg = &CFG;
7345         uint32_t fabric;
7346         uint32_t fabric_switch;
7347         emlxs_auth_cfg_t *auth_cfg = NULL;
7348         emlxs_auth_key_t *auth_key = NULL;

7350         fabric = ((did & FABRIC_DID_MASK) == FABRIC_DID_MASK) ? 1 : 0;
7351         fabric_switch = ((did == FABRIC_DID) ? 1 : 0);

7353         /* Return is authentication is not enabled */
7354         if (cfg[CFG_AUTH_ENABLE].current == 0) {
7355                 sp->cmn.fcsp_support = 0;
7356                 bcopy("fcsp:Disabled (0)", (void *) &msg[0],
7357                     sizeof ("fcsp:Disabled (0)"));
7358                 return;
7359         }

7361         if (port->vpi != 0 && cfg[CFG_AUTH_NPIV].current == 0) {
7362                 sp->cmn.fcsp_support = 0;
7363                 bcopy("fcsp:Disabled (npiv)", (void *) &msg[0],
7364                     sizeof ("fcsp:Disabled (npiv)"));
7365                 return;
7366         }
7367         if (!fabric_switch && fabric) {
7368                 sp->cmn.fcsp_support = 0;
7369                 bcopy("fcsp:Disabled (fs)", (void *) &msg[0],
7370                     sizeof ("fcsp:Disabled (fs)"));
7371                 return;
7372         }
7373         /* Return if fcsp support to this node is not enabled */
7374         if (!fabric_switch && cfg[CFG_AUTH_E2E].current == 0) {
7375                 sp->cmn.fcsp_support = 0;
7376                 bcopy("fcsp:Disabled (e2e)", (void *) &msg[0],
7377                     sizeof ("fcsp:Disabled (e2e)"));
7378                 return;
7379         }

7381         mutex_enter(&hba->auth_lock);
7382         if (fabric_switch) {
7383                 auth_cfg = emlxs_auth_cfg_find(port,
7384                     (uint8_t *)emlxs_fabric_wwn);
7385                 auth_key = emlxs_auth_key_find(port,
7386                     (uint8_t *)emlxs_fabric_wwn);
7387                 if ((!auth_cfg) || (!auth_key)) {
```

```
7388                        sp->cmn.fcsp_support = 0;
7389                        bcopy("fcsp:Disabled (1)", (void *) &msg[0],
7390                            sizeof ("fcsp:Disabled (1)"));
7391                        mutex_exit(&hba->auth_lock);
7392                        return;
7393                }
7394        }
7395        mutex_exit(&hba->auth_lock);

7397        sp->cmn.fcsp_support = 1;

7399        return;

7401 } /* emlxs_dhc_init_sp() */


7404 extern uint32_t
7405 emlxs_dhc_verify_login(emlxs_port_t *port, uint32_t sid, SERV_PARM *sp)
7406 {
7407        emlxs_hba_t *hba = HBA;
7408        emlxs_config_t *cfg = &CFG;
7409        emlxs_auth_cfg_t *auth_cfg;
7410        emlxs_auth_key_t *auth_key;
7411        uint32_t fabric;
7412        uint32_t fabric_switch;

7414        fabric = ((sid & FABRIC_DID_MASK) == FABRIC_DID_MASK) ? 1 : 0;
7415        fabric_switch = ((sid == FABRIC_DID) ? 1 : 0);

7417        if (port->port_dhc.state == ELX_FABRIC_AUTH_FAILED) {
7418                /* Reject login */
7419                return (1);
7420        }
7421        /* Remote host supports FCSP */
7422        if (sp->cmn.fcsp_support) {
7423                /* Continue login */
7424                return (0);
7425        }
7426        /* Auth disabled in host */
7427        if (cfg[CFG_AUTH_ENABLE].current == 0) {
7428                /* Continue login */
7429                return (0);
7430        }
7431        /* Auth disabled for npiv */
7432        if (port->vpi != 0 && cfg[CFG_AUTH_NPIV].current == 0) {
7433                /* Continue login */
7434                return (0);
7435        }
7436        if (!fabric_switch && fabric) {
7437                /* Continue login */
7438                return (0);
7439        }
7440        /* Auth disabled for p2p */
7441        if (!fabric_switch && cfg[CFG_AUTH_E2E].current == 0) {
7442                /* Continue login */
7443                return (0);
7444        }

7446        /* Remote port does NOT support FCSP */
7447        /* Host has FCSP enabled */
7448        /* Now check to make sure auth mode for this port is also enabled */

7450        mutex_enter(&hba->auth_lock);

7452        /* Acquire auth configuration */
7453        if (fabric_switch) {
```



```
7454                auth_cfg = emlxs_auth_cfg_find(port,
7455                    (uint8_t *)emlxs_fabric_wwn);
7456                auth_key = emlxs_auth_key_find(port,
7457                    (uint8_t *)emlxs_fabric_wwn);
7458        } else {
7459                auth_cfg = emlxs_auth_cfg_find(port,
7460                    (uint8_t *)&sp->portName);
7461                auth_key = emlxs_auth_key_find(port,
7462                    (uint8_t *)&sp->portName);
7463        }

7465        if (auth_key && auth_cfg &&
7466            (auth_cfg->authentication_mode == AUTH_MODE_ACTIVE)) {
7467                mutex_exit(&hba->auth_lock);

7469                /* Reject login */
7470                return (1);
7471        }
7472        mutex_exit(&hba->auth_lock);

7474        return (0);

7476 } /* emlxs_dhc_verify_login() */


7479 /*
7480  * ! emlxs_dhc_reauth_timeout
7481  *
7482  * \pre \post \param phba \param arg1: \param arg2: ndlp to which the host
7483  * is to be authenticated. \return void
7484  *
7485  * \b Description:
7486  *
7487  * Timeout handler for reauthentication heartbeat.
7488  *
7489  * The reauthentication heart beat will be triggered 1 min by default after
7490  * the first authentication success. reauth_intval is
7491  * configurable. if reauth_intval is set to zero, it means no reauth heart
7492  * beat anymore.
7493  *
7494  * reauth heart beat will be triggered by IOCTL call from user space. Reauth
7495  * heart beat will go through the authentication process
7496  * all over again without causing IO traffic disruption. Initially it should
7497  * be triggered after authentication success.
7498  * Subsequently disable/enable reauth heart beat will be performed by
7499  * HBAnyware or other utility.
7500  *
7501  */
7502 /* ARGSUSED */
7503 extern void
7504 emlxs_dhc_reauth_timeout(
7505        emlxs_port_t *port,
7506        void *arg1,
7507        void *arg2)
7508 {
7509        emlxs_port_dhc_t *port_dhc = &port->port_dhc;
7510        NODELIST *ndlp = (NODELIST *) arg2;
7511        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;

7513        if (node_dhc->auth_cfg.reauthenticate_time_interval == 0) {
7514                EMLXS_MSGF(EMLXS_CONTEXT,
7515                    &emlxs_fcsp_debug_msg,
7516                    "Reauth timeout. Reauth no longer enabled. 0x%x %x",
7517                    ndlp->nlp_DID, node_dhc->state);

7519                emlxs_dhc_set_reauth_time(port, ndlp, DISABLE);
```

```
7521                return;
7522        }
7523        /* This should not happen!! */
7524        if (port_dhc->state == ELX_FABRIC_IN_AUTH) {
7525                EMLXS_MSGF(EMLXS_CONTEXT,
7526                    &emlxs_fcsp_error_msg,
7527                    "Reauth timeout. Fabric in auth. Quiting. 0x%x %x",
7528                    ndlp->nlp_DID, node_dhc->state);

7530                emlxs_dhc_set_reauth_time(port, ndlp, DISABLE);

7532                return;
7533        }
7534        if (node_dhc->state != NODE_STATE_AUTH_SUCCESS) {
7535                EMLXS_MSGF(EMLXS_CONTEXT,
7536                    &emlxs_fcsp_debug_msg,
7537                    "Reauth timeout. Auth not done. Restarting. 0x%x %x",
7538                    ndlp->nlp_DID, node_dhc->state);

7540                goto restart;
7541        }
7542        /*
7543         * This might happen, the ndlp is doing reauthencation. meaning ndlp
7544         * is being re-authenticated to the host. Thus not necessary to have
7545         * host re-authenticated to the ndlp at this point because ndlp might
7546         * support bi-directional auth. we can just simply donothing and
7547         * restart the timer.
7548         */
7549        if (port_dhc->state == ELX_FABRIC_IN_REAUTH) {
7550                EMLXS_MSGF(EMLXS_CONTEXT,
7551                    &emlxs_fcsp_debug_msg,
7552                    "Reauth timeout. Fabric in reauth. Restarting. 0x%x %x",
7553                    ndlp->nlp_DID, node_dhc->state);

7555                goto restart;
7556        }
7557        /*
7558         * node's reauth heart beat is running already, cancel it first and
7559         * then restart
7560         */
7561        if (node_dhc->nlp_reauth_status == NLP_HOST_REAUTH_IN_PROGRESS) {
7562                EMLXS_MSGF(EMLXS_CONTEXT,
7563                    &emlxs_fcsp_debug_msg,
7564                    "Reauth timeout. Fabric in reauth. Restarting. 0x%x %x",
7565                    ndlp->nlp_DID, node_dhc->state);

7567                goto restart;
7568        }
7569        EMLXS_MSGF(EMLXS_CONTEXT,
7570            &emlxs_fcsp_debug_msg,
7571            "Reauth timeout. Auth initiated. did=0x%x",
7572            ndlp->nlp_DID);

7574        emlxs_dhc_set_reauth_time(port, ndlp, ENABLE);
7575        node_dhc->nlp_reauth_status = NLP_HOST_REAUTH_IN_PROGRESS;

7577        /* Attempt to restart authentication */
7578        if (emlxs_dhc_auth_start(port, ndlp, NULL, NULL) != 0) {
7579                EMLXS_MSGF(EMLXS_CONTEXT,
7580                    &emlxs_fcsp_debug_msg,
7581                    "Reauth timeout. Auth initiation failed. 0x%x %x",
7582                    ndlp->nlp_DID, node_dhc->state);

7584                return;
7585        }
```



```
7586        return;

7588 restart:

7590        emlxs_dhc_set_reauth_time(port, ndlp, ENABLE);

7592        return;

7594 } /* emlxs_dhc_reauth_timeout */


7597 static void
7598 emlxs_dhc_set_reauth_time(
7599        emlxs_port_t *port,
7600        emlxs_node_t *ndlp,
7601        uint32_t status)
7602 {
7603        emlxs_port_dhc_t *port_dhc = &port->port_dhc;
7604        emlxs_node_dhc_t *node_dhc = &ndlp->node_dhc;
7605        uint32_t drv_time;
7606        uint32_t timeout;
7607        uint32_t reauth_tmo;
7608        uint32_t last_auth_time;
  26        time_t last_auth_time;

7610        node_dhc->flag &= ~NLP_SET_REAUTH_TIME;

7612        if ((status == ENABLE) &&
7613            node_dhc->auth_cfg.reauthenticate_time_interval) {

7615                timeout =
7616                    (60 * node_dhc->auth_cfg.reauthenticate_time_interval);
7617                drv_time = DRV_TIME;

7619                /* Get last successful auth time */
7620                if (ndlp->nlp_DID == FABRIC_DID) {
7621                        last_auth_time = port_dhc->auth_time;
7622                } else if (node_dhc->parent_auth_cfg) {
7623                        last_auth_time = node_dhc->parent_auth_cfg->auth_time;
7624                } else {
7625                        last_auth_time = 0;
7626                }

7628                if (last_auth_time) {
7629                        reauth_tmo = last_auth_time + timeout;

7631                        /* Validate reauth_tmo */
7632                        if ((reauth_tmo < drv_time) ||
7633                            (reauth_tmo > drv_time + timeout)) {
7634                                reauth_tmo = drv_time + timeout;
7635                        }
7636                } else {
7637                        reauth_tmo = drv_time + timeout;
7638                }

7640                node_dhc->nlp_reauth_tmo = reauth_tmo;
7641                node_dhc->nlp_reauth_status = NLP_HOST_REAUTH_ENABLED;

7643                EMLXS_MSGF(EMLXS_CONTEXT,
7644                    &emlxs_fcsp_debug_msg,
7645                    "Reauth enabled. did=0x%x state=%x tmo=%d,%d",
7646                    ndlp->nlp_DID, node_dhc->state,
7647                    node_dhc->auth_cfg.reauthenticate_time_interval,
7648                    (reauth_tmo - drv_time));

7650        } else {
```

```
7651                node_dhc->nlp_reauth_tmo = 0;
7652                node_dhc->nlp_reauth_status = NLP_HOST_REAUTH_DISABLED;

7654                EMLXS_MSGF(EMLXS_CONTEXT,
7655                    &emlxs_fcsp_debug_msg,
7656                    "Reauth disabled. did=0x%x state=%x",
7657                    ndlp->nlp_DID, node_dhc->state);
7658        }

7660        return;

7662 } /* emlxs_dhc_set_reauth_time */
______unchanged_portion_omitted_


9651 /* Provides DFC support for emlxs_dfc_get_auth_status() */
9652 extern uint32_t
9653 emlxs_dhc_get_auth_status(emlxs_hba_t *hba, dfc_auth_status_t *fcsp_status)
9654 {
9655        emlxs_port_t *port = &PPORT;
9656        emlxs_config_t *cfg = &CFG;
9657        char s_lwwpn[64];
9658        char s_rwwpn[64];
9659        emlxs_auth_cfg_t *auth_cfg;
9660        dfc_auth_status_t *auth_status;
9661        NODELIST *ndlp;
9662        uint32_t rc;
9663        uint32_t auth_time;
2081        time_t auth_time;
9664        uint32_t update;

9666        /* Return is authentication is not enabled */
9667        if (cfg[CFG_AUTH_ENABLE].current == 0) {
9668                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_fcsp_debug_msg,
9669                    "emlxs_dhc_get_auth_status. Auth disabled.");

9671                return (DFC_AUTH_AUTHENTICATION_DISABLED);
9672        }
9673        mutex_enter(&hba->auth_lock);

9675        auth_cfg = emlxs_auth_cfg_get(hba, (uint8_t *)&fcsp_status->lwwpn,
9676            (uint8_t *)&fcsp_status->rwwpn);

9678        if (!auth_cfg) {
9679                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_dfc_error_msg,
9680                    "emlxs_dhc_get_auth_status: entry not found. %s:%s",
9681                    emlxs_wwn_xlate(s_lwwpn, (uint8_t *)&fcsp_status->lwwpn),
9682                    emlxs_wwn_xlate(s_rwwpn, (uint8_t *)&fcsp_status->rwwpn));

9684                mutex_exit(&hba->auth_lock);

9686                return (DFC_AUTH_NOT_CONFIGURED);
9687        }
9688        if (bcmp((uint8_t *)&fcsp_status->rwwpn,
9689            (uint8_t *)emlxs_fabric_wwn, 8) == 0) {
9690                auth_status = &port->port_dhc.auth_status;
9691                auth_time = port->port_dhc.auth_time;
9692                ndlp = emlxs_node_find_did(port, FABRIC_DID);
9693        } else {
9694                auth_status = &auth_cfg->auth_status;
9695                auth_time = auth_cfg->auth_time;
9696                ndlp = auth_cfg->node;
9697        }

9699        update = 0;
```



```
9701        /* Check if node is still available */
9702        if (ndlp && ndlp->nlp_active) {
9703                emlxs_dhc_status(port, ndlp, 0, 0);
9704                update = 1;
9705        } else {
9706                rc = DFC_AUTH_WWN_NOT_FOUND;
9707        }


9710        if (update) {
9711                fcsp_status->auth_state = auth_status->auth_state;
9712                fcsp_status->auth_failReason = auth_status->auth_failReason;
9713                fcsp_status->type_priority = auth_status->type_priority;
9714                fcsp_status->group_priority = auth_status->group_priority;
9715                fcsp_status->hash_priority = auth_status->hash_priority;
9716                fcsp_status->localAuth = auth_status->localAuth;
9717                fcsp_status->remoteAuth = auth_status->remoteAuth;
9718                fcsp_status->time_from_last_auth = DRV_TIME - auth_time;
9719                fcsp_status->time_until_next_auth =
9720                    auth_status->time_until_next_auth;

9722                rc = 0;
9723        } else {
9724                rc = DFC_AUTH_WWN_NOT_FOUND;
9725        }

9727        mutex_exit(&hba->auth_lock);

9729        return (rc);

9731 } /* emlxs_dhc_get_auth_status() */
______unchanged_portion_omitted_
```

```
**********************************************************
  275456 Mon May  5 14:29:42 2014
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_solaris.c
4786 emlxs shouldn't abuse ddi_get_time(9f)
**********************************************************
/*
 * 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 Emulex.  All rights reserved.
 * Use is subject to license terms.
 * Copyright (c) 2011 Bayard G. Bell. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */


#define DEF_ICFG        1

#include <emlxs.h>
#include <emlxs_version.h>


char emlxs_revision[] = EMLXS_REVISION;
char emlxs_version[] = EMLXS_VERSION;
char emlxs_name[] = EMLXS_NAME;
char emlxs_label[] = EMLXS_LABEL;

/* Required for EMLXS_CONTEXT in EMLXS_MSGF calls */
EMLXS_MSG_DEF(EMLXS_SOLARIS_C);

#ifdef MENLO_SUPPORT
static int32_t  emlxs_send_menlo(emlxs_port_t *port, emlxs_buf_t *sbp);
#endif /* MENLO_SUPPORT */

static void     emlxs_fca_attach(emlxs_hba_t *hba);
static void     emlxs_fca_detach(emlxs_hba_t *hba);
static void     emlxs_drv_banner(emlxs_hba_t *hba);

static int32_t  emlxs_get_props(emlxs_hba_t *hba);
static int32_t  emlxs_send_fcp_cmd(emlxs_port_t *port, emlxs_buf_t *sbp,
                    uint32_t *pkt_flags);
static int32_t  emlxs_send_fct_status(emlxs_port_t *port, emlxs_buf_t *sbp);
static int32_t  emlxs_send_fct_abort(emlxs_port_t *port, emlxs_buf_t *sbp);
static int32_t  emlxs_send_ip(emlxs_port_t *port, emlxs_buf_t *sbp);
static int32_t  emlxs_send_els(emlxs_port_t *port, emlxs_buf_t *sbp);
static int32_t  emlxs_send_els_rsp(emlxs_port_t *port, emlxs_buf_t *sbp);
static int32_t  emlxs_send_ct(emlxs_port_t *port, emlxs_buf_t *sbp);
```



```
static int32_t  emlxs_send_ct_rsp(emlxs_port_t *port, emlxs_buf_t *sbp);
static uint32_t emlxs_add_instance(int32_t ddiinst);
static void     emlxs_iodone(emlxs_buf_t *sbp);
static int      emlxs_pm_lower_power(dev_info_t *dip);
static int      emlxs_pm_raise_power(dev_info_t *dip);
static void     emlxs_driver_remove(dev_info_t *dip, uint32_t init_flag,
                    uint32_t failed);
static void     emlxs_iodone_server(void *arg1, void *arg2, void *arg3);
static uint32_t emlxs_integrity_check(emlxs_hba_t *hba);
static uint32_t emlxs_test(emlxs_hba_t *hba, uint32_t test_code,
                    uint32_t args, uint32_t *arg);

#if (EMLXS_MODREV >= EMLXS_MODREV3) && (EMLXS_MODREV <= EMLXS_MODREV4)
static void     emlxs_read_vport_prop(emlxs_hba_t *hba);
#endif  /* EMLXS_MODREV3 || EMLXS_MODREV4 */



extern int
emlxs_msiid_to_chan(emlxs_hba_t *hba, int msi_id);
extern int
emlxs_select_msiid(emlxs_hba_t *hba);

/*
 * Driver Entry Routines.
 */
static int32_t  emlxs_detach(dev_info_t *, ddi_detach_cmd_t);
static int32_t  emlxs_attach(dev_info_t *, ddi_attach_cmd_t);
static int32_t  emlxs_open(dev_t *, int32_t, int32_t, cred_t *);
static int32_t  emlxs_close(dev_t, int32_t, int32_t, cred_t *);
static int32_t  emlxs_ioctl(dev_t, int32_t, intptr_t, int32_t,
                    cred_t *, int32_t *);
static int32_t  emlxs_info(dev_info_t *, ddi_info_cmd_t, void *, void **);


/*
 * FC_AL Transport Functions.
 */
static opaque_t emlxs_fca_bind_port(dev_info_t *, fc_fca_port_info_t *,
                    fc_fca_bind_info_t *);
static void     emlxs_fca_unbind_port(opaque_t);
static void     emlxs_initialize_pkt(emlxs_port_t *, emlxs_buf_t *);
static int32_t  emlxs_fca_get_cap(opaque_t, char *, void *);
static int32_t  emlxs_fca_set_cap(opaque_t, char *, void *);
static int32_t  emlxs_fca_get_map(opaque_t, fc_lilpmap_t *);
static int32_t  emlxs_fca_ub_alloc(opaque_t, uint64_t *, uint32_t,
                    uint32_t *, uint32_t);
static int32_t  emlxs_fca_ub_free(opaque_t, uint32_t, uint64_t *);

static opaque_t emlxs_fca_get_device(opaque_t, fc_portid_t);
static int32_t  emlxs_fca_notify(opaque_t, uint32_t);
static void     emlxs_ub_els_reject(emlxs_port_t *, fc_unsol_buf_t *);

/*
 * Driver Internal Functions.
 */

static void     emlxs_poll(emlxs_port_t *, emlxs_buf_t *);
static int32_t  emlxs_power(dev_info_t *, int32_t, int32_t);
#ifdef EMLXS_I386
#ifdef S11
static int32_t  emlxs_quiesce(dev_info_t *);
#endif
#endif
static int32_t  emlxs_hba_resume(dev_info_t *);
static int32_t  emlxs_hba_suspend(dev_info_t *);
```

```
128 static int32_t  emlxs_hba_detach(dev_info_t *);
129 static int32_t  emlxs_hba_attach(dev_info_t *);
130 static void     emlxs_lock_destroy(emlxs_hba_t *);
131 static void     emlxs_lock_init(emlxs_hba_t *);

133 char *emlxs_pm_components[] = {
134         "NAME=emlxx000",
135         "0=Device D3 State",
136         "1=Device D0 State"
137 };


140 /*
141  * Default emlx dma limits
142  */
143 ddi_dma_lim_t emlxs_dma_lim = {
144         (uint32_t)0,                            /* dlim_addr_lo */
145         (uint32_t)0xffffffff,                   /* dlim_addr_hi */
146         (uint_t)0x00ffffff,                     /* dlim_cntr_max */
147         DEFAULT_BURSTSIZE | BURST32 | BURST64,  /* dlim_burstsizes */
148         1,                                      /* dlim_minxfer */
149         0x00ffffff                              /* dlim_dmaspeed */
150 };

152 /*
153  * Be careful when using these attributes; the defaults listed below are
154  * (almost) the most general case, permitting allocation in almost any
155  * way supported by the LightPulse family.  The sole exception is the
156  * alignment specified as requiring memory allocation on a 4-byte boundary;
157  * the Lightpulse can DMA memory on any byte boundary.
158  *
159  * The LightPulse family currently is limited to 16M transfers;
160  * this restriction affects the dma_attr_count_max and dma_attr_maxxfer fields.
161  */
162 ddi_dma_attr_t emlxs_dma_attr = {
163         DMA_ATTR_V0,                            /* dma_attr_version */
164         (uint64_t)0,                            /* dma_attr_addr_lo */
165         (uint64_t)0xffffffffffffffff,           /* dma_attr_addr_hi */
166         (uint64_t)0x00ffffff,                   /* dma_attr_count_max */
167         1,                                      /* dma_attr_align */
168         DEFAULT_BURSTSIZE | BURST32 | BURST64,  /* dma_attr_burstsizes */
169         1,                                      /* dma_attr_minxfer */
170         (uint64_t)0x00ffffff,                   /* dma_attr_maxxfer */
171         (uint64_t)0xffffffff,                   /* dma_attr_seg */
172         EMLXS_SGLLEN,                           /* dma_attr_sgllen */
173         1,                                      /* dma_attr_granular */
174         0                                       /* dma_attr_flags */
175 };

177 ddi_dma_attr_t emlxs_dma_attr_ro = {
178         DMA_ATTR_V0,                            /* dma_attr_version */
179         (uint64_t)0,                            /* dma_attr_addr_lo */
180         (uint64_t)0xffffffffffffffff,           /* dma_attr_addr_hi */
181         (uint64_t)0x00ffffff,                   /* dma_attr_count_max */
182         1,                                      /* dma_attr_align */
183         DEFAULT_BURSTSIZE | BURST32 | BURST64,  /* dma_attr_burstsizes */
184         1,                                      /* dma_attr_minxfer */
185         (uint64_t)0x00ffffff,                   /* dma_attr_maxxfer */
186         (uint64_t)0xffffffff,                   /* dma_attr_seg */
187         EMLXS_SGLLEN,                           /* dma_attr_sgllen */
188         1,                                      /* dma_attr_granular */
189         DDI_DMA_RELAXED_ORDERING                /* dma_attr_flags */
190 };

192 ddi_dma_attr_t emlxs_dma_attr_1sg = {
193         DMA_ATTR_V0,                            /* dma_attr_version */
```



```
194         (uint64_t)0,                            /* dma_attr_addr_lo */
195         (uint64_t)0xffffffffffffffff,           /* dma_attr_addr_hi */
196         (uint64_t)0x00ffffff,                   /* dma_attr_count_max */
197         1,                                      /* dma_attr_align */
198         DEFAULT_BURSTSIZE | BURST32 | BURST64,  /* dma_attr_burstsizes */
199         1,                                      /* dma_attr_minxfer */
200         (uint64_t)0x00ffffff,                   /* dma_attr_maxxfer */
201         (uint64_t)0xffffffff,                   /* dma_attr_seg */
202         1,                                      /* dma_attr_sgllen */
203         1,                                      /* dma_attr_granular */
204         0                                       /* dma_attr_flags */
205 };

207 #if (EMLXS_MODREV >= EMLXS_MODREV3)
208 ddi_dma_attr_t emlxs_dma_attr_fcip_rsp = {
209         DMA_ATTR_V0,                            /* dma_attr_version */
210         (uint64_t)0,                            /* dma_attr_addr_lo */
211         (uint64_t)0xffffffffffffffff,           /* dma_attr_addr_hi */
212         (uint64_t)0x00ffffff,                   /* dma_attr_count_max */
213         1,                                      /* dma_attr_align */
214         DEFAULT_BURSTSIZE | BURST32 | BURST64,  /* dma_attr_burstsizes */
215         1,                                      /* dma_attr_minxfer */
216         (uint64_t)0x00ffffff,                   /* dma_attr_maxxfer */
217         (uint64_t)0xffffffff,                   /* dma_attr_seg */
218         EMLXS_SGLLEN,                           /* dma_attr_sgllen */
219         1,                                      /* dma_attr_granular */
220         0                                       /* dma_attr_flags */
221 };
222 #endif  /* >= EMLXS_MODREV3 */

224 /*
225  * DDI access attributes for device
226  */
227 ddi_device_acc_attr_t emlxs_dev_acc_attr = {
228         DDI_DEVICE_ATTR_V1,     /* devacc_attr_version          */
229         DDI_STRUCTURE_LE_ACC,   /* PCI is Little Endian         */
230         DDI_STRICTORDER_ACC,    /* devacc_attr_dataorder        */
231         DDI_DEFAULT_ACC         /* devacc_attr_access           */
232 };

234 /*
235  * DDI access attributes for data
236  */
237 ddi_device_acc_attr_t emlxs_data_acc_attr = {
238         DDI_DEVICE_ATTR_V1,     /* devacc_attr_version          */
239         DDI_NEVERSWAP_ACC,      /* don't swap for Data          */
240         DDI_STRICTORDER_ACC,    /* devacc_attr_dataorder        */
241         DDI_DEFAULT_ACC         /* devacc_attr_access           */
242 };

244 /*
245  * Fill in the FC Transport structure,
246  * as defined in the Fibre Channel Transport Programmming Guide.
247  */
248 #if (EMLXS_MODREV == EMLXS_MODREV5)
249         static fc_fca_tran_t emlxs_fca_tran = {
250         FCTL_FCA_MODREV_5,              /* fca_version, with SUN NPIV support */
251         MAX_VPORTS,                     /* fca numerb of ports */
252         sizeof (emlxs_buf_t),           /* fca pkt size */
253         2048,                           /* fca cmd max */
254         &emlxs_dma_lim,                 /* fca dma limits */
255         0,                              /* fca iblock, to be filled in later */
256         &emlxs_dma_attr,                /* fca dma attributes */
257         &emlxs_dma_attr_1sg,            /* fca dma fcp cmd attributes */
258         &emlxs_dma_attr_1sg,            /* fca dma fcp rsp attributes */
259         &emlxs_dma_attr_ro,             /* fca dma fcp data attributes */
```

```
260         &emlxs_dma_attr_1sg,            /* fca dma fcip cmd attributes */
261         &emlxs_dma_attr_fcip_rsp,       /* fca dma fcip rsp attributes */
262         &emlxs_dma_attr_1sg,            /* fca dma fcsm cmd attributes */
263         &emlxs_dma_attr,                /* fca dma fcsm rsp attributes */
264         &emlxs_data_acc_attr,           /* fca access atributes */
265         0,                              /* fca_num_npivports */
266         {0, 0, 0, 0, 0, 0, 0, 0},       /* Physical port WWPN */
267         emlxs_fca_bind_port,
268         emlxs_fca_unbind_port,
269         emlxs_fca_pkt_init,
270         emlxs_fca_pkt_uninit,
271         emlxs_fca_transport,
272         emlxs_fca_get_cap,
273         emlxs_fca_set_cap,
274         emlxs_fca_get_map,
275         emlxs_fca_transport,
276         emlxs_fca_ub_alloc,
277         emlxs_fca_ub_free,
278         emlxs_fca_ub_release,
279         emlxs_fca_pkt_abort,
280         emlxs_fca_reset,
281         emlxs_fca_port_manage,
282         emlxs_fca_get_device,
283         emlxs_fca_notify
284 };
285 #endif  /* EMLXS_MODREV5 */


288 #if (EMLXS_MODREV == EMLXS_MODREV4)
289 static fc_fca_tran_t emlxs_fca_tran = {
290         FCTL_FCA_MODREV_4,              /* fca_version */
291         MAX_VPORTS,                     /* fca numerb of ports */
292         sizeof (emlxs_buf_t),           /* fca pkt size */
293         2048,                           /* fca cmd max */
294         &emlxs_dma_lim,                 /* fca dma limits */
295         0,                              /* fca iblock, to be filled in later */
296         &emlxs_dma_attr,                /* fca dma attributes */
297         &emlxs_dma_attr_1sg,            /* fca dma fcp cmd attributes */
298         &emlxs_dma_attr_1sg,            /* fca dma fcp rsp attributes */
299         &emlxs_dma_attr_ro,             /* fca dma fcp data attributes */
300         &emlxs_dma_attr_1sg,            /* fca dma fcip cmd attributes */
301         &emlxs_dma_attr_fcip_rsp,       /* fca dma fcip rsp attributes */
302         &emlxs_dma_attr_1sg,            /* fca dma fcsm cmd attributes */
303         &emlxs_dma_attr,                /* fca dma fcsm rsp attributes */
304         &emlxs_data_acc_attr,           /* fca access atributes */
305         emlxs_fca_bind_port,
306         emlxs_fca_unbind_port,
307         emlxs_fca_pkt_init,
308         emlxs_fca_pkt_uninit,
309         emlxs_fca_transport,
310         emlxs_fca_get_cap,
311         emlxs_fca_set_cap,
312         emlxs_fca_get_map,
313         emlxs_fca_transport,
314         emlxs_fca_ub_alloc,
315         emlxs_fca_ub_free,
316         emlxs_fca_ub_release,
317         emlxs_fca_pkt_abort,
318         emlxs_fca_reset,
319         emlxs_fca_port_manage,
320         emlxs_fca_get_device,
321         emlxs_fca_notify
322 };
323 #endif  /* EMLXS_MODEREV4 */
```



```
326 #if (EMLXS_MODREV == EMLXS_MODREV3)
327 static fc_fca_tran_t emlxs_fca_tran = {
328         FCTL_FCA_MODREV_3,              /* fca_version */
329         MAX_VPORTS,                     /* fca numerb of ports */
330         sizeof (emlxs_buf_t),           /* fca pkt size */
331         2048,                           /* fca cmd max */
332         &emlxs_dma_lim,                 /* fca dma limits */
333         0,                              /* fca iblock, to be filled in later */
334         &emlxs_dma_attr,                /* fca dma attributes */
335         &emlxs_dma_attr_1sg,            /* fca dma fcp cmd attributes */
336         &emlxs_dma_attr_1sg,            /* fca dma fcp rsp attributes */
337         &emlxs_dma_attr_ro,             /* fca dma fcp data attributes */
338         &emlxs_dma_attr_1sg,            /* fca dma fcip cmd attributes */
339         &emlxs_dma_attr_fcip_rsp,       /* fca dma fcip rsp attributes */
340         &emlxs_dma_attr_1sg,            /* fca dma fcsm cmd attributes */
341         &emlxs_dma_attr,                /* fca dma fcsm rsp attributes */
342         &emlxs_data_acc_attr,           /* fca access atributes */
343         emlxs_fca_bind_port,
344         emlxs_fca_unbind_port,
345         emlxs_fca_pkt_init,
346         emlxs_fca_pkt_uninit,
347         emlxs_fca_transport,
348         emlxs_fca_get_cap,
349         emlxs_fca_set_cap,
350         emlxs_fca_get_map,
351         emlxs_fca_transport,
352         emlxs_fca_ub_alloc,
353         emlxs_fca_ub_free,
354         emlxs_fca_ub_release,
355         emlxs_fca_pkt_abort,
356         emlxs_fca_reset,
357         emlxs_fca_port_manage,
358         emlxs_fca_get_device,
359         emlxs_fca_notify
360 };
361 #endif  /* EMLXS_MODREV3 */


364 #if (EMLXS_MODREV == EMLXS_MODREV2)
365 static fc_fca_tran_t emlxs_fca_tran = {
366         FCTL_FCA_MODREV_2,              /* fca_version */
367         MAX_VPORTS,                     /* number of ports */
368         sizeof (emlxs_buf_t),           /* pkt size */
369         2048,                           /* max cmds */
370         &emlxs_dma_lim,                 /* DMA limits */
371         0,                              /* iblock, to be filled in later */
372         &emlxs_dma_attr,                /* dma attributes */
373         &emlxs_data_acc_attr,           /* access atributes */
374         emlxs_fca_bind_port,
375         emlxs_fca_unbind_port,
376         emlxs_fca_pkt_init,
377         emlxs_fca_pkt_uninit,
378         emlxs_fca_transport,
379         emlxs_fca_get_cap,
380         emlxs_fca_set_cap,
381         emlxs_fca_get_map,
382         emlxs_fca_transport,
383         emlxs_fca_ub_alloc,
384         emlxs_fca_ub_free,
385         emlxs_fca_ub_release,
386         emlxs_fca_pkt_abort,
387         emlxs_fca_reset,
388         emlxs_fca_port_manage,
389         emlxs_fca_get_device,
390         emlxs_fca_notify
391 };
```

```
392 #endif  /* EMLXS_MODREV2 */

394 /*
395  * state pointer which the implementation uses as a place to
396  * hang a set of per-driver structures;
397  *
398  */
399 void            *emlxs_soft_state = NULL;

401 /*
402  * Driver Global variables.
403  */
404 int32_t         emlxs_scsi_reset_delay = 3000;  /* milliseconds */

406 emlxs_device_t  emlxs_device;

408 uint32_t        emlxs_instance[MAX_FC_BRDS];    /* uses emlxs_device.lock */
409 uint32_t        emlxs_instance_count = 0;       /* uses emlxs_device.lock */
410 uint32_t        emlxs_instance_flag = 0;        /* uses emlxs_device.lock */
411 #define EMLXS_FW_SHOW           0x00000001


414 /*
415  * Single private "global" lock used to gain access to
416  * the hba_list and/or any other case where we want need to be
417  * single-threaded.
418  */
419 uint32_t        emlxs_diag_state;

421 /*
422  * CB ops vector.  Used for administration only.
423  */
424 static struct cb_ops emlxs_cb_ops = {
425         emlxs_open,     /* cb_open      */
426         emlxs_close,    /* cb_close     */
427         nodev,          /* cb_strategy  */
428         nodev,          /* cb_print     */
429         nodev,          /* cb_dump      */
430         nodev,          /* cb_read      */
431         nodev,          /* cb_write     */
432         emlxs_ioctl,    /* cb_ioctl     */
433         nodev,          /* cb_devmap    */
434         nodev,          /* cb_mmap      */
435         nodev,          /* cb_segmap    */
436         nochpoll,       /* cb_chpoll    */
437         ddi_prop_op,    /* cb_prop_op   */
438         0,              /* cb_stream    */
439 #ifdef _LP64
440         D_64BIT | D_HOTPLUG | D_MP | D_NEW,     /* cb_flag */
441 #else
442         D_HOTPLUG | D_MP | D_NEW,               /* cb_flag */
443 #endif
444         CB_REV,         /* rev          */
445         nodev,          /* cb_aread     */
446         nodev           /* cb_awrite    */
447 };

449 static struct dev_ops emlxs_ops = {
450         DEVO_REV,       /* rev */
451         0,      /* refcnt */
452         emlxs_info,     /* getinfo      */
453         nulldev,        /* identify     */
454         nulldev,        /* probe        */
455         emlxs_attach,   /* attach       */
456         emlxs_detach,   /* detach       */
457         nodev,          /* reset        */
```



```
458         &emlxs_cb_ops,  /* devo_cb_ops  */
459         NULL,           /* devo_bus_ops */
460         emlxs_power,    /* power ops    */
461 #ifdef EMLXS_I386
462 #ifdef S11
463         emlxs_quiesce,  /* quiesce      */
464 #endif
465 #endif
466 };

468 #include <sys/modctl.h>
469 extern struct mod_ops mod_driverops;

471 #ifdef SAN_DIAG_SUPPORT
472 extern kmutex_t         sd_bucket_mutex;
473 extern sd_bucket_info_t sd_bucket;
474 #endif /* SAN_DIAG_SUPPORT */

476 /*
477  * Module linkage information for the kernel.
478  */
479 static struct modldrv emlxs_modldrv = {
480         &mod_driverops, /* module type - driver */
481         emlxs_name,     /* module name */
482         &emlxs_ops,     /* driver ops */
483 };


486 /*
487  * Driver module linkage structure
488  */
489 static struct modlinkage emlxs_modlinkage = {
490         MODREV_1,       /* ml_rev - must be MODREV_1 */
491         &emlxs_modldrv, /* ml_linkage */
492         NULL    /* end of driver linkage */
493 };


496 /* We only need to add entries for non-default return codes. */
497 /* Entries do not need to be in order. */
498 /* Default:     FC_PKT_TRAN_ERROR,      FC_REASON_ABORTED, */
499 /*              FC_EXPLN_NONE,          FC_ACTION_RETRYABLE */

501 emlxs_xlat_err_t emlxs_iostat_tbl[] = {
502 /*      {f/w code, pkt_state, pkt_reason,       */
503 /*              pkt_expln, pkt_action}          */

505         /* 0x00 - Do not remove */
506         {IOSTAT_SUCCESS, FC_PKT_SUCCESS, FC_REASON_NONE,
507                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

509         /* 0x01 - Do not remove */
510         {IOSTAT_FCP_RSP_ERROR, FC_PKT_SUCCESS, FC_REASON_NONE,
511                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

513         /* 0x02 */
514         {IOSTAT_REMOTE_STOP, FC_PKT_REMOTE_STOP, FC_REASON_ABTS,
515                 FC_EXPLN_NONE, FC_ACTION_NON_RETRYABLE},

517         /*
518          * This is a default entry.
519          * The real codes are written dynamically in emlxs_els.c
520          */
521         /* 0x09 */
522         {IOSTAT_LS_RJT, FC_PKT_LS_RJT, FC_REASON_CMD_UNABLE,
523                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},
```

```
525         /* Special error code */
526         /* 0x10 */
527         {IOSTAT_DATA_OVERRUN, FC_PKT_TRAN_ERROR, FC_REASON_OVERRUN,
528                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

530         /* Special error code */
531         /* 0x11 */
532         {IOSTAT_DATA_UNDERRUN, FC_PKT_TRAN_ERROR, FC_REASON_ABORTED,
533                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

535         /* CLASS 2 only */
536         /* 0x04 */
537         {IOSTAT_NPORT_RJT, FC_PKT_NPORT_RJT, FC_REASON_PROTOCOL_ERROR,
538                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

540         /* CLASS 2 only */
541         /* 0x05 */
542         {IOSTAT_FABRIC_RJT, FC_PKT_FABRIC_RJT, FC_REASON_PROTOCOL_ERROR,
543                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

545         /* CLASS 2 only */
546         /* 0x06 */
547         {IOSTAT_NPORT_BSY, FC_PKT_NPORT_BSY, FC_REASON_PHYSICAL_BUSY,
548                 FC_EXPLN_NONE, FC_ACTION_SEQ_TERM_RETRY},

550         /* CLASS 2 only */
551         /* 0x07 */
552         {IOSTAT_FABRIC_BSY, FC_PKT_FABRIC_BSY, FC_REASON_FABRIC_BSY,
553                 FC_EXPLN_NONE, FC_ACTION_SEQ_TERM_RETRY},
554 };

556 #define IOSTAT_MAX (sizeof (emlxs_iostat_tbl)/sizeof (emlxs_xlat_err_t))


559 /* We only need to add entries for non-default return codes. */
560 /* Entries do not need to be in order. */
561 /* Default:     FC_PKT_TRAN_ERROR,      FC_REASON_ABORTED, */
562 /*              FC_EXPLN_NONE,          FC_ACTION_RETRYABLE} */

564 emlxs_xlat_err_t emlxs_ioerr_tbl[] = {
565 /*      {f/w code, pkt_state, pkt_reason,       */
566 /*              pkt_expln, pkt_action}          */

568         /* 0x01 */
569         {IOERR_MISSING_CONTINUE, FC_PKT_TRAN_ERROR, FC_REASON_OVERRUN,
570                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

572         /* 0x02 */
573         {IOERR_SEQUENCE_TIMEOUT, FC_PKT_TIMEOUT, FC_REASON_SEQ_TIMEOUT,
574                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

576         /* 0x04 */
577         {IOERR_INVALID_RPI, FC_PKT_PORT_OFFLINE, FC_REASON_OFFLINE,
578                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

580         /* 0x05 */
581         {IOERR_NO_XRI, FC_PKT_LOCAL_RJT, FC_REASON_XCHG_DROPPED,
582                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

584         /* 0x06 */
585         {IOERR_ILLEGAL_COMMAND, FC_PKT_LOCAL_RJT, FC_REASON_ILLEGAL_REQ,
586                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

588         /* 0x07 */
589         {IOERR_XCHG_DROPPED, FC_PKT_LOCAL_RJT,  FC_REASON_XCHG_DROPPED,
```



```
590                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

592         /* 0x08 */
593         {IOERR_ILLEGAL_FIELD, FC_PKT_LOCAL_RJT, FC_REASON_ILLEGAL_REQ,
594                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

596         /* 0x0B */
597         {IOERR_RCV_BUFFER_WAITING, FC_PKT_LOCAL_RJT, FC_REASON_NOMEM,
598                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

600         /* 0x0D */
601         {IOERR_TX_DMA_FAILED, FC_PKT_LOCAL_RJT, FC_REASON_DMA_ERROR,
602                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

604         /* 0x0E */
605         {IOERR_RX_DMA_FAILED, FC_PKT_LOCAL_RJT, FC_REASON_DMA_ERROR,
606                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

608         /* 0x0F */
609         {IOERR_ILLEGAL_FRAME, FC_PKT_LOCAL_RJT, FC_REASON_ILLEGAL_FRAME,
610                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

612         /* 0x11 */
613         {IOERR_NO_RESOURCES, FC_PKT_LOCAL_RJT,  FC_REASON_NOMEM,
614                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

616         /* 0x13 */
617         {IOERR_ILLEGAL_LENGTH, FC_PKT_LOCAL_RJT, FC_REASON_ILLEGAL_LENGTH,
618                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

620         /* 0x14 */
621         {IOERR_UNSUPPORTED_FEATURE, FC_PKT_LOCAL_RJT, FC_REASON_UNSUPPORTED,
622                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

624         /* 0x15 */
625         {IOERR_ABORT_IN_PROGRESS, FC_PKT_LOCAL_RJT, FC_REASON_ABORTED,
626                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

628         /* 0x16 */
629         {IOERR_ABORT_REQUESTED, FC_PKT_LOCAL_RJT, FC_REASON_ABORTED,
630                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

632         /* 0x17 */
633         {IOERR_RCV_BUFFER_TIMEOUT, FC_PKT_LOCAL_RJT, FC_REASON_RX_BUF_TIMEOUT,
634                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

636         /* 0x18 */
637         {IOERR_LOOP_OPEN_FAILURE, FC_PKT_LOCAL_RJT, FC_REASON_FCAL_OPN_FAIL,
638                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

640         /* 0x1A */
641         {IOERR_LINK_DOWN, FC_PKT_PORT_OFFLINE, FC_REASON_OFFLINE,
642                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

644         /* 0x21 */
645         {IOERR_BAD_HOST_ADDRESS, FC_PKT_LOCAL_RJT, FC_REASON_BAD_SID,
646                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

648         /* Occurs at link down */
649         /* 0x28 */
650         {IOERR_BUFFER_SHORTAGE, FC_PKT_PORT_OFFLINE, FC_REASON_OFFLINE,
651                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},

653         /* 0xF0 */
654         {IOERR_ABORT_TIMEOUT, FC_PKT_TIMEOUT, FC_REASON_SEQ_TIMEOUT,
655                 FC_EXPLN_NONE, FC_ACTION_RETRYABLE},
```

```
};

#define IOERR_MAX    (sizeof (emlxs_ioerr_tbl)/sizeof (emlxs_xlat_err_t))



emlxs_table_t emlxs_error_table[] = {
        {IOERR_SUCCESS, "No error."},
        {IOERR_MISSING_CONTINUE, "Missing continue."},
        {IOERR_SEQUENCE_TIMEOUT, "Sequence timeout."},
        {IOERR_INTERNAL_ERROR, "Internal error."},
        {IOERR_INVALID_RPI, "Invalid RPI."},
        {IOERR_NO_XRI, "No XRI."},
        {IOERR_ILLEGAL_COMMAND, "Illegal command."},
        {IOERR_XCHG_DROPPED, "Exchange dropped."},
        {IOERR_ILLEGAL_FIELD, "Illegal field."},
        {IOERR_RCV_BUFFER_WAITING, "RX buffer waiting."},
        {IOERR_TX_DMA_FAILED, "TX DMA failed."},
        {IOERR_RX_DMA_FAILED, "RX DMA failed."},
        {IOERR_ILLEGAL_FRAME, "Illegal frame."},
        {IOERR_NO_RESOURCES, "No resources."},
        {IOERR_ILLEGAL_LENGTH, "Illegal length."},
        {IOERR_UNSUPPORTED_FEATURE, "Unsupported feature."},
        {IOERR_ABORT_IN_PROGRESS, "Abort in progess."},
        {IOERR_ABORT_REQUESTED, "Abort requested."},
        {IOERR_RCV_BUFFER_TIMEOUT, "RX buffer timeout."},
        {IOERR_LOOP_OPEN_FAILURE, "Loop open failed."},
        {IOERR_RING_RESET, "Ring reset."},
        {IOERR_LINK_DOWN, "Link down."},
        {IOERR_CORRUPTED_DATA, "Corrupted data."},
        {IOERR_CORRUPTED_RPI, "Corrupted RPI."},
        {IOERR_OUT_OF_ORDER_DATA, "Out-of-order data."},
        {IOERR_OUT_OF_ORDER_ACK, "Out-of-order ack."},
        {IOERR_DUP_FRAME, "Duplicate frame."},
        {IOERR_LINK_CONTROL_FRAME, "Link control frame."},
        {IOERR_BAD_HOST_ADDRESS, "Bad host address."},
        {IOERR_RCV_HDRBUF_WAITING, "RX header buffer waiting."},
        {IOERR_MISSING_HDR_BUFFER, "Missing header buffer."},
        {IOERR_MSEQ_CHAIN_CORRUPTED, "MSEQ chain corrupted."},
        {IOERR_ABORTMULT_REQUESTED, "Abort multiple requested."},
        {IOERR_BUFFER_SHORTAGE, "Buffer shortage."},
        {IOERR_XRIBUF_WAITING, "XRI buffer shortage"},
        {IOERR_XRIBUF_MISSING, "XRI buffer missing"},
        {IOERR_ROFFSET_INVAL, "Relative offset invalid."},
        {IOERR_ROFFSET_MISSING, "Relative offset missing."},
        {IOERR_INSUF_BUFFER, "Buffer too small."},
        {IOERR_MISSING_SI, "ELS frame missing SI"},
        {IOERR_MISSING_ES, "Exhausted burst without ES"},
        {IOERR_INCOMP_XFER, "Transfer incomplete."},
        {IOERR_ABORT_TIMEOUT, "Abort timeout."}

};      /* emlxs_error_table */


emlxs_table_t emlxs_state_table[] = {
        {IOSTAT_SUCCESS, "Success."},
        {IOSTAT_FCP_RSP_ERROR, "FCP response error."},
        {IOSTAT_REMOTE_STOP, "Remote stop."},
        {IOSTAT_LOCAL_REJECT, "Local reject."},
        {IOSTAT_NPORT_RJT, "NPort reject."},
        {IOSTAT_FABRIC_RJT, "Fabric reject."},
        {IOSTAT_NPORT_BSY, "Nport busy."},
        {IOSTAT_FABRIC_BSY, "Fabric busy."},
        {IOSTAT_INTERMED_RSP, "Intermediate response."},
        {IOSTAT_LS_RJT, "LS reject."},
        {IOSTAT_CMD_REJECT,             "Cmd reject."},
```



```
        {IOSTAT_FCP_TGT_LENCHK, "TGT length check."},
        {IOSTAT_NEED_BUFF_ENTRY, "Need buffer entry."},
        {IOSTAT_DATA_UNDERRUN, "Data underrun."},
        {IOSTAT_DATA_OVERRUN,  "Data overrun."},

};      /* emlxs_state_table */


#ifdef MENLO_SUPPORT
emlxs_table_t emlxs_menlo_cmd_table[] = {
        {MENLO_CMD_INITIALIZE,          "MENLO_INIT"},
        {MENLO_CMD_FW_DOWNLOAD,         "MENLO_FW_DOWNLOAD"},
        {MENLO_CMD_READ_MEMORY,         "MENLO_READ_MEM"},
        {MENLO_CMD_WRITE_MEMORY,        "MENLO_WRITE_MEM"},
        {MENLO_CMD_FTE_INSERT,          "MENLO_FTE_INSERT"},
        {MENLO_CMD_FTE_DELETE,          "MENLO_FTE_DELETE"},

        {MENLO_CMD_GET_INIT,            "MENLO_GET_INIT"},
        {MENLO_CMD_GET_CONFIG,          "MENLO_GET_CONFIG"},
        {MENLO_CMD_GET_PORT_STATS,      "MENLO_GET_PORT_STATS"},
        {MENLO_CMD_GET_LIF_STATS,       "MENLO_GET_LIF_STATS"},
        {MENLO_CMD_GET_ASIC_STATS,      "MENLO_GET_ASIC_STATS"},
        {MENLO_CMD_GET_LOG_CONFIG,      "MENLO_GET_LOG_CFG"},
        {MENLO_CMD_GET_LOG_DATA,        "MENLO_GET_LOG_DATA"},
        {MENLO_CMD_GET_PANIC_LOG,       "MENLO_GET_PANIC_LOG"},
        {MENLO_CMD_GET_LB_MODE,         "MENLO_GET_LB_MODE"},

        {MENLO_CMD_SET_PAUSE,           "MENLO_SET_PAUSE"},
        {MENLO_CMD_SET_FCOE_COS,        "MENLO_SET_FCOE_COS"},
        {MENLO_CMD_SET_UIF_PORT_TYPE,   "MENLO_SET_UIF_TYPE"},

        {MENLO_CMD_DIAGNOSTICS,         "MENLO_DIAGNOSTICS"},
        {MENLO_CMD_LOOPBACK,            "MENLO_LOOPBACK"},

        {MENLO_CMD_RESET,               "MENLO_RESET"},
        {MENLO_CMD_SET_MODE,            "MENLO_SET_MODE"}

};      /* emlxs_menlo_cmd_table */

emlxs_table_t emlxs_menlo_rsp_table[] = {
        {MENLO_RSP_SUCCESS,             "SUCCESS"},
        {MENLO_ERR_FAILED,              "FAILED"},
        {MENLO_ERR_INVALID_CMD,         "INVALID_CMD"},
        {MENLO_ERR_INVALID_CREDIT,      "INVALID_CREDIT"},
        {MENLO_ERR_INVALID_SIZE,        "INVALID_SIZE"},
        {MENLO_ERR_INVALID_ADDRESS,     "INVALID_ADDRESS"},
        {MENLO_ERR_INVALID_CONTEXT,     "INVALID_CONTEXT"},
        {MENLO_ERR_INVALID_LENGTH,      "INVALID_LENGTH"},
        {MENLO_ERR_INVALID_TYPE,        "INVALID_TYPE"},
        {MENLO_ERR_INVALID_DATA,        "INVALID_DATA"},
        {MENLO_ERR_INVALID_VALUE1,      "INVALID_VALUE1"},
        {MENLO_ERR_INVALID_VALUE2,      "INVALID_VALUE2"},
        {MENLO_ERR_INVALID_MASK,        "INVALID_MASK"},
        {MENLO_ERR_CHECKSUM,            "CHECKSUM_ERROR"},
        {MENLO_ERR_UNKNOWN_FCID,        "UNKNOWN_FCID"},
        {MENLO_ERR_UNKNOWN_WWN,         "UNKNOWN_WWN"},
        {MENLO_ERR_BUSY,                "BUSY"},

};      /* emlxs_menlo_rsp_table */

#endif /* MENLO_SUPPORT */


emlxs_table_t emlxs_mscmd_table[] = {
        {SLI_CT_RESPONSE_FS_ACC, "CT_ACC"},
        {SLI_CT_RESPONSE_FS_RJT, "CT_RJT"},
```

```
788         {MS_GTIN, "MS_GTIN"},
789         {MS_GIEL, "MS_GIEL"},
790         {MS_GIET, "MS_GIET"},
791         {MS_GDID, "MS_GDID"},
792         {MS_GMID, "MS_GMID"},
793         {MS_GFN, "MS_GFN"},
794         {MS_GIELN, "MS_GIELN"},
795         {MS_GMAL, "MS_GMAL"},
796         {MS_GIEIL, "MS_GIEIL"},
797         {MS_GPL, "MS_GPL"},
798         {MS_GPT, "MS_GPT"},
799         {MS_GPPN, "MS_GPPN"},
800         {MS_GAPNL, "MS_GAPNL"},
801         {MS_GPS, "MS_GPS"},
802         {MS_GPSC, "MS_GPSC"},
803         {MS_GATIN, "MS_GATIN"},
804         {MS_GSES, "MS_GSES"},
805         {MS_GPLNL, "MS_GPLNL"},
806         {MS_GPLT, "MS_GPLT"},
807         {MS_GPLML, "MS_GPLML"},
808         {MS_GPAB, "MS_GPAB"},
809         {MS_GNPL, "MS_GNPL"},
810         {MS_GPNL, "MS_GPNL"},
811         {MS_GPFCP, "MS_GPFCP"},
812         {MS_GPLI, "MS_GPLI"},
813         {MS_GNID, "MS_GNID"},
814         {MS_RIELN, "MS_RIELN"},
815         {MS_RPL, "MS_RPL"},
816         {MS_RPLN, "MS_RPLN"},
817         {MS_RPLT, "MS_RPLT"},
818         {MS_RPLM, "MS_RPLM"},
819         {MS_RPAB, "MS_RPAB"},
820         {MS_RPFCP, "MS_RPFCP"},
821         {MS_RPLI, "MS_RPLI"},
822         {MS_DPL, "MS_DPL"},
823         {MS_DPLN, "MS_DPLN"},
824         {MS_DPLM, "MS_DPLM"},
825         {MS_DPLML, "MS_DPLML"},
826         {MS_DPLI, "MS_DPLI"},
827         {MS_DPAB, "MS_DPAB"},
828         {MS_DPALL, "MS_DPALL"}

830 };      /* emlxs_mscmd_table */


833 emlxs_table_t emlxs_ctcmd_table[] = {
834         {SLI_CT_RESPONSE_FS_ACC, "CT_ACC"},
835         {SLI_CT_RESPONSE_FS_RJT, "CT_RJT"},
836         {SLI_CTNS_GA_NXT, "GA_NXT"},
837         {SLI_CTNS_GPN_ID, "GPN_ID"},
838         {SLI_CTNS_GNN_ID, "GNN_ID"},
839         {SLI_CTNS_GCS_ID, "GCS_ID"},
840         {SLI_CTNS_GFT_ID, "GFT_ID"},
841         {SLI_CTNS_GSPN_ID, "GSPN_ID"},
842         {SLI_CTNS_GPT_ID, "GPT_ID"},
843         {SLI_CTNS_GID_PN, "GID_PN"},
844         {SLI_CTNS_GID_NN, "GID_NN"},
845         {SLI_CTNS_GIP_NN, "GIP_NN"},
846         {SLI_CTNS_GIPA_NN, "GIPA_NN"},
847         {SLI_CTNS_GSNN_NN, "GSNN_NN"},
848         {SLI_CTNS_GNN_IP, "GNN_IP"},
849         {SLI_CTNS_GIPA_IP, "GIPA_IP"},
850         {SLI_CTNS_GID_FT, "GID_FT"},
851         {SLI_CTNS_GID_PT, "GID_PT"},
852         {SLI_CTNS_RPN_ID, "RPN_ID"},
853         {SLI_CTNS_RNN_ID, "RNN_ID"},
```



```
854         {SLI_CTNS_RCS_ID, "RCS_ID"},
855         {SLI_CTNS_RFT_ID, "RFT_ID"},
856         {SLI_CTNS_RSPN_ID, "RSPN_ID"},
857         {SLI_CTNS_RPT_ID, "RPT_ID"},
858         {SLI_CTNS_RIP_NN, "RIP_NN"},
859         {SLI_CTNS_RIPA_NN, "RIPA_NN"},
860         {SLI_CTNS_RSNN_NN, "RSNN_NN"},
861         {SLI_CTNS_DA_ID, "DA_ID"},
862         {SLI_CT_LOOPBACK, "LOOPBACK"} /* Driver special */

864 };      /* emlxs_ctcmd_table */



868 emlxs_table_t emlxs_rmcmd_table[] = {
869         {SLI_CT_RESPONSE_FS_ACC, "CT_ACC"},
870         {SLI_CT_RESPONSE_FS_RJT, "CT_RJT"},
871         {CT_OP_GSAT, "RM_GSAT"},
872         {CT_OP_GHAT, "RM_GHAT"},
873         {CT_OP_GPAT, "RM_GPAT"},
874         {CT_OP_GDAT, "RM_GDAT"},
875         {CT_OP_GPST, "RM_GPST"},
876         {CT_OP_GDP, "RM_GDP"},
877         {CT_OP_GDPG, "RM_GDPG"},
878         {CT_OP_GEPS, "RM_GEPS"},
879         {CT_OP_GLAT, "RM_GLAT"},
880         {CT_OP_SSAT, "RM_SSAT"},
881         {CT_OP_SHAT, "RM_SHAT"},
882         {CT_OP_SPAT, "RM_SPAT"},
883         {CT_OP_SDAT, "RM_SDAT"},
884         {CT_OP_SDP, "RM_SDP"},
885         {CT_OP_SBBS, "RM_SBBS"},
886         {CT_OP_RPST, "RM_RPST"},
887         {CT_OP_VFW, "RM_VFW"},
888         {CT_OP_DFW, "RM_DFW"},
889         {CT_OP_RES, "RM_RES"},
890         {CT_OP_RHD, "RM_RHD"},
891         {CT_OP_UFW, "RM_UFW"},
892         {CT_OP_RDP, "RM_RDP"},
893         {CT_OP_GHDR, "RM_GHDR"},
894         {CT_OP_CHD, "RM_CHD"},
895         {CT_OP_SSR, "RM_SSR"},
896         {CT_OP_RSAT, "RM_RSAT"},
897         {CT_OP_WSAT, "RM_WSAT"},
898         {CT_OP_RSAH, "RM_RSAH"},
899         {CT_OP_WSAH, "RM_WSAH"},
900         {CT_OP_RACT, "RM_RACT"},
901         {CT_OP_WACT, "RM_WACT"},
902         {CT_OP_RKT, "RM_RKT"},
903         {CT_OP_WKT, "RM_WKT"},
904         {CT_OP_SSC, "RM_SSC"},
905         {CT_OP_QHBA, "RM_QHBA"},
906         {CT_OP_GST, "RM_GST"},
907         {CT_OP_GFTM, "RM_GFTM"},
908         {CT_OP_SRL, "RM_SRL"},
909         {CT_OP_SI, "RM_SI"},
910         {CT_OP_SRC, "RM_SRC"},
911         {CT_OP_GPB, "RM_GPB"},
912         {CT_OP_SPB, "RM_SPB"},
913         {CT_OP_RPB, "RM_RPB"},
914         {CT_OP_RAPB, "RM_RAPB"},
915         {CT_OP_GBC, "RM_GBC"},
916         {CT_OP_GBS, "RM_GBS"},
917         {CT_OP_SBS, "RM_SBS"},
918         {CT_OP_GANI, "RM_GANI"},
919         {CT_OP_GRV, "RM_GRV"},
```

```
920         {CT_OP_GAPBS, "RM_GAPBS"},
921         {CT_OP_APBC, "RM_APBC"},
922         {CT_OP_GDT, "RM_GDT"},
923         {CT_OP_GDLMI, "RM_GDLMI"},
924         {CT_OP_GANA, "RM_GANA"},
925         {CT_OP_GDLV, "RM_GDLV"},
926         {CT_OP_GWUP, "RM_GWUP"},
927         {CT_OP_GLM, "RM_GLM"},
928         {CT_OP_GABS, "RM_GABS"},
929         {CT_OP_SABS, "RM_SABS"},
930         {CT_OP_RPR, "RM_RPR"},
931         {SLI_CT_LOOPBACK, "LOOPBACK"} /* Driver special */

933 };      /* emlxs_rmcmd_table */


936 emlxs_table_t emlxs_elscmd_table[] = {
937         {ELS_CMD_ACC, "ACC"},
938         {ELS_CMD_LS_RJT, "LS_RJT"},
939         {ELS_CMD_PLOGI, "PLOGI"},
940         {ELS_CMD_FLOGI, "FLOGI"},
941         {ELS_CMD_LOGO, "LOGO"},
942         {ELS_CMD_ABTX, "ABTX"},
943         {ELS_CMD_RCS, "RCS"},
944         {ELS_CMD_RES, "RES"},
945         {ELS_CMD_RSS, "RSS"},
946         {ELS_CMD_RSI, "RSI"},
947         {ELS_CMD_ESTS, "ESTS"},
948         {ELS_CMD_ESTC, "ESTC"},
949         {ELS_CMD_ADVC, "ADVC"},
950         {ELS_CMD_RTV, "RTV"},
951         {ELS_CMD_RLS, "RLS"},
952         {ELS_CMD_ECHO, "ECHO"},
953         {ELS_CMD_TEST, "TEST"},
954         {ELS_CMD_RRQ, "RRQ"},
955         {ELS_CMD_REC, "REC"},
956         {ELS_CMD_PRLI, "PRLI"},
957         {ELS_CMD_PRLO, "PRLO"},
958         {ELS_CMD_SCN, "SCN"},
959         {ELS_CMD_TPLS, "TPLS"},
960         {ELS_CMD_GPRLO, "GPRLO"},
961         {ELS_CMD_GAID, "GAID"},
962         {ELS_CMD_FACT, "FACT"},
963         {ELS_CMD_FDACT, "FDACT"},
964         {ELS_CMD_NACT, "NACT"},
965         {ELS_CMD_NDACT, "NDACT"},
966         {ELS_CMD_QoSR, "QoSR"},
967         {ELS_CMD_RVCS, "RVCS"},
968         {ELS_CMD_PDISC, "PDISC"},
969         {ELS_CMD_FDISC, "FDISC"},
970         {ELS_CMD_ADISC, "ADISC"},
971         {ELS_CMD_FARP, "FARP"},
972         {ELS_CMD_FARPR, "FARPR"},
973         {ELS_CMD_FAN, "FAN"},
974         {ELS_CMD_RSCN, "RSCN"},
975         {ELS_CMD_SCR, "SCR"},
976         {ELS_CMD_LINIT, "LINIT"},
977         {ELS_CMD_RNID, "RNID"},
978         {ELS_CMD_AUTH, "AUTH"}

980 };      /* emlxs_elscmd_table */


983 /*
984  *
985  *      Device Driver Entry Routines
```



```
986  *
987  */

989 #ifdef MODSYM_SUPPORT
990 static void emlxs_fca_modclose();
991 static int  emlxs_fca_modopen();
992 emlxs_modsym_t emlxs_modsym;    /* uses emlxs_device.lock */

994 static int
995 emlxs_fca_modopen()
996 {
997         int err;

999         if (emlxs_modsym.mod_fctl) {
1000                return (0);
1001        }

1003        /* Leadville (fctl) */
1004        err = 0;
1005        emlxs_modsym.mod_fctl =
1006            ddi_modopen("misc/fctl", KRTLD_MODE_FIRST, &err);
1007        if (!emlxs_modsym.mod_fctl) {
1008                cmn_err(CE_WARN,
1009                    "?%s: misc/fctl: ddi_modopen misc/fctl failed: error=%d",
1010                    DRIVER_NAME, err);

1012                goto failed;
1013        }

1015        err = 0;
1016        /* Check if the fctl fc_fca_attach is present */
1017        emlxs_modsym.fc_fca_attach =
1018            (int (*)())ddi_modsym(emlxs_modsym.mod_fctl, "fc_fca_attach",
1019            &err);
1020        if ((void *)emlxs_modsym.fc_fca_attach == NULL) {
1021                cmn_err(CE_WARN,
1022                    "?%s: misc/fctl: fc_fca_attach not present", DRIVER_NAME);
1023                goto failed;
1024        }

1026        err = 0;
1027        /* Check if the fctl fc_fca_detach is present */
1028        emlxs_modsym.fc_fca_detach =
1029            (int (*)())ddi_modsym(emlxs_modsym.mod_fctl, "fc_fca_detach",
1030            &err);
1031        if ((void *)emlxs_modsym.fc_fca_detach == NULL) {
1032                cmn_err(CE_WARN,
1033                    "?%s: misc/fctl: fc_fca_detach not present", DRIVER_NAME);
1034                goto failed;
1035        }

1037        err = 0;
1038        /* Check if the fctl fc_fca_init is present */
1039        emlxs_modsym.fc_fca_init =
1040            (int (*)())ddi_modsym(emlxs_modsym.mod_fctl, "fc_fca_init", &err);
1041        if ((void *)emlxs_modsym.fc_fca_init == NULL) {
1042                cmn_err(CE_WARN,
1043                    "?%s: misc/fctl: fc_fca_init not present", DRIVER_NAME);
1044                goto failed;
1045        }

1047        return (0);

1049 failed:

1051        emlxs_fca_modclose();
```

```
1053         return (1);


1056 } /* emlxs_fca_modopen() */


1059 static void
1060 emlxs_fca_modclose()
1061 {
1062         if (emlxs_modsym.mod_fctl) {
1063                 (void) ddi_modclose(emlxs_modsym.mod_fctl);
1064                 emlxs_modsym.mod_fctl = 0;
1065         }

1067         emlxs_modsym.fc_fca_attach = NULL;
1068         emlxs_modsym.fc_fca_detach = NULL;
1069         emlxs_modsym.fc_fca_init   = NULL;

1071         return;

1073 } /* emlxs_fca_modclose() */

1075 #endif /* MODSYM_SUPPORT */



1079 /*
1080  * Global driver initialization, called once when driver is loaded
1081  */
1082 int
1083 _init(void)
1084 {
1085         int ret;
1086         char buf[64];

1088         /*
1089          * First init call for this driver,
1090          * so initialize the emlxs_dev_ctl structure.
1091          */
1092         bzero(&emlxs_device, sizeof (emlxs_device));

1094 #ifdef MODSYM_SUPPORT
1095         bzero(&emlxs_modsym, sizeof (emlxs_modsym_t));
1096 #endif /* MODSYM_SUPPORT */

1098         (void) sprintf(buf, "%s_device mutex", DRIVER_NAME);
1099         mutex_init(&emlxs_device.lock, buf, MUTEX_DRIVER, NULL);

1101         (void) drv_getparm(LBOLT, &emlxs_device.log_timestamp);
1102         emlxs_device.drv_timestamp = gethrtime();
  26         emlxs_device.drv_timestamp = ddi_get_time();

1104         for (ret = 0; ret < MAX_FC_BRDS; ret++) {
1105                 emlxs_instance[ret] = (uint32_t)-1;
1106         }

1108         /*
1109          * Provide for one ddiinst of the emlxs_dev_ctl structure
1110          * for each possible board in the system.
1111          */
1112         if ((ret = ddi_soft_state_init(&emlxs_soft_state,
1113             sizeof (emlxs_hba_t), MAX_FC_BRDS)) != 0) {
1114                 cmn_err(CE_WARN,
1115                     "?%s: _init: ddi_soft_state_init failed. rval=%x",
1116                     DRIVER_NAME, ret);
```



```
1118                 return (ret);
1119         }

1121 #ifdef MODSYM_SUPPORT
1122         /* Open SFS */
1123         (void) emlxs_fca_modopen();
1124 #endif /* MODSYM_SUPPORT */

1126         /* Setup devops for SFS */
1127         MODSYM(fc_fca_init)(&emlxs_ops);

1129         if ((ret = mod_install(&emlxs_modlinkage)) != 0) {
1130                 (void) ddi_soft_state_fini(&emlxs_soft_state);
1131 #ifdef MODSYM_SUPPORT
1132                 /* Close SFS */
1133                 emlxs_fca_modclose();
1134 #endif /* MODSYM_SUPPORT */

1136                 return (ret);
1137         }

1139 #ifdef SAN_DIAG_SUPPORT
1140         (void) sprintf(buf, "%s_sd_bucket mutex", DRIVER_NAME);
1141         mutex_init(&sd_bucket_mutex, buf, MUTEX_DRIVER, NULL);
1142 #endif /* SAN_DIAG_SUPPORT */

1144         return (ret);

1146 } /* _init() */
```

*unchanged_portion_omitted_*

```
**********************************************************
  213056 Mon May  5 14:29:42 2014
new/usr/src/uts/common/io/ib/mgt/ibdm/ibdm.c
4777 ibdm shouldn't abuse ddi_get_time(9f)
Reviewed by: Rob Gittins <rob.gittins@nexenta.com>
Reviewed by: Albert Lee <albert.lee@nexenta.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

/*
 * ibdm.c
 *
 * This file contains the InifiniBand Device Manager (IBDM) support functions.
 * IB nexus driver will only be the client for the IBDM module.
 *
 * IBDM registers with IBTF for HCA arrival/removal notification.
 * IBDM registers with SA access to send DM MADs to discover the IOC's behind
 * the IOU's.
 *
 * IB nexus driver registers with IBDM to find the information about the
 * HCA's and IOC's (behind the IOU) present on the IB fabric.
 */

#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/systm.h>
#include <sys/taskq.h>
#include <sys/ib/mgt/ibdm/ibdm_impl.h>
#include <sys/ib/mgt/ibmf/ibmf_impl.h>
#include <sys/ib/ibtl/impl/ibtl_ibnex.h>
#include <sys/modctl.h>

/* Function Prototype declarations */
static int      ibdm_free_iou_info(ibdm_dp_gidinfo_t *, ibdm_iou_info_t **);
static int      ibdm_fini(void);
static int      ibdm_init(void);
static int      ibdm_get_reachable_ports(ibdm_port_attr_t *,
                        ibdm_hca_list_t *);
static ibdm_dp_gidinfo_t *ibdm_check_dgid(ib_guid_t, ib_sn_prefix_t);
```



```
static ibdm_dp_gidinfo_t *ibdm_check_dest_nodeguid(ibdm_dp_gidinfo_t *);
static boolean_t ibdm_is_cisco(ib_guid_t);
static boolean_t ibdm_is_cisco_switch(ibdm_dp_gidinfo_t *);
static void     ibdm_wait_cisco_probe_completion(ibdm_dp_gidinfo_t *);
static int      ibdm_set_classportinfo(ibdm_dp_gidinfo_t *);
static int      ibdm_send_classportinfo(ibdm_dp_gidinfo_t *);
static int      ibdm_send_iounitinfo(ibdm_dp_gidinfo_t *);
static int      ibdm_is_dev_mgt_supported(ibdm_dp_gidinfo_t *);
static int      ibdm_get_node_port_guids(ibmf_saa_handle_t, ib_lid_t,
                    ib_guid_t *, ib_guid_t *);
static int      ibdm_retry_command(ibdm_timeout_cb_args_t *);
static int      ibdm_get_diagcode(ibdm_dp_gidinfo_t *, int);
static int      ibdm_verify_mad_status(ib_mad_hdr_t *);
static int      ibdm_handle_redirection(ibmf_msg_t *,
                    ibdm_dp_gidinfo_t *, int *);
static void     ibdm_wait_probe_completion(void);
static void     ibdm_sweep_fabric(int);
static void     ibdm_probe_gid_thread(void *);
static void     ibdm_wakeup_probe_gid_cv(void);
static void     ibdm_port_attr_ibmf_init(ibdm_port_attr_t *, ib_pkey_t, int);
static int      ibdm_port_attr_ibmf_fini(ibdm_port_attr_t *, int);
static void     ibdm_update_port_attr(ibdm_port_attr_t *);
static void     ibdm_handle_hca_attach(ib_guid_t);
static void     ibdm_handle_srventry_mad(ibmf_msg_t *,
                    ibdm_dp_gidinfo_t *, int *);
static void     ibdm_ibmf_recv_cb(ibmf_handle_t, ibmf_msg_t *, void *);
static void     ibdm_recv_incoming_mad(void *);
static void     ibdm_process_incoming_mad(ibmf_handle_t, ibmf_msg_t *, void *);
static void     ibdm_ibmf_send_cb(ibmf_handle_t, ibmf_msg_t *, void *);
static void     ibdm_pkt_timeout_hdlr(void *arg);
static void     ibdm_initialize_port(ibdm_port_attr_t *);
static void     ibdm_update_port_pkeys(ibdm_port_attr_t *port);
static void     ibdm_handle_diagcode(ibmf_msg_t *, ibdm_dp_gidinfo_t *, int *);
static void     ibdm_probe_gid(ibdm_dp_gidinfo_t *);
static void     ibdm_alloc_send_buffers(ibmf_msg_t *);
static void     ibdm_free_send_buffers(ibmf_msg_t *);
static void     ibdm_handle_hca_detach(ib_guid_t);
static void     ibdm_handle_port_change_event(ibt_async_event_t *);
static int      ibdm_fini_port(ibdm_port_attr_t *);
static int      ibdm_uninit_hca(ibdm_hca_list_t *);
static void     ibdm_handle_setclassportinfo(ibmf_handle_t, ibmf_msg_t *,
                    ibdm_dp_gidinfo_t *, int *);
static void     ibdm_handle_iounitinfo(ibmf_handle_t,
                    ibmf_msg_t *, ibdm_dp_gidinfo_t *, int *);
static void     ibdm_handle_ioc_profile(ibmf_handle_t,
                    ibmf_msg_t *, ibdm_dp_gidinfo_t *, int *);
static void     ibdm_event_hdlr(void *, ibt_hca_hdl_t,
                    ibt_async_code_t, ibt_async_event_t *);
static void     ibdm_handle_classportinfo(ibmf_handle_t,
                    ibmf_msg_t *, ibdm_dp_gidinfo_t *, int *);
static void     ibdm_update_ioc_port_gidlist(ibdm_ioc_info_t *,
                    ibdm_dp_gidinfo_t *);

static ibdm_hca_list_t          *ibdm_dup_hca_attr(ibdm_hca_list_t *);
static ibdm_ioc_info_t          *ibdm_dup_ioc_info(ibdm_ioc_info_t *,
                                    ibdm_dp_gidinfo_t *gid_list);
static void                     ibdm_probe_ioc(ib_guid_t, ib_guid_t, int);
static ibdm_ioc_info_t          *ibdm_is_ioc_present(ib_guid_t,
                                    ibdm_dp_gidinfo_t *, int *);
static ibdm_port_attr_t         *ibdm_get_port_attr(ibt_async_event_t *,
                                    ibdm_hca_list_t **);
static sa_node_record_t         *ibdm_get_node_records(ibmf_saa_handle_t,
                                    size_t *, ib_guid_t);
static int                      ibdm_get_node_record_by_port(ibmf_saa_handle_t,
                                    ib_guid_t, sa_node_record_t **, size_t *);
static sa_portinfo_record_t     *ibdm_get_portinfo(ibmf_saa_handle_t, size_t *,
```

```
125                                     ib_lid_t);
126 static ibdm_dp_gidinfo_t        *ibdm_create_gid_info(ibdm_port_attr_t *,
127                                     ib_gid_t, ib_gid_t);
128 static ibdm_dp_gidinfo_t        *ibdm_find_gid(ib_guid_t, ib_guid_t);
129 static int      ibdm_send_ioc_profile(ibdm_dp_gidinfo_t *, uint8_t);
130 static ibdm_ioc_info_t  *ibdm_update_ioc_gidlist(ibdm_dp_gidinfo_t *, int);
131 static void     ibdm_saa_event_cb(ibmf_saa_handle_t, ibmf_saa_subnet_event_t,
132                     ibmf_saa_event_details_t *, void *);
133 static void     ibdm_reprobe_update_port_srv(ibdm_ioc_info_t *,
134     ibdm_dp_gidinfo_t *);
135 static ibdm_dp_gidinfo_t *ibdm_handle_gid_rm(ibdm_dp_gidinfo_t *);
136 static void ibdm_rmfrom_glgid_list(ibdm_dp_gidinfo_t *,
137     ibdm_dp_gidinfo_t *);
138 static void ibdm_addto_gidlist(ibdm_gid_t **, ibdm_gid_t *);
139 static void ibdm_free_gid_list(ibdm_gid_t *);
140 static void ibdm_rescan_gidlist(ib_guid_t *ioc_guid);
141 static void ibdm_notify_newgid_iocs(ibdm_dp_gidinfo_t *);
142 static void ibdm_saa_event_taskq(void *);
143 static void ibdm_free_saa_event_arg(ibdm_saa_event_arg_t *);
144 static void ibdm_get_next_port(ibdm_hca_list_t **,
145     ibdm_port_attr_t **, int);
146 static void ibdm_add_to_gl_gid(ibdm_dp_gidinfo_t *,
147     ibdm_dp_gidinfo_t *);
148 static void ibdm_addto_glhcalist(ibdm_dp_gidinfo_t *,
149     ibdm_hca_list_t *);
150 static void ibdm_delete_glhca_list(ibdm_dp_gidinfo_t *);
151 static void ibdm_saa_handle_new_gid(void *);
152 static void ibdm_reset_all_dgids(ibmf_saa_handle_t);
153 static void ibdm_reset_gidinfo(ibdm_dp_gidinfo_t *);
154 static void ibdm_delete_gidinfo(ibdm_dp_gidinfo_t *);
155 static void ibdm_fill_srv_attr_mod(ib_mad_hdr_t *, ibdm_timeout_cb_args_t *);
156 static void ibdm_bump_transactionID(ibdm_dp_gidinfo_t *);
157 static ibdm_ioc_info_t  *ibdm_handle_prev_iou();
158 static int ibdm_serv_cmp(ibdm_srvents_info_t *, ibdm_srvents_info_t *,
159     int);
160 static ibdm_ioc_info_t *ibdm_get_ioc_info_with_gid(ib_guid_t,
161     ibdm_dp_gidinfo_t **);

163 int     ibdm_dft_timeout        = IBDM_DFT_TIMEOUT;
164 int     ibdm_dft_retry_cnt      = IBDM_DFT_NRETRIES;
165 #ifdef DEBUG
166 int     ibdm_ignore_saa_event = 0;
167 #endif
168 int     ibdm_enumerate_iocs = 0;

170 /* Modload support */
171 static struct modlmisc ibdm_modlmisc    = {
172         &mod_miscops,
173         "InfiniBand Device Manager"
174 };

176 struct modlinkage ibdm_modlinkage = {
177         MODREV_1,
178         (void *)&ibdm_modlmisc,
179         NULL
180 };

182 static ibt_clnt_modinfo_t ibdm_ibt_modinfo = {
183         IBTI_V_CURR,
184         IBT_DM,
185         ibdm_event_hdlr,
186         NULL,
187         "ibdm"
188 };

190 /* Global variables */
```



```
191 ibdm_t  ibdm;
192 int     ibdm_taskq_enable = IBDM_ENABLE_TASKQ_HANDLING;
193 char    *ibdm_string = "ibdm";

195 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv",
196     ibdm.ibdm_dp_gidlist_head))

198 /*
199  * _init
200  *      Loadable module init, called before any other module.
201  *      Initialize mutex
202  *      Register with IBTF
203  */
204 int
205 _init(void)
206 {
207         int             err;

209         IBTF_DPRINTF_L4("ibdm", "\t_init: addr of ibdm %p", &ibdm);

211         if ((err = ibdm_init()) != IBDM_SUCCESS) {
212                 IBTF_DPRINTF_L2("ibdm", "_init: ibdm_init failed 0x%x", err);
213                 (void) ibdm_fini();
214                 return (DDI_FAILURE);
215         }

217         if ((err = mod_install(&ibdm_modlinkage)) != 0) {
218                 IBTF_DPRINTF_L2("ibdm", "_init: mod_install failed 0x%x", err);
219                 (void) ibdm_fini();
220         }
221         return (err);
222 }


225 int
226 _fini(void)
227 {
228         int err;

230         if ((err = ibdm_fini()) != IBDM_SUCCESS) {
231                 IBTF_DPRINTF_L2("ibdm", "_fini: ibdm_fini failed 0x%x", err);
232                 (void) ibdm_init();
233                 return (EBUSY);
234         }

236         if ((err = mod_remove(&ibdm_modlinkage)) != 0) {
237                 IBTF_DPRINTF_L2("ibdm", "_fini: mod_remove failed 0x%x", err);
238                 (void) ibdm_init();
239         }
240         return (err);
241 }


244 int
245 _info(struct modinfo *modinfop)
246 {
247         return (mod_info(&ibdm_modlinkage, modinfop));
248 }


251 /*
252  * ibdm_init():
253  *      Register with IBTF
254  *      Allocate memory for the HCAs
255  *      Allocate minor-nodes for the HCAs
256  */
```

```
257 static int
258 ibdm_init(void)
259 {
260         int                     i, hca_count;
261         ib_guid_t               *hca_guids;
262         ibt_status_t            status;
263
264         IBTF_DPRINTF_L4("ibdm", "\tibdm_init:");
265         if (!(ibdm.ibdm_state & IBDM_LOCKS_ALLOCED)) {
266                 mutex_init(&ibdm.ibdm_mutex, NULL, MUTEX_DEFAULT, NULL);
267                 mutex_init(&ibdm.ibdm_hl_mutex, NULL, MUTEX_DEFAULT, NULL);
268                 mutex_init(&ibdm.ibdm_ibnex_mutex, NULL, MUTEX_DEFAULT, NULL);
269                 cv_init(&ibdm.ibdm_port_settle_cv, NULL, CV_DRIVER, NULL);
270                 mutex_enter(&ibdm.ibdm_mutex);
271                 ibdm.ibdm_state |= IBDM_LOCKS_ALLOCED;
272         }
273
274         if (!(ibdm.ibdm_state & IBDM_IBT_ATTACHED)) {
275                 if ((status = ibt_attach(&ibdm_ibt_modinfo, NULL, NULL,
276                     (void *)&ibdm.ibdm_ibt_clnt_hdl)) != IBT_SUCCESS) {
277                         IBTF_DPRINTF_L2("ibdm", "ibdm_init: ibt_attach "
278                             "failed %x", status);
279                         mutex_exit(&ibdm.ibdm_mutex);
280                         return (IBDM_FAILURE);
281                 }
282
283                 ibdm.ibdm_state |= IBDM_IBT_ATTACHED;
284                 mutex_exit(&ibdm.ibdm_mutex);
285         }
286
287
288         if (!(ibdm.ibdm_state & IBDM_HCA_ATTACHED)) {
289                 hca_count = ibt_get_hca_list(&hca_guids);
290                 IBTF_DPRINTF_L4("ibdm", "ibdm_init: num_hcas = %d", hca_count);
291                 for (i = 0; i < hca_count; i++)
292                         (void) ibdm_handle_hca_attach(hca_guids[i]);
293                 if (hca_count)
294                         ibt_free_hca_list(hca_guids, hca_count);
295
296                 mutex_enter(&ibdm.ibdm_mutex);
297                 ibdm.ibdm_state |= IBDM_HCA_ATTACHED;
298                 mutex_exit(&ibdm.ibdm_mutex);
299         }
300
301         if (!(ibdm.ibdm_state & IBDM_CVS_ALLOCED)) {
302                 cv_init(&ibdm.ibdm_probe_cv, NULL, CV_DRIVER, NULL);
303                 cv_init(&ibdm.ibdm_busy_cv, NULL, CV_DRIVER, NULL);
304                 mutex_enter(&ibdm.ibdm_mutex);
305                 ibdm.ibdm_state |= IBDM_CVS_ALLOCED;
306                 mutex_exit(&ibdm.ibdm_mutex);
307         }
308         return (IBDM_SUCCESS);
309 }
310
311
312 static int
313 ibdm_free_iou_info(ibdm_dp_gidinfo_t *gid_info, ibdm_iou_info_t **ioup)
314 {
315         int                     ii, k, niocs;
316         size_t                  size;
317         ibdm_gid_t              *delete, *head;
318         timeout_id_t            timeout_id;
319         ibdm_ioc_info_t         *ioc;
320         ibdm_iou_info_t         *gl_iou = *ioup;
321
322         ASSERT(mutex_owned(&gid_info->gl_mutex));
```



```
323         if (gl_iou == NULL) {
324                 IBTF_DPRINTF_L4("ibdm", "\tibdm_free_iou_info: No IOU");
325                 return (0);
326         }
327
328         niocs = gl_iou->iou_info.iou_num_ctrl_slots;
329         IBTF_DPRINTF_L4("ibdm", "\tfree_iou_info: gid_info = %p, niocs %d",
330             gid_info, niocs);
331
332         for (ii = 0; ii < niocs; ii++) {
333                 ioc = (ibdm_ioc_info_t *)&gl_iou->iou_ioc_info[ii];
334
335                 /* handle the case where an ioc_timeout_id is scheduled */
336                 if (ioc->ioc_timeout_id) {
337                         timeout_id = ioc->ioc_timeout_id;
338                         ioc->ioc_timeout_id = 0;
339                         mutex_exit(&gid_info->gl_mutex);
340                         IBTF_DPRINTF_L5("ibdm", "free_iou_info: "
341                             "ioc_timeout_id = 0x%x", timeout_id);
342                         if (untimeout(timeout_id) == -1) {
343                                 IBTF_DPRINTF_L2("ibdm", "free_iou_info: "
344                                     "untimeout ioc_timeout_id failed");
345                                 mutex_enter(&gid_info->gl_mutex);
346                                 return (-1);
347                         }
348                         mutex_enter(&gid_info->gl_mutex);
349                 }
350
351                 /* handle the case where an ioc_dc_timeout_id is scheduled */
352                 if (ioc->ioc_dc_timeout_id) {
353                         timeout_id = ioc->ioc_dc_timeout_id;
354                         ioc->ioc_dc_timeout_id = 0;
355                         mutex_exit(&gid_info->gl_mutex);
356                         IBTF_DPRINTF_L5("ibdm", "free_iou_info: "
357                             "ioc_dc_timeout_id = 0x%x", timeout_id);
358                         if (untimeout(timeout_id) == -1) {
359                                 IBTF_DPRINTF_L2("ibdm", "free_iou_info: "
360                                     "untimeout ioc_dc_timeout_id failed");
361                                 mutex_enter(&gid_info->gl_mutex);
362                                 return (-1);
363                         }
364                         mutex_enter(&gid_info->gl_mutex);
365                 }
366
367                 /* handle the case where serv[k].se_timeout_id is scheduled */
368                 for (k = 0; k < ioc->ioc_profile.ioc_service_entries; k++) {
369                         if (ioc->ioc_serv[k].se_timeout_id) {
370                                 timeout_id = ioc->ioc_serv[k].se_timeout_id;
371                                 ioc->ioc_serv[k].se_timeout_id = 0;
372                                 mutex_exit(&gid_info->gl_mutex);
373                                 IBTF_DPRINTF_L5("ibdm", "free_iou_info: "
374                                     "ioc->ioc_serv[%d].se_timeout_id = 0x%x",
375                                     k, timeout_id);
376                                 if (untimeout(timeout_id) == -1) {
377                                         IBTF_DPRINTF_L2("ibdm", "free_iou_info:"
378                                             " untimeout se_timeout_id failed");
379                                         mutex_enter(&gid_info->gl_mutex);
380                                         return (-1);
381                                 }
382                                 mutex_enter(&gid_info->gl_mutex);
383                         }
384                 }
385
386                 /* delete GID list in IOC */
387                 head = ioc->ioc_gid_list;
388                 while (head) {
```

```
389                         IBTF_DPRINTF_L4("ibdm", "\tibdm_free_iou_info: "
390                             "Deleting gid_list struct %p", head);
391                         delete = head;
392                         head = head->gid_next;
393                         kmem_free(delete, sizeof (ibdm_gid_t));
394                 }
395                 ioc->ioc_gid_list = NULL;
396
397                 /* delete ioc_serv */
398                 size = ioc->ioc_profile.ioc_service_entries *
399                     sizeof (ibdm_srvents_info_t);
400                 if (ioc->ioc_serv && size) {
401                         kmem_free(ioc->ioc_serv, size);
402                         ioc->ioc_serv = NULL;
403                 }
404         }
405         /*
406          * Clear the IBDM_CISCO_PROBE_DONE flag to get the IO Unit information
407          * via the switch during the probe process.
408          */
409         gid_info->gl_flag &= ~IBDM_CISCO_PROBE_DONE;
410
411         IBTF_DPRINTF_L4("ibdm", "\tibdm_free_iou_info: deleting IOU & IOC");
412         size = sizeof (ibdm_iou_info_t) + niocs * sizeof (ibdm_ioc_info_t);
413         kmem_free(gl_iou, size);
414         *ioup = NULL;
415         return (0);
416 }
417
418
419 /*
420  * ibdm_fini():
421  *      Un-register with IBTF
422  *      De allocate memory for the GID info
423  */
424 static int
425 ibdm_fini()
426 {
427         int                     ii;
428         ibdm_hca_list_t         *hca_list, *temp;
429         ibdm_dp_gidinfo_t       *gid_info, *tmp;
430         ibdm_gid_t              *head, *delete;
431
432         IBTF_DPRINTF_L4("ibdm", "\tibdm_fini");
433
434         mutex_enter(&ibdm.ibdm_hl_mutex);
435         if (ibdm.ibdm_state & IBDM_IBT_ATTACHED) {
436                 if (ibt_detach(ibdm.ibdm_ibt_clnt_hdl) != IBT_SUCCESS) {
437                         IBTF_DPRINTF_L2("ibdm", "\t_fini: ibt_detach failed");
438                         mutex_exit(&ibdm.ibdm_hl_mutex);
439                         return (IBDM_FAILURE);
440                 }
441                 ibdm.ibdm_state &= ~IBDM_IBT_ATTACHED;
442                 ibdm.ibdm_ibt_clnt_hdl = NULL;
443         }
444
445         hca_list = ibdm.ibdm_hca_list_head;
446         IBTF_DPRINTF_L4("ibdm", "\tibdm_fini: nhcas %d", ibdm.ibdm_hca_count);
447         for (ii = 0; ii < ibdm.ibdm_hca_count; ii++) {
448                 temp = hca_list;
449                 hca_list = hca_list->hl_next;
450                 IBTF_DPRINTF_L4("ibdm", "\tibdm_fini: hca %p", temp);
451                 if (ibdm_uninit_hca(temp) != IBDM_SUCCESS) {
452                         IBTF_DPRINTF_L2("ibdm", "\tibdm_fini: "
453                             "uninit_hca %p failed", temp);
454                         mutex_exit(&ibdm.ibdm_hl_mutex);
```



```
455                         return (IBDM_FAILURE);
456                 }
457         }
458         mutex_exit(&ibdm.ibdm_hl_mutex);
459
460         mutex_enter(&ibdm.ibdm_mutex);
461         if (ibdm.ibdm_state & IBDM_HCA_ATTACHED)
462                 ibdm.ibdm_state &= ~IBDM_HCA_ATTACHED;
463
464         gid_info = ibdm.ibdm_dp_gidlist_head;
465         while (gid_info) {
466                 mutex_enter(&gid_info->gl_mutex);
467                 (void) ibdm_free_iou_info(gid_info, &gid_info->gl_iou);
468                 mutex_exit(&gid_info->gl_mutex);
469                 ibdm_delete_glhca_list(gid_info);
470
471                 tmp = gid_info;
472                 gid_info = gid_info->gl_next;
473                 mutex_destroy(&tmp->gl_mutex);
474                 head = tmp->gl_gid;
475                 while (head) {
476                         IBTF_DPRINTF_L4("ibdm",
477                             "\tibdm_fini: Deleting gid structs");
478                         delete = head;
479                         head = head->gid_next;
480                         kmem_free(delete, sizeof (ibdm_gid_t));
481                 }
482                 kmem_free(tmp, sizeof (ibdm_dp_gidinfo_t));
483         }
484         mutex_exit(&ibdm.ibdm_mutex);
485
486         if (ibdm.ibdm_state & IBDM_LOCKS_ALLOCED) {
487                 ibdm.ibdm_state &= ~IBDM_LOCKS_ALLOCED;
488                 mutex_destroy(&ibdm.ibdm_mutex);
489                 mutex_destroy(&ibdm.ibdm_hl_mutex);
490                 mutex_destroy(&ibdm.ibdm_ibnex_mutex);
491                 cv_destroy(&ibdm.ibdm_port_settle_cv);
492         }
493         if (ibdm.ibdm_state & IBDM_CVS_ALLOCED) {
494                 ibdm.ibdm_state &= ~IBDM_CVS_ALLOCED;
495                 cv_destroy(&ibdm.ibdm_probe_cv);
496                 cv_destroy(&ibdm.ibdm_busy_cv);
497         }
498         return (IBDM_SUCCESS);
499 }
500
501
502 /*
503  * ibdm_event_hdlr()
504  *
505  *      IBDM registers  this asynchronous event handler at the time of
506  *      ibt_attach. IBDM support the following async events. For other
507  *      event, simply returns success.
508  *      IBT_HCA_ATTACH_EVENT:
509  *              Retrieves the  information about all the port that are
510  *              present on this HCA,  allocates  the  port  attributes
511  *              structure  and calls IB  nexus  callback  routine with
512  *              the port attributes structure as an input argument.
513  *      IBT_HCA_DETACH_EVENT:
514  *              Retrieves the information about all the ports that are
515  *              present on  this HCA and  calls IB nexus callback with
516  *              port guid as an argument
517  *      IBT_EVENT_PORT_UP:
518  *              Register with IBMF and SA access
519  *              Setup IBMF receive callback routine
520  *      IBT_EVENT_PORT_DOWN:
```

```
 *              Un-Register with IBMF and SA access
 *              Teardown IBMF receive callback routine
 */
/*ARGSUSED*/
static void
ibdm_event_hdlr(void *clnt_hdl,
    ibt_hca_hdl_t hca_hdl, ibt_async_code_t code, ibt_async_event_t *event)
{
        ibdm_hca_list_t         *hca_list;
        ibdm_port_attr_t        *port;
        ibmf_saa_handle_t       port_sa_hdl;

        IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: async code 0x%x", code);

        switch (code) {
        case IBT_HCA_ATTACH_EVENT:      /* New HCA registered with IBTF */
                ibdm_handle_hca_attach(event->ev_hca_guid);
                break;

        case IBT_HCA_DETACH_EVENT:      /* HCA unregistered with IBTF */
                ibdm_handle_hca_detach(event->ev_hca_guid);
                mutex_enter(&ibdm.ibdm_ibnex_mutex);
                if (ibdm.ibdm_ibnex_callback != NULL) {
                        (*ibdm.ibdm_ibnex_callback)((void *)
                            &event->ev_hca_guid, IBDM_EVENT_HCA_REMOVED);
                }
                mutex_exit(&ibdm.ibdm_ibnex_mutex);
                break;

        case IBT_EVENT_PORT_UP:
                IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: PORT_UP");
                mutex_enter(&ibdm.ibdm_hl_mutex);
                port = ibdm_get_port_attr(event, &hca_list);
                if (port == NULL) {
                        IBTF_DPRINTF_L2("ibdm",
                            "\tevent_hdlr: HCA not present");
                        mutex_exit(&ibdm.ibdm_hl_mutex);
                        break;
                }
                ibdm_initialize_port(port);
                hca_list->hl_nports_active++;
                cv_broadcast(&ibdm.ibdm_port_settle_cv);
                mutex_exit(&ibdm.ibdm_hl_mutex);

                /* Inform IB nexus driver */
                mutex_enter(&ibdm.ibdm_ibnex_mutex);
                if (ibdm.ibdm_ibnex_callback != NULL) {
                        (*ibdm.ibdm_ibnex_callback)((void *)
                            &event->ev_hca_guid, IBDM_EVENT_PORT_UP);
                }
                mutex_exit(&ibdm.ibdm_ibnex_mutex);
                break;

        case IBT_ERROR_PORT_DOWN:
                IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: PORT_DOWN");
                mutex_enter(&ibdm.ibdm_hl_mutex);
                port = ibdm_get_port_attr(event, &hca_list);
                if (port == NULL) {
                        IBTF_DPRINTF_L2("ibdm",
                            "\tevent_hdlr: HCA not present");
                        mutex_exit(&ibdm.ibdm_hl_mutex);
                        break;
                }
                hca_list->hl_nports_active--;
                port_sa_hdl = port->pa_sa_hdl;
                (void) ibdm_fini_port(port);
```



```
                port->pa_state = IBT_PORT_DOWN;
                cv_broadcast(&ibdm.ibdm_port_settle_cv);
                mutex_exit(&ibdm.ibdm_hl_mutex);
                ibdm_reset_all_dgids(port_sa_hdl);
                break;

        case IBT_PORT_CHANGE_EVENT:
                IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: PORT_CHANGE");
                if (event->ev_port_flags & IBT_PORT_CHANGE_PKEY)
                        ibdm_handle_port_change_event(event);
                break;

        default:                /* Ignore all other events/errors */
                break;
        }
}

static void
ibdm_handle_port_change_event(ibt_async_event_t *event)
{
        ibdm_port_attr_t        *port;
        ibdm_hca_list_t         *hca_list;

        IBTF_DPRINTF_L2("ibdm", "\tibdm_handle_port_change_event:"
            " HCA guid  %llx", event->ev_hca_guid);
        mutex_enter(&ibdm.ibdm_hl_mutex);
        port = ibdm_get_port_attr(event, &hca_list);
        if (port == NULL) {
                IBTF_DPRINTF_L2("ibdm", "\tevent_hdlr: HCA not present");
                mutex_exit(&ibdm.ibdm_hl_mutex);
                return;
        }
        ibdm_update_port_pkeys(port);
        cv_broadcast(&ibdm.ibdm_port_settle_cv);
        mutex_exit(&ibdm.ibdm_hl_mutex);

        /* Inform IB nexus driver */
        mutex_enter(&ibdm.ibdm_ibnex_mutex);
        if (ibdm.ibdm_ibnex_callback != NULL) {
                (*ibdm.ibdm_ibnex_callback)((void *)
                    &event->ev_hca_guid, IBDM_EVENT_PORT_PKEY_CHANGE);
        }
        mutex_exit(&ibdm.ibdm_ibnex_mutex);
}

/*
 * ibdm_update_port_pkeys()
 *      Update the pkey table
 *      Update the port attributes
 */
static void
ibdm_update_port_pkeys(ibdm_port_attr_t *port)
{
        uint_t                          nports, size;
        uint_t                          pkey_idx, opkey_idx;
        uint16_t                        npkeys;
        ibt_hca_portinfo_t              *pinfop;
        ib_pkey_t                       pkey;
        ibdm_pkey_tbl_t                 *pkey_tbl;
        ibdm_port_attr_t                newport;

        IBTF_DPRINTF_L4("ibdm", "\tupdate_port_pkeys:");
        ASSERT(MUTEX_HELD(&ibdm.ibdm_hl_mutex));

        /* Check whether the port is active */
        if (ibt_get_port_state(port->pa_hca_hdl, port->pa_port_num, NULL,
```

```
653             NULL) != IBT_SUCCESS)
654                 return;
655
656         if (ibt_query_hca_ports(port->pa_hca_hdl, port->pa_port_num,
657             &pinfop, &nports, &size) != IBT_SUCCESS) {
658                 /* This should not occur */
659                 port->pa_npkeys = 0;
660                 port->pa_pkey_tbl = NULL;
661                 return;
662         }
663
664         npkeys = pinfop->p_pkey_tbl_sz;
665         pkey_tbl = kmem_zalloc(npkeys * sizeof (ibdm_pkey_tbl_t), KM_SLEEP);
666         newport.pa_pkey_tbl = pkey_tbl;
667         newport.pa_ibmf_hdl = port->pa_ibmf_hdl;
668
669         for (pkey_idx = 0; pkey_idx < npkeys; pkey_idx++) {
670                 pkey = pkey_tbl[pkey_idx].pt_pkey =
671                     pinfop->p_pkey_tbl[pkey_idx];
672                 /*
673                  * Is this pkey present in the current table ?
674                  */
675                 for (opkey_idx = 0; opkey_idx < port->pa_npkeys; opkey_idx++) {
676                         if (pkey == port->pa_pkey_tbl[opkey_idx].pt_pkey) {
677                                 pkey_tbl[pkey_idx].pt_qp_hdl =
678                                     port->pa_pkey_tbl[opkey_idx].pt_qp_hdl;
679                                 port->pa_pkey_tbl[opkey_idx].pt_qp_hdl = NULL;
680                                 break;
681                         }
682                 }
683
684                 if (opkey_idx == port->pa_npkeys) {
685                         pkey = pkey_tbl[pkey_idx].pt_pkey;
686                         if (IBDM_INVALID_PKEY(pkey)) {
687                                 pkey_tbl[pkey_idx].pt_qp_hdl = NULL;
688                                 continue;
689                         }
690                         ibdm_port_attr_ibmf_init(&newport, pkey, pkey_idx);
691                 }
692         }
693
694         for (opkey_idx = 0; opkey_idx < port->pa_npkeys; opkey_idx++) {
695                 if (port->pa_pkey_tbl[opkey_idx].pt_qp_hdl != NULL) {
696                         if (ibdm_port_attr_ibmf_fini(port, opkey_idx) !=
697                             IBDM_SUCCESS) {
698                                 IBTF_DPRINTF_L2("ibdm", "\tupdate_port_pkeys: "
699                                     "ibdm_port_attr_ibmf_fini failed for "
700                                     "port pkey 0x%x",
701                                     port->pa_pkey_tbl[opkey_idx].pt_pkey);
702                         }
703                 }
704         }
705
706         if (port->pa_pkey_tbl != NULL) {
707                 kmem_free(port->pa_pkey_tbl,
708                     port->pa_npkeys * sizeof (ibdm_pkey_tbl_t));
709         }
710
711         port->pa_npkeys = npkeys;
712         port->pa_pkey_tbl = pkey_tbl;
713         port->pa_sn_prefix = pinfop->p_sgid_tbl[0].gid_prefix;
714         port->pa_state = pinfop->p_linkstate;
715         ibt_free_portinfo(pinfop, size);
716 }
717
718 /*
```



```
719  * ibdm_initialize_port()
720  *      Register with IBMF
721  *      Register with SA access
722  *      Register a receive callback routine with IBMF. IBMF invokes
723  *      this routine whenever a MAD arrives at this port.
724  *      Update the port attributes
725  */
726 static void
727 ibdm_initialize_port(ibdm_port_attr_t *port)
728 {
729         int                             ii;
730         uint_t                          nports, size;
731         uint_t                          pkey_idx;
732         ib_pkey_t                       pkey;
733         ibt_hca_portinfo_t              *pinfop;
734         ibmf_register_info_t            ibmf_reg;
735         ibmf_saa_subnet_event_args_t    event_args;
736
737         IBTF_DPRINTF_L4("ibdm", "\tinitialize_port:");
738         ASSERT(MUTEX_HELD(&ibdm.ibdm_hl_mutex));
739
740         /* Check whether the port is active */
741         if (ibt_get_port_state(port->pa_hca_hdl, port->pa_port_num, NULL,
742             NULL) != IBT_SUCCESS)
743                 return;
744
745         if (port->pa_sa_hdl != NULL || port->pa_pkey_tbl != NULL)
746                 return;
747
748         if (ibt_query_hca_ports(port->pa_hca_hdl, port->pa_port_num,
749             &pinfop, &nports, &size) != IBT_SUCCESS) {
750                 /* This should not occur */
751                 port->pa_npkeys         = 0;
752                 port->pa_pkey_tbl       = NULL;
753                 return;
754         }
755         port->pa_sn_prefix = pinfop->p_sgid_tbl[0].gid_prefix;
756
757         port->pa_state          = pinfop->p_linkstate;
758         port->pa_npkeys         = pinfop->p_pkey_tbl_sz;
759         port->pa_pkey_tbl       = (ibdm_pkey_tbl_t *)kmem_zalloc(
760             port->pa_npkeys * sizeof (ibdm_pkey_tbl_t), KM_SLEEP);
761
762         for (pkey_idx = 0; pkey_idx < port->pa_npkeys; pkey_idx++)
763                 port->pa_pkey_tbl[pkey_idx].pt_pkey =
764                     pinfop->p_pkey_tbl[pkey_idx];
765
766         ibt_free_portinfo(pinfop, size);
767
768         if (ibdm_enumerate_iocs) {
769                 event_args.is_event_callback = ibdm_saa_event_cb;
770                 event_args.is_event_callback_arg = port;
771                 if (ibmf_sa_session_open(port->pa_port_guid, 0, &event_args,
772                     IBMF_VERSION, 0, &port->pa_sa_hdl) != IBMF_SUCCESS) {
773                         IBTF_DPRINTF_L2("ibdm", "\tinitialize_port: "
774                             "sa access registration failed");
775                         (void) ibdm_fini_port(port);
776                         return;
777                 }
778
779                 ibmf_reg.ir_ci_guid             = port->pa_hca_guid;
780                 ibmf_reg.ir_port_num            = port->pa_port_num;
781                 ibmf_reg.ir_client_class        = DEV_MGT_MANAGER;
782
783                 if (ibmf_register(&ibmf_reg, IBMF_VERSION, 0, NULL, NULL,
784                     &port->pa_ibmf_hdl, &port->pa_ibmf_caps) != IBMF_SUCCESS) {
```

```
785                         IBTF_DPRINTF_L2("ibdm", "\tinitialize_port: "
786                             "IBMF registration failed");
787                         (void) ibdm_fini_port(port);
788                         return;
789                 }
790
791                 if (ibmf_setup_async_cb(port->pa_ibmf_hdl,
792                     IBMF_QP_HANDLE_DEFAULT,
793                     ibdm_ibmf_recv_cb, 0, 0) != IBMF_SUCCESS) {
794                         IBTF_DPRINTF_L2("ibdm", "\tinitialize_port: "
795                             "IBMF setup recv cb failed");
796                         (void) ibdm_fini_port(port);
797                         return;
798                 }
799         } else {
800                 port->pa_sa_hdl = NULL;
801                 port->pa_ibmf_hdl = NULL;
802         }
803
804         for (ii = 0; ii < port->pa_npkeys; ii++) {
805                 pkey = port->pa_pkey_tbl[ii].pt_pkey;
806                 if (IBDM_INVALID_PKEY(pkey)) {
807                         port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
808                         continue;
809                 }
810                 ibdm_port_attr_ibmf_init(port, pkey, ii);
811         }
812 }
813
814
815 /*
816  * ibdm_port_attr_ibmf_init:
817  *      With IBMF - Alloc QP Handle and Setup Async callback
818  */
819 static void
820 ibdm_port_attr_ibmf_init(ibdm_port_attr_t *port, ib_pkey_t pkey, int ii)
821 {
822         int ret;
823
824         if (ibdm_enumerate_iocs == 0) {
825                 port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
826                 return;
827         }
828
829         if ((ret = ibmf_alloc_qp(port->pa_ibmf_hdl, pkey, IB_GSI_QKEY,
830             IBMF_ALT_QP_MAD_NO_RMPP, &port->pa_pkey_tbl[ii].pt_qp_hdl)) !=
831             IBMF_SUCCESS) {
832                 IBTF_DPRINTF_L2("ibdm", "\tport_attr_ibmf_init: "
833                     "IBMF failed to alloc qp %d", ret);
834                 port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
835                 return;
836         }
837
838         IBTF_DPRINTF_L4("ibdm", "\tport_attr_ibmf_init: QP handle is %p",
839             port->pa_ibmf_hdl);
840
841         if ((ret = ibmf_setup_async_cb(port->pa_ibmf_hdl,
842             port->pa_pkey_tbl[ii].pt_qp_hdl, ibdm_ibmf_recv_cb, 0, 0)) !=
843             IBMF_SUCCESS) {
844                 IBTF_DPRINTF_L2("ibdm", "\tport_attr_ibmf_init: "
845                     "IBMF setup recv cb failed %d", ret);
846                 (void) ibmf_free_qp(port->pa_ibmf_hdl,
847                     &port->pa_pkey_tbl[ii].pt_qp_hdl, 0);
848                 port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
849         }
850 }
```



```
851
852
853 /*
854  * ibdm_get_port_attr()
855  *      Get port attributes from HCA guid and port number
856  *      Return pointer to ibdm_port_attr_t on Success
857  *      and NULL on failure
858  */
859 static ibdm_port_attr_t *
860 ibdm_get_port_attr(ibt_async_event_t *event, ibdm_hca_list_t **retval)
861 {
862         ibdm_hca_list_t         *hca_list;
863         ibdm_port_attr_t        *port_attr;
864         int                     ii;
865
866         IBTF_DPRINTF_L4("ibdm", "\tget_port_attr: port# %d", event->ev_port);
867         ASSERT(MUTEX_HELD(&ibdm.ibdm_hl_mutex));
868         hca_list = ibdm.ibdm_hca_list_head;
869         while (hca_list) {
870                 if (hca_list->hl_hca_guid == event->ev_hca_guid) {
871                         for (ii = 0; ii < hca_list->hl_nports; ii++) {
872                                 port_attr = &hca_list->hl_port_attr[ii];
873                                 if (port_attr->pa_port_num == event->ev_port) {
874                                         *retval = hca_list;
875                                         return (port_attr);
876                                 }
877                         }
878                 }
879                 hca_list = hca_list->hl_next;
880         }
881         return (NULL);
882 }
883
884
885 /*
886  * ibdm_update_port_attr()
887  *      Update the port attributes
888  */
889 static void
890 ibdm_update_port_attr(ibdm_port_attr_t *port)
891 {
892         uint_t                  nports, size;
893         uint_t                  pkey_idx;
894         ibt_hca_portinfo_t      *portinfop;
895
896         IBTF_DPRINTF_L4("ibdm", "\tupdate_port_attr: Begin");
897         if (ibt_query_hca_ports(port->pa_hca_hdl,
898             port->pa_port_num, &portinfop, &nports, &size) != IBT_SUCCESS) {
899                 /* This should not occur */
900                 port->pa_npkeys         = 0;
901                 port->pa_pkey_tbl       = NULL;
902                 return;
903         }
904         port->pa_sn_prefix = portinfop->p_sgid_tbl[0].gid_prefix;
905
906         port->pa_state          = portinfop->p_linkstate;
907
908         /*
909          * PKey information in portinfo valid only if port is
910          * ACTIVE. Bail out if not.
911          */
912         if (port->pa_state != IBT_PORT_ACTIVE) {
913                 port->pa_npkeys         = 0;
914                 port->pa_pkey_tbl       = NULL;
915                 ibt_free_portinfo(portinfop, size);
916                 return;
```

```
917         }

919         port->pa_npkeys         = portinfop->p_pkey_tbl_sz;
920         port->pa_pkey_tbl       = (ibdm_pkey_tbl_t *)kmem_zalloc(
921             port->pa_npkeys * sizeof (ibdm_pkey_tbl_t), KM_SLEEP);

923         for (pkey_idx = 0; pkey_idx < port->pa_npkeys; pkey_idx++) {
924                 port->pa_pkey_tbl[pkey_idx].pt_pkey =
925                     portinfop->p_pkey_tbl[pkey_idx];
926         }
927         ibt_free_portinfo(portinfop, size);
928 }


931 /*
932  * ibdm_handle_hca_attach()
933  */
934 static void
935 ibdm_handle_hca_attach(ib_guid_t hca_guid)
936 {
937         uint_t                  size;
938         uint_t                  ii, nports;
939         ibt_status_t            status;
940         ibt_hca_hdl_t           hca_hdl;
941         ibt_hca_attr_t          *hca_attr;
942         ibdm_hca_list_t         *hca_list, *temp;
943         ibdm_port_attr_t        *port_attr;
944         ibt_hca_portinfo_t      *portinfop;

946         IBTF_DPRINTF_L4("ibdm",
947             "\thandle_hca_attach: hca_guid = 0x%llX", hca_guid);

949         /* open the HCA first */
950         if ((status = ibt_open_hca(ibdm.ibdm_ibt_clnt_hdl, hca_guid,
951             &hca_hdl)) != IBT_SUCCESS) {
952                 IBTF_DPRINTF_L2("ibdm", "\thandle_hca_attach: "
953                     "open_hca failed, status 0x%x", status);
954                 return;
955         }

957         hca_attr = (ibt_hca_attr_t *)
958             kmem_alloc(sizeof (ibt_hca_attr_t), KM_SLEEP);
959         /* ibt_query_hca always returns IBT_SUCCESS */
960         (void) ibt_query_hca(hca_hdl, hca_attr);

962         IBTF_DPRINTF_L4("ibdm", "\tvid: 0x%x, pid: 0x%x, ver: 0x%x,"
963             " #ports: %d", hca_attr->hca_vendor_id, hca_attr->hca_device_id,
964             hca_attr->hca_version_id, hca_attr->hca_nports);

966         if ((status = ibt_query_hca_ports(hca_hdl, 0, &portinfop, &nports,
967             &size)) != IBT_SUCCESS) {
968                 IBTF_DPRINTF_L2("ibdm", "\thandle_hca_attach: "
969                     "ibt_query_hca_ports failed, status 0x%x", status);
970                 kmem_free(hca_attr, sizeof (ibt_hca_attr_t));
971                 (void) ibt_close_hca(hca_hdl);
972                 return;
973         }
974         hca_list = (ibdm_hca_list_t *)
975             kmem_zalloc((sizeof (ibdm_hca_list_t)), KM_SLEEP);
976         hca_list->hl_port_attr = (ibdm_port_attr_t *)kmem_zalloc(
977             (sizeof (ibdm_port_attr_t) * hca_attr->hca_nports), KM_SLEEP);
978         hca_list->hl_hca_guid = hca_attr->hca_node_guid;
979         hca_list->hl_nports = hca_attr->hca_nports;
980         hca_list->hl_attach_time = gethrtime();
 24         hca_list->hl_attach_time = ddi_get_time();
981         hca_list->hl_hca_hdl = hca_hdl;
```



```
983         /*
984          * Init a dummy port attribute for the HCA node
985          * This is for Per-HCA Node. Initialize port_attr :
986          *      hca_guid & port_guid -> hca_guid
987          *      npkeys, pkey_tbl is NULL
988          *      port_num, sn_prefix is 0
989          *      vendorid, product_id, dev_version from HCA
990          *      pa_state is IBT_PORT_ACTIVE
991          */
992         hca_list->hl_hca_port_attr = (ibdm_port_attr_t *)kmem_zalloc(
993             sizeof (ibdm_port_attr_t), KM_SLEEP);
994         port_attr = hca_list->hl_hca_port_attr;
995         port_attr->pa_vendorid  = hca_attr->hca_vendor_id;
996         port_attr->pa_productid = hca_attr->hca_device_id;
997         port_attr->pa_dev_version = hca_attr->hca_version_id;
998         port_attr->pa_hca_guid  = hca_attr->hca_node_guid;
999         port_attr->pa_hca_hdl   = hca_list->hl_hca_hdl;
1000        port_attr->pa_port_guid = hca_attr->hca_node_guid;
1001        port_attr->pa_state     = IBT_PORT_ACTIVE;


1004        for (ii = 0; ii < nports; ii++) {
1005                port_attr               = &hca_list->hl_port_attr[ii];
1006                port_attr->pa_vendorid  = hca_attr->hca_vendor_id;
1007                port_attr->pa_productid = hca_attr->hca_device_id;
1008                port_attr->pa_dev_version = hca_attr->hca_version_id;
1009                port_attr->pa_hca_guid  = hca_attr->hca_node_guid;
1010                port_attr->pa_hca_hdl   = hca_list->hl_hca_hdl;
1011                port_attr->pa_port_guid = portinfop[ii].p_sgid_tbl->gid_guid;
1012                port_attr->pa_sn_prefix = portinfop[ii].p_sgid_tbl->gid_prefix;
1013                port_attr->pa_port_num  = portinfop[ii].p_port_num;
1014                port_attr->pa_state     = portinfop[ii].p_linkstate;

1016                /*
1017                 * Register with IBMF, SA access when the port is in
1018                 * ACTIVE state. Also register a callback routine
1019                 * with IBMF to receive incoming DM MAD's.
1020                 * The IBDM event handler takes care of registration of
1021                 * port which are not active.
1022                 */
1023                IBTF_DPRINTF_L4("ibdm",
1024                    "\thandle_hca_attach: port guid %llx Port state 0x%x",
1025                    port_attr->pa_port_guid, portinfop[ii].p_linkstate);

1027                if (portinfop[ii].p_linkstate == IBT_PORT_ACTIVE) {
1028                        mutex_enter(&ibdm.ibdm_hl_mutex);
1029                        hca_list->hl_nports_active++;
1030                        ibdm_initialize_port(port_attr);
1031                        cv_broadcast(&ibdm.ibdm_port_settle_cv);
1032                        mutex_exit(&ibdm.ibdm_hl_mutex);
1033                }
1034        }
1035        mutex_enter(&ibdm.ibdm_hl_mutex);
1036        for (temp = ibdm.ibdm_hca_list_head; temp; temp = temp->hl_next) {
1037                if (temp->hl_hca_guid == hca_guid) {
1038                        IBTF_DPRINTF_L2("ibdm", "hca_attach: HCA %llX "
1039                            "already seen by IBDM", hca_guid);
1040                        mutex_exit(&ibdm.ibdm_hl_mutex);
1041                        (void) ibdm_uninit_hca(hca_list);
1042                        return;
1043                }
1044        }
1045        ibdm.ibdm_hca_count++;
1046        if (ibdm.ibdm_hca_list_head == NULL) {
1047                ibdm.ibdm_hca_list_head = hca_list;
```

```
1048                    ibdm.ibdm_hca_list_tail = hca_list;
1049            } else {
1050                    ibdm.ibdm_hca_list_tail->hl_next = hca_list;
1051                    ibdm.ibdm_hca_list_tail = hca_list;
1052            }
1053            mutex_exit(&ibdm.ibdm_hl_mutex);
1054            mutex_enter(&ibdm.ibdm_ibnex_mutex);
1055            if (ibdm.ibdm_ibnex_callback != NULL) {
1056                    (*ibdm.ibdm_ibnex_callback)((void *)
1057                        &hca_guid, IBDM_EVENT_HCA_ADDED);
1058            }
1059            mutex_exit(&ibdm.ibdm_ibnex_mutex);

1061            kmem_free(hca_attr, sizeof (ibt_hca_attr_t));
1062            ibt_free_portinfo(portinfop, size);
1063    }
______unchanged_portion_omitted_

4696    /*
4697     * ibdm_get_waittime()
4698     *      Calculates the wait time based on the last HCA attach time
4699     */
4700    static clock_t
4701    ibdm_get_waittime(ib_guid_t hca_guid, int dft_wait_sec)
3744    static time_t
3745    ibdm_get_waittime(ib_guid_t hca_guid, int dft_wait)
4702    {
4703            const hrtime_t  dft_wait = dft_wait_sec * NANOSEC;
4704            hrtime_t        temp, wait_time = 0;
4705            clock_t         usecs;
4706            int             i;
3747            int             ii;
3748            time_t          temp, wait_time = 0;
4707            ibdm_hca_list_t *hca;

4709            IBTF_DPRINTF_L4("ibdm", "\tget_waittime hcaguid:%llx"
4710                "\tport settling time %d", hca_guid, dft_wait);

4712            ASSERT(mutex_owned(&ibdm.ibdm_hl_mutex));

4714            hca = ibdm.ibdm_hca_list_head;

4716            for (i = 0; i < ibdm.ibdm_hca_count; i++, hca = hca->hl_next) {
4717                    if (hca->hl_nports == hca->hl_nports_active)
4718                            continue;

4720                    if (hca_guid && (hca_guid != hca->hl_hca_guid))
4721                            continue;

4723                    temp = gethrtime() - hca->hl_attach_time;
4724                    temp = MAX(0, (dft_wait - temp));

4726    #endif /* ! codereview */
4727                    if (hca_guid) {
4728                            wait_time = temp;
3758                    for (ii = 0; ii < ibdm.ibdm_hca_count; ii++) {
3759                            if ((hca_guid == hca->hl_hca_guid) &&
3760                                (hca->hl_nports != hca->hl_nports_active)) {
3761                                    wait_time =
3762                                        ddi_get_time() - hca->hl_attach_time;
3763                                    wait_time = ((wait_time >= dft_wait) ?
3764                                        0 : (dft_wait - wait_time));
4729                            break;
4730                    }

4732                    wait_time = MAX(temp, wait_time);
```



```
3767                            hca = hca->hl_next;
3768                    }
3769                    IBTF_DPRINTF_L2("ibdm", "\tget_waittime: wait_time = %ld secs",
3770                        (long)wait_time);
3771                    return (wait_time);
4733            }

4735            /* convert to microseconds */
4736            usecs = MIN(wait_time, dft_wait) / (NANOSEC / MICROSEC);

4738            IBTF_DPRINTF_L2("ibdm", "\tget_waittime: wait_time = %ld usecs",
4739                (long)usecs);

4741            return (drv_usectohz(usecs));
3774            for (ii = 0; ii < ibdm.ibdm_hca_count; ii++) {
3775                    if (hca->hl_nports != hca->hl_nports_active) {
3776                            temp = ddi_get_time() - hca->hl_attach_time;
3777                            temp = ((temp >= dft_wait) ? 0 : (dft_wait - temp));
3778                            wait_time = (temp > wait_time) ? temp : wait_time;
3779                    }
3780                    hca = hca->hl_next;
3781            }
3782            IBTF_DPRINTF_L2("ibdm", "\tget_waittime: wait_time = %ld secs",
3783                (long)wait_time);
3784            return (wait_time);
4742    }

4744    void
4745    ibdm_ibnex_port_settle_wait(ib_guid_t hca_guid, int dft_wait)
4746    {
4747            clock_t wait_time;
3790            time_t wait_time;
3791            clock_t delta;

4749            mutex_enter(&ibdm.ibdm_hl_mutex);

4751            while ((wait_time = ibdm_get_waittime(hca_guid, dft_wait)) > 0)
3795            while ((wait_time = ibdm_get_waittime(hca_guid, dft_wait)) > 0) {
3796                    if (wait_time > dft_wait) {
3797                            IBTF_DPRINTF_L1("ibdm",
3798                                "\tibnex_port_settle_wait: wait_time = %ld secs; "
3799                                "Resetting to %d secs",
3800                                (long)wait_time, dft_wait);
3801                            wait_time = dft_wait;
3802                    }
3803                    delta = drv_usectohz(wait_time * 1000000);
4752                    (void) cv_reltimedwait(&ibdm.ibdm_port_settle_cv,
4753                        &ibdm.ibdm_hl_mutex, wait_time, TR_CLOCK_TICK);
3805                    &ibdm.ibdm_hl_mutex, delta, TR_CLOCK_TICK);
3806            }

4755            mutex_exit(&ibdm.ibdm_hl_mutex);
4756    }
______unchanged_portion_omitted_
```

```
**********************************************************
   41057 Mon May  5 14:29:43 2014
new/usr/src/uts/common/io/iprb/iprb.c
4778 iprb shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
**********************************************************
______unchanged_portion_omitted_

 258 int
 259 iprb_attach(dev_info_t *dip)
 260 {
 261         iprb_t          *ip;
 262         uint16_t        w;
 263         int             i;
 264         mac_register_t  *macp;

 266         ip = kmem_zalloc(sizeof (*ip), KM_SLEEP);
 267         ddi_set_driver_private(dip, ip);
 268         ip->dip = dip;

 270         list_create(&ip->mcast, sizeof (struct iprb_mcast),
 271             offsetof(struct iprb_mcast, node));

 273         /* we don't support high level interrupts, so we don't need cookies */
 274         mutex_init(&ip->culock, NULL, MUTEX_DRIVER, NULL);
 275         mutex_init(&ip->rulock, NULL, MUTEX_DRIVER, NULL);

 277         if (pci_config_setup(dip, &ip->pcih) != DDI_SUCCESS) {
 278                 iprb_error(ip, "unable to map configuration space");
 279                 iprb_destroy(ip);
 280                 return (DDI_FAILURE);
 281         }

 283         if (ddi_regs_map_setup(dip, 1, &ip->regs, 0, 0, &acc_attr,
 284             &ip->regsh) != DDI_SUCCESS) {
 285                 iprb_error(ip, "unable to map device registers");
 286                 iprb_destroy(ip);
 287                 return (DDI_FAILURE);
 288         }

 290         /* Reset, but first go into idle state */
 291         PUT32(ip, CSR_PORT, PORT_SEL_RESET);
 292         drv_usecwait(10);
 293         PUT32(ip, CSR_PORT, PORT_SW_RESET);
 294         drv_usecwait(10);
 295         PUT8(ip, CSR_INTCTL, INTCTL_MASK);
 296         (void) GET8(ip, CSR_INTCTL);

 298         /*
 299          * Precalculate watchdog times.
 300          */
 301         ip->tx_timeout = TX_WATCHDOG;
 302         ip->rx_timeout = RX_WATCHDOG;
 301         ip->tx_timeout = drv_usectohz(TX_WATCHDOG * 1000000);
 302         ip->rx_timeout = drv_usectohz(RX_WATCHDOG * 1000000);

 304         iprb_identify(ip);

 306         /* Obtain our factory MAC address */
 307         w = iprb_eeprom_read(ip, 0);
 308         ip->factaddr[0] = w & 0xff;
 309         ip->factaddr[1] = w >> 8;
 310         w = iprb_eeprom_read(ip, 1);
 311         ip->factaddr[2] = w & 0xff;
 312         ip->factaddr[3] = w >> 8;
```



```
 313         w = iprb_eeprom_read(ip, 2);
 314         ip->factaddr[4] = w & 0xff;
 315         ip->factaddr[5] = w >> 8;
 316         bcopy(ip->factaddr, ip->curraddr, 6);

 318         if (ip->resumebug) {
 319                 /*
 320                  * Generally, most devices we will ever see will
 321                  * already have fixed firmware.  Since I can't verify
 322                  * the validity of the fix (no suitably downrev
 323                  * hardware), we'll just do our best to avoid it for
 324                  * devices that exhibit this behavior.
 325                  */
 326                 if ((iprb_eeprom_read(ip, 10) & 0x02) == 0) {
 327                         /* EEPROM fix was already applied, assume safe. */
 328                         ip->resumebug = B_FALSE;
 329                 }
 330         }

 332         if ((iprb_eeprom_read(ip, 3) & 0x3) != 0x3) {
 333                 cmn_err(CE_CONT, "?Enabling RX errata workaround.\n");
 334                 ip->rxhangbug = B_TRUE;
 335         }

 337         /* Determine whether we have an MII or a legacy 80c24 */
 338         w = iprb_eeprom_read(ip, 6);
 339         if ((w & 0x3f00) != 0x0600) {
 340                 if ((ip->miih = mii_alloc(ip, dip, &iprb_mii_ops)) == NULL) {
 341                         iprb_error(ip, "unable to allocate MII ops vector");
 342                         iprb_destroy(ip);
 343                         return (DDI_FAILURE);
 344                 }
 345                 if (ip->canpause) {
 346                         mii_set_pauseable(ip->miih, B_TRUE, B_FALSE);
 347                 }
 348         }

 350         /* Allocate cmds and tx region */
 351         for (i = 0; i < NUM_TX; i++) {
 352                 /* Command blocks */
 353                 if (iprb_dma_alloc(ip, &ip->cmds[i], CB_SIZE) != DDI_SUCCESS) {
 354                         iprb_destroy(ip);
 355                         return (DDI_FAILURE);
 356                 }
 357         }

 359         for (i = 0; i < NUM_TX; i++) {
 360                 iprb_dma_t *cb = &ip->cmds[i];
 361                 /* Link the command blocks into a ring */
 362                 PUTCB32(cb, CB_LNK_OFFSET, (ip->cmds[(i + 1) % NUM_TX].paddr));
 363         }

 365         for (i = 0; i < NUM_RX; i++) {
 366                 /* Rx packet buffers */
 367                 if (iprb_dma_alloc(ip, &ip->rxb[i], RFD_SIZE) != DDI_SUCCESS) {
 368                         iprb_destroy(ip);
 369                         return (DDI_FAILURE);
 370                 }
 371         }
 372         if (iprb_dma_alloc(ip, &ip->stats, STATS_SIZE) != DDI_SUCCESS) {
 373                 iprb_destroy(ip);
 374                 return (DDI_FAILURE);
 375         }

 377         if (iprb_add_intr(ip) != DDI_SUCCESS) {
 378                 iprb_destroy(ip);
```

```
379                 return (DDI_FAILURE);
380         }

382         if ((macp = mac_alloc(MAC_VERSION)) == NULL) {
383                 iprb_error(ip, "unable to allocate mac structure");
384                 iprb_destroy(ip);
385                 return (DDI_FAILURE);
386         }

388         macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
389         macp->m_driver = ip;
390         macp->m_dip = dip;
391         macp->m_src_addr = ip->curraddr;
392         macp->m_callbacks = &iprb_m_callbacks;
393         macp->m_min_sdu = 0;
394         macp->m_max_sdu = ETHERMTU;
395         macp->m_margin = VLAN_TAGSZ;
396         if (mac_register(macp, &ip->mach) != 0) {
397                 iprb_error(ip, "unable to register mac with framework");
398                 mac_free(macp);
399                 iprb_destroy(ip);
400                 return (DDI_FAILURE);
401         }

403         mac_free(macp);
404         return (DDI_SUCCESS);
405 }
```

*unchanged_portion_omitted_*

```
689 void
690 iprb_cmd_reclaim(iprb_t *ip)
691 {
692         while (ip->cmd_count) {
693                 iprb_dma_t *cb = &ip->cmds[ip->cmd_tail];

695                 SYNCCB(cb, CB_STS_OFFSET, 2, DDI_DMA_SYNC_FORKERNEL);
696                 if ((GETCB16(cb, CB_STS_OFFSET) & CB_STS_C) == 0) {
697                         break;
698                 }

700                 ip->cmd_tail++;
701                 ip->cmd_tail %= NUM_TX;
702                 ip->cmd_count--;
703                 if (ip->cmd_count == 0) {
704                         ip->tx_wdog = 0;
705                 } else {
706                         ip->tx_wdog = gethrtime();
706                         ip->tx_wdog = ddi_get_time();
707                 }
708         }
709 }
```

*unchanged_portion_omitted_*

```
724 int
725 iprb_cmd_submit(iprb_t *ip, uint16_t cmd)
726 {
727         iprb_dma_t      *ncb = &ip->cmds[ip->cmd_head];
728         iprb_dma_t      *lcb = &ip->cmds[ip->cmd_last];

730         /* If this command will consume the last CB, interrupt when done */
731         ASSERT((ip->cmd_count) < NUM_TX);
732         if (ip->cmd_count == (NUM_TX - 1)) {
733                 cmd |= CB_CMD_I;
734         }

736         /* clear the status entry */
```



```
737         PUTCB16(ncb, CB_STS_OFFSET, 0);

739         /* suspend upon completion of this new command */
740         cmd |= CB_CMD_S;
741         PUTCB16(ncb, CB_CMD_OFFSET, cmd);
742         SYNCCB(ncb, 0, 0, DDI_DMA_SYNC_FORDEV);

744         /* clear the suspend flag from the last submitted command */
745         SYNCCB(lcb, CB_CMD_OFFSET, 2, DDI_DMA_SYNC_FORKERNEL);
746         PUTCB16(lcb, CB_CMD_OFFSET, GETCB16(lcb, CB_CMD_OFFSET) & ~CB_CMD_S);
747         SYNCCB(lcb, CB_CMD_OFFSET, 2, DDI_DMA_SYNC_FORDEV);


750         /*
751          * If the chip has a resume bug, then we need to try this as a work
752          * around.  Some anecdotal evidence is that this will help solve
753          * the resume bug.  Its a performance hit, but only if the EEPROM
754          * is not updated.  (In theory we could do this only for 10Mbps HDX,
755          * but since it should just about never get used, we keep it simple.)
756          */
757         if (ip->resumebug) {
758                 if (iprb_cmd_ready(ip) != DDI_SUCCESS)
759                         return (DDI_FAILURE);
760                 PUT8(ip, CSR_CMD, CUC_NOP);
761                 (void) GET8(ip, CSR_CMD);
762                 drv_usecwait(1);
763         }

765         /* wait for the SCB to be ready to accept a new command */
766         if (iprb_cmd_ready(ip) != DDI_SUCCESS)
767                 return (DDI_FAILURE);

769         /*
770          * Finally we can resume the CU.  Note that if this the first
771          * command in the sequence (i.e. if the CU is IDLE), or if the
772          * CU is already busy working, then this CU resume command
773          * will not have any effect.
774          */
775         PUT8(ip, CSR_CMD, CUC_RESUME);
776         (void) GET8(ip, CSR_CMD);       /* flush CSR */

778         ip->tx_wdog = gethrtime();
778         ip->tx_wdog = ddi_get_time();
779         ip->cmd_last = ip->cmd_head;
780         ip->cmd_head++;
781         ip->cmd_head %= NUM_TX;
782         ip->cmd_count++;

784         return (DDI_SUCCESS);
785 }
```

*unchanged_portion_omitted_*

```
1009 void
1010 iprb_update_stats(iprb_t *ip)
1011 {
1012         iprb_dma_t      *sp = &ip->stats;
1013         hrtime_t        tstamp;
1013         time_t          tstamp;
1014         int             i;

1016         ASSERT(mutex_owned(&ip->culock));

1018         /* Collect the hardware stats, but don't keep redoing it */
1019         tstamp = gethrtime();
1020         if (tstamp / NANOSEC == ip->stats_time / NANOSEC)
1019         if ((tstamp = ddi_get_time()) == ip->stats_time) {
```

```
1021                return;
1021        }

1023        PUTSTAT(sp, STATS_DONE_OFFSET, 0);
1024        SYNCSTATS(sp, 0, 0, DDI_DMA_SYNC_FORDEV);

1026        if (iprb_cmd_ready(ip) != DDI_SUCCESS)
1027                return;
1028        PUT32(ip, CSR_GEN_PTR, sp->paddr);
1029        PUT8(ip, CSR_CMD, CUC_STATSBASE);
1030        (void) GET8(ip, CSR_CMD);

1032        if (iprb_cmd_ready(ip) != DDI_SUCCESS)
1033                return;
1034        PUT8(ip, CSR_CMD, CUC_STATS_RST);
1035        (void) GET8(ip, CSR_CMD);       /* flush wb */

1037        for (i = 10000; i; i -= 10) {
1038                SYNCSTATS(sp, 0, 0, DDI_DMA_SYNC_FORKERNEL);
1039                if (GETSTAT(sp, STATS_DONE_OFFSET) == STATS_RST_DONE) {
1040                        /* yay stats are updated */
1041                        break;
1042                }
1043                drv_usecwait(10);
1044        }
1045        if (i == 0) {
1046                iprb_error(ip, "time out acquiring hardware statistics");
1047                return;
1048        }

1050        ip->ex_coll += GETSTAT(sp, STATS_TX_MAXCOL_OFFSET);
1051        ip->late_coll += GETSTAT(sp, STATS_TX_LATECOL_OFFSET);
1052        ip->uflo += GETSTAT(sp, STATS_TX_UFLO_OFFSET);
1053        ip->defer_xmt += GETSTAT(sp, STATS_TX_DEFER_OFFSET);
1054        ip->one_coll += GETSTAT(sp, STATS_TX_ONECOL_OFFSET);
1055        ip->multi_coll += GETSTAT(sp, STATS_TX_MULTCOL_OFFSET);
1056        ip->collisions += GETSTAT(sp, STATS_TX_TOTCOL_OFFSET);
1057        ip->fcs_errs += GETSTAT(sp, STATS_RX_FCS_OFFSET);
1058        ip->align_errs += GETSTAT(sp, STATS_RX_ALIGN_OFFSET);
1059        ip->norcvbuf += GETSTAT(sp, STATS_RX_NOBUF_OFFSET);
1060        ip->oflo += GETSTAT(sp, STATS_RX_OFLO_OFFSET);
1061        ip->runt += GETSTAT(sp, STATS_RX_SHORT_OFFSET);

1063        ip->stats_time = tstamp;
1064 }
```

*unchanged_portion_omitted_*

```
1156 mblk_t *
1157 iprb_rx(iprb_t *ip)
1158 {
1159        iprb_dma_t      *rfd;
1160        uint16_t        cnt;
1161        uint16_t        sts;
1162        int             i;
1163        mblk_t          *mplist;
1164        mblk_t          **mpp;
1165        mblk_t          *mp;

1167        mplist = NULL;
1168        mpp = &mplist;

1170        for (i = 0; i < NUM_RX; i++) {
1171                rfd = &ip->rxb[ip->rx_index];
1172                SYNCRFD(rfd, RFD_STS_OFFSET, 2, DDI_DMA_SYNC_FORKERNEL);
1173                if ((GETRFD16(rfd, RFD_STS_OFFSET) & RFD_STS_C) == 0) {
1174                        break;
```



```
1175                }

1177                ip->rx_wdog = gethrtime();
1177                ip->rx_wdog = ddi_get_time();

1179                SYNCRFD(rfd, 0, 0, DDI_DMA_SYNC_FORKERNEL);
1180                cnt = GETRFD16(rfd, RFD_CNT_OFFSET);
1181                cnt &= ~(RFD_CNT_EOF | RFD_CNT_F);
1182                sts = GETRFD16(rfd, RFD_STS_OFFSET);

1184                if (cnt > (ETHERMAX + VLAN_TAGSZ)) {
1185                        ip->toolong++;
1186                        iprb_rx_add(ip);
1187                        continue;
1188                }
1189                if (((sts & RFD_STS_OK) == 0) && (sts & RFD_STS_ERRS)) {
1190                        iprb_rx_add(ip);
1191                        continue;
1192                }
1193                if ((mp = allocb(cnt, BPRI_MED)) == NULL) {
1194                        ip->norcvbuf++;
1195                        iprb_rx_add(ip);
1196                        continue;
1197                }
1198                bcopy(rfd->vaddr + RFD_PKT_OFFSET, mp->b_wptr, cnt);

1200                /* return it to the RFD list */
1201                iprb_rx_add(ip);

1203                mp->b_wptr += cnt;
1204                ip->ipackets++;
1205                ip->rbytes += cnt;
1206                if (mp->b_rptr[0] & 0x1) {
1207                        if (bcmp(mp->b_rptr, &iprb_bcast, 6) != 0) {
1208                                ip->multircv++;
1209                        } else {
1210                                ip->brdcstrcv++;
1211                        }
1212                }
1213                *mpp = mp;
1214                mpp = &mp->b_next;
1215        }
1216        return (mplist);
1217 }
```

*unchanged_portion_omitted_*

```
1647 void
1648 iprb_periodic(void *arg)
1649 {
1650        iprb_t *ip = arg;
1651        boolean_t reset = B_FALSE;

1653        mutex_enter(&ip->rulock);
1654        if (ip->suspended || !ip->running) {
1655                mutex_exit(&ip->rulock);
1656                return;
1657        }

1659        /*
1660         * If we haven't received a packet in a while, and if the link
1661         * is up, then it might be a hung chip.  This problem
1662         * reportedly only occurs at 10 Mbps.
1663         */
1664        if (ip->rxhangbug &&
1665            ((ip->miih == NULL) || (mii_get_speed(ip->miih) == 10000000)) &&
1666            ((gethrtime() - ip->rx_wdog) > ip->rx_timeout)) {
```

```
1666             ((ddi_get_time() - ip->rx_wdog) > ip->rx_timeout)) {
1667                 cmn_err(CE_CONT, "?Possible RU hang, resetting.\n");
1668                 reset = B_TRUE;
1669         }

1671         /* update the statistics */
1672         mutex_enter(&ip->culock);

1674         if (ip->tx_wdog && ((gethrtime() - ip->tx_wdog) > ip->tx_timeout)) {
1674         if (ip->tx_wdog && ((ddi_get_time() - ip->tx_wdog) > ip->tx_timeout)) {
1675                 /* transmit/CU hang? */
1676                 cmn_err(CE_CONT, "?CU stalled, resetting.\n");
1677                 reset = B_TRUE;
1678         }

1680         if (reset) {
1681                 /* We want to reconfigure */
1682                 iprb_stop(ip);
1683                 if (iprb_start(ip) != DDI_SUCCESS) {
1684                         iprb_error(ip, "unable to restart chip");
1685                 }
1686         }

1688         iprb_update_stats(ip);

1690         mutex_exit(&ip->culock);
1691         mutex_exit(&ip->rulock);
1692 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    9904 Mon May  5 14:29:43 2014
new/usr/src/uts/common/io/iprb/iprb.h
4778 iprb shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
**********************************************************
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2010 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _IPRB_H
#define _IPRB_H

/*
 * iprb - Intel Pro/100B Ethernet Driver
 */

/*
 * Tunables.
 */
#define NUM_TX          128     /* outstanding tx queue */
#define NUM_RX          128     /* outstanding rx queue */

/* timeouts for the rx and tx watchdogs (nsec) */
#define RX_WATCHDOG     (15 * NANOSEC)
#define TX_WATCHDOG     (15 * NANOSEC)
#define RX_WATCHDOG     15      /* timeout for rx watchdog (sec) */
#define TX_WATCHDOG     15      /* timeout for tx watchdog (sec) */

/*
 * Driver structures.
 */
typedef struct {
        ddi_acc_handle_t        acch;
        ddi_dma_handle_t        dmah;
        caddr_t                 vaddr;
        uint32_t                paddr;
} iprb_dma_t;
_____unchanged_portion_omitted_

typedef struct iprb {
        dev_info_t              *dip;
        ddi_acc_handle_t        pcih;
        ddi_acc_handle_t        regsh;
        caddr_t                 regs;

        uint16_t                devid;
        uint8_t                 revid;

        mac_handle_t            mach;
        mii_handle_t            miih;

        ddi_intr_handle_t       intrh;
```



```
        ddi_periodic_t          perh;

        kmutex_t                culock;
        kmutex_t                rulock;

        uint8_t                 factaddr[6];
        uint8_t                 curraddr[6];

        int                     nmcast;
        list_t                  mcast;
        boolean_t               promisc;
        iprb_dma_t              cmds[NUM_TX];
        iprb_dma_t              rxb[NUM_RX];
        iprb_dma_t              stats;
        hrtime_t                stats_time;
        time_t                  stats_time;

        uint16_t                cmd_head;
        uint16_t                cmd_last;
        uint16_t                cmd_tail;
        uint16_t                cmd_count;

        uint16_t                rx_index;
        uint16_t                rx_last;
        hrtime_t                rx_wdog;
        hrtime_t                rx_timeout;
        hrtime_t                tx_wdog;
        hrtime_t                tx_timeout;
        time_t                  rx_wdog;
        time_t                  rx_timeout;
        time_t                  tx_wdog;
        time_t                  tx_timeout;

        uint16_t                eeprom_bits;

        boolean_t               running;
        boolean_t               suspended;
        boolean_t               wantw;
        boolean_t               rxhangbug;
        boolean_t               resumebug;
        boolean_t               is557;
        boolean_t               canpause;
        boolean_t               canmwi;

        /*
         * Statistics
         */
        uint64_t                ipackets;
        uint64_t                rbytes;
        uint64_t                multircv;
        uint64_t                brdcstrcv;
        uint64_t                opackets;
        uint64_t                obytes;
        uint64_t                multixmt;
        uint64_t                brdcstxmt;
        uint64_t                ex_coll;
        uint64_t                late_coll;
        uint64_t                uflo;
        uint64_t                defer_xmt;
        uint64_t                one_coll;
        uint64_t                multi_coll;
        uint64_t                collisions;
        uint64_t                fcs_errs;
        uint64_t                align_errs;
        uint64_t                norcvbuf;
```

```
 122         uint64_t                oflo;
 123         uint64_t                runt;
 124         uint64_t                nocarrier;
 125         uint64_t                toolong;
 126         uint64_t                macxmt_errs;
 127         uint64_t                macrcv_errs;
 128 } iprb_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   57931 Mon May  5 14:29:43 2014
new/usr/src/uts/common/io/mac/mac_protect.c
4788 mac shouldn't abuse ddi_get_time(9f)
**********************************************************
```

```
/*
 * 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) 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#include <sys/strsun.h>
#include <sys/sdt.h>
#include <sys/mac.h>
#include <sys/mac_impl.h>
#include <sys/mac_client_impl.h>
#include <sys/mac_client_priv.h>
#include <sys/ethernet.h>
#include <sys/vlan.h>
#include <sys/dlpi.h>
#include <sys/avl.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/arp.h>
#include <netinet/arp.h>
#include <netinet/udp.h>
#include <netinet/dhcp.h>
#include <netinet/dhcp6.h>

/*
 * Implementation overview for DHCP address detection
 *
 * The purpose of DHCP address detection is to relieve the user of having to
 * manually configure static IP addresses when ip-nospoof protection is turned
 * on. To achieve this, the mac layer needs to intercept DHCP packets to
 * determine the assigned IP addresses.
 *
 * A DHCP handshake between client and server typically requires at least
 * 4 messages:
 *
 * 1. DISCOVER - client attempts to locate DHCP servers via a
 *               broadcast message to its subnet.
 * 2. OFFER    - server responds to client with an IP address and
```



```
 *               other parameters.
 * 3. REQUEST  - client requests the offered address.
 * 4. ACK      - server verifies that the requested address matches
 *               the one it offered.
 *
 * DHCPv6 behaves pretty much the same way aside from different message names.
 *
 * Address information is embedded in either the OFFER or REQUEST message.
 * We chose to intercept REQUEST because this is at the last part of the
 * handshake and it indicates that the client intends to keep the address.
 * Intercepting OFFERs is unreliable because the client may receive multiple
 * offers from different servers, and we can't tell which address the client
 * will keep.
 *
 * Each DHCP message has a transaction ID. We use this transaction ID to match
 * REQUESTs with ACKs received from servers.
 *
 * For IPv4, the process to acquire a DHCP-assigned address is as follows:
 *
 * 1. Client sends REQUEST. a new dhcpv4_txn_t object is created and inserted
 *    in the the mci_v4_pending_txn table (keyed by xid). This object represents
 *    a new transaction. It contains the xid, the client ID and requested IP
 *    address.
 *
 * 2. Server responds with an ACK. The xid from this ACK is used to lookup the
 *    pending transaction from the mci_v4_pending_txn table. Once the object is
 *    found, it is removed from the pending table and inserted into the
 *    completed table (mci_v4_completed_txn, keyed by client ID) and the dynamic
 *    IP table (mci_v4_dyn_ip, keyed by IP address).
 *
 * 3. An outgoing packet that goes through the ip-nospoof path will be checked
 *    against the dynamic IP table. Packets that have the assigned DHCP address
 *    as the source IP address will pass the check and be admitted onto the
 *    network.
 *
 * IPv4 notes:
 *
 * If the server never responds with an ACK, there is a timer that is set after
 * the insertion of the transaction into the pending table. When the timer
 * fires, it will check whether the transaction is old (by comparing current
 * time and the txn's timestamp), if so the transaction will be freed. along
 * with this, any transaction in the completed/dyn-ip tables matching the client
 * ID of this stale transaction will also be freed. If the client fails to
 * extend a lease, we want to stop the client from using any IP addresses that
 * were granted previously.
 *
 * A RELEASE message from the client will not cause a transaction to be created.
 * The client ID in the RELEASE message will be used for finding and removing
 * transactions in the completed and dyn-ip tables.
 *
 *
 * For IPv6, the process to acquire a DHCPv6-assigned address is as follows:
 *
 * 1. Client sends REQUEST. The DUID is extracted and stored into a dhcpv6_cid_t
 *    structure. A new transaction structure (dhcpv6_txn_t) is also created and
 *    it will point to the dhcpv6_cid_t. If an existing transaction with a
 *    matching xid is not found, this dhcpv6_txn_t will be inserted into the
 *    mci_v6_pending_txn table (keyed by xid).
 *
 * 2. Server responds with a REPLY. If a pending transaction is found, the
 *    addresses in the reply will be placed into the dhcpv6_cid_t pointed to by
 *    the transaction. The dhcpv6_cid_t will then be moved to the mci_v6_cid
 *    table (keyed by cid). The associated addresses will be added to the
 *    mci_v6_dyn_ip table (while still being pointed to by the dhcpv6_cid_t).
 *
 * 3. IPv6 ip-nospoof will now check mci_v6_dyn_ip for matching packets.
```

```
 *    Packets with a source address matching one of the DHCPv6-assigned
 *    addresses will be allowed through.
 *
 * IPv6 notes:
 *
 * The v6 code shares the same timer as v4 for scrubbing stale transactions.
 * Just like v4, as part of removing an expired transaction, a RELEASE will be
 * be triggered on the cid associated with the expired transaction.
 *
 * The data structures used for v6 are slightly different because a v6 client
 * may have multiple addresses associated with it.
 */

/*
 * These are just arbitrary limits meant for preventing abuse (e.g. a user
 * flooding the network with bogus transactions). They are not meant to be
 * user-modifiable so they are not exposed as linkprops.
 */
static ulong_t  dhcp_max_pending_txn = 512;
static ulong_t  dhcp_max_completed_txn = 512;
static hrtime_t txn_cleanup_interval = 60 * NANOSEC;
static time_t   txn_cleanup_interval = 60;

/*
 * DHCPv4 transaction. It may be added to three different tables
 * (keyed by different fields).
 */
typedef struct dhcpv4_txn {
        uint32_t                dt_xid;
        hrtime_t                dt_timestamp;
        time_t                  dt_timestamp;
        uint8_t                 dt_cid[DHCP_MAX_OPT_SIZE];
        uint8_t                 dt_cid_len;
        ipaddr_t                dt_ipaddr;
        avl_node_t              dt_node;
        avl_node_t              dt_ipnode;
        struct dhcpv4_txn       *dt_next;
} dhcpv4_txn_t;
```

*unchanged_portion_omitted_*

```
/*
 * DHCPv6 transaction. Unlike its v4 counterpart, this object gets freed up
 * as soon as the transaction completes or expires.
 */
typedef struct dhcpv6_txn {
        uint32_t                dt_xid;
        hrtime_t                dt_timestamp;
        time_t                  dt_timestamp;
        dhcpv6_cid_t            *dt_cid;
        avl_node_t              dt_node;
        struct dhcpv6_txn       *dt_next;
} dhcpv6_txn_t;
```

*unchanged_portion_omitted_*

```
/*
 * Create/destroy a DHCPv4 transaction.
 */
static dhcpv4_txn_t *
create_dhcpv4_txn(uint32_t xid, uint8_t *cid, uint8_t cid_len, ipaddr_t ipaddr)
{
        dhcpv4_txn_t    *txn;

        if ((txn = kmem_zalloc(sizeof (*txn), KM_NOSLEEP)) == NULL)
                return (NULL);

        txn->dt_xid = xid;
```



```
        txn->dt_timestamp = gethrtime();
        txn->dt_timestamp = ddi_get_time();
        if (cid_len > 0)
                bcopy(cid, &txn->dt_cid, cid_len);
        txn->dt_cid_len = cid_len;
        txn->dt_ipaddr = ipaddr;
        return (txn);
}
```

*unchanged_portion_omitted_*

```
/*
 * Cleanup stale DHCPv4 transactions.
 */
static void
txn_cleanup_v4(mac_client_impl_t *mcip)
{
        dhcpv4_txn_t            *txn, *ctxn, *next, *txn_list = NULL;

        /*
         * Find stale pending transactions and place them on a list
         * to be removed.
         */
        for (txn = avl_first(&mcip->mci_v4_pending_txn); txn != NULL;
            txn = avl_walk(&mcip->mci_v4_pending_txn, txn, AVL_AFTER)) {
                if (gethrtime() - txn->dt_timestamp > txn_cleanup_interval) {
                if (ddi_get_time() - txn->dt_timestamp >
                    txn_cleanup_interval) {
                        DTRACE_PROBE2(found__expired__txn,
                            mac_client_impl_t *, mcip,
                            dhcpv4_txn_t *, txn);

                        txn->dt_next = txn_list;
                        txn_list = txn;
                }
        }

        /*
         * Remove and free stale pending transactions and completed
         * transactions with the same client IDs as the stale transactions.
         */
        for (txn = txn_list; txn != NULL; txn = next) {
                avl_remove(&mcip->mci_v4_pending_txn, txn);

                ctxn = find_dhcpv4_completed_txn(mcip, txn->dt_cid,
                    txn->dt_cid_len);
                if (ctxn != NULL) {
                        DTRACE_PROBE2(removing__completed__txn,
                            mac_client_impl_t *, mcip,
                            dhcpv4_txn_t *, ctxn);

                        remove_dhcpv4_completed_txn(mcip, ctxn);
                        free_dhcpv4_txn(ctxn);
                }
                next = txn->dt_next;
                txn->dt_next = NULL;

                DTRACE_PROBE2(freeing__txn, mac_client_impl_t *, mcip,
                    dhcpv4_txn_t *, txn);
                free_dhcpv4_txn(txn);
        }
}

/*
 * Core logic for intercepting outbound DHCPv4 packets.
 */
static boolean_t
```

```
559 intercept_dhcpv4_outbound(mac_client_impl_t *mcip, ipha_t *ipha, uchar_t *end)
560 {
561         struct dhcp             *dh4;
562         uchar_t                 *opt;
563         dhcpv4_txn_t            *txn, *ctxn;
564         ipaddr_t                ipaddr;
565         uint8_t                 opt_len, mtype, cid[DHCP_MAX_OPT_SIZE], cid_len;
566         mac_resource_props_t    *mrp = MCIP_RESOURCE_PROPS(mcip);

568         if (get_dhcpv4_info(ipha, end, &dh4) != 0)
569                 return (B_TRUE);

571         /* ip_nospoof/allowed-ips and DHCP are mutually exclusive by default */
572         if (allowed_ips_set(mrp, IPV4_VERSION))
573                 return (B_FALSE);

575         if (get_dhcpv4_option(dh4, end, CD_DHCP_TYPE, &opt, &opt_len) != 0 ||
576             opt_len != 1) {
577                 DTRACE_PROBE2(mtype__not__found, mac_client_impl_t *, mcip,
578                     struct dhcp *, dh4);
579                 return (B_TRUE);
580         }
581         mtype = *opt;
582         if (mtype != REQUEST && mtype != RELEASE) {
583                 DTRACE_PROBE3(ignored__mtype, mac_client_impl_t *, mcip,
584                     struct dhcp *, dh4, uint8_t, mtype);
585                 return (B_TRUE);
586         }

588         /* client ID is optional for IPv4 */
589         if (get_dhcpv4_option(dh4, end, CD_CLIENT_ID, &opt, &opt_len) == 0 &&
590             opt_len >= 2) {
591                 bcopy(opt, cid, opt_len);
592                 cid_len = opt_len;
593         } else {
594                 bzero(cid, DHCP_MAX_OPT_SIZE);
595                 cid_len = 0;
596         }

598         mutex_enter(&mcip->mci_protect_lock);
599         if (mtype == RELEASE) {
600                 DTRACE_PROBE2(release, mac_client_impl_t *, mcip,
601                     struct dhcp *, dh4);

603                 /* flush any completed txn with this cid */
604                 ctxn = find_dhcpv4_completed_txn(mcip, cid, cid_len);
605                 if (ctxn != NULL) {
606                         DTRACE_PROBE2(release__successful, mac_client_impl_t *,
607                             mcip, struct dhcp *, dh4);

609                         remove_dhcpv4_completed_txn(mcip, ctxn);
610                         free_dhcpv4_txn(ctxn);
611                 }
612                 goto done;
613         }

615         /*
616          * If a pending txn already exists, we'll update its timestamp so
617          * it won't get flushed by the timer. We don't need to create new
618          * txns for retransmissions.
619          */
620         if ((txn = find_dhcpv4_pending_txn(mcip, dh4->xid)) != NULL) {
621                 DTRACE_PROBE2(update, mac_client_impl_t *, mcip,
622                     dhcpv4_txn_t *, txn);
623                 txn->dt_timestamp = gethrtime();
501                 txn->dt_timestamp = ddi_get_time();
```



```
624                 goto done;
625         }

627         if (get_dhcpv4_option(dh4, end, CD_REQUESTED_IP_ADDR,
628             &opt, &opt_len) != 0 || opt_len != sizeof (ipaddr)) {
629                 DTRACE_PROBE2(ipaddr__not__found, mac_client_impl_t *, mcip,
630                     struct dhcp *, dh4);
631                 goto done;
632         }
633         bcopy(opt, &ipaddr, sizeof (ipaddr));
634         if ((txn = create_dhcpv4_txn(dh4->xid, cid, cid_len, ipaddr)) == NULL)
635                 goto done;

637         if (insert_dhcpv4_pending_txn(mcip, txn) != 0) {
638                 DTRACE_PROBE2(insert__failed, mac_client_impl_t *, mcip,
639                     dhcpv4_txn_t *, txn);
640                 free_dhcpv4_txn(txn);
641                 goto done;
642         }
643         start_txn_cleanup_timer(mcip);

645         DTRACE_PROBE2(txn__pending, mac_client_impl_t *, mcip,
646             dhcpv4_txn_t *, txn);

648 done:
649         mutex_exit(&mcip->mci_protect_lock);
650         return (B_TRUE);
651 }
______unchanged_portion_omitted_

1112 static dhcpv6_txn_t *
1113 create_dhcpv6_txn(uint32_t xid, dhcpv6_cid_t *cid)
1114 {
1115         dhcpv6_txn_t    *txn;

1117         if ((txn = kmem_zalloc(sizeof (dhcpv6_txn_t), KM_NOSLEEP)) == NULL)
1118                 return (NULL);

1120         txn->dt_xid = xid;
1121         txn->dt_cid = cid;
1122         txn->dt_timestamp = gethrtime();
1000         txn->dt_timestamp = ddi_get_time();
1123         return (txn);
1124 }
______unchanged_portion_omitted_

1175 /*
1176  * Cleanup stale DHCPv6 transactions.
1177  */
1178 static void
1179 txn_cleanup_v6(mac_client_impl_t *mcip)
1180 {
1181         dhcpv6_txn_t            *txn, *next, *txn_list = NULL;

1183         /*
1184          * Find stale pending transactions and place them on a list
1185          * to be removed.
1186          */
1187         for (txn = avl_first(&mcip->mci_v6_pending_txn); txn != NULL;
1188             txn = avl_walk(&mcip->mci_v6_pending_txn, txn, AVL_AFTER)) {
1189                 if (gethrtime() - txn->dt_timestamp > txn_cleanup_interval) {
1067                 if (ddi_get_time() - txn->dt_timestamp >
1068                     txn_cleanup_interval) {
1190                         DTRACE_PROBE2(found__expired__txn,
1191                             mac_client_impl_t *, mcip,
1192                             dhcpv6_txn_t *, txn);
```

```
1194                        txn->dt_next = txn_list;
1195                        txn_list = txn;
1196                }
1197        }

1199        /*
1200         * Remove and free stale pending transactions.
1201         * Release any existing cids matching the stale transactions.
1202         */
1203        for (txn = txn_list; txn != NULL; txn = next) {
1204                avl_remove(&mcip->mci_v6_pending_txn, txn);
1205                release_dhcpv6_cid(mcip, txn->dt_cid);
1206                next = txn->dt_next;
1207                txn->dt_next = NULL;

1209                DTRACE_PROBE2(freeing__txn, mac_client_impl_t *, mcip,
1210                    dhcpv6_txn_t *, txn);
1211                free_dhcpv6_txn(txn);
1212        }

1214 }

1216 /*
1217  * Core logic for intercepting outbound DHCPv6 packets.
1218  */
1219 static boolean_t
1220 intercept_dhcpv6_outbound(mac_client_impl_t *mcip, ip6_t *ip6h, uchar_t *end)
1221 {
1222        dhcpv6_message_t        *dh6;
1223        dhcpv6_txn_t            *txn;
1224        dhcpv6_cid_t            *cid = NULL;
1225        uint32_t                xid;
1226        uint8_t                 mtype;
1227        mac_resource_props_t *mrp = MCIP_RESOURCE_PROPS(mcip);

1229        if (get_dhcpv6_info(ip6h, end, &dh6) != 0)
1230                return (B_TRUE);

1232        /* ip_nospoof/allowed-ips and DHCP are mutually exclusive by default */
1233        if (allowed_ips_set(mrp, IPV6_VERSION))
1234                return (B_FALSE);

1236        mtype = dh6->d6m_msg_type;
1237        if (mtype != DHCPV6_MSG_REQUEST && mtype != DHCPV6_MSG_RENEW &&
1238            mtype != DHCPV6_MSG_REBIND && mtype != DHCPV6_MSG_RELEASE)
1239                return (B_TRUE);

1241        if ((cid = create_dhcpv6_cid(dh6, end)) == NULL)
1242                return (B_TRUE);

1244        mutex_enter(&mcip->mci_protect_lock);
1245        if (mtype == DHCPV6_MSG_RELEASE) {
1246                release_dhcpv6_cid(mcip, cid);
1247                goto done;
1248        }
1249        xid = DHCPV6_GET_TRANSID(dh6);
1250        if ((txn = find_dhcpv6_pending_txn(mcip, xid)) != NULL) {
1251                DTRACE_PROBE2(update, mac_client_impl_t *, mcip,
1252                    dhcpv6_txn_t *, txn);
1253                txn->dt_timestamp = gethrtime();
1132                txn->dt_timestamp = ddi_get_time();
1254                goto done;
1255        }
1256        if ((txn = create_dhcpv6_txn(xid, cid)) == NULL)
1257                goto done;
```



```
1259        cid = NULL;
1260        if (insert_dhcpv6_pending_txn(mcip, txn) != 0) {
1261                DTRACE_PROBE2(insert__failed, mac_client_impl_t *, mcip,
1262                    dhcpv6_txn_t *, txn);
1263                free_dhcpv6_txn(txn);
1264                goto done;
1265        }
1266        start_txn_cleanup_timer(mcip);

1268        DTRACE_PROBE2(txn__pending, mac_client_impl_t *, mcip,
1269            dhcpv6_txn_t *, txn);

1271 done:
1272        if (cid != NULL)
1273                free_dhcpv6_cid(cid);

1275        mutex_exit(&mcip->mci_protect_lock);
1276        return (B_TRUE);
1277 }
```

_____unchanged_portion_omitted_

```
1335 /*
1336  * Timer for cleaning up stale transactions.
1337  */
1338 static void
1339 txn_cleanup_timer(void *arg)
1340 {
1341        mac_client_impl_t       *mcip = arg;

1343        mutex_enter(&mcip->mci_protect_lock);
1344        if (mcip->mci_txn_cleanup_tid == 0) {
1345                /* do nothing if timer got cancelled */
1346                mutex_exit(&mcip->mci_protect_lock);
1347                return;
1348        }
1349        mcip->mci_txn_cleanup_tid = 0;

1351        txn_cleanup_v4(mcip);
1352        txn_cleanup_v6(mcip);

1354        /*
1355         * Restart timer if pending transactions still exist.
1356         */
1357        if (!avl_is_empty(&mcip->mci_v4_pending_txn) ||
1358            !avl_is_empty(&mcip->mci_v6_pending_txn)) {
1359                DTRACE_PROBE1(restarting__timer, mac_client_impl_t *, mcip);

1361                mcip->mci_txn_cleanup_tid = timeout(txn_cleanup_timer, mcip,
1362                    drv_usectohz(txn_cleanup_interval / (NANOSEC / MICROSEC)));
1241                    drv_usectohz(txn_cleanup_interval * 1000000));
1363        }
1364        mutex_exit(&mcip->mci_protect_lock);
1365 }

1367 static void
1368 start_txn_cleanup_timer(mac_client_impl_t *mcip)
1369 {
1370        ASSERT(MUTEX_HELD(&mcip->mci_protect_lock));
1371        if (mcip->mci_txn_cleanup_tid == 0) {
1372                mcip->mci_txn_cleanup_tid = timeout(txn_cleanup_timer, mcip,
1373                    drv_usectohz(txn_cleanup_interval / (NANOSEC / MICROSEC)));
1252                    drv_usectohz(txn_cleanup_interval * 1000000));
1374        }
1375 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  246182 Mon May  5 14:29:44 2014
new/usr/src/uts/common/io/scsi/adapters/scsi_vhci/scsi_vhci.c
4779 vhci shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2001, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

/*
 * Multiplexed I/O SCSI vHCI implementation
 */

#include <sys/conf.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/scsi/scsi.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/scsi/impl/services.h>
#include <sys/sunmdi.h>
#include <sys/mdi_impldefs.h>
#include <sys/scsi/adapters/scsi_vhci.h>
#include <sys/disp.h>
#include <sys/byteorder.h>

extern uintptr_t scsi_callback_id;
extern ddi_dma_attr_t scsi_alloc_attr;

#ifdef  DEBUG
int     vhci_debug = VHCI_DEBUG_DEFAULT_VAL;
#endif

/* retry for the vhci_do_prout command when a not ready is returned */
int vhci_prout_not_ready_retry = 180;

/*
 * These values are defined to support the internal retry of
 * SCSI packets for better sense code handling.
 */
#define VHCI_CMD_CMPLT  0
```



```
#define VHCI_CMD_RETRY  1
#define VHCI_CMD_ERROR  -1

#define PROPFLAGS (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM)
#define VHCI_SCSI_PERR          0x47
#define VHCI_PGR_ILLEGALOP      -2
#define VHCI_NUM_UPDATE_TASKQ   8
/* changed to 132 to accomodate HDS */

/*
 * Version Macros
 */
#define VHCI_NAME_VERSION       "SCSI VHCI Driver"
char            vhci_version_name[] = VHCI_NAME_VERSION;

int             vhci_first_time = 0;
clock_t         vhci_to_ticks = 0;
int             vhci_init_wait_timeout = VHCI_INIT_WAIT_TIMEOUT;
kcondvar_t      vhci_cv;
kmutex_t        vhci_global_mutex;
void            *vhci_softstate = NULL; /* for soft state */

/*
 * Flag to delay the retry of the reserve command
 */
int             vhci_reserve_delay = 100000;
static int      vhci_path_quiesce_timeout = 60;
static uchar_t  zero_key[MHIOC_RESV_KEY_SIZE];

/* uscsi delay for a TRAN_BUSY */
static int vhci_uscsi_delay = 100000;
static int vhci_uscsi_retry_count = 180;
/* uscsi_restart_sense timeout id in case it needs to get canceled */
static timeout_id_t vhci_restart_timeid = 0;

static int      vhci_bus_config_debug = 0;

/*
 * Bidirectional map of 'target-port' to port id <pid> for support of
 * iostat(1M) '-Xx' and '-Yx' output.
 */
static kmutex_t         vhci_targetmap_mutex;
static uint_t           vhci_targetmap_pid = 1;
static mod_hash_t       *vhci_targetmap_bypid;  /* <pid> -> 'target-port' */
static mod_hash_t       *vhci_targetmap_byport; /* 'target-port' -> <pid> */

/*
 * functions exported by scsi_vhci struct cb_ops
 */
static int vhci_open(dev_t *, int, int, cred_t *);
static int vhci_close(dev_t, int, int, cred_t *);
static int vhci_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

/*
 * functions exported by scsi_vhci struct dev_ops
 */
static int vhci_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int vhci_attach(dev_info_t *, ddi_attach_cmd_t);
static int vhci_detach(dev_info_t *, ddi_detach_cmd_t);

/*
 * functions exported by scsi_vhci scsi_hba_tran_t transport table
 */
static int vhci_scsi_tgt_init(dev_info_t *, dev_info_t *,
    scsi_hba_tran_t *, struct scsi_device *);
static void vhci_scsi_tgt_free(dev_info_t *, dev_info_t *, scsi_hba_tran_t *,
```

```
127     struct scsi_device *);
128 static int vhci_pgr_register_start(scsi_vhci_lun_t *, struct scsi_pkt *);
129 static int vhci_scsi_start(struct scsi_address *, struct scsi_pkt *);
130 static int vhci_scsi_abort(struct scsi_address *, struct scsi_pkt *);
131 static int vhci_scsi_reset(struct scsi_address *, int);
132 static int vhci_scsi_reset_target(struct scsi_address *, int level,
133     uint8_t select_path);
134 static int vhci_scsi_reset_bus(struct scsi_address *);
135 static int vhci_scsi_getcap(struct scsi_address *, char *, int);
136 static int vhci_scsi_setcap(struct scsi_address *, char *, int, int);
137 static int vhci_commoncap(struct scsi_address *, char *, int, int, int);
138 static int vhci_pHCI_cap(struct scsi_address *ap, char *cap, int val, int whom,
139     mdi_pathinfo_t *pip);
140 static struct scsi_pkt *vhci_scsi_init_pkt(struct scsi_address *,
141     struct scsi_pkt *, struct buf *, int, int, int, int, int (*)(), caddr_t);
142 static void vhci_scsi_destroy_pkt(struct scsi_address *, struct scsi_pkt *);
143 static void vhci_scsi_dmafree(struct scsi_address *, struct scsi_pkt *);
144 static void vhci_scsi_sync_pkt(struct scsi_address *, struct scsi_pkt *);
145 static int vhci_scsi_reset_notify(struct scsi_address *, int, void (*)(caddr_t),
146     caddr_t);
147 static int vhci_scsi_get_bus_addr(struct scsi_device *, char *, int);
148 static int vhci_scsi_get_name(struct scsi_device *, char *, int);
149 static int vhci_scsi_bus_power(dev_info_t *, void *, pm_bus_power_op_t,
150     void *, void *);
151 static int vhci_scsi_bus_config(dev_info_t *, uint_t, ddi_bus_config_op_t,
152     void *, dev_info_t **);
153 static int vhci_scsi_bus_unconfig(dev_info_t *, uint_t, ddi_bus_config_op_t,
154     void *);
155 static struct scsi_failover_ops *vhci_dev_fo(dev_info_t *, struct scsi_device *,
156     void **, char **);

158 /*
159  * functions registered with the mpxio framework via mdi_vhci_ops_t
160  */
161 static int vhci_pathinfo_init(dev_info_t *, mdi_pathinfo_t *, int);
162 static int vhci_pathinfo_uninit(dev_info_t *, mdi_pathinfo_t *, int);
163 static int vhci_pathinfo_state_change(dev_info_t *, mdi_pathinfo_t *,
164                 mdi_pathinfo_state_t, uint32_t, int);
165 static int vhci_pathinfo_online(dev_info_t *, mdi_pathinfo_t *, int);
166 static int vhci_pathinfo_offline(dev_info_t *, mdi_pathinfo_t *, int);
167 static int vhci_failover(dev_info_t *, dev_info_t *, int);
168 static void vhci_client_attached(dev_info_t *);
169 static int vhci_is_dev_supported(dev_info_t *, dev_info_t *, void *);

171 static int vhci_ctl(dev_t, int, intptr_t, int, cred_t *, int *);
172 static int vhci_devctl(dev_t, int, intptr_t, int, cred_t *, int *);
173 static int vhci_ioc_get_phci_path(sv_iocdata_t *, caddr_t, int, caddr_t);
174 static int vhci_ioc_get_client_path(sv_iocdata_t *, caddr_t, int, caddr_t);
175 static int vhci_ioc_get_paddr(sv_iocdata_t *, caddr_t, int, caddr_t);
176 static int vhci_ioc_send_client_path(caddr_t, sv_iocdata_t *, int, caddr_t);
177 static void vhci_ioc_devi_to_path(dev_info_t *, caddr_t);
178 static int vhci_get_phci_path_list(dev_info_t *, sv_path_info_t *, uint_t);
179 static int vhci_get_client_path_list(dev_info_t *, sv_path_info_t *, uint_t);
180 static int vhci_get_iocdata(const void *, sv_iocdata_t *, int, caddr_t);
181 static int vhci_get_iocswitchdata(const void *, sv_switch_to_cntlr_iocdata_t *,
182     int, caddr_t);
183 static int vhci_ioc_alloc_pathinfo(sv_path_info_t **, sv_path_info_t **,
184     uint_t, sv_iocdata_t *, int, caddr_t);
185 static void vhci_ioc_free_pathinfo(sv_path_info_t *, sv_path_info_t *, uint_t);
186 static int vhci_ioc_send_pathinfo(sv_path_info_t *, sv_path_info_t *, uint_t,
187     sv_iocdata_t *, int, caddr_t);
188 static int vhci_handle_ext_fo(struct scsi_pkt *, int);
189 static int vhci_efo_watch_cb(caddr_t, struct scsi_watch_result *);
190 static int vhci_quiesce_lun(struct scsi_vhci_lun *);
191 static int vhci_pgr_validate_and_register(scsi_vhci_priv_t *);
192 static void vhci_dispatch_scsi_start(void *);
```



```
193 static void vhci_efo_done(void *);
194 static void vhci_initiate_auto_failback(void *);
195 static void vhci_update_pHCI_pkt(struct vhci_pkt *, struct scsi_pkt *);
196 static int vhci_update_pathinfo(struct scsi_device *, mdi_pathinfo_t *,
197     struct scsi_failover_ops *, scsi_vhci_lun_t *, struct scsi_vhci *);
198 static void vhci_kstat_create_pathinfo(mdi_pathinfo_t *);
199 static int vhci_quiesce_paths(dev_info_t *, dev_info_t *,
200     scsi_vhci_lun_t *, char *, char *);

202 static char *vhci_devnm_to_guid(char *);
203 static int vhci_bind_transport(struct scsi_address *, struct vhci_pkt *,
204     int, int (*func)(caddr_t));
205 static void vhci_intr(struct scsi_pkt *);
206 static int vhci_do_prout(scsi_vhci_priv_t *);
207 static void vhci_run_cmd(void *);
208 static int vhci_do_prin(struct vhci_pkt **);
209 static struct scsi_pkt *vhci_create_retry_pkt(struct vhci_pkt *);
210 static struct vhci_pkt *vhci_sync_retry_pkt(struct vhci_pkt *);
211 static struct scsi_vhci_lun *vhci_lun_lookup(dev_info_t *);
212 static struct scsi_vhci_lun *vhci_lun_lookup_alloc(dev_info_t *, char *, int *);
213 static void vhci_lun_free(struct scsi_vhci_lun *dvlp, struct scsi_device *sd);
214 static int vhci_recovery_reset(scsi_vhci_lun_t *, struct scsi_address *,
215     uint8_t, uint8_t);
216 void vhci_update_pathstates(void *);

218 #ifdef DEBUG
219 static void vhci_print_prin_keys(vhci_prin_readkeys_t *, int);
220 static void vhci_print_cdb(dev_info_t *dip, uint_t level,
221     char *title, uchar_t *cdb);
222 static void vhci_clean_print(dev_info_t *dev, uint_t level,
223     char *title, uchar_t *data, int len);
224 #endif
225 static void vhci_print_prout_keys(scsi_vhci_lun_t *, char *);
226 static void vhci_uscsi_iodone(struct scsi_pkt *pkt);
227 static void vhci_invalidate_mpapi_lu(struct scsi_vhci *, scsi_vhci_lun_t *);

229 /*
230  * MP-API related functions
231  */
232 extern int vhci_mpapi_init(struct scsi_vhci *);
233 extern void vhci_mpapi_add_dev_prod(struct scsi_vhci *, char *);
234 extern int vhci_mpapi_ctl(dev_t, int, intptr_t, int, cred_t *, int *);
235 extern void vhci_update_mpapi_data(struct scsi_vhci *,
236     scsi_vhci_lun_t *, mdi_pathinfo_t *);
237 extern void* vhci_get_mpapi_item(struct scsi_vhci *, mpapi_list_header_t *,
238     uint8_t, void*);
239 extern void vhci_mpapi_set_path_state(dev_info_t *, mdi_pathinfo_t *, int);
240 extern int vhci_mpapi_update_tpg_acc_state_for_lu(struct scsi_vhci *,
241     scsi_vhci_lun_t *);

243 #define VHCI_DMA_MAX_XFER_CAP   INT_MAX

245 #define VHCI_MAX_PGR_RETRIES    3

247 /*
248  * Macros for the device-type mpxio options
249  */
250 #define LOAD_BALANCE_OPTIONS            "load-balance-options"
251 #define LOGICAL_BLOCK_REGION_SIZE       "region-size"
252 #define MPXIO_OPTIONS_LIST              "device-type-mpxio-options-list"
253 #define DEVICE_TYPE_STR                 "device-type"
254 #define isdigit(ch)                     ((ch) >= '0' && (ch) <= '9')

256 static struct cb_ops vhci_cb_ops = {
257         vhci_open,                      /* open */
258         vhci_close,                     /* close */
```

```
259         nodev,                          /* strategy */
260         nodev,                          /* print */
261         nodev,                          /* dump */
262         nodev,                          /* read */
263         nodev,                          /* write */
264         vhci_ioctl,                     /* ioctl */
265         nodev,                          /* devmap */
266         nodev,                          /* mmap */
267         nodev,                          /* segmap */
268         nochpoll,                       /* chpoll */
269         ddi_prop_op,                    /* cb_prop_op */
270         0,                              /* streamtab */
271         D_NEW | D_MP,                   /* cb_flag */
272         CB_REV,                         /* rev */
273         nodev,                          /* aread */
274         nodev                           /* awrite */
275 };

277 static struct dev_ops vhci_ops = {
278         DEVO_REV,
279         0,
280         vhci_getinfo,
281         nulldev,                /* identify */
282         nulldev,                /* probe */
283         vhci_attach,            /* attach and detach are mandatory */
284         vhci_detach,
285         nodev,                  /* reset */
286         &vhci_cb_ops,           /* cb_ops */
287         NULL,                   /* bus_ops */
288         NULL,                   /* power */
289         ddi_quiesce_not_needed, /* quiesce */
290 };

292 extern struct mod_ops mod_driverops;

294 static struct modldrv modldrv = {
295         &mod_driverops,
296         vhci_version_name,      /* module name */
297         &vhci_ops
298 };

300 static struct modlinkage modlinkage = {
301         MODREV_1,
302         &modldrv,
303         NULL
304 };

306 static mdi_vhci_ops_t vhci_opinfo = {
307         MDI_VHCI_OPS_REV,
308         vhci_pathinfo_init,             /* Pathinfo node init callback */
309         vhci_pathinfo_uninit,           /* Pathinfo uninit callback */
310         vhci_pathinfo_state_change,     /* Pathinfo node state change */
311         vhci_failover,                  /* failover callback */
312         vhci_client_attached,           /* client attached callback     */
313         vhci_is_dev_supported           /* is device supported by mdi */
314 };

316 /*
317  * The scsi_failover table defines an ordered set of 'fops' modules supported
318  * by scsi_vhci.  Currently, initialize this table from the 'ddi-forceload'
319  * property specified in scsi_vhci.conf.
320  */
321 static struct scsi_failover {
322         ddi_modhandle_t                 sf_mod;
323         struct scsi_failover_ops        *sf_sfo;
324 } *scsi_failover_table;
```



```
325 static uint_t   scsi_nfailover;

327 int
328 _init(void)
329 {
330         int     rval;

332         /*
333          * Allocate soft state and prepare to do ddi_soft_state_zalloc()
334          * before registering with the transport first.
335          */
336         if ((rval = ddi_soft_state_init(&vhci_softstate,
337             sizeof (struct scsi_vhci), 1)) != 0) {
338                 VHCI_DEBUG(1, (CE_NOTE, NULL,
339                     "!_init:soft state init failed\n"));
340                 return (rval);
341         }

343         if ((rval = scsi_hba_init(&modlinkage)) != 0) {
344                 VHCI_DEBUG(1, (CE_NOTE, NULL,
345                     "!_init: scsi hba init failed\n"));
346                 ddi_soft_state_fini(&vhci_softstate);
347                 return (rval);
348         }

350         mutex_init(&vhci_global_mutex, NULL, MUTEX_DRIVER, NULL);
351         cv_init(&vhci_cv, NULL, CV_DRIVER, NULL);

353         mutex_init(&vhci_targetmap_mutex, NULL, MUTEX_DRIVER, NULL);
354         vhci_targetmap_byport = mod_hash_create_strhash(
355             "vhci_targetmap_byport", 256, mod_hash_null_valdtor);
356         vhci_targetmap_bypid = mod_hash_create_idhash(
357             "vhci_targetmap_bypid", 256, mod_hash_null_valdtor);

359         if ((rval = mod_install(&modlinkage)) != 0) {
360                 VHCI_DEBUG(1, (CE_NOTE, NULL, "!_init: mod_install failed\n"));
361                 if (vhci_targetmap_bypid)
362                         mod_hash_destroy_idhash(vhci_targetmap_bypid);
363                 if (vhci_targetmap_byport)
364                         mod_hash_destroy_strhash(vhci_targetmap_byport);
365                 mutex_destroy(&vhci_targetmap_mutex);
366                 cv_destroy(&vhci_cv);
367                 mutex_destroy(&vhci_global_mutex);
368                 scsi_hba_fini(&modlinkage);
369                 ddi_soft_state_fini(&vhci_softstate);
370         }
371         return (rval);
372 }


375 /*
376  * the system is done with us as a driver, so clean up
377  */
378 int
379 _fini(void)
380 {
381         int rval;

383         /*
384          * don't start cleaning up until we know that the module remove
385          * has worked  -- if this works, then we know that each instance
386          * has successfully been DDI_DETACHed
387          */
388         if ((rval = mod_remove(&modlinkage)) != 0) {
389                 VHCI_DEBUG(4, (CE_NOTE, NULL, "!_fini: mod_remove failed\n"));
390                 return (rval);
```

```
391         }
393         if (vhci_targetmap_bypid)
394                 mod_hash_destroy_idhash(vhci_targetmap_bypid);
395         if (vhci_targetmap_byport)
396                 mod_hash_destroy_strhash(vhci_targetmap_byport);
397         mutex_destroy(&vhci_targetmap_mutex);
398         cv_destroy(&vhci_cv);
399         mutex_destroy(&vhci_global_mutex);
400         scsi_hba_fini(&modlinkage);
401         ddi_soft_state_fini(&vhci_softstate);
403         return (rval);
404 }
406 int
407 _info(struct modinfo *modinfop)
408 {
409         return (mod_info(&modlinkage, modinfop));
410 }
412 /*
413  * Lookup scsi_failover by "short name" of failover module.
414  */
415 struct scsi_failover_ops *
416 vhci_failover_ops_by_name(char *name)
417 {
418         struct scsi_failover    *sf;
420         for (sf = scsi_failover_table; sf->sf_mod; sf++) {
421                 if (sf->sf_sfo == NULL)
422                         continue;
423                 if (strcmp(sf->sf_sfo->sfo_name, name) == 0)
424                         return (sf->sf_sfo);
425         }
426         return (NULL);
427 }
429 /*
430  * Load all scsi_failover_ops 'fops' modules.
431  */
432 static void
433 vhci_failover_modopen(struct scsi_vhci *vhci)
434 {
435         char                    **module;
436         int                     i;
437         struct scsi_failover    *sf;
438         char                    **dt;
439         int                     e;
441         if (scsi_failover_table)
442                 return;
444         /* Get the list of modules from scsi_vhci.conf */
445         if (ddi_prop_lookup_string_array(DDI_DEV_T_ANY,
446             vhci->vhci_dip, DDI_PROP_DONTPASS, "ddi-forceload",
447             &module, &scsi_nfailover) != DDI_PROP_SUCCESS) {
448                 cmn_err(CE_WARN, "scsi_vhci: "
449                     "scsi_vhci.conf is missing 'ddi-forceload'");
450                 return;
451         }
452         if (scsi_nfailover == 0) {
453                 cmn_err(CE_WARN, "scsi_vhci: "
454                     "scsi_vhci.conf has empty 'ddi-forceload'");
455                 ddi_prop_free(module);
456                 return;
```



```
457         }
459         /* allocate failover table based on number of modules */
460         scsi_failover_table = (struct scsi_failover *)
461             kmem_zalloc(sizeof (struct scsi_failover) * (scsi_nfailover + 1),
462             KM_SLEEP);
464         /* loop over modules specified in scsi_vhci.conf and open each module */
465         for (i = 0, sf = scsi_failover_table; i < scsi_nfailover; i++) {
466                 if (module[i] == NULL)
467                         continue;
469                 sf->sf_mod = ddi_modopen(module[i], KRTLD_MODE_FIRST, &e);
470                 if (sf->sf_mod == NULL) {
471                         /*
472                          * A module returns EEXIST if other software is
473                          * supporting the intended function: for example
474                          * the scsi_vhci_f_sum_emc module returns EEXIST
475                          * from _init if EMC powerpath software is installed.
476                          */
477                         if (e != EEXIST)
478                                 cmn_err(CE_WARN, "scsi_vhci: unable to open "
479                                     "module '%s', error %d", module[i], e);
480                         continue;
481                 }
482                 sf->sf_sfo = ddi_modsym(sf->sf_mod,
483                     "scsi_vhci_failover_ops", &e);
484                 if (sf->sf_sfo == NULL) {
485                         cmn_err(CE_WARN, "scsi_vhci: "
486                             "unable to import 'scsi_failover_ops' from '%s', "
487                             "error %d", module[i], e);
488                         (void) ddi_modclose(sf->sf_mod);
489                         sf->sf_mod = NULL;
490                         continue;
491                 }
493                 /* register vid/pid of devices supported with mpapi */
494                 for (dt = sf->sf_sfo->sfo_devices; *dt; dt++)
495                         vhci_mpapi_add_dev_prod(vhci, *dt);
496                 sf++;
497         }
499         /* verify that at least the "well-known" modules were there */
500         if (vhci_failover_ops_by_name(SFO_NAME_SYM) == NULL)
501                 cmn_err(CE_WARN, "scsi_vhci: well-known module \""
502                     SFO_NAME_SYM "\" not defined in scsi_vhci.conf's "
503                     "'ddi-forceload'");
504         if (vhci_failover_ops_by_name(SFO_NAME_TPGS) == NULL)
505                 cmn_err(CE_WARN, "scsi_vhci: well-known module \""
506                     SFO_NAME_TPGS "\" not defined in scsi_vhci.conf's "
507                     "'ddi-forceload'");
509         /* call sfo_init for modules that need it */
510         for (sf = scsi_failover_table; sf->sf_mod; sf++) {
511                 if (sf->sf_sfo && sf->sf_sfo->sfo_init)
512                         sf->sf_sfo->sfo_init();
513         }
515         ddi_prop_free(module);
516 }
518 /*
519  * unload all loaded scsi_failover_ops modules
520  */
521 static void
522 vhci_failover_modclose()
```

```
{
        struct scsi_failover    *sf;

        for (sf = scsi_failover_table; sf->sf_mod; sf++) {
                if ((sf->sf_mod == NULL) || (sf->sf_sfo == NULL))
                        continue;
                (void) ddi_modclose(sf->sf_mod);
                sf->sf_mod = NULL;
                sf->sf_sfo = NULL;
        }

        if (scsi_failover_table && scsi_nfailover)
                kmem_free(scsi_failover_table,
                    sizeof (struct scsi_failover) * (scsi_nfailover + 1));
        scsi_failover_table = NULL;
        scsi_nfailover = 0;
}

/* ARGSUSED */
static int
vhci_open(dev_t *devp, int flag, int otype, cred_t *credp)
{
        struct scsi_vhci        *vhci;

        if (otype != OTYP_CHR) {
                return (EINVAL);
        }

        vhci = ddi_get_soft_state(vhci_softstate, MINOR2INST(getminor(*devp)));
        if (vhci == NULL) {
                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_open: failed ENXIO\n"));
                return (ENXIO);
        }

        mutex_enter(&vhci->vhci_mutex);
        if ((flag & FEXCL) && (vhci->vhci_state & VHCI_STATE_OPEN)) {
                mutex_exit(&vhci->vhci_mutex);
                vhci_log(CE_NOTE, vhci->vhci_dip,
                    "!vhci%d: Already open\n", getminor(*devp));
                return (EBUSY);
        }

        vhci->vhci_state |= VHCI_STATE_OPEN;
        mutex_exit(&vhci->vhci_mutex);
        return (0);
}


/* ARGSUSED */
static int
vhci_close(dev_t dev, int flag, int otype, cred_t *credp)
{
        struct scsi_vhci        *vhci;

        if (otype != OTYP_CHR) {
                return (EINVAL);
        }

        vhci = ddi_get_soft_state(vhci_softstate, MINOR2INST(getminor(dev)));
        if (vhci == NULL) {
                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_close: failed ENXIO\n"));
                return (ENXIO);
        }

        mutex_enter(&vhci->vhci_mutex);
        vhci->vhci_state &= ~VHCI_STATE_OPEN;
```



```
        mutex_exit(&vhci->vhci_mutex);

        return (0);
}

/* ARGSUSED */
static int
vhci_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
        cred_t *credp, int *rval)
{
        if (IS_DEVCTL(cmd)) {
                return (vhci_devctl(dev, cmd, data, mode, credp, rval));
        } else if (cmd == MP_CMD) {
                return (vhci_mpapi_ctl(dev, cmd, data, mode, credp, rval));
        } else {
                return (vhci_ctl(dev, cmd, data, mode, credp, rval));
        }
}

/*
 * attach the module
 */
static int
vhci_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int                     rval = DDI_FAILURE;
        int                     scsi_hba_attached = 0;
        int                     vhci_attached = 0;
        int                     mutex_initted = 0;
        int                     instance;
        struct scsi_vhci        *vhci;
        scsi_hba_tran_t         *tran;
        char                    cache_name_buf[64];
        char                    *data;

        VHCI_DEBUG(4, (CE_NOTE, NULL, "vhci_attach: cmd=0x%x\n", cmd));

        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_ATTACH:
                break;

        case DDI_RESUME:
        case DDI_PM_RESUME:
                VHCI_DEBUG(1, (CE_NOTE, NULL, "!vhci_attach: resume not yet"
                    "implemented\n"));
                return (rval);

        default:
                VHCI_DEBUG(1, (CE_NOTE, NULL,
                    "!vhci_attach: unknown ddi command\n"));
                return (rval);
        }

        /*
         * Allocate vhci data structure.
         */
        if (ddi_soft_state_zalloc(vhci_softstate, instance) != DDI_SUCCESS) {
                VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
                    "soft state alloc failed\n"));
                return (DDI_FAILURE);
        }

        if ((vhci = ddi_get_soft_state(vhci_softstate, instance)) == NULL) {
                VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
```

```
655                     "bad soft state\n"));
656                 ddi_soft_state_free(vhci_softstate, instance);
657                 return (DDI_FAILURE);
658         }
659
660         /* Allocate packet cache */
661         (void) snprintf(cache_name_buf, sizeof (cache_name_buf),
662             "vhci%d_cache", instance);
663
664         mutex_init(&vhci->vhci_mutex, NULL, MUTEX_DRIVER, NULL);
665         mutex_initted++;
666
667         /*
668          * Allocate a transport structure
669          */
670         tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);
671         ASSERT(tran != NULL);
672
673         vhci->vhci_tran         = tran;
674         vhci->vhci_dip          = dip;
675         vhci->vhci_instance     = instance;
676
677         tran->tran_hba_private  = vhci;
678         tran->tran_tgt_init     = vhci_scsi_tgt_init;
679         tran->tran_tgt_probe    = NULL;
680         tran->tran_tgt_free     = vhci_scsi_tgt_free;
681
682         tran->tran_start        = vhci_scsi_start;
683         tran->tran_abort        = vhci_scsi_abort;
684         tran->tran_reset        = vhci_scsi_reset;
685         tran->tran_getcap       = vhci_scsi_getcap;
686         tran->tran_setcap       = vhci_scsi_setcap;
687         tran->tran_init_pkt     = vhci_scsi_init_pkt;
688         tran->tran_destroy_pkt  = vhci_scsi_destroy_pkt;
689         tran->tran_dmafree      = vhci_scsi_dmafree;
690         tran->tran_sync_pkt     = vhci_scsi_sync_pkt;
691         tran->tran_reset_notify = vhci_scsi_reset_notify;
692
693         tran->tran_get_bus_addr = vhci_scsi_get_bus_addr;
694         tran->tran_get_name     = vhci_scsi_get_name;
695         tran->tran_bus_reset    = NULL;
696         tran->tran_quiesce      = NULL;
697         tran->tran_unquiesce    = NULL;
698
699         /*
700          * register event notification routines with scsa
701          */
702         tran->tran_get_eventcookie = NULL;
703         tran->tran_add_eventcall = NULL;
704         tran->tran_remove_eventcall = NULL;
705         tran->tran_post_event   = NULL;
706
707         tran->tran_bus_power    = vhci_scsi_bus_power;
708
709         tran->tran_bus_config   = vhci_scsi_bus_config;
710         tran->tran_bus_unconfig = vhci_scsi_bus_unconfig;
711
712         /*
713          * Attach this instance with the mpxio framework
714          */
715         if (mdi_vhci_register(MDI_HCI_CLASS_SCSI, dip, &vhci_opinfo, 0)
716             != MDI_SUCCESS) {
717                 VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
718                     "mdi_vhci_register failed\n"));
719                 goto attach_fail;
720         }
```



```
721         vhci_attached++;
722
723         /*
724          * Attach this instance of the hba.
725          *
726          * Regarding dma attributes: Since scsi_vhci is a virtual scsi HBA
727          * driver, it has nothing to do with DMA. However, when calling
728          * scsi_hba_attach_setup() we need to pass something valid in the
729          * dma attributes parameter. So we just use scsi_alloc_attr.
730          * SCSA itself seems to care only for dma_attr_minxfer and
731          * dma_attr_burstsizes fields of dma attributes structure.
732          * It expects those fileds to be non-zero.
733          */
734         if (scsi_hba_attach_setup(dip, &scsi_alloc_attr, tran,
735             SCSI_HBA_ADDR_COMPLEX) != DDI_SUCCESS) {
736                 VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
737                     "hba attach failed\n"));
738                 goto attach_fail;
739         }
740         scsi_hba_attached++;
741
742         if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
743             INST2DEVCTL(instance), DDI_NT_SCSI_NEXUS, 0) != DDI_SUCCESS) {
744                 VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
745                     " ddi_create_minor_node failed\n"));
746                 goto attach_fail;
747         }
748
749         /*
750          * Set pm-want-child-notification property for
751          * power management of the phci and client
752          */
753         if (ddi_prop_create(DDI_DEV_T_NONE, dip, DDI_PROP_CANSLEEP,
754             "pm-want-child-notification?", NULL, NULL) != DDI_PROP_SUCCESS) {
755                 cmn_err(CE_WARN,
756                     "%s%d fail to create pm-want-child-notification? prop",
757                     ddi_driver_name(dip), ddi_get_instance(dip));
758                 goto attach_fail;
759         }
760
761         vhci->vhci_taskq = taskq_create("vhci_taskq", 1, MINCLSYSPRI, 1, 4, 0);
762         vhci->vhci_update_pathstates_taskq =
763             taskq_create("vhci_update_pathstates", VHCI_NUM_UPDATE_TASKQ,
764             MINCLSYSPRI, 1, 4, 0);
765         ASSERT(vhci->vhci_taskq);
766         ASSERT(vhci->vhci_update_pathstates_taskq);
767
768         /*
769          * Set appropriate configuration flags based on options set in
770          * conf file.
771          */
772         vhci->vhci_conf_flags = 0;
773         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, PROPFLAGS,
774             "auto-failback", &data) == DDI_SUCCESS) {
775                 if (strcmp(data, "enable") == 0)
776                         vhci->vhci_conf_flags |= VHCI_CONF_FLAGS_AUTO_FAILBACK;
777                 ddi_prop_free(data);
778         }
779
780         if (!(vhci->vhci_conf_flags & VHCI_CONF_FLAGS_AUTO_FAILBACK))
781                 vhci_log(CE_NOTE, dip, "!Auto-failback capability "
782                     "disabled through scsi_vhci.conf file.");
783
784         /*
785          * Allocate an mpapi private structure
786          */
```

```
787         vhci->mp_priv = kmem_zalloc(sizeof (mpapi_priv_t), KM_SLEEP);
788         if (vhci_mpapi_init(vhci) != 0) {
789                 VHCI_DEBUG(1, (CE_WARN, NULL, "!vhci_attach: "
790                     "vhci_mpapi_init() failed"));
791         }

793         vhci_failover_modopen(vhci);            /* load failover modules */

795         ddi_report_dev(dip);
796         return (DDI_SUCCESS);

798 attach_fail:
799         if (vhci_attached)
800                 (void) mdi_vhci_unregister(dip, 0);

802         if (scsi_hba_attached)
803                 (void) scsi_hba_detach(dip);

805         if (vhci->vhci_tran)
806                 scsi_hba_tran_free(vhci->vhci_tran);

808         if (mutex_initted) {
809                 mutex_destroy(&vhci->vhci_mutex);
810         }

812         ddi_soft_state_free(vhci_softstate, instance);
813         return (DDI_FAILURE);
814 }


817 /*ARGSUSED*/
818 static int
819 vhci_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
820 {
821         int                     instance = ddi_get_instance(dip);
822         scsi_hba_tran_t         *tran;
823         struct scsi_vhci        *vhci;

825         VHCI_DEBUG(4, (CE_NOTE, NULL, "vhci_detach: cmd=0x%x\n", cmd));

827         if ((tran = ddi_get_driver_private(dip)) == NULL)
828                 return (DDI_FAILURE);

830         vhci = TRAN2HBAPRIVATE(tran);
831         if (!vhci) {
832                 return (DDI_FAILURE);
833         }

835         switch (cmd) {
836         case DDI_DETACH:
837                 break;

839         case DDI_SUSPEND:
840         case DDI_PM_SUSPEND:
841                 VHCI_DEBUG(1, (CE_NOTE, NULL, "!vhci_detach: suspend/pm not yet"
842                     "implemented\n"));
843                 return (DDI_FAILURE);

845         default:
846                 VHCI_DEBUG(1, (CE_NOTE, NULL,
847                     "!vhci_detach: unknown ddi command\n"));
848                 return (DDI_FAILURE);
849         }

851         (void) mdi_vhci_unregister(dip, 0);
852         (void) scsi_hba_detach(dip);
```



```
853         scsi_hba_tran_free(tran);

855         if (ddi_prop_remove(DDI_DEV_T_NONE, dip,
856             "pm-want-child-notification?") != DDI_PROP_SUCCESS) {
857                 cmn_err(CE_WARN,
858                     "%s%d unable to remove prop pm-want_child_notification?",
859                     ddi_driver_name(dip), ddi_get_instance(dip));
860         }
861         if (vhci_restart_timeid != 0) {
862                 (void) untimeout(vhci_restart_timeid);
863         }
864         vhci_restart_timeid = 0;

866         mutex_destroy(&vhci->vhci_mutex);
867         vhci->vhci_dip = NULL;
868         vhci->vhci_tran = NULL;
869         taskq_destroy(vhci->vhci_taskq);
870         taskq_destroy(vhci->vhci_update_pathstates_taskq);
871         ddi_remove_minor_node(dip, NULL);
872         ddi_soft_state_free(vhci_softstate, instance);

874         vhci_failover_modclose();               /* unload failover modules */
875         return (DDI_SUCCESS);
876 }

878 /*
879  * vhci_getinfo()
880  * Given the device number, return the devinfo pointer or the
881  * instance number.
882  * Note: always succeed DDI_INFO_DEVT2INSTANCE, even before attach.
883  */

885 /*ARGSUSED*/
886 static int
887 vhci_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void **result)
888 {
889         struct scsi_vhci        *vhcip;
890         int                     instance = MINOR2INST(getminor((dev_t)arg));

892         switch (cmd) {
893         case DDI_INFO_DEVT2DEVINFO:
894                 vhcip = ddi_get_soft_state(vhci_softstate, instance);
895                 if (vhcip != NULL)
896                         *result = vhcip->vhci_dip;
897                 else {
898                         *result = NULL;
899                         return (DDI_FAILURE);
900                 }
901                 break;

903         case DDI_INFO_DEVT2INSTANCE:
904                 *result = (void *)(uintptr_t)instance;
905                 break;

907         default:
908                 return (DDI_FAILURE);
909         }

911         return (DDI_SUCCESS);
912 }

914 /*ARGSUSED*/
915 static int
916 vhci_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
917         scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
918 {
```

```
        char                    *guid;
        scsi_vhci_lun_t         *vlun;
        struct scsi_vhci        *vhci;
        clock_t                 from_ticks;
        mdi_pathinfo_t          *pip;
        int                     rval;

        ASSERT(hba_dip != NULL);
        ASSERT(tgt_dip != NULL);

        if (ddi_prop_lookup_string(DDI_DEV_T_ANY, tgt_dip, PROPFLAGS,
            MDI_CLIENT_GUID_PROP, &guid) != DDI_SUCCESS) {
                /*
                 * This must be the .conf node without GUID property.
                 * The node under fp already inserts a delay, so we
                 * just return from here. We rely on this delay to have
                 * all dips be posted to the ndi hotplug thread's newdev
                 * list. This is necessary for the deferred attach
                 * mechanism to work and opens() done soon after boot to
                 * succeed.
                 */
                VHCI_DEBUG(4, (CE_WARN, hba_dip, "tgt_init: lun guid "
                    "property failed"));
                return (DDI_NOT_WELL_FORMED);
        }

        if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
                /*
                 * This must be .conf node with the GUID property. We don't
                 * merge property by ndi_merge_node() here  because the
                 * devi_addr_buf of .conf node is "" always according the
                 * implementation of vhci_scsi_get_name_bus_addr().
                 */
                ddi_set_name_addr(tgt_dip, NULL);
                return (DDI_FAILURE);
        }

        vhci = ddi_get_soft_state(vhci_softstate, ddi_get_instance(hba_dip));
        ASSERT(vhci != NULL);

        VHCI_DEBUG(4, (CE_NOTE, hba_dip,
            "!tgt_init: called for %s (instance %d)\n",
            ddi_driver_name(tgt_dip), ddi_get_instance(tgt_dip)));

        vlun = vhci_lun_lookup(tgt_dip);

        mutex_enter(&vhci_global_mutex);

        from_ticks = ddi_get_lbolt();
        if (vhci_to_ticks == 0) {
                vhci_to_ticks = from_ticks +
                    drv_usectohz(vhci_init_wait_timeout);
        }

#if DEBUG
        if (vlun) {
                VHCI_DEBUG(1, (CE_WARN, hba_dip, "tgt_init: "
                    "vhci_scsi_tgt_init: guid %s : found vlun 0x%p "
                    "from_ticks %lx to_ticks %lx",
                    guid, (void *)vlun, from_ticks, vhci_to_ticks));
        } else {
                VHCI_DEBUG(1, (CE_WARN, hba_dip, "tgt_init: "
                    "vhci_scsi_tgt_init: guid %s : vlun not found "
                    "from_ticks %lx to_ticks %lx", guid, from_ticks,
                    vhci_to_ticks));
        }
```



```
#endif

        rval = mdi_select_path(tgt_dip, NULL,
            (MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH), NULL, &pip);
        if (rval == MDI_SUCCESS) {
                mdi_rele_path(pip);
        }

        /*
         * Wait for the following conditions :
         *      1. no vlun available yet
         *      2. no path established
         *      3. timer did not expire
         */
        while ((vlun == NULL) || (mdi_client_get_path_count(tgt_dip) == 0) ||
            (rval != MDI_SUCCESS)) {
                if (vlun && vlun->svl_not_supported) {
                        VHCI_DEBUG(1, (CE_WARN, hba_dip, "tgt_init: "
                            "vlun 0x%p lun guid %s not supported!",
                            (void *)vlun, guid));
                        mutex_exit(&vhci_global_mutex);
                        ddi_prop_free(guid);
                        return (DDI_NOT_WELL_FORMED);
                }
                if ((vhci_first_time == 0) && (from_ticks >= vhci_to_ticks)) {
                        vhci_first_time = 1;
                }
                if (vhci_first_time == 1) {
                        VHCI_DEBUG(1, (CE_WARN, hba_dip, "vhci_scsi_tgt_init: "
                            "no wait for %s. from_tick %lx, to_tick %lx",
                            guid, from_ticks, vhci_to_ticks));
                        mutex_exit(&vhci_global_mutex);
                        ddi_prop_free(guid);
                        return (DDI_NOT_WELL_FORMED);
                }

                if (cv_timedwait(&vhci_cv,
                    &vhci_global_mutex, vhci_to_ticks) == -1) {
                        /* Timed out */
#ifdef DEBUG
                        if (vlun == NULL) {
                                VHCI_DEBUG(1, (CE_WARN, hba_dip,
                                    "tgt_init: no vlun for %s!", guid));
                        } else if (mdi_client_get_path_count(tgt_dip) == 0) {
                                VHCI_DEBUG(1, (CE_WARN, hba_dip,
                                    "tgt_init: client path count is "
                                    "zero for %s!", guid));
                        } else {
                                VHCI_DEBUG(1, (CE_WARN, hba_dip,
                                    "tgt_init: client path not "
                                    "available yet for %s!", guid));
                        }
#endif /* DEBUG */
                        mutex_exit(&vhci_global_mutex);
                        ddi_prop_free(guid);
                        return (DDI_NOT_WELL_FORMED);
                }
                vlun = vhci_lun_lookup(tgt_dip);
                rval = mdi_select_path(tgt_dip, NULL,
                    (MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH),
                    NULL, &pip);
                if (rval == MDI_SUCCESS) {
                        mdi_rele_path(pip);
                }
                from_ticks = ddi_get_lbolt();
        }
```

```
1051         mutex_exit(&vhci_global_mutex);

1053         ASSERT(vlun != NULL);
1054         ddi_prop_free(guid);

1056         scsi_device_hba_private_set(sd, vlun);

1058         return (DDI_SUCCESS);
1059 }

1061 /*ARGSUSED*/
1062 static void
1063 vhci_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1064         scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1065 {
1066         struct scsi_vhci_lun *dvlp;
1067         ASSERT(mdi_client_get_path_count(tgt_dip) <= 0);
1068         dvlp = (struct scsi_vhci_lun *)scsi_device_hba_private_get(sd);
1069         ASSERT(dvlp != NULL);

1071         vhci_lun_free(dvlp, sd);
1072 }

1074 /*
1075  * a PGR register command has started; copy the info we need
1076  */
1077 int
1078 vhci_pgr_register_start(scsi_vhci_lun_t *vlun, struct scsi_pkt *pkt)
1079 {
1080         struct vhci_pkt         *vpkt = TGTPKT2VHCIPKT(pkt);
1081         void                    *addr;

1083         if (!vpkt->vpkt_tgt_init_bp)
1084                 return (TRAN_BADPKT);

1086         addr = bp_mapin_common(vpkt->vpkt_tgt_init_bp,
1087             (vpkt->vpkt_flags & CFLAG_NOWAIT) ? VM_NOSLEEP : VM_SLEEP);
1088         if (addr == NULL)
1089                 return (TRAN_BUSY);

1091         mutex_enter(&vlun->svl_mutex);

1093         vhci_print_prout_keys(vlun, "v_pgr_reg_start: before bcopy:");

1095         bcopy(addr, &vlun->svl_prout, sizeof (vhci_prout_t) -
1096             (2 * MHIOC_RESV_KEY_SIZE*sizeof (char)));
1097         bcopy(pkt->pkt_cdbp, vlun->svl_cdb, sizeof (vlun->svl_cdb));

1099         vhci_print_prout_keys(vlun, "v_pgr_reg_start: after bcopy:");

1101         vlun->svl_time = pkt->pkt_time;
1102         vlun->svl_bcount = vpkt->vpkt_tgt_init_bp->b_bcount;
1103         vlun->svl_first_path = vpkt->vpkt_path;
1104         mutex_exit(&vlun->svl_mutex);
1105         return (0);
1106 }

1108 /*
1109  * Function name : vhci_scsi_start()
1110  *
1111  * Return Values : TRAN_FATAL_ERROR     - vhci has been shutdown
1112  *                                        or other fatal failure
1113  *                                        preventing packet transportation
1114  *                 TRAN_BUSY            - request queue is full
1115  *                 TRAN_ACCEPT          - pkt has been submitted to phci
1116  *                                        (or is held in the waitQ)
```



```
1117  * Description   : Implements SCSA's tran_start() entry point for
1118  *                 packet transport
1119  *
1120  */
1121 static int
1122 vhci_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
1123 {
1124         int                     rval = TRAN_ACCEPT;
1125         int                     instance, held;
1126         struct scsi_vhci        *vhci = ADDR2VHCI(ap);
1127         struct scsi_vhci_lun    *vlun = ADDR2VLUN(ap);
1128         struct vhci_pkt         *vpkt = TGTPKT2VHCIPKT(pkt);
1129         int                     flags = 0;
1130         scsi_vhci_priv_t        *svp, *svp_resrv;
1131         dev_info_t              *cdip;
1132         client_lb_t             lbp;
1133         int                     restore_lbp = 0;
1134         /* set if pkt is SCSI-II RESERVE cmd */
1135         int                     pkt_reserve_cmd = 0;
1136         int                     reserve_failed = 0;
1137         int                     resrv_instance = 0;
1138         mdi_pathinfo_t          *pip;
1139         struct scsi_pkt         *rel_pkt;

1141         ASSERT(vhci != NULL);
1142         ASSERT(vpkt != NULL);
1143         ASSERT(vpkt->vpkt_state != VHCI_PKT_ISSUED);
1144         cdip = ADDR2DIP(ap);

1146         /*
1147          * Block IOs if LUN is held or QUIESCED for IOs.
1148          */
1149         if ((VHCI_LUN_IS_HELD(vlun)) ||
1150             ((vlun->svl_flags & VLUN_QUIESCED_FLG) == VLUN_QUIESCED_FLG)) {
1151                 return (TRAN_BUSY);
1152         }

1154         /*
1155          * vhci_lun needs to be quiesced before SCSI-II RESERVE command
1156          * can be issued.  This may require a cv_timedwait, which is
1157          * dangerous to perform in an interrupt context.  So if this
1158          * is a RESERVE command a taskq is dispatched to service it.
1159          * This taskq shall again call vhci_scsi_start, but we shall be
1160          * sure its not in an interrupt context.
1161          */
1162         if ((pkt->pkt_cdbp[0] == SCMD_RESERVE) ||
1163             (pkt->pkt_cdbp[0] == SCMD_RESERVE_G1)) {
1164                 if (!(vpkt->vpkt_state & VHCI_PKT_THRU_TASKQ)) {
1165                         if (taskq_dispatch(vhci->vhci_taskq,
1166                             vhci_dispatch_scsi_start, (void *) vpkt,
1167                             KM_NOSLEEP)) {
1168                                 return (TRAN_ACCEPT);
1169                         } else {
1170                                 return (TRAN_BUSY);
1171                         }
1172                 }

1174                 /*
1175                  * Here we ensure that simultaneous SCSI-II RESERVE cmds don't
1176                  * get serviced for a lun.
1177                  */
1178                 VHCI_HOLD_LUN(vlun, VH_NOSLEEP, held);
1179                 if (!held) {
1180                         return (TRAN_BUSY);
1181                 } else if ((vlun->svl_flags & VLUN_QUIESCED_FLG) ==
1182                     VLUN_QUIESCED_FLG) {
```

```
1183                        VHCI_RELEASE_LUN(vlun);
1184                        return (TRAN_BUSY);
1185                }

1187                /*
1188                 * To ensure that no IOs occur for this LUN for the duration
1189                 * of this pkt set the VLUN_QUIESCED_FLG.
1190                 * In case this routine needs to exit on error make sure that
1191                 * this flag is cleared.
1192                 */
1193                vlun->svl_flags |= VLUN_QUIESCED_FLG;
1194                pkt_reserve_cmd = 1;

1196                /*
1197                 * if this is a SCSI-II RESERVE command, set load balancing
1198                 * policy to be ALTERNATE PATH to ensure that all subsequent
1199                 * IOs are routed on the same path.  This is because if commands
1200                 * are routed across multiple paths then IOs on paths other than
1201                 * the one on which the RESERVE was executed will get a
1202                 * RESERVATION CONFLICT
1203                 */
1204                lbp = mdi_get_lb_policy(cdip);
1205                if (lbp != LOAD_BALANCE_NONE) {
1206                        if (vhci_quiesce_lun(vlun) != 1) {
1207                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1208                                VHCI_RELEASE_LUN(vlun);
1209                                return (TRAN_FATAL_ERROR);
1210                        }
1211                        vlun->svl_lb_policy_save = lbp;
1212                        if (mdi_set_lb_policy(cdip, LOAD_BALANCE_NONE) !=
1213                            MDI_SUCCESS) {
1214                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1215                                VHCI_RELEASE_LUN(vlun);
1216                                return (TRAN_FATAL_ERROR);
1217                        }
1218                        restore_lbp = 1;
1219                }

1221                VHCI_DEBUG(2, (CE_NOTE, vhci->vhci_dip,
1222                    "!vhci_scsi_start: sending SCSI-2 RESERVE, vlun 0x%p, "
1223                    "svl_resrv_pip 0x%p, svl_flags: %x, lb_policy %x",
1224                    (void *)vlun, (void *)vlun->svl_resrv_pip, vlun->svl_flags,
1225                    mdi_get_lb_policy(cdip)));

1227                /*
1228                 * See comments for VLUN_RESERVE_ACTIVE_FLG in scsi_vhci.h
1229                 * To narrow this window where a reserve command may be sent
1230                 * down an inactive path the path states first need to be
1231                 * updated.  Before calling vhci_update_pathstates reset
1232                 * VLUN_RESERVE_ACTIVE_FLG, just in case it was already set
1233                 * for this lun.  This shall prevent an unnecessary reset
1234                 * from being sent out.  Also remember currently reserved path
1235                 * just for a case the new reservation will go to another path.
1236                 */
1237                if (vlun->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
1238                        resrv_instance = mdi_pi_get_path_instance(
1239                            vlun->svl_resrv_pip);
1240                }
1241                vlun->svl_flags &= ~VLUN_RESERVE_ACTIVE_FLG;
1242                vhci_update_pathstates((void *)vlun);
1243        }

1245        instance = ddi_get_instance(vhci->vhci_dip);

1247        /*
1248         * If the command is PRIN with action of zero, then the cmd
```



```
1249         * is reading PR keys which requires filtering on completion.
1250         * Data cache sync must be guaranteed.
1251         */
1252        if ((pkt->pkt_cdbp[0] == SCMD_PRIN) && (pkt->pkt_cdbp[1] == 0) &&
1253            (vpkt->vpkt_org_vpkt == NULL)) {
1254                vpkt->vpkt_tgt_init_pkt_flags |= PKT_CONSISTENT;
1255        }

1257        /*
1258         * Do not defer bind for PKT_DMA_PARTIAL
1259         */
1260        if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {

1262                /* This is a non pkt_dma_partial case */
1263                if ((rval = vhci_bind_transport(
1264                    ap, vpkt, vpkt->vpkt_tgt_init_pkt_flags, NULL_FUNC))
1265                    != TRAN_ACCEPT) {
1266                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1267                            "!vhci%d %x: failed to bind transport: "
1268                            "vlun 0x%p pkt_reserved %x restore_lbp %x,"
1269                            "lbp %x", instance, rval, (void *)vlun,
1270                            pkt_reserve_cmd, restore_lbp, lbp));
1271                        if (restore_lbp)
1272                                (void) mdi_set_lb_policy(cdip, lbp);
1273                        if (pkt_reserve_cmd)
1274                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1275                        return (rval);
1276                }
1277                VHCI_DEBUG(8, (CE_NOTE, NULL,
1278                    "vhci_scsi_start: v_b_t called 0x%p\n", (void *)vpkt));
1279        }
1280        ASSERT(vpkt->vpkt_hba_pkt != NULL);
1281        ASSERT(vpkt->vpkt_path != NULL);

1283        /*
1284         * This is the chance to adjust the pHCI's pkt and other information
1285         * from target driver's pkt.
1286         */
1287        VHCI_DEBUG(8, (CE_NOTE, vhci->vhci_dip, "vhci_scsi_start vpkt %p\n",
1288            (void *)vpkt));
1289        vhci_update_pHCI_pkt(vpkt, pkt);

1291        if (vlun->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
1292                if (vpkt->vpkt_path != vlun->svl_resrv_pip) {
1293                        VHCI_DEBUG(1, (CE_WARN, vhci->vhci_dip,
1294                            "!vhci_bind: reserve flag set for vlun 0x%p, but, "
1295                            "pktpath 0x%p resrv path 0x%p differ. lb_policy %x",
1296                            (void *)vlun, (void *)vpkt->vpkt_path,
1297                            (void *)vlun->svl_resrv_pip,
1298                            mdi_get_lb_policy(cdip)));
1299                        reserve_failed = 1;
1300                }
1301        }

1303        svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(vpkt->vpkt_path);
1304        if (svp == NULL || reserve_failed) {
1305                if (pkt_reserve_cmd) {
1306                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1307                            "!vhci_bind returned null svp vlun 0x%p",
1308                            (void *)vlun));
1309                        vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1310                        if (restore_lbp)
1311                                (void) mdi_set_lb_policy(cdip, lbp);
1312                }
1313 pkt_cleanup:
1314                if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {
```

```
1315                        scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
1316                        vpkt->vpkt_hba_pkt = NULL;
1317                        if (vpkt->vpkt_path) {
1318                                mdi_rele_path(vpkt->vpkt_path);
1319                                vpkt->vpkt_path = NULL;
1320                        }
1321                }
1322                if ((pkt->pkt_cdbp[0] == SCMD_PROUT) &&
1323                    (((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_REGISTER) ||
1324                    ((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_R_AND_IGNORE))) {
1325                        sema_v(&vlun->svl_pgr_sema);
1326                }
1327                return (TRAN_BUSY);
1328        }

1330        if ((resrv_instance != 0) && (resrv_instance !=
1331            mdi_pi_get_path_instance(vpkt->vpkt_path))) {
1332                /*
1333                 * This is an attempt to reserve vpkt->vpkt_path.  But the
1334                 * previously reserved path referred by resrv_instance might
1335                 * still be reserved.  Hence we will send a release command
1336                 * there in order to avoid a reservation conflict.
1337                 */
1338                VHCI_DEBUG(1, (CE_NOTE, vhci->vhci_dip, "!vhci_scsi_start: "
1339                    "conflicting reservation on another path, vlun 0x%p, "
1340                    "reserved instance %d, new instance: %d, pip: 0x%p",
1341                    (void *)vlun, resrv_instance,
1342                    mdi_pi_get_path_instance(vpkt->vpkt_path),
1343                    (void *)vpkt->vpkt_path));

1345                /*
1346                 * In rare cases, the path referred by resrv_instance could
1347                 * disappear in the meantime. Calling mdi_select_path() below
1348                 * is an attempt to find out if the path still exists. It also
1349                 * ensures that the path will be held when the release is sent.
1350                 */
1351                rval = mdi_select_path(cdip, NULL, MDI_SELECT_PATH_INSTANCE,
1352                    (void *)(intptr_t)resrv_instance, &pip);

1354                if ((rval == MDI_SUCCESS) && (pip != NULL)) {
1355                        svp_resrv = (scsi_vhci_priv_t *)
1356                            mdi_pi_get_vhci_private(pip);
1357                        rel_pkt = scsi_init_pkt(&svp_resrv->svp_psd->sd_address,
1358                            NULL, NULL, CDB_GROUP0,
1359                            sizeof (struct scsi_arq_status), 0, 0, SLEEP_FUNC,
1360                            NULL);

1362                        if (rel_pkt == NULL) {
1363                                char    *p_path;

1365                                /*
1366                                 * This is very unlikely.
1367                                 * scsi_init_pkt(SLEEP_FUNC) does not fail
1368                                 * because of resources. But in theory it could
1369                                 * fail for some other reason. There is not an
1370                                 * easy way how to recover though. Log a warning
1371                                 * and return.
1372                                 */
1373                                p_path = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
1374                                vhci_log(CE_WARN, vhci->vhci_dip, "!Sending "
1375                                    "RELEASE(6) to %s failed, a potential "
1376                                    "reservation conflict ahead.",
1377                                    ddi_pathname(mdi_pi_get_phci(pip), p_path));
1378                                kmem_free(p_path, MAXPATHLEN);

1380                                if (restore_lbp)
```



```
1381                                        (void) mdi_set_lb_policy(cdip, lbp);

1383                                /* no need to check pkt_reserve_cmd here */
1384                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1385                                return (TRAN_FATAL_ERROR);
1386                        }

1388                        rel_pkt->pkt_cdbp[0] = SCMD_RELEASE;
1389                        rel_pkt->pkt_time = 60;

1391                        /*
1392                         * Ignore the return value.  If it will fail
1393                         * then most likely it is no longer reserved
1394                         * anyway.
1395                         */
1396                        (void) vhci_do_scsi_cmd(rel_pkt);
1397                        VHCI_DEBUG(1, (CE_NOTE, NULL,
1398                            "!vhci_scsi_start: path 0x%p, issued SCSI-2"
1399                            " RELEASE\n", (void *)pip));
1400                        scsi_destroy_pkt(rel_pkt);
1401                        mdi_rele_path(pip);
1402                }
1403        }

1405        VHCI_INCR_PATH_CMDCOUNT(svp);

1407        /*
1408         * Ensure that no other IOs raced ahead, while a RESERVE cmd was
1409         * QUIESCING the same lun.
1410         */
1411        if ((!pkt_reserve_cmd) &&
1412            ((vlun->svl_flags & VLUN_QUIESCED_FLG) == VLUN_QUIESCED_FLG)) {
1413                VHCI_DECR_PATH_CMDCOUNT(svp);
1414                goto pkt_cleanup;
1415        }

1417        if ((pkt->pkt_cdbp[0] == SCMD_PRIN) ||
1418            (pkt->pkt_cdbp[0] == SCMD_PROUT)) {
1419                /*
1420                 * currently this thread only handles running PGR
1421                 * commands, so don't bother creating it unless
1422                 * something interesting is going to happen (like
1423                 * either a PGR out, or a PGR in with enough space
1424                 * to hold the keys that are getting returned)
1425                 */
1426                mutex_enter(&vlun->svl_mutex);
1427                if (((vlun->svl_flags & VLUN_TASK_D_ALIVE_FLG) == 0) &&
1428                    (pkt->pkt_cdbp[0] == SCMD_PROUT)) {
1429                        vlun->svl_taskq = taskq_create("vlun_pgr_task_daemon",
1430                            1, MINCLSYSPRI, 1, 4, 0);
1431                        vlun->svl_flags |= VLUN_TASK_D_ALIVE_FLG;
1432                }
1433                mutex_exit(&vlun->svl_mutex);
1434                if ((pkt->pkt_cdbp[0] == SCMD_PROUT) &&
1435                    (((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_REGISTER) ||
1436                    ((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_R_AND_IGNORE))) {
1437                        if (rval = vhci_pgr_register_start(vlun, pkt)) {
1438                                /* an error */
1439                                sema_v(&vlun->svl_pgr_sema);
1440                                return (rval);
1441                        }
1442                }
1443        }

1445        /*
1446         * SCSI-II RESERVE cmd is not expected in polled mode.
```

```
1447          * If this changes it needs to be handled for the polled scenario.
1448          */
1449         flags = vpkt->vpkt_hba_pkt->pkt_flags;

1451         /*
1452          * Set the path_instance *before* sending the scsi_pkt down the path
1453          * to mpxio's pHCI so that additional path abstractions at a pHCI
1454          * level (like maybe iSCSI at some point in the future) can update
1455          * the path_instance.
1456          */
1457         if (scsi_pkt_allocated_correctly(vpkt->vpkt_hba_pkt))
1458                 vpkt->vpkt_hba_pkt->pkt_path_instance =
1459                     mdi_pi_get_path_instance(vpkt->vpkt_path);

1461         rval = scsi_transport(vpkt->vpkt_hba_pkt);
1462         if (rval == TRAN_ACCEPT) {
1463                 if (flags & FLAG_NOINTR) {
1464                         struct scsi_pkt *tpkt = vpkt->vpkt_tgt_pkt;
1465                         struct scsi_pkt *pkt = vpkt->vpkt_hba_pkt;

1467                         ASSERT(tpkt != NULL);
1468                         *(tpkt->pkt_scbp) = *(pkt->pkt_scbp);
1469                         tpkt->pkt_resid = pkt->pkt_resid;
1470                         tpkt->pkt_state = pkt->pkt_state;
1471                         tpkt->pkt_statistics = pkt->pkt_statistics;
1472                         tpkt->pkt_reason = pkt->pkt_reason;

1474                         if ((*(pkt->pkt_scbp) == STATUS_CHECK) &&
1475                             (pkt->pkt_state & STATE_ARQ_DONE)) {
1476                                 bcopy(pkt->pkt_scbp, tpkt->pkt_scbp,
1477                                     vpkt->vpkt_tgt_init_scblen);
1478                         }

1480                         VHCI_DECR_PATH_CMDCOUNT(svp);
1481                         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {
1482                                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
1483                                 vpkt->vpkt_hba_pkt = NULL;
1484                                 if (vpkt->vpkt_path) {
1485                                         mdi_rele_path(vpkt->vpkt_path);
1486                                         vpkt->vpkt_path = NULL;
1487                                 }
1488                         }
1489                         /*
1490                          * This path will not automatically retry pkts
1491                          * internally, therefore, vpkt_org_vpkt should
1492                          * never be set.
1493                          */
1494                         ASSERT(vpkt->vpkt_org_vpkt == NULL);
1495                         scsi_hba_pkt_comp(tpkt);
1496                 }
1497                 return (rval);
1498         } else if ((pkt->pkt_cdbp[0] == SCMD_PROUT) &&
1499             (((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_REGISTER) ||
1500             ((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_R_AND_IGNORE))) {
1501                 /* the command exited with bad status */
1502                 sema_v(&vlun->svl_pgr_sema);
1503         } else if (vpkt->vpkt_tgt_pkt->pkt_cdbp[0] == SCMD_PRIN) {
1504                 /* the command exited with bad status */
1505                 sema_v(&vlun->svl_pgr_sema);
1506         } else if (pkt_reserve_cmd) {
1507                 VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1508                     "!vhci_scsi_start: reserve failed vlun 0x%p",
1509                     (void *)vlun));
1510                 vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1511                 if (restore_lbp)
1512                         (void) mdi_set_lb_policy(cdip, lbp);
```



```
1513         }

1515         ASSERT(vpkt->vpkt_hba_pkt != NULL);
1516         VHCI_DECR_PATH_CMDCOUNT(svp);

1518         /* Do not destroy phci packet information for PKT_DMA_PARTIAL */
1519         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {
1520                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
1521                 vpkt->vpkt_hba_pkt = NULL;
1522                 if (vpkt->vpkt_path) {
1523                         MDI_PI_ERRSTAT(vpkt->vpkt_path, MDI_PI_TRANSERR);
1524                         mdi_rele_path(vpkt->vpkt_path);
1525                         vpkt->vpkt_path = NULL;
1526                 }
1527         }
1528         return (TRAN_BUSY);
1529 }

1531 /*
1532  * Function name : vhci_scsi_reset()
1533  *
1534  * Return Values : 0 - reset failed
1535  *                 1 - reset succeeded
1536  */

1538 /* ARGSUSED */
1539 static int
1540 vhci_scsi_reset(struct scsi_address *ap, int level)
1541 {
1542         int rval = 0;

1544         cmn_err(CE_WARN, "!vhci_scsi_reset 0x%x", level);
1545         if ((level == RESET_TARGET) || (level == RESET_LUN)) {
1546                 return (vhci_scsi_reset_target(ap, level, TRUE));
1547         } else if (level == RESET_ALL) {
1548                 return (vhci_scsi_reset_bus(ap));
1549         }

1551         return (rval);
1552 }

1554 /*
1555  * vhci_recovery_reset:
1556  *      Issues reset to the device
1557  * Input:
1558  *      vlun - vhci lun pointer of the device
1559  *      ap - address of the device
1560  *      select_path:
1561  *              If select_path is FALSE, then the address specified in ap is
1562  *              the path on which reset will be issued.
1563  *              If select_path is TRUE, then path is obtained by calling
1564  *              mdi_select_path.
1565  *
1566  *      recovery_depth:
1567  *              Caller can specify the level of reset.
1568  *              VHCI_DEPTH_LUN -
1569  *                      Issues LUN RESET if device supports lun reset.
1570  *              VHCI_DEPTH_TARGET -
1571  *                      If Lun Reset fails or the device does not support
1572  *                      Lun Reset, issues TARGET RESET
1573  *              VHCI_DEPTH_ALL -
1574  *                      If Lun Reset fails or the device does not support
1575  *                      Lun Reset, issues TARGET RESET.
1576  *                      If TARGET RESET does not succeed, issues Bus Reset.
1577  */
```

```
1579 static int
1580 vhci_recovery_reset(scsi_vhci_lun_t *vlun, struct scsi_address *ap,
1581         uint8_t select_path, uint8_t recovery_depth)
1582 {
1583         int     ret = 0;

1585         ASSERT(ap != NULL);

1587         if (vlun && vlun->svl_support_lun_reset == 1) {
1588                 ret = vhci_scsi_reset_target(ap, RESET_LUN,
1589                     select_path);
1590         }

1592         recovery_depth--;

1594         if ((ret == 0) && recovery_depth) {
1595                 ret = vhci_scsi_reset_target(ap, RESET_TARGET,
1596                     select_path);
1597                 recovery_depth--;
1598         }

1600         if ((ret == 0) && recovery_depth) {
1601                 (void) scsi_reset(ap, RESET_ALL);
1602         }

1604         return (ret);
1605 }

1607 /*
1608  * Note: The scsi_address passed to this routine could be the scsi_address
1609  * for the virtual device or the physical device. No assumptions should be
1610  * made in this routine about the contents of the ap structure.
1611  * Further, note that the child dip would be the dip of the ssd node regardless
1612  * of the scsi_address passed in.
1613  */
1614 static int
1615 vhci_scsi_reset_target(struct scsi_address *ap, int level, uint8_t select_path)
1616 {
1617         dev_info_t              *vdip, *cdip;
1618         mdi_pathinfo_t          *pip = NULL;
1619         mdi_pathinfo_t          *npip = NULL;
1620         int                     rval = -1;
1621         scsi_vhci_priv_t        *svp = NULL;
1622         struct scsi_address     *pap = NULL;
1623         scsi_hba_tran_t         *hba = NULL;
1624         int                     sps;
1625         struct scsi_vhci        *vhci = NULL;

1627         if (select_path != TRUE) {
1628                 ASSERT(ap != NULL);
1629                 if (level == RESET_LUN) {
1630                         hba = ap->a_hba_tran;
1631                         ASSERT(hba != NULL);
1632                         return (hba->tran_reset(ap, RESET_LUN));
1633                 }
1634                 return (scsi_reset(ap, level));
1635         }

1637         cdip = ADDR2DIP(ap);
1638         ASSERT(cdip != NULL);
1639         vdip = ddi_get_parent(cdip);
1640         ASSERT(vdip != NULL);
1641         vhci = ddi_get_soft_state(vhci_softstate, ddi_get_instance(vdip));
1642         ASSERT(vhci != NULL);

1644         rval = mdi_select_path(cdip, NULL, MDI_SELECT_ONLINE_PATH, NULL, &pip);
```



```
1645         if ((rval != MDI_SUCCESS) || (pip == NULL)) {
1646                 VHCI_DEBUG(2, (CE_WARN, NULL, "!vhci_scsi_reset_target: "
1647                     "Unable to get a path, dip 0x%p", (void *)cdip));
1648                 return (0);
1649         }
1650 again:
1651         svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(pip);
1652         if (svp == NULL) {
1653                 VHCI_DEBUG(2, (CE_WARN, NULL, "!vhci_scsi_reset_target: "
1654                     "priv is NULL, pip 0x%p", (void *)pip));
1655                 mdi_rele_path(pip);
1656                 return (0);
1657         }

1659         if (svp->svp_psd == NULL) {
1660                 VHCI_DEBUG(2, (CE_WARN, NULL, "!vhci_scsi_reset_target: "
1661                     "psd is NULL, pip 0x%p, svp 0x%p",
1662                     (void *)pip, (void *)svp));
1663                 mdi_rele_path(pip);
1664                 return (0);
1665         }

1667         pap = &svp->svp_psd->sd_address;
1668         hba = pap->a_hba_tran;

1670         ASSERT(pap != NULL);
1671         ASSERT(hba != NULL);

1673         if (hba->tran_reset != NULL) {
1674                 if (hba->tran_reset(pap, level) == 0) {
1675                         vhci_log(CE_WARN, vdip, "!%s%d: "
1676                             "path %s, reset %d failed",
1677                             ddi_driver_name(cdip), ddi_get_instance(cdip),
1678                             mdi_pi_spathname(pip), level);

1680                         /*
1681                          * Select next path and issue the reset, repeat
1682                          * until all paths are exhausted
1683                          */
1684                         sps = mdi_select_path(cdip, NULL,
1685                             MDI_SELECT_ONLINE_PATH, pip, &npip);
1686                         if ((sps != MDI_SUCCESS) || (npip == NULL)) {
1687                                 mdi_rele_path(pip);
1688                                 return (0);
1689                         }
1690                         mdi_rele_path(pip);
1691                         pip = npip;
1692                         goto again;
1693                 }
1694                 mdi_rele_path(pip);
1695                 mutex_enter(&vhci->vhci_mutex);
1696                 scsi_hba_reset_notify_callback(&vhci->vhci_mutex,
1697                     &vhci->vhci_reset_notify_listf);
1698                 mutex_exit(&vhci->vhci_mutex);
1699                 VHCI_DEBUG(6, (CE_NOTE, NULL, "!vhci_scsi_reset_target: "
1700                     "reset %d sent down pip:%p for cdip:%p\n", level,
1701                     (void *)pip, (void *)cdip));
1702                 return (1);
1703         }
1704         mdi_rele_path(pip);
1705         return (0);
1706 }


1709 /* ARGSUSED */
1710 static int
```

```
vhci_scsi_reset_bus(struct scsi_address *ap)
{
        return (1);
}


/*
 * called by vhci_getcap and vhci_setcap to get and set (respectively)
 * SCSI capabilities
 */
/* ARGSUSED */
static int
vhci_commoncap(struct scsi_address *ap, char *cap,
    int val, int tgtonly, int doset)
{
        struct scsi_vhci                *vhci = ADDR2VHCI(ap);
        struct scsi_vhci_lun            *vlun = ADDR2VLUN(ap);
        int                     cidx;
        int                     rval = 0;

        if (cap == (char *)0) {
                VHCI_DEBUG(3, (CE_WARN, vhci->vhci_dip,
                    "!vhci_commoncap: invalid arg"));
                return (rval);
        }

        if (vlun == NULL) {
                VHCI_DEBUG(3, (CE_WARN, vhci->vhci_dip,
                    "!vhci_commoncap: vlun is null"));
                return (rval);
        }

        if ((cidx = scsi_hba_lookup_capstr(cap)) == -1) {
                return (UNDEFINED);
        }

        /*
         * Process setcap request.
         */
        if (doset) {
                /*
                 * At present, we can only set binary (0/1) values
                 */
                switch (cidx) {
                case SCSI_CAP_ARQ:
                        if (val == 0) {
                                rval = 0;
                        } else {
                                rval = 1;
                        }
                        break;

                case SCSI_CAP_LUN_RESET:
                        if (tgtonly == 0) {
                                VHCI_DEBUG(1, (CE_WARN, vhci->vhci_dip,
                                    "scsi_vhci_setcap: "
                                    "Returning error since whom = 0"));
                                rval = -1;
                                break;
                        }
                        /*
                         * Set the capability accordingly.
                         */
                        mutex_enter(&vlun->svl_mutex);
                        vlun->svl_support_lun_reset = val;
                        rval = val;
```



```
                        mutex_exit(&vlun->svl_mutex);
                        break;

                case SCSI_CAP_SECTOR_SIZE:
                        mutex_enter(&vlun->svl_mutex);
                        vlun->svl_sector_size = val;
                        vlun->svl_setcap_done = 1;
                        mutex_exit(&vlun->svl_mutex);
                        (void) vhci_pHCI_cap(ap, cap, val, tgtonly, NULL);

                        /* Always return success */
                        rval = 1;
                        break;

                default:
                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
                            "!vhci_setcap: unsupported %d", cidx));
                        rval = UNDEFINED;
                        break;
                }

                VHCI_DEBUG(6, (CE_NOTE, vhci->vhci_dip,
                    "!set cap: cap=%s, val/tgtonly/doset/rval = "
                    "0x%x/0x%x/0x%x/%d\n",
                    cap, val, tgtonly, doset, rval));

        } else {
                /*
                 * Process getcap request.
                 */
                switch (cidx) {
                case SCSI_CAP_DMA_MAX:
                        /*
                         * For X86 this capability is caught in scsi_ifgetcap().
                         * XXX Should this be getting the value from the pHCI?
                         */
                        rval = (int)VHCI_DMA_MAX_XFER_CAP;
                        break;

                case SCSI_CAP_INITIATOR_ID:
                        rval = 0x00;
                        break;

                case SCSI_CAP_ARQ:
                case SCSI_CAP_RESET_NOTIFICATION:
                case SCSI_CAP_TAGGED_QING:
                        rval = 1;
                        break;

                case SCSI_CAP_SCSI_VERSION:
                        rval = 3;
                        break;

                case SCSI_CAP_INTERCONNECT_TYPE:
                        rval = INTERCONNECT_FABRIC;
                        break;

                case SCSI_CAP_LUN_RESET:
                        /*
                         * scsi_vhci will always return success for LUN reset.
                         * When request for doing LUN reset comes
                         * through scsi_reset entry point, at that time attempt
                         * will be made to do reset through all the possible
                         * paths.
                         */
                        mutex_enter(&vlun->svl_mutex);
```

```
1843                        rval = vlun->svl_support_lun_reset;
1844                        mutex_exit(&vlun->svl_mutex);
1845                        VHCI_DEBUG(4, (CE_WARN, vhci->vhci_dip,
1846                            "scsi_vhci_getcap:"
1847                            "Getting the Lun reset capability %d", rval));
1848                        break;

1850                case SCSI_CAP_SECTOR_SIZE:
1851                        mutex_enter(&vlun->svl_mutex);
1852                        rval = vlun->svl_sector_size;
1853                        mutex_exit(&vlun->svl_mutex);
1854                        break;

1856                case SCSI_CAP_CDB_LEN:
1857                        rval = VHCI_SCSI_CDB_SIZE;
1858                        break;

1860                case SCSI_CAP_DMA_MAX_ARCH:
1861                        /*
1862                         * For X86 this capability is caught in scsi_ifgetcap().
1863                         * XXX Should this be getting the value from the pHCI?
1864                         */
1865                        rval = 0;
1866                        break;

1868                default:
1869                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1870                            "!vhci_getcap: unsupported %d", cidx));
1871                        rval = UNDEFINED;
1872                        break;
1873                }

1875                VHCI_DEBUG(6, (CE_NOTE, vhci->vhci_dip,
1876                    "!get cap: cap=%s, val/tgtonly/doset/rval = "
1877                    "0x%x/0x%x/0x%x/%d\n",
1878                    cap, val, tgtonly, doset, rval));
1879        }
1880        return (rval);
1881 }


1884 /*
1885  * Function name : vhci_scsi_getcap()
1886  *
1887  */
1888 static int
1889 vhci_scsi_getcap(struct scsi_address *ap, char *cap, int whom)
1890 {
1891        return (vhci_commoncap(ap, cap, 0, whom, 0));
1892 }

1894 static int
1895 vhci_scsi_setcap(struct scsi_address *ap, char *cap, int value, int whom)
1896 {
1897        return (vhci_commoncap(ap, cap, value, whom, 1));
1898 }

1900 /*
1901  * Function name : vhci_scsi_abort()
1902  */
1903 /* ARGSUSED */
1904 static int
1905 vhci_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
1906 {
1907        return (0);
1908 }
```



```
1910 /*
1911  * Function name : vhci_scsi_init_pkt
1912  *
1913  * Return Values : pointer to scsi_pkt, or NULL
1914  */
1915 /* ARGSUSED */
1916 static struct scsi_pkt *
1917 vhci_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
1918        struct buf *bp, int cmdlen, int statuslen, int tgtlen,
1919        int flags, int (*callback)(caddr_t), caddr_t arg)
1920 {
1921        struct scsi_vhci        *vhci = ADDR2VHCI(ap);
1922        struct vhci_pkt         *vpkt;
1923        int                     rval;
1924        int                     newpkt = 0;
1925        struct scsi_pkt         *pktp;


1928        if (pkt == NULL) {
1929                if (cmdlen > VHCI_SCSI_CDB_SIZE) {
1930                        if ((cmdlen != VHCI_SCSI_OSD_CDB_SIZE) ||
1931                            ((flags & VHCI_SCSI_OSD_PKT_FLAGS) !=
1932                            VHCI_SCSI_OSD_PKT_FLAGS)) {
1933                                VHCI_DEBUG(1, (CE_NOTE, NULL,
1934                                    "!init pkt: cdb size not supported\n"));
1935                                return (NULL);
1936                        }
1937                }

1939                pktp = scsi_hba_pkt_alloc(vhci->vhci_dip,
1940                    ap, cmdlen, statuslen, tgtlen, sizeof (*vpkt), callback,
1941                    arg);

1943                if (pktp == NULL) {
1944                        return (NULL);
1945                }

1947                /* Get the vhci's private structure */
1948                vpkt = (struct vhci_pkt *)(pktp->pkt_ha_private);
1949                ASSERT(vpkt);

1951                /* Save the target driver's packet */
1952                vpkt->vpkt_tgt_pkt = pktp;

1954                /*
1955                 * Save pkt_tgt_init_pkt fields if deferred binding
1956                 * is needed or for other purposes.
1957                 */
1958                vpkt->vpkt_tgt_init_pkt_flags = flags;
1959                vpkt->vpkt_flags = (callback == NULL_FUNC) ? CFLAG_NOWAIT : 0;
1960                vpkt->vpkt_state = VHCI_PKT_IDLE;
1961                vpkt->vpkt_tgt_init_cdblen = cmdlen;
1962                vpkt->vpkt_tgt_init_scblen = statuslen;
1963                newpkt = 1;
1964        } else { /* pkt not NULL */
1965                vpkt = pkt->pkt_ha_private;
1966        }

1968        VHCI_DEBUG(8, (CE_NOTE, NULL, "vhci_scsi_init_pkt "
1969            "vpkt %p flags %x\n", (void *)vpkt, flags));

1971        /* Clear any stale error flags */
1972        if (bp) {
1973                bioerror(bp, 0);
1974        }
```

```
1976         vpkt->vpkt_tgt_init_bp = bp;

1978         if (flags & PKT_DMA_PARTIAL) {

1980                 /*
1981                  * Immediate binding is needed.
1982                  * Target driver may not set this flag in next invocation.
1983                  * vhci has to remember this flag was set during first
1984                  * invocation of vhci_scsi_init_pkt.
1985                  */
1986                 vpkt->vpkt_flags |= CFLAG_DMA_PARTIAL;
1987         }

1989         if (vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) {

1991                 /*
1992                  * Re-initialize some of the target driver packet state
1993                  * information.
1994                  */
1995                 vpkt->vpkt_tgt_pkt->pkt_state = 0;
1996                 vpkt->vpkt_tgt_pkt->pkt_statistics = 0;
1997                 vpkt->vpkt_tgt_pkt->pkt_reason = 0;

1999                 /*
2000                  * Binding a vpkt->vpkt_path for this IO at init_time.
2001                  * If an IO error happens later, target driver will clear
2002                  * this vpkt->vpkt_path binding before re-init IO again.
2003                  */
2004                 VHCI_DEBUG(8, (CE_NOTE, NULL,
2005                     "vhci_scsi_init_pkt: calling v_b_t %p, newpkt %d\n",
2006                     (void *)vpkt, newpkt));
2007                 if (pkt && vpkt->vpkt_hba_pkt) {
2008                         VHCI_DEBUG(4, (CE_NOTE, NULL,
2009                             "v_s_i_p calling update_pHCI_pkt resid %ld\n",
2010                             pkt->pkt_resid));
2011                         vhci_update_pHCI_pkt(vpkt, pkt);
2012                 }
2013                 if (callback == SLEEP_FUNC) {
2014                         rval = vhci_bind_transport(
2015                             ap, vpkt, flags, callback);
2016                 } else {
2017                         rval = vhci_bind_transport(
2018                             ap, vpkt, flags, NULL_FUNC);
2019                 }
2020                 VHCI_DEBUG(8, (CE_NOTE, NULL,
2021                     "vhci_scsi_init_pkt: v_b_t called 0x%p rval 0x%x\n",
2022                     (void *)vpkt, rval));
2023                 if (bp) {
2024                         if (rval == TRAN_FATAL_ERROR) {
2025                                 /*
2026                                  * No paths available. Could not bind
2027                                  * any pHCI. Setting EFAULT as a way
2028                                  * to indicate no DMA is mapped.
2029                                  */
2030                                 bioerror(bp, EFAULT);
2031                         } else {
2032                                 /*
2033                                  * Do not indicate any pHCI errors to
2034                                  * target driver otherwise.
2035                                  */
2036                                 bioerror(bp, 0);
2037                         }
2038                 }
2039                 if (rval != TRAN_ACCEPT) {
2040                         VHCI_DEBUG(8, (CE_NOTE, NULL,
```



```
2041                             "vhci_scsi_init_pkt: "
2042                             "v_b_t failed 0x%p newpkt %x\n",
2043                             (void *)vpkt, newpkt));
2044                         if (newpkt) {
2045                                 scsi_hba_pkt_free(ap,
2046                                     vpkt->vpkt_tgt_pkt);
2047                         }
2048                         return (NULL);
2049                 }
2050                 ASSERT(vpkt->vpkt_hba_pkt != NULL);
2051                 ASSERT(vpkt->vpkt_path != NULL);

2053                 /* Update the resid for the target driver */
2054                 vpkt->vpkt_tgt_pkt->pkt_resid =
2055                     vpkt->vpkt_hba_pkt->pkt_resid;
2056         }

2058         return (vpkt->vpkt_tgt_pkt);
2059 }

2061 /*
2062  * Function name : vhci_scsi_destroy_pkt
2063  *
2064  * Return Values : none
2065  */
2066 static void
2067 vhci_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
2068 {
2069         struct vhci_pkt         *vpkt = (struct vhci_pkt *)pkt->pkt_ha_private;

2071         VHCI_DEBUG(8, (CE_NOTE, NULL,
2072             "vhci_scsi_destroy_pkt: vpkt 0x%p\n", (void *)vpkt));

2074         vpkt->vpkt_tgt_init_pkt_flags = 0;
2075         if (vpkt->vpkt_hba_pkt) {
2076                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
2077                 vpkt->vpkt_hba_pkt = NULL;
2078         }
2079         if (vpkt->vpkt_path) {
2080                 mdi_rele_path(vpkt->vpkt_path);
2081                 vpkt->vpkt_path = NULL;
2082         }

2084         ASSERT(vpkt->vpkt_state != VHCI_PKT_ISSUED);
2085         scsi_hba_pkt_free(ap, vpkt->vpkt_tgt_pkt);
2086 }

2088 /*
2089  * Function name : vhci_scsi_dmafree()
2090  *
2091  * Return Values : none
2092  */
2093 /*ARGSUSED*/
2094 static void
2095 vhci_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
2096 {
2097         struct vhci_pkt *vpkt = (struct vhci_pkt *)pkt->pkt_ha_private;

2099         VHCI_DEBUG(6, (CE_NOTE, NULL,
2100             "vhci_scsi_dmafree: vpkt 0x%p\n", (void *)vpkt));

2102         ASSERT(vpkt != NULL);
2103         if (vpkt->vpkt_hba_pkt) {
2104                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
2105                 vpkt->vpkt_hba_pkt = NULL;
2106         }
```

```
2107         if (vpkt->vpkt_path) {
2108                 mdi_rele_path(vpkt->vpkt_path);
2109                 vpkt->vpkt_path = NULL;
2110         }
2111 }

2113 /*
2114  * Function name : vhci_scsi_sync_pkt()
2115  *
2116  * Return Values : none
2117  */
2118 /*ARGSUSED*/
2119 static void
2120 vhci_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
2121 {
2122         struct vhci_pkt *vpkt = (struct vhci_pkt *)pkt->pkt_ha_private;

2124         ASSERT(vpkt != NULL);
2125         if (vpkt->vpkt_hba_pkt) {
2126                 scsi_sync_pkt(vpkt->vpkt_hba_pkt);
2127         }
2128 }

2130 /*
2131  * routine for reset notification setup, to register or cancel.
2132  */
2133 static int
2134 vhci_scsi_reset_notify(struct scsi_address *ap, int flag,
2135     void (*callback)(caddr_t), caddr_t arg)
2136 {
2137         struct scsi_vhci *vhci = ADDR2VHCI(ap);
2138         return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
2139             &vhci->vhci_mutex, &vhci->vhci_reset_notify_listf));
2140 }

2142 static int
2143 vhci_scsi_get_name_bus_addr(struct scsi_device *sd,
2144     char *name, int len, int bus_addr)
2145 {
2146         dev_info_t              *cdip;
2147         char                    *guid;
2148         scsi_vhci_lun_t         *vlun;

2150         ASSERT(sd != NULL);
2151         ASSERT(name != NULL);

2153         *name = 0;
2154         cdip = sd->sd_dev;

2156         ASSERT(cdip != NULL);

2158         if (mdi_component_is_client(cdip, NULL) != MDI_SUCCESS)
2159                 return (1);

2161         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, cdip, PROPFLAGS,
2162             MDI_CLIENT_GUID_PROP, &guid) != DDI_SUCCESS)
2163                 return (1);

2165         /*
2166          * Message is "sd# at scsi_vhci0: unit-address <guid>: <bus_addr>".
2167          *      <guid>          bus_addr argument == 0
2168          *      <bus_addr>      bus_addr argument != 0
2169          * Since the <guid> is already provided with unit-address, we just
2170          * provide failover module in <bus_addr> to keep output shorter.
2171          */
2172         vlun = ADDR2VLUN(&sd->sd_address);
```



```
2173         if (bus_addr == 0) {
2174                 /* report the guid:  */
2175                 (void) snprintf(name, len, "g%s", guid);
2176         } else if (vlun && vlun->svl_fops_name) {
2177                 /* report the name of the failover module */
2178                 (void) snprintf(name, len, "%s", vlun->svl_fops_name);
2179         }

2181         ddi_prop_free(guid);
2182         return (1);
2183 }

2185 static int
2186 vhci_scsi_get_bus_addr(struct scsi_device *sd, char *name, int len)
2187 {
2188         return (vhci_scsi_get_name_bus_addr(sd, name, len, 1));
2189 }

2191 static int
2192 vhci_scsi_get_name(struct scsi_device *sd, char *name, int len)
2193 {
2194         return (vhci_scsi_get_name_bus_addr(sd, name, len, 0));
2195 }

2197 /*
2198  * Return a pointer to the guid part of the devnm.
2199  * devnm format is "nodename@busaddr", busaddr format is "gGUID".
2200  */
2201 static char *
2202 vhci_devnm_to_guid(char *devnm)
2203 {
2204         char *cp = devnm;

2206         if (devnm == NULL)
2207                 return (NULL);

2209         while (*cp != '\0' && *cp != '@')
2210                 cp++;
2211         if (*cp == '@' && *(cp + 1) == 'g')
2212                 return (cp + 2);
2213         return (NULL);
2214 }

2216 static int
2217 vhci_bind_transport(struct scsi_address *ap, struct vhci_pkt *vpkt, int flags,
2218     int (*func)(caddr_t))
2219 {
2220         struct scsi_vhci        *vhci = ADDR2VHCI(ap);
2221         dev_info_t              *cdip = ADDR2DIP(ap);
2222         mdi_pathinfo_t          *pip = NULL;
2223         mdi_pathinfo_t          *npip = NULL;
2224         scsi_vhci_priv_t        *svp = NULL;
2225         struct scsi_device      *psd = NULL;
2226         struct scsi_address     *address = NULL;
2227         struct scsi_pkt         *pkt = NULL;
2228         int                     rval = -1;
2229         int                     pgr_sema_held = 0;
2230         int                     held;
2231         int                     mps_flag = MDI_SELECT_ONLINE_PATH;
2232         struct scsi_vhci_lun    *vlun;
  24         time_t                  tnow;
2233         int                     path_instance = 0;

2235         vlun = ADDR2VLUN(ap);
2236         ASSERT(vlun != 0);
```

```
2238         if ((vpkt->vpkt_tgt_pkt->pkt_cdbp[0] == SCMD_PROUT) &&
2239             (((vpkt->vpkt_tgt_pkt->pkt_cdbp[1] & 0x1f) ==
2240             VHCI_PROUT_REGISTER) ||
2241             ((vpkt->vpkt_tgt_pkt->pkt_cdbp[1] & 0x1f) ==
2242             VHCI_PROUT_R_AND_IGNORE))) {
2243                 if (!sema_tryp(&vlun->svl_pgr_sema))
2244                         return (TRAN_BUSY);
2245                 pgr_sema_held = 1;
2246                 if (vlun->svl_first_path != NULL) {
2247                         rval = mdi_select_path(cdip, NULL,
2248                             MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH,
2249                             NULL, &pip);
2250                         if ((rval != MDI_SUCCESS) || (pip == NULL)) {
2251                                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2252                                     "vhci_bind_transport: path select fail\n"));
2253                         } else {
2254                                 npip = pip;
2255                                 do {
2256                                         if (npip == vlun->svl_first_path) {
2257                                                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2258                                                     "vhci_bind_transport: "
2259                                                     "valid first path 0x%p\n",
2260                                                     (void *)
2261                                                     vlun->svl_first_path));
2262                                                 pip = vlun->svl_first_path;
2263                                                 goto bind_path;
2264                                         }
2265                                         pip = npip;
2266                                         rval = mdi_select_path(cdip, NULL,
2267                                             MDI_SELECT_ONLINE_PATH |
2268                                             MDI_SELECT_STANDBY_PATH,
2269                                             pip, &npip);
2270                                         mdi_rele_path(pip);
2271                                 } while ((rval == MDI_SUCCESS) &&
2272                                     (npip != NULL));
2273                         }
2274                 }

2276                 if (vlun->svl_first_path) {
2277                         VHCI_DEBUG(4, (CE_NOTE, NULL,
2278                             "vhci_bind_transport: invalid first path 0x%p\n",
2279                             (void *)vlun->svl_first_path));
2280                         vlun->svl_first_path = NULL;
2281                 }
2282         } else if (vpkt->vpkt_tgt_pkt->pkt_cdbp[0] == SCMD_PRIN) {
2283                 if ((vpkt->vpkt_state & VHCI_PKT_THRU_TASKQ) == 0) {
2284                         if (!sema_tryp(&vlun->svl_pgr_sema))
2285                                 return (TRAN_BUSY);
2286                 }
2287                 pgr_sema_held = 1;
2288         }

2290         /*
2291          * If the path is already bound for PKT_PARTIAL_DMA case,
2292          * try to use the same path.
2293          */
2294         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) && vpkt->vpkt_path) {
2295                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2296                     "vhci_bind_transport: PKT_PARTIAL_DMA "
2297                     "vpkt 0x%p, path 0x%p\n",
2298                     (void *)vpkt, (void *)vpkt->vpkt_path));
2299                 pip = vpkt->vpkt_path;
2300                 goto bind_path;
2301         }

2303         /*
```



```
2304          * Get path_instance. Non-zero with FLAG_PKT_PATH_INSTANCE set
2305          * indicates that mdi_select_path should be called to select a
2306          * specific instance.
2307          *
2308          * NB: Condition pkt_path_instance reference on proper allocation.
2309          */
2310         if ((vpkt->vpkt_tgt_pkt->pkt_flags & FLAG_PKT_PATH_INSTANCE) &&
2311             scsi_pkt_allocated_correctly(vpkt->vpkt_tgt_pkt)) {
2312                 path_instance = vpkt->vpkt_tgt_pkt->pkt_path_instance;
2313         }

2315         /*
2316          * If reservation is active bind the transport directly to the pip
2317          * with the reservation.
2318          */
2319         if (vpkt->vpkt_hba_pkt == NULL) {
2320                 if (vlun->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
2321                         if (MDI_PI_IS_ONLINE(vlun->svl_resrv_pip)) {
2322                                 pip = vlun->svl_resrv_pip;
2323                                 mdi_hold_path(pip);
2324                                 vlun->svl_waiting_for_activepath = 0;
2325                                 rval = MDI_SUCCESS;
2326                                 goto bind_path;
2327                         } else {
2328                                 if (pgr_sema_held) {
2329                                         sema_v(&vlun->svl_pgr_sema);
2330                                 }
2331                                 return (TRAN_BUSY);
2332                         }
2333                 }
2334 try_again:
2335                 rval = mdi_select_path(cdip, vpkt->vpkt_tgt_init_bp,
2336                     path_instance ? MDI_SELECT_PATH_INSTANCE : 0,
2337                     (void *)(intptr_t)path_instance, &pip);
2338                 if (rval == MDI_BUSY) {
2339                         if (pgr_sema_held) {
2340                                 sema_v(&vlun->svl_pgr_sema);
2341                         }
2342                         return (TRAN_BUSY);
2343                 } else if (rval == MDI_DEVI_ONLINING) {
2344                         /*
2345                          * if we are here then we are in the midst of
2346                          * an attach/probe of the client device.
2347                          * We attempt to bind to ONLINE path if available,
2348                          * else it is OK to bind to a STANDBY path (instead
2349                          * of triggering a failover) because IO associated
2350                          * with attach/probe (eg. INQUIRY, block 0 read)
2351                          * are completed by targets even on passive paths
2352                          * If no ONLINE paths available, it is important
2353                          * to set svl_waiting_for_activepath for two
2354                          * reasons: (1) avoid sense analysis in the
2355                          * "external failure detection" codepath in
2356                          * vhci_intr().  Failure to do so will result in
2357                          * infinite loop (unless an ONLINE path becomes
2358                          * available at some point) (2) avoid
2359                          * unnecessary failover (see "---Waiting For Active
2360                          * Path---" comment below).
2361                          */
2362                         VHCI_DEBUG(1, (CE_NOTE, NULL, "!%p in onlining "
2363                             "state\n", (void *)cdip));
2364                         pip = NULL;
2365                         rval = mdi_select_path(cdip, vpkt->vpkt_tgt_init_bp,
2366                             mps_flag, NULL, &pip);
2367                         if ((rval != MDI_SUCCESS) || (pip == NULL)) {
2368                                 if (vlun->svl_waiting_for_activepath == 0) {
2369                                         vlun->svl_waiting_for_activepath = 1;
```

```
2370                                    vlun->svl_wfa_time = gethrtime();
 162                                    vlun->svl_wfa_time = ddi_get_time();
2371                            }
2372                            mps_flag |= MDI_SELECT_STANDBY_PATH;
2373                            rval = mdi_select_path(cdip,
2374                                vpkt->vpkt_tgt_init_bp,
2375                                mps_flag, NULL, &pip);
2376                            if ((rval != MDI_SUCCESS) || (pip == NULL)) {
2377                                    if (pgr_sema_held) {
2378                                            sema_v(&vlun->svl_pgr_sema);
2379                                    }
2380                                    return (TRAN_FATAL_ERROR);
2381                            }
2382                            goto bind_path;
2383                    }
2384            } else if ((rval == MDI_FAILURE) ||
2385                ((rval == MDI_NOPATH) && (path_instance))) {
2386                    if (pgr_sema_held) {
2387                            sema_v(&vlun->svl_pgr_sema);
2388                    }
2389                    return (TRAN_FATAL_ERROR);
2390            }

2392             if ((pip == NULL) || (rval == MDI_NOPATH)) {
2393                    while (vlun->svl_waiting_for_activepath) {
2394                            /*
2395                             * ---Waiting For Active Path---
2396                             * This device was discovered across a
2397                             * passive path; lets wait for a little
2398                             * bit, hopefully an active path will
2399                             * show up obviating the need for a
2400                             * failover
2401                             */
2402                            if ((gethrtime() - vlun->svl_wfa_time) >=
2403                                (60 * NANOSEC)) {
 194                            tnow = ddi_get_time();
 195                            if (tnow - vlun->svl_wfa_time >= 60) {
2404                                    vlun->svl_waiting_for_activepath = 0;
2405                            } else {
2406                                    drv_usecwait(1000);
2407                                    if (vlun->svl_waiting_for_activepath
2408                                        == 0) {
2409                                            /*
2410                                             * an active path has come
2411                                             * online!
2412                                             */
2413                                            goto try_again;
2414                                    }
2415                            }
2416                    }
2417                    VHCI_HOLD_LUN(vlun, VH_NOSLEEP, held);
2418                    if (!held) {
2419                            VHCI_DEBUG(4, (CE_NOTE, NULL,
2420                                "!Lun not held\n"));
2421                            if (pgr_sema_held) {
2422                                    sema_v(&vlun->svl_pgr_sema);
2423                            }
2424                            return (TRAN_BUSY);
2425                    }
2426                    /*
2427                     * now that the LUN is stable, one last check
2428                     * to make sure no other changes sneaked in
2429                     * (like a path coming online or a
2430                     * failover initiated by another thread)
2431                     */
2432                    pip = NULL;
```



```
2433                            rval = mdi_select_path(cdip, vpkt->vpkt_tgt_init_bp,
2434                                0, NULL, &pip);
2435                            if (pip != NULL) {
2436                                    VHCI_RELEASE_LUN(vlun);
2437                                    vlun->svl_waiting_for_activepath = 0;
2438                                    goto bind_path;
2439                            }

2441                            /*
2442                             * Check if there is an ONLINE path OR a STANDBY path
2443                             * available. If none is available, do not attempt
2444                             * to do a failover, just return a fatal error at this
2445                             * point.
2446                             */
2447                            npip = NULL;
2448                            rval = mdi_select_path(cdip, NULL,
2449                                (MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH),
2450                                NULL, &npip);
2451                            if ((npip == NULL) || (rval != MDI_SUCCESS)) {
2452                                    /*
2453                                     * No paths available, jus return FATAL error.
2454                                     */
2455                                    VHCI_RELEASE_LUN(vlun);
2456                                    if (pgr_sema_held) {
2457                                            sema_v(&vlun->svl_pgr_sema);
2458                                    }
2459                                    return (TRAN_FATAL_ERROR);
2460                            }
2461                            mdi_rele_path(npip);
2462                            if (!(vpkt->vpkt_state & VHCI_PKT_IN_FAILOVER)) {
2463                                    VHCI_DEBUG(1, (CE_NOTE, NULL, "!invoking "
2464                                        "mdi_failover\n"));
2465                                    rval = mdi_failover(vhci->vhci_dip, cdip,
2466                                        MDI_FAILOVER_ASYNC);
2467                            } else {
2468                                    rval = vlun->svl_failover_status;
2469                            }
2470                            if (rval == MDI_FAILURE) {
2471                                    VHCI_RELEASE_LUN(vlun);
2472                                    if (pgr_sema_held) {
2473                                            sema_v(&vlun->svl_pgr_sema);
2474                                    }
2475                                    return (TRAN_FATAL_ERROR);
2476                            } else if (rval == MDI_BUSY) {
2477                                    VHCI_RELEASE_LUN(vlun);
2478                                    if (pgr_sema_held) {
2479                                            sema_v(&vlun->svl_pgr_sema);
2480                                    }
2481                                    return (TRAN_BUSY);
2482                            } else {
2483                                    if (pgr_sema_held) {
2484                                            sema_v(&vlun->svl_pgr_sema);
2485                                    }
2486                                    vpkt->vpkt_state |= VHCI_PKT_IN_FAILOVER;
2487                                    return (TRAN_BUSY);
2488                            }
2489                    }
2490                    vlun->svl_waiting_for_activepath = 0;
2491 bind_path:
2492                    vpkt->vpkt_path = pip;
2493                    svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(pip);
2494                    ASSERT(svp != NULL);

2496                    psd = svp->svp_psd;
2497                    ASSERT(psd != NULL);
2498                    address = &psd->sd_address;
```

```
2499         } else {
2500                 pkt = vpkt->vpkt_hba_pkt;
2501                 address = &pkt->pkt_address;
2502         }

2504         /* Verify match of specified path_instance and selected path_instance */
2505         ASSERT((path_instance == 0) ||
2506             (path_instance == mdi_pi_get_path_instance(vpkt->vpkt_path)));

2508         /*
2509          * For PKT_PARTIAL_DMA case, call pHCI's scsi_init_pkt whenever
2510          * target driver calls vhci_scsi_init_pkt.
2511          */
2512         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) &&
2513             vpkt->vpkt_path && vpkt->vpkt_hba_pkt) {
2514                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2515                     "vhci_bind_transport: PKT_PARTIAL_DMA "
2516                     "vpkt 0x%p, path 0x%p hba_pkt 0x%p\n",
2517                     (void *)vpkt, (void *)vpkt->vpkt_path, (void *)pkt));
2518                 pkt = vpkt->vpkt_hba_pkt;
2519                 address = &pkt->pkt_address;
2520         }

2522         if (pkt == NULL || (vpkt->vpkt_flags & CFLAG_DMA_PARTIAL)) {
2523                 pkt = scsi_init_pkt(address, pkt,
2524                     vpkt->vpkt_tgt_init_bp, vpkt->vpkt_tgt_init_cdblen,
2525                     vpkt->vpkt_tgt_init_scblen, 0, flags, func, NULL);

2527                 if (pkt == NULL) {
2528                         VHCI_DEBUG(4, (CE_NOTE, NULL,
2529                             "!bind transport: 0x%p 0x%p 0x%p\n",
2530                             (void *)vhci, (void *)psd, (void *)vpkt));
2531                         if ((vpkt->vpkt_hba_pkt == NULL) && vpkt->vpkt_path) {
2532                                 MDI_PI_ERRSTAT(vpkt->vpkt_path,
2533                                     MDI_PI_TRANSERR);
2534                                 mdi_rele_path(vpkt->vpkt_path);
2535                                 vpkt->vpkt_path = NULL;
2536                         }
2537                         if (pgr_sema_held) {
2538                                 sema_v(&vlun->svl_pgr_sema);
2539                         }
2540                         /*
2541                          * Consider it a fatal error if b_error is
2542                          * set as a result of DMA binding failure
2543                          * vs. a condition of being temporarily out of
2544                          * some resource
2545                          */
2546                         if (vpkt->vpkt_tgt_init_bp == NULL ||
2547                             geterror(vpkt->vpkt_tgt_init_bp))
2548                                 return (TRAN_FATAL_ERROR);
2549                         else
2550                                 return (TRAN_BUSY);
2551                 }
2552         }

2554         pkt->pkt_private = vpkt;
2555         vpkt->vpkt_hba_pkt = pkt;
2556         return (TRAN_ACCEPT);
2557 }
```

*unchanged_portion_omitted_*

```
3571 /*
3572  * two possibilities: (1) failover has completed
3573  * or (2) is in progress; update our path states for
3574  * the former case; for the latter case,
3575  * initiate a scsi_watch request to
```



```
3576  * determine when failover completes - vlun is HELD
3577  * until failover completes; BUSY is returned to upper
3578  * layer in both the cases
3579  */
3580 static int
3581 vhci_handle_ext_fo(struct scsi_pkt *pkt, int fostat)
3582 {
3583         struct vhci_pkt         *vpkt = (struct vhci_pkt *)pkt->pkt_private;
3584         struct scsi_pkt         *tpkt;
3585         scsi_vhci_priv_t        *svp;
3586         scsi_vhci_lun_t         *vlun;
3587         struct scsi_vhci        *vhci;
3588         scsi_vhci_swarg_t       *swarg;
3589         char                    *path;

3591         ASSERT(vpkt != NULL);
3592         tpkt = vpkt->vpkt_tgt_pkt;
3593         ASSERT(tpkt != NULL);
3594         svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(vpkt->vpkt_path);
3595         ASSERT(svp != NULL);
3596         vlun = svp->svp_svl;
3597         ASSERT(vlun != NULL);
3598         ASSERT(VHCI_LUN_IS_HELD(vlun));

3600         vhci = ADDR2VHCI(&tpkt->pkt_address);

3602         if (fostat == SCSI_SENSE_INACTIVE) {
3603                 VHCI_DEBUG(1, (CE_NOTE, NULL, "!Failover "
3604                     "detected for %s; updating path states...\n",
3605                     vlun->svl_lun_wwn));
3606                 /*
3607                  * set the vlun flag to indicate to the task that the target
3608                  * port group needs updating
3609                  */
3610                 vlun->svl_flags |= VLUN_UPDATE_TPG;
3611                 (void) taskq_dispatch(vhci->vhci_update_pathstates_taskq,
3612                     vhci_update_pathstates, (void *)vlun, KM_SLEEP);
3613         } else {
3614                 path = kmem_alloc(MAXPATHLEN, KM_SLEEP);
3615                 vhci_log(CE_NOTE, ddi_get_parent(vlun->svl_dip),
3616                     "!%s (%s%d): Waiting for externally initiated failover "
3617                     "to complete", ddi_pathname(vlun->svl_dip, path),
3618                     ddi_driver_name(vlun->svl_dip),
3619                     ddi_get_instance(vlun->svl_dip));
3620                 kmem_free(path, MAXPATHLEN);
3621                 swarg = kmem_alloc(sizeof (*swarg), KM_NOSLEEP);
3622                 if (swarg == NULL) {
3623                         VHCI_DEBUG(1, (CE_NOTE, NULL, "!vhci_handle_ext_fo: "
3624                             "request packet allocation for %s failed....\n",
3625                             vlun->svl_lun_wwn));
3626                         VHCI_RELEASE_LUN(vlun);
3627                         return (PKT_RETURN);
3628                 }
3629                 swarg->svs_svp = svp;
3630                 swarg->svs_tos = gethrtime();
1422                 swarg->svs_tos = ddi_get_time();
3631                 swarg->svs_pi = vpkt->vpkt_path;
3632                 swarg->svs_release_lun = 0;
3633                 swarg->svs_done = 0;
3634                 /*
3635                  * place a hold on the path...we don't want it to
3636                  * vanish while scsi_watch is in progress
3637                  */
3638                 mdi_hold_path(vpkt->vpkt_path);
3639                 svp->svp_sw_token = scsi_watch_request_submit(svp->svp_psd,
3640                     VHCI_FOWATCH_INTERVAL, SENSE_LENGTH, vhci_efo_watch_cb,
```

```
3641                    (caddr_t)swarg);
3642        }
3643        return (BUSY_RETURN);
3644 }

3646 /*
3647  * vhci_efo_watch_cb:
3648  *      Callback from scsi_watch request to check the failover status.
3649  *      Completion is either due to successful failover or timeout.
3650  *      Upon successful completion, vhci_update_path_states is called.
3651  *      For timeout condition, vhci_efo_done is called.
3652  *      Always returns 0 to scsi_watch to keep retrying till vhci_efo_done
3653  *      terminates this request properly in a separate thread.
3654  */

3656 static int
3657 vhci_efo_watch_cb(caddr_t arg, struct scsi_watch_result *resultp)
3658 {
3659        struct scsi_status              *statusp = resultp->statusp;
3660        uint8_t                         *sensep = (uint8_t *)resultp->sensep;
3661        struct scsi_pkt                 *pkt = resultp->pkt;
3662        scsi_vhci_swarg_t               *swarg;
3663        scsi_vhci_priv_t                *svp;
3664        scsi_vhci_lun_t                 *vlun;
3665        struct scsi_vhci                *vhci;
3666        dev_info_t                      *vdip;
3667        int                             rval, updt_paths;

3669        swarg = (scsi_vhci_swarg_t *)(uintptr_t)arg;
3670        svp = swarg->svs_svp;
3671        if (swarg->svs_done) {
3672                /*
3673                 * Already completed failover or timedout.
3674                 * Waiting for vhci_efo_done to terminate this scsi_watch.
3675                 */
3676                return (0);
3677        }

3679        ASSERT(svp != NULL);
3680        vlun = svp->svp_svl;
3681        ASSERT(vlun != NULL);
3682        ASSERT(VHCI_LUN_IS_HELD(vlun));
3683        vlun->svl_efo_update_path = 0;
3684        vdip = ddi_get_parent(vlun->svl_dip);
3685        vhci = ddi_get_soft_state(vhci_softstate,
3686            ddi_get_instance(vdip));

3688        updt_paths = 0;

3690        if (pkt->pkt_reason != CMD_CMPLT) {
3691                if ((gethrtime() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
1483                if ((ddi_get_time() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
3692                        swarg->svs_release_lun = 1;
3693                        goto done;
3694                }
3695                return (0);
3696        }
3697        if (*((unsigned char *)statusp) == STATUS_CHECK) {
3698                rval = vlun->svl_fops->sfo_analyze_sense(svp->svp_psd, sensep,
3699                    vlun->svl_fops_ctpriv);
3700                switch (rval) {
3701                        /*
3702                         * Only update path states in case path is definitely
3703                         * inactive, or no failover occurred.  For all other
3704                         * check conditions continue pinging.  A unexpected
3705                         * check condition shouldn't cause pinging to complete
```



```
3706                         * prematurely.
3707                         */
3708                        case SCSI_SENSE_INACTIVE:
3709                        case SCSI_SENSE_NOFAILOVER:
3710                                updt_paths = 1;
3711                                break;
3712                        default:
3713                                if ((gethrtime() - swarg->svs_tos)
1505                                if ((ddi_get_time() - swarg->svs_tos)
3714                                    >= VHCI_EXTFO_TIMEOUT) {
3715                                        swarg->svs_release_lun = 1;
3716                                        goto done;
3717                                }
3718                                return (0);
3719                        }
3720                } else if (*((unsigned char *)statusp) ==
3721                    STATUS_RESERVATION_CONFLICT) {
3722                        updt_paths = 1;
3723                } else if ((*((unsigned char *)statusp)) &
3724                    (STATUS_BUSY | STATUS_QFULL)) {
3725                        return (0);
3726                }
3727                if ((*((unsigned char *)statusp) == STATUS_GOOD) ||
3728                    (updt_paths == 1)) {
3729                        /*
3730                         * we got here because we had detected an
3731                         * externally initiated failover; things
3732                         * have settled down now, so let's
3733                         * start up a task to update the
3734                         * path states and target port group
3735                         */
3736                        vlun->svl_efo_update_path = 1;
3737                        swarg->svs_done = 1;
3738                        vlun->svl_swarg = swarg;
3739                        vlun->svl_flags |= VLUN_UPDATE_TPG;
3740                        (void) taskq_dispatch(vhci->vhci_update_pathstates_taskq,
3741                            vhci_update_pathstates, (void *)vlun,
3742                            KM_SLEEP);
3743                        return (0);
3744                }
3745                if ((gethrtime() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
1537                if ((ddi_get_time() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
3746                        swarg->svs_release_lun = 1;
3747                        goto done;
3748                }
3749                return (0);
3750 done:
3751                swarg->svs_done = 1;
3752                (void) taskq_dispatch(vhci->vhci_taskq,
3753                    vhci_efo_done, (void *)swarg, KM_SLEEP);
3754                return (0);
3755 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  911463 Mon May  5 14:29:44 2014
new/usr/src/uts/common/io/scsi/targets/sd.c
4781 sd shouldn't abuse ddi_get_time(9f)
Reviewed by: Richard Elling <richard.elling@gmail.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

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

/*
 * Copyright (c) 1990, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright (c) 2011 Bayard G. Bell.  All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2012 DEY Storage Systems, Inc.  All rights reserved.
 */
/*
 * Copyright 2011 cyril.galibern@opensvc.com
 */

/*
 * SCSI disk target driver.
 */
#include <sys/scsi/scsi.h>
#include <sys/dkbad.h>
#include <sys/dklabel.h>
#include <sys/dkio.h>
#include <sys/fdio.h>
#include <sys/cdio.h>
#include <sys/mhd.h>
#include <sys/vtoc.h>
#include <sys/dktp/fdisk.h>
#include <sys/kstat.h>
#include <sys/vtrace.h>
#include <sys/note.h>
#include <sys/thread.h>
#include <sys/proc.h>
#include <sys/efi_partition.h>
#include <sys/var.h>
#include <sys/aio_req.h>

#ifdef __lock_lint
#define _LP64
#define __amd64
```



```
#endif

#if (defined(__fibre))
/* Note: is there a leadville version of the following? */
#include <sys/fc4/fcal_linkapp.h>
#endif
#include <sys/taskq.h>
#include <sys/uuid.h>
#include <sys/byteorder.h>
#include <sys/sdt.h>

#include "sd_xbuf.h"

#include <sys/scsi/targets/sddef.h>
#include <sys/cmlb.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/dev.h>

#include <sys/fm/protocol.h>

/*
 * Loadable module info.
 */
#if (defined(__fibre))
#define SD_MODULE_NAME  "SCSI SSA/FCAL Disk Driver"
#else /* !__fibre */
#define SD_MODULE_NAME  "SCSI Disk Driver"
#endif /* !__fibre */

/*
 * Define the interconnect type, to allow the driver to distinguish
 * between parallel SCSI (sd) and fibre channel (ssd) behaviors.
 *
 * This is really for backward compatibility. In the future, the driver
 * should actually check the "interconnect-type" property as reported by
 * the HBA; however at present this property is not defined by all HBAs,
 * so we will use this #define (1) to permit the driver to run in
 * backward-compatibility mode; and (2) to print a notification message
 * if an FC HBA does not support the "interconnect-type" property.  The
 * behavior of the driver will be to assume parallel SCSI behaviors unless
 * the "interconnect-type" property is defined by the HBA **AND** has a
 * value of either INTERCONNECT_FIBRE, INTERCONNECT_SSA, or
 * INTERCONNECT_FABRIC, in which case the driver will assume Fibre
 * Channel behaviors (as per the old ssd).  (Note that the
 * INTERCONNECT_1394 and INTERCONNECT_USB types are not supported and
 * will result in the driver assuming parallel SCSI behaviors.)
 *
 * (see common/sys/scsi/impl/services.h)
 *
 * Note: For ssd semantics, don't use INTERCONNECT_FABRIC as the default
 * since some FC HBAs may already support that, and there is some code in
 * the driver that already looks for it.  Using INTERCONNECT_FABRIC as the
 * default would confuse that code, and besides things should work fine
 * anyways if the FC HBA already reports INTERCONNECT_FABRIC for the
 * "interconnect_type" property.
 *
 */
#if (defined(__fibre))
#define SD_DEFAULT_INTERCONNECT_TYPE    SD_INTERCONNECT_FIBRE
#else
#define SD_DEFAULT_INTERCONNECT_TYPE    SD_INTERCONNECT_PARALLEL
#endif

/*
 * The name of the driver, established from the module name in _init.
 */
```

```
125 static  char *sd_label                 = NULL;

127 /*
128  * Driver name is unfortunately prefixed on some driver.conf properties.
129  */
130 #if (defined(__fibre))
131 #define sd_max_xfer_size              ssd_max_xfer_size
132 #define sd_config_list                ssd_config_list
133 static  char *sd_max_xfer_size        = "ssd_max_xfer_size";
134 static  char *sd_config_list          = "ssd-config-list";
135 #else
136 static  char *sd_max_xfer_size        = "sd_max_xfer_size";
137 static  char *sd_config_list          = "sd-config-list";
138 #endif

140 /*
141  * Driver global variables
142  */

144 #if (defined(__fibre))
145 /*
146  * These #defines are to avoid namespace collisions that occur because this
147  * code is currently used to compile two separate driver modules: sd and ssd.
148  * All global variables need to be treated this way (even if declared static)
149  * in order to allow the debugger to resolve the names properly.
150  * It is anticipated that in the near future the ssd module will be obsoleted,
151  * at which time this namespace issue should go away.
152  */
153 #define sd_state                      ssd_state
154 #define sd_io_time                    ssd_io_time
155 #define sd_failfast_enable            ssd_failfast_enable
156 #define sd_ua_retry_count             ssd_ua_retry_count
157 #define sd_report_pfa                 ssd_report_pfa
158 #define sd_max_throttle               ssd_max_throttle
159 #define sd_min_throttle               ssd_min_throttle
160 #define sd_rot_delay                  ssd_rot_delay

162 #define sd_retry_on_reservation_conflict        \
163                                       ssd_retry_on_reservation_conflict
164 #define sd_reinstate_resv_delay       ssd_reinstate_resv_delay
165 #define sd_resv_conflict_name         ssd_resv_conflict_name

167 #define sd_component_mask             ssd_component_mask
168 #define sd_level_mask                 ssd_level_mask
169 #define sd_debug_un                   ssd_debug_un
170 #define sd_error_level                ssd_error_level

172 #define sd_xbuf_active_limit          ssd_xbuf_active_limit
173 #define sd_xbuf_reserve_limit         ssd_xbuf_reserve_limit

175 #define sd_tr                         ssd_tr
176 #define sd_reset_throttle_timeout     ssd_reset_throttle_timeout
177 #define sd_qfull_throttle_timeout     ssd_qfull_throttle_timeout
178 #define sd_qfull_throttle_enable      ssd_qfull_throttle_enable
179 #define sd_check_media_time           ssd_check_media_time
180 #define sd_wait_cmds_complete         ssd_wait_cmds_complete
181 #define sd_label_mutex                ssd_label_mutex
182 #define sd_detach_mutex               ssd_detach_mutex
183 #define sd_log_buf                    ssd_log_buf
184 #define sd_log_mutex                  ssd_log_mutex

186 #define sd_disk_table                 ssd_disk_table
187 #define sd_disk_table_size            ssd_disk_table_size
188 #define sd_sense_mutex                ssd_sense_mutex
189 #define sd_cdbtab                     ssd_cdbtab
```



```
191 #define sd_cb_ops                     ssd_cb_ops
192 #define sd_ops                        ssd_ops
193 #define sd_additional_codes           ssd_additional_codes
194 #define sd_tgops                      ssd_tgops

196 #define sd_minor_data                 ssd_minor_data
197 #define sd_minor_data_efi             ssd_minor_data_efi

199 #define sd_tq                         ssd_tq
200 #define sd_wmr_tq                     ssd_wmr_tq
201 #define sd_taskq_name                 ssd_taskq_name
202 #define sd_wmr_taskq_name             ssd_wmr_taskq_name
203 #define sd_taskq_minalloc             ssd_taskq_minalloc
204 #define sd_taskq_maxalloc             ssd_taskq_maxalloc

206 #define sd_dump_format_string         ssd_dump_format_string

208 #define sd_iostart_chain              ssd_iostart_chain
209 #define sd_iodone_chain               ssd_iodone_chain

211 #define sd_pm_idletime                ssd_pm_idletime

213 #define sd_force_pm_supported         ssd_force_pm_supported

215 #define sd_dtype_optical_bind         ssd_dtype_optical_bind

217 #define sd_ssc_init                   ssd_ssc_init
218 #define sd_ssc_send                   ssd_ssc_send
219 #define sd_ssc_fini                   ssd_ssc_fini
220 #define sd_ssc_assessment             ssd_ssc_assessment
221 #define sd_ssc_post                   ssd_ssc_post
222 #define sd_ssc_print                  ssd_ssc_print
223 #define sd_ssc_ereport_post           ssd_ssc_ereport_post
224 #define sd_ssc_set_info               ssd_ssc_set_info
225 #define sd_ssc_extract_info           ssd_ssc_extract_info

227 #endif

229 #ifdef  SDDEBUG
230 int     sd_force_pm_supported         = 0;
231 #endif  /* SDDEBUG */

233 void *sd_state                        = NULL;
234 int sd_io_time                        = SD_IO_TIME;
235 int sd_failfast_enable                = 1;
236 int sd_ua_retry_count                 = SD_UA_RETRY_COUNT;
237 int sd_report_pfa                     = 1;
238 int sd_max_throttle                   = SD_MAX_THROTTLE;
239 int sd_min_throttle                   = SD_MIN_THROTTLE;
240 int sd_rot_delay                      = 4; /* Default 4ms Rotation delay */
241 int sd_qfull_throttle_enable          = TRUE;

243 int sd_retry_on_reservation_conflict  = 1;
244 int sd_reinstate_resv_delay           = SD_REINSTATE_RESV_DELAY;
245 _NOTE(SCHEME_PROTECTS_DATA("safe sharing", sd_reinstate_resv_delay))

247 static int sd_dtype_optical_bind      = -1;

249 /* Note: the following is not a bug, it really is "sd_" and not "ssd_" */
250 static  char *sd_resv_conflict_name   = "sd_retry_on_reservation_conflict";

252 /*
253  * Global data for debug logging. To enable debug printing, sd_component_mask
254  * and sd_level_mask should be set to the desired bit patterns as outlined in
255  * sddef.h.
256  */
```

```
 257 uint_t  sd_component_mask               = 0x0;
 258 uint_t  sd_level_mask                   = 0x0;
 259 struct  sd_lun *sd_debug_un             = NULL;
 260 uint_t  sd_error_level                  = SCSI_ERR_RETRYABLE;

 262 /* Note: these may go away in the future... */
 263 static uint32_t sd_xbuf_active_limit    = 512;
 264 static uint32_t sd_xbuf_reserve_limit   = 16;

 266 static struct sd_resv_reclaim_request   sd_tr = { NULL, NULL, NULL, 0, 0, 0 };

 268 /*
 269  * Timer value used to reset the throttle after it has been reduced
 270  * (typically in response to TRAN_BUSY or STATUS_QFULL)
 271  */
 272 static int sd_reset_throttle_timeout    = SD_RESET_THROTTLE_TIMEOUT;
 273 static int sd_qfull_throttle_timeout    = SD_QFULL_THROTTLE_TIMEOUT;

 275 /*
 276  * Interval value associated with the media change scsi watch.
 277  */
 278 static int sd_check_media_time          = 3000000;

 280 /*
 281  * Wait value used for in progress operations during a DDI_SUSPEND
 282  */
 283 static int sd_wait_cmds_complete        = SD_WAIT_CMDS_COMPLETE;

 285 /*
 286  * sd_label_mutex protects a static buffer used in the disk label
 287  * component of the driver
 288  */
 289 static kmutex_t sd_label_mutex;

 291 /*
 292  * sd_detach_mutex protects un_layer_count, un_detach_count, and
 293  * un_opens_in_progress in the sd_lun structure.
 294  */
 295 static kmutex_t sd_detach_mutex;

 297 _NOTE(MUTEX_PROTECTS_DATA(sd_detach_mutex,
 298         sd_lun::{un_layer_count un_detach_count un_opens_in_progress}))

 300 /*
 301  * Global buffer and mutex for debug logging
 302  */
 303 static char     sd_log_buf[1024];
 304 static kmutex_t sd_log_mutex;

 306 /*
 307  * Structs and globals for recording attached lun information.
 308  * This maintains a chain. Each node in the chain represents a SCSI controller.
 309  * The structure records the number of luns attached to each target connected
 310  * with the controller.
 311  * For parallel scsi device only.
 312  */
 313 struct sd_scsi_hba_tgt_lun {
 314         struct sd_scsi_hba_tgt_lun      *next;
 315         dev_info_t                      *pdip;
 316         int                             nlun[NTARGETS_WIDE];
 317 };
```

*unchanged_portion_omitted_*

```
6523 /*
6524  *    Function: sd_pm_idletimeout_handler
```



```
6525  *
6526  * Description: A timer routine that's active only while a device is busy.
6527  *              The purpose is to extend slightly the pm framework's busy
6528  *              view of the device to prevent busy/idle thrashing for
6529  *              back-to-back commands. Do this by comparing the current time
6530  *              to the time at which the last command completed and when the
6531  *              difference is greater than sd_pm_idletime, call
6532  *              pm_idle_component. In addition to indicating idle to the pm
6533  *              framework, update the chain type to again use the internal pm
6534  *              layers of the driver.
6535  *
6536  *   Arguments: arg - driver soft state (unit) structure
6537  *
6538  *     Context: Executes in a timeout(9F) thread context
6539  */

6541 static void
6542 sd_pm_idletimeout_handler(void *arg)
6543 {
6544         const hrtime_t idletime = sd_pm_idletime * NANOSEC;
6545 #endif /* ! codereview */
6546         struct sd_lun *un = arg;

6544         time_t  now;

6548         mutex_enter(&sd_detach_mutex);
6549         if (un->un_detach_count != 0) {
6550                 /* Abort if the instance is detaching */
6551                 mutex_exit(&sd_detach_mutex);
6552                 return;
6553         }
6554         mutex_exit(&sd_detach_mutex);

6554         now = ddi_get_time();
6556         /*
6557          * Grab both mutexes, in the proper order, since we're accessing
6558          * both PM and softstate variables.
6559          */
6560         mutex_enter(SD_MUTEX(un));
6561         mutex_enter(&un->un_pm_mutex);
6562         if (((gethrtime() - un->un_pm_idle_time) > idletime) &&
6561         if (((now - un->un_pm_idle_time) > sd_pm_idletime) &&
6563             (un->un_ncmds_in_driver == 0) && (un->un_pm_count == 0)) {
6564                 /*
6565                  * Update the chain types.
6566                  * This takes affect on the next new command received.
6567                  */
6568                 if (un->un_f_non_devbsize_supported) {
6569                         un->un_buf_chain_type = SD_CHAIN_INFO_RMMEDIA;
6570                 } else {
6571                         un->un_buf_chain_type = SD_CHAIN_INFO_DISK;
6572                 }
6573                 un->un_uscsi_chain_type = SD_CHAIN_INFO_USCSI_CMD;

6575                 SD_TRACE(SD_LOG_IO_PM, un,
6576                     "sd_pm_idletimeout_handler: idling device\n");
6577                 (void) pm_idle_component(SD_DEVINFO(un), 0);
6578                 un->un_pm_idle_timeid = NULL;
6579         } else {
6580                 un->un_pm_idle_timeid =
6581                     timeout(sd_pm_idletimeout_handler, un,
6582                     (drv_usectohz((clock_t)300000))); /* 300 ms. */
6583         }
6584         mutex_exit(&un->un_pm_mutex);
6585         mutex_exit(SD_MUTEX(un));
6586 }
```

*unchanged_portion_omitted_*

```
12430 /*
12431  *    Function: sd_buf_iodone
12432  *
12433  * Description: Frees the sd_xbuf & returns the buf to its originator.
12434  *
12435  *     Context: May be called from interrupt context.
12436  */
12437 /* ARGSUSED */
12438 static void
12439 sd_buf_iodone(int index, struct sd_lun *un, struct buf *bp)
12440 {
12441         struct sd_xbuf *xp;

12443         ASSERT(un != NULL);
12444         ASSERT(bp != NULL);
12445         ASSERT(!mutex_owned(SD_MUTEX(un)));

12447         SD_TRACE(SD_LOG_IO_CORE, un, "sd_buf_iodone: entry.\n");

12449         xp = SD_GET_XBUF(bp);
12450         ASSERT(xp != NULL);

12452         /* xbuf is gone after this */
12453         if (ddi_xbuf_done(bp, un->un_xbuf_attr)) {
12454                 mutex_enter(SD_MUTEX(un));

12456                 /*
12457                  * Grab time when the cmd completed.
12458                  * This is used for determining if the system has been
12459                  * idle long enough to make it idle to the PM framework.
12460                  * This is for lowering the overhead, and therefore improving
12461                  * performance per I/O operation.
12462                  */
12463                 un->un_pm_idle_time = gethrtime();
12462                 un->un_pm_idle_time = ddi_get_time();

12465                 un->un_ncmds_in_driver--;
12466                 ASSERT(un->un_ncmds_in_driver >= 0);
12467                 SD_INFO(SD_LOG_IO, un,
12468                     "sd_buf_iodone: un_ncmds_in_driver = %ld\n",
12469                     un->un_ncmds_in_driver);

12471                 mutex_exit(SD_MUTEX(un));
12472         }

12474         biodone(bp);                            /* bp is gone after this */

12476         SD_TRACE(SD_LOG_IO_CORE, un, "sd_buf_iodone: exit.\n");
12477 }


12480 /*
12481  *    Function: sd_uscsi_iodone
12482  *
12483  * Description: Frees the sd_xbuf & returns the buf to its originator.
12484  *
12485  *     Context: May be called from interrupt context.
12486  */
12487 /* ARGSUSED */
12488 static void
12489 sd_uscsi_iodone(int index, struct sd_lun *un, struct buf *bp)
12490 {
12491         struct sd_xbuf *xp;

12493         ASSERT(un != NULL);
```



```
12494         ASSERT(bp != NULL);

12496         xp = SD_GET_XBUF(bp);
12497         ASSERT(xp != NULL);
12498         ASSERT(!mutex_owned(SD_MUTEX(un)));

12500         SD_INFO(SD_LOG_IO, un, "sd_uscsi_iodone: entry.\n");

12502         bp->b_private = xp->xb_private;

12504         mutex_enter(SD_MUTEX(un));

12506         /*
12507          * Grab time when the cmd completed.
12508          * This is used for determining if the system has been
12509          * idle long enough to make it idle to the PM framework.
12510          * This is for lowering the overhead, and therefore improving
12511          * performance per I/O operation.
12512          */
12513         un->un_pm_idle_time = gethrtime();
12512         un->un_pm_idle_time = ddi_get_time();

12515         un->un_ncmds_in_driver--;
12516         ASSERT(un->un_ncmds_in_driver >= 0);
12517         SD_INFO(SD_LOG_IO, un, "sd_uscsi_iodone: un_ncmds_in_driver = %ld\n",
12518             un->un_ncmds_in_driver);

12520         mutex_exit(SD_MUTEX(un));

12522         if (((struct uscsi_cmd *)(xp->xb_pktinfo))->uscsi_rqlen >
12523             SENSE_LENGTH) {
12524                 kmem_free(xp, sizeof (struct sd_xbuf) - SENSE_LENGTH +
12525                     MAX_SENSE_LENGTH);
12526         } else {
12527                 kmem_free(xp, sizeof (struct sd_xbuf));
12528         }

12530         biodone(bp);

12532         SD_INFO(SD_LOG_IO, un, "sd_uscsi_iodone: exit.\n");
12533 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  235818 Mon May  5 14:29:45 2014
new/usr/src/uts/common/io/usb/usba/hubdi.c
4782 usba shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1998, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2012 Garrett D'Amore <garrett@damore.org>.  All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 * USBA: Solaris USB Architecture support for the hub
 * including root hub
 * Most of the code for hubd resides in this file and
 * is shared between the HCD root hub support and hubd
 */
#define USBA_FRAMEWORK
#include <sys/usb/usba.h>
#include <sys/usb/usba/usba_devdb.h>
#include <sys/sunndi.h>
#include <sys/usb/usba/usba_impl.h>
#include <sys/usb/usba/usba_types.h>
#include <sys/usb/usba/hubdi.h>
#include <sys/usb/usba/hcdi_impl.h>
#include <sys/usb/hubd/hub.h>
#include <sys/usb/hubd/hubdvar.h>
#include <sys/usb/hubd/hubd_impl.h>
#include <sys/kobj.h>
#include <sys/kobj_lex.h>
#include <sys/fs/dv_node.h>
#include <sys/strsun.h>

/*
 * External functions
 */
extern boolean_t consconfig_console_is_ready(void);

/*
 * Prototypes for static functions
 */
static  int     usba_hubdi_bus_ctl(
                        dev_info_t              *dip,
                        dev_info_t              *rdip,
```



```
                        ddi_ctl_enum_t          op,
                        void                    *arg,
                        void                    *result);

static int      usba_hubdi_map_fault(
                        dev_info_t              *dip,
                        dev_info_t              *rdip,
                        struct hat              *hat,
                        struct seg              *seg,
                        caddr_t                 addr,
                        struct devpage          *dp,
                        pfn_t                   pfn,
                        uint_t                  prot,
                        uint_t                  lock);

static int hubd_busop_get_eventcookie(dev_info_t *dip,
                        dev_info_t *rdip,
                        char *eventname,
                        ddi_eventcookie_t *cookie);
static int hubd_busop_add_eventcall(dev_info_t *dip,
                        dev_info_t *rdip,
                        ddi_eventcookie_t cookie,
                        void (*callback)(dev_info_t *dip,
                                ddi_eventcookie_t cookie, void *arg,
                                void *bus_impldata),
                        void *arg, ddi_callback_id_t *cb_id);
static int hubd_busop_remove_eventcall(dev_info_t *dip,
                        ddi_callback_id_t cb_id);
static int hubd_bus_config(dev_info_t *dip,
                        uint_t flag,
                        ddi_bus_config_op_t op,
                        void *arg,
                        dev_info_t **child);
static int hubd_bus_unconfig(dev_info_t *dip,
                        uint_t flag,
                        ddi_bus_config_op_t op,
                        void *arg);
static int hubd_bus_power(dev_info_t *dip, void *impl_arg,
                        pm_bus_power_op_t op, void *arg, void *result);

static usb_port_t  hubd_get_port_num(hubd_t *, struct devctl_iocdata *);
static dev_info_t *hubd_get_child_dip(hubd_t *, usb_port_t);
static uint_t hubd_cfgadm_state(hubd_t *, usb_port_t);
static int hubd_toggle_port(hubd_t *, usb_port_t);
static void hubd_register_cpr_callback(hubd_t *);
static void hubd_unregister_cpr_callback(hubd_t *);

/*
 * Busops vector for USB HUB's
 */
struct bus_ops usba_hubdi_busops =      {
        BUSO_REV,
        nullbusmap,                     /* bus_map */
        NULL,                           /* bus_get_intrspec */
        NULL,                           /* bus_add_intrspec */
        NULL,                           /* bus_remove_intrspec */
        usba_hubdi_map_fault,           /* bus_map_fault */
        NULL,                           /* bus_dma_map */
        ddi_dma_allochdl,
        ddi_dma_freehdl,
        ddi_dma_bindhdl,
        ddi_dma_unbindhdl,
        ddi_dma_flush,
        ddi_dma_win,
        ddi_dma_mctl,                   /* bus_dma_ctl */
        usba_hubdi_bus_ctl,             /* bus_ctl */
```

```
127         ddi_bus_prop_op,                /* bus_prop_op */
128         hubd_busop_get_eventcookie,
129         hubd_busop_add_eventcall,
130         hubd_busop_remove_eventcall,
131         NULL,                           /* bus_post_event */
132         NULL,                           /* bus_intr_ctl */
133         hubd_bus_config,                /* bus_config */
134         hubd_bus_unconfig,              /* bus_unconfig */
135         NULL,                           /* bus_fm_init */
136         NULL,                           /* bus_fm_fini */
137         NULL,                           /* bus_fm_access_enter */
138         NULL,                           /* bus_fm_access_exit */
139         hubd_bus_power                  /* bus_power */
140 };

142 #define USB_HUB_INTEL_VID       0x8087
143 #define USB_HUB_INTEL_PID       0x0020

145 /*
146  * local variables
147  */
148 static kmutex_t usba_hubdi_mutex;       /* protects USBA HUB data structures */

150 static usba_list_entry_t        usba_hubdi_list;

152 usb_log_handle_t        hubdi_log_handle;
153 uint_t                  hubdi_errlevel = USB_LOG_L4;
154 uint_t                  hubdi_errmask = (uint_t)-1;
155 uint8_t                 hubdi_min_pm_threshold = 5; /* seconds */
156 uint8_t                 hubdi_reset_delay = 20; /* seconds */
157 extern int modrootloaded;

159 /*
160  * initialize private data
161  */
162 void
163 usba_hubdi_initialization()
164 {
165         hubdi_log_handle = usb_alloc_log_hdl(NULL, "hubdi", &hubdi_errlevel,
166             &hubdi_errmask, NULL, 0);

168         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
169             "usba_hubdi_initialization");

171         mutex_init(&usba_hubdi_mutex, NULL, MUTEX_DRIVER, NULL);

173         usba_init_list(&usba_hubdi_list, NULL, NULL);
174 }


177 void
178 usba_hubdi_destroy()
179 {
180         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
181             "usba_hubdi_destroy");

183         mutex_destroy(&usba_hubdi_mutex);
184         usba_destroy_list(&usba_hubdi_list);

186         usb_free_log_hdl(hubdi_log_handle);
187 }


190 /*
191  * Called by an HUB to attach an instance of the driver
192  *      make this instance known to USBA
```



```
193  *      the HUB should initialize usba_hubdi structure prior
194  *      to calling this interface
195  */
196 int
197 usba_hubdi_register(dev_info_t  *dip,
198                 uint_t          flags)
199 {
200         usba_hubdi_t *hubdi = kmem_zalloc(sizeof (usba_hubdi_t), KM_SLEEP);
201         usba_device_t *usba_device = usba_get_usba_device(dip);

203         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
204             "usba_hubdi_register: %s", ddi_node_name(dip));

206         hubdi->hubdi_dip = dip;
207         hubdi->hubdi_flags = flags;

209         usba_device->usb_hubdi = hubdi;

211         /*
212          * add this hubdi instance to the list of known hubdi's
213          */
214         usba_init_list(&hubdi->hubdi_list, (usb_opaque_t)hubdi,
215             usba_hcdi_get_hcdi(usba_device->usb_root_hub_dip)->
216             hcdi_iblock_cookie);
217         mutex_enter(&usba_hubdi_mutex);
218         usba_add_to_list(&usba_hubdi_list, &hubdi->hubdi_list);
219         mutex_exit(&usba_hubdi_mutex);

221         return (DDI_SUCCESS);
222 }


225 /*
226  * Called by an HUB to detach an instance of the driver
227  */
228 int
229 usba_hubdi_unregister(dev_info_t *dip)
230 {
231         usba_device_t *usba_device = usba_get_usba_device(dip);
232         usba_hubdi_t *hubdi = usba_device->usb_hubdi;

234         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
235             "usba_hubdi_unregister: %s", ddi_node_name(dip));

237         mutex_enter(&usba_hubdi_mutex);
238         (void) usba_rm_from_list(&usba_hubdi_list, &hubdi->hubdi_list);
239         mutex_exit(&usba_hubdi_mutex);

241         usba_destroy_list(&hubdi->hubdi_list);

243         kmem_free(hubdi, sizeof (usba_hubdi_t));

245         return (DDI_SUCCESS);
246 }


249 /*
250  * misc bus routines currently not used
251  */
252 /*ARGSUSED*/
253 static int
254 usba_hubdi_map_fault(dev_info_t *dip,
255         dev_info_t      *rdip,
256         struct hat      *hat,
257         struct seg      *seg,
258         caddr_t         addr,
```

```
259         struct devpage  *dp,
260         pfn_t           pfn,
261         uint_t          prot,
262         uint_t          lock)
263 {
264         return (DDI_FAILURE);
265 }


268 /*
269  * root hub support. the root hub uses the same devi as the HCD
270  */
271 int
272 usba_hubdi_bind_root_hub(dev_info_t *dip,
273         uchar_t *root_hub_config_descriptor,
274         size_t config_length,
275         usb_dev_descr_t *root_hub_device_descriptor)
276 {
277         usba_device_t *usba_device;
278         usba_hcdi_t *hcdi = usba_hcdi_get_hcdi(dip);
279         hubd_t  *root_hubd;
280         usb_pipe_handle_t ph = NULL;
281         dev_info_t *child = ddi_get_child(dip);

283         if (ndi_prop_create_boolean(DDI_DEV_T_NONE, dip,
284             "root-hub") != NDI_SUCCESS) {

286                 return (USB_FAILURE);
287         }

289         usba_add_root_hub(dip);

291         root_hubd = kmem_zalloc(sizeof (hubd_t), KM_SLEEP);

293         /*
294          * create and initialize a usba_device structure
295          */
296         usba_device = usba_alloc_usba_device(dip);

298         mutex_enter(&usba_device->usb_mutex);
299         usba_device->usb_hcdi_ops = hcdi->hcdi_ops;
300         usba_device->usb_cfg = root_hub_config_descriptor;
301         usba_device->usb_cfg_length = config_length;
302         usba_device->usb_dev_descr = root_hub_device_descriptor;
303         usba_device->usb_port = 1;
304         usba_device->usb_addr = ROOT_HUB_ADDR;
305         usba_device->usb_root_hubd = root_hubd;
306         usba_device->usb_cfg_array = kmem_zalloc(sizeof (uchar_t *),
307             KM_SLEEP);
308         usba_device->usb_cfg_array_length = sizeof (uchar_t *);

310         usba_device->usb_cfg_array_len = kmem_zalloc(sizeof (uint16_t),
311             KM_SLEEP);
312         usba_device->usb_cfg_array_len_length = sizeof (uint16_t);

314         usba_device->usb_cfg_array[0] = root_hub_config_descriptor;
315         usba_device->usb_cfg_array_len[0] =
316             sizeof (root_hub_config_descriptor);

318         usba_device->usb_cfg_str_descr = kmem_zalloc(sizeof (uchar_t *),
319             KM_SLEEP);
320         usba_device->usb_n_cfgs = 1;
321         usba_device->usb_n_ifs = 1;
322         usba_device->usb_dip = dip;

324         usba_device->usb_client_flags = kmem_zalloc(
```



```
325             usba_device->usb_n_ifs * USBA_CLIENT_FLAG_SIZE, KM_SLEEP);

327         usba_device->usb_client_attach_list = kmem_zalloc(
328             usba_device->usb_n_ifs *
329             sizeof (*usba_device->usb_client_attach_list), KM_SLEEP);

331         usba_device->usb_client_ev_cb_list = kmem_zalloc(
332             usba_device->usb_n_ifs *
333             sizeof (*usba_device->usb_client_ev_cb_list), KM_SLEEP);

335         /*
336          * The bDeviceProtocol field of root hub device specifies,
337          * whether root hub is a High or Full speed usb device.
338          */
339         if (root_hub_device_descriptor->bDeviceProtocol) {
340                 usba_device->usb_port_status = USBA_HIGH_SPEED_DEV;
341         } else {
342                 usba_device->usb_port_status = USBA_FULL_SPEED_DEV;
343         }

345         mutex_exit(&usba_device->usb_mutex);

347         usba_set_usba_device(dip, usba_device);

349         /*
350          * For the root hub the default pipe is not yet open
351          */
352         if (usb_pipe_open(dip, NULL, NULL,
353             USB_FLAGS_SLEEP | USBA_FLAGS_PRIVILEGED, &ph) != USB_SUCCESS) {
354                 goto fail;
355         }

357         /*
358          * kill off all OBP children, they may not be fully
359          * enumerated
360          */
361         while (child) {
362                 dev_info_t *next = ddi_get_next_sibling(child);
363                 (void) ddi_remove_child(child, 0);
364                 child = next;
365         }

367         /*
368          * "attach" the root hub driver
369          */
370         if (usba_hubdi_attach(dip, DDI_ATTACH) != DDI_SUCCESS) {
371                 goto fail;
372         }

374         return (USB_SUCCESS);

376 fail:
377         (void) ndi_prop_remove(DDI_DEV_T_NONE, dip, "root-hub");

379         usba_rem_root_hub(dip);

381         if (ph) {
382                 usb_pipe_close(dip, ph,
383                     USB_FLAGS_SLEEP | USBA_FLAGS_PRIVILEGED, NULL, NULL);
384         }

386         kmem_free(usba_device->usb_cfg_array,
387             usba_device->usb_cfg_array_length);
388         kmem_free(usba_device->usb_cfg_array_len,
389             usba_device->usb_cfg_array_len_length);
```

```
391         kmem_free(usba_device->usb_cfg_str_descr, sizeof (uchar_t *));

393         usba_free_usba_device(usba_device);

395         usba_set_usba_device(dip, NULL);
396         if (root_hubd) {
397                 kmem_free(root_hubd, sizeof (hubd_t));
398         }

400         return (USB_FAILURE);
401 }


404 int
405 usba_hubdi_unbind_root_hub(dev_info_t *dip)
406 {
407         usba_device_t *usba_device;

409         /* was root hub attached? */
410         if (!(usba_is_root_hub(dip))) {

412                 /* return success anyway */
413                 return (USB_SUCCESS);
414         }

416         /*
417          * usba_hubdi_detach also closes the default pipe
418          * and removes properties so there is no need to
419          * do it here
420          */
421         if (usba_hubdi_detach(dip, DDI_DETACH) != DDI_SUCCESS) {

423                 if (DEVI_IS_ATTACHING(dip)) {
424                         USB_DPRINTF_L2(DPRINT_MASK_ATTA, hubdi_log_handle,
425                             "failure to unbind root hub after attach failure");
426                 }

428                 return (USB_FAILURE);
429         }

431         usba_device = usba_get_usba_device(dip);

433         kmem_free(usba_device->usb_root_hubd, sizeof (hubd_t));

435         kmem_free(usba_device->usb_cfg_array,
436             usba_device->usb_cfg_array_length);
437         kmem_free(usba_device->usb_cfg_array_len,
438             usba_device->usb_cfg_array_len_length);

440         kmem_free(usba_device->usb_cfg_str_descr, sizeof (uchar_t *));

442         usba_free_usba_device(usba_device);

444         usba_rem_root_hub(dip);

446         (void) ndi_prop_remove(DDI_DEV_T_NONE, dip, "root-hub");

448         return (USB_SUCCESS);
449 }


452 /*
453  * Actual Hub Driver support code:
454  *      shared by root hub and non-root hubs
455  */
456 #include <sys/usb/usba/usbai_version.h>
```



```
458 /* Debugging support */
459 uint_t hubd_errlevel    = USB_LOG_L4;
460 uint_t hubd_errmask     = (uint_t)DPRINT_MASK_ALL;
461 uint_t hubd_instance_debug = (uint_t)-1;
462 static uint_t hubdi_bus_config_debug = 0;

464 _NOTE(DATA_READABLE_WITHOUT_LOCK(hubd_errlevel))
465 _NOTE(DATA_READABLE_WITHOUT_LOCK(hubd_errmask))
466 _NOTE(DATA_READABLE_WITHOUT_LOCK(hubd_instance_debug))

468 _NOTE(SCHEME_PROTECTS_DATA("unique", msgb))
469 _NOTE(SCHEME_PROTECTS_DATA("unique", dev_info))


472 /*
473  * local variables:
474  *
475  * Amount of time to wait between resetting the port and accessing
476  * the device.  The value is in microseconds.
477  */
478 static uint_t hubd_device_delay = 1000000;

480 /*
481  * enumeration retry
482  */
483 #define HUBD_PORT_RETRY 5
484 static uint_t hubd_retry_enumerate = HUBD_PORT_RETRY;

486 /*
487  * Stale hotremoved device cleanup delay
488  */
489 #define HUBD_STALE_DIP_CLEANUP_DELAY    5000000
490 static uint_t hubd_dip_cleanup_delay = HUBD_STALE_DIP_CLEANUP_DELAY;

492 /*
493  * retries for USB suspend and resume
494  */
495 #define HUBD_SUS_RES_RETRY      2

497 void    *hubd_statep;

499 /*
500  * prototypes
501  */
502 static int hubd_cleanup(dev_info_t *dip, hubd_t  *hubd);
503 static int hubd_check_ports(hubd_t  *hubd);

505 static int  hubd_open_intr_pipe(hubd_t *hubd);
506 static void hubd_start_polling(hubd_t *hubd, int always);
507 static void hubd_stop_polling(hubd_t *hubd);
508 static void hubd_close_intr_pipe(hubd_t *hubd);

510 static void hubd_read_cb(usb_pipe_handle_t pipe, usb_intr_req_t *req);
511 static void hubd_exception_cb(usb_pipe_handle_t pipe,
512                                                 usb_intr_req_t *req);
513 static void hubd_hotplug_thread(void *arg);
514 static void hubd_reset_thread(void *arg);
515 static int hubd_create_child(dev_info_t *dip,
516                 hubd_t          *hubd,
517                 usba_device_t   *usba_device,
518                 usb_port_status_t port_status,
519                 usb_port_t      port,
520                 int             iteration);

522 static int hubd_delete_child(hubd_t *hubd, usb_port_t port, uint_t flag,
```

```
523         boolean_t retry);

525 static int hubd_get_hub_descriptor(hubd_t *hubd);

527 static int hubd_get_hub_status_words(hubd_t *hubd, uint16_t *status);

529 static int hubd_reset_port(hubd_t *hubd, usb_port_t port);

531 static int hubd_get_hub_status(hubd_t *hubd);

533 static int hubd_handle_port_connect(hubd_t *hubd, usb_port_t port);

535 static int hubd_disable_port(hubd_t *hubd, usb_port_t port);

537 static int hubd_enable_port(hubd_t *hubd, usb_port_t port);
538 static int hubd_recover_disabled_port(hubd_t *hubd, usb_port_t port);

540 static int hubd_determine_port_status(hubd_t *hubd, usb_port_t port,
541         uint16_t *status, uint16_t *change, uint_t ack_flag);

543 static int hubd_enable_all_port_power(hubd_t *hubd);
544 static int hubd_disable_all_port_power(hubd_t *hubd);
545 static int hubd_disable_port_power(hubd_t *hubd, usb_port_t port);
546 static int hubd_enable_port_power(hubd_t *hubd, usb_port_t port);

548 static void hubd_free_usba_device(hubd_t *hubd, usba_device_t *usba_device);

550 static int hubd_can_suspend(hubd_t *hubd);
551 static void hubd_restore_device_state(dev_info_t *dip, hubd_t *hubd);
552 static int hubd_setdevaddr(hubd_t *hubd, usb_port_t port);
553 static void hubd_setdevconfig(hubd_t *hubd, usb_port_t port);

555 static int hubd_register_events(hubd_t *hubd);
556 static void hubd_do_callback(hubd_t *hubd, dev_info_t *dip,
557         ddi_eventcookie_t cookie);
558 static void hubd_run_callbacks(hubd_t *hubd, usba_event_t type);
559 static void hubd_post_event(hubd_t *hubd, usb_port_t port, usba_event_t type);
560 static void hubd_create_pm_components(dev_info_t *dip, hubd_t *hubd);

562 static int hubd_disconnect_event_cb(dev_info_t *dip);
563 static int hubd_reconnect_event_cb(dev_info_t *dip);
564 static int hubd_pre_suspend_event_cb(dev_info_t *dip);
565 static int hubd_post_resume_event_cb(dev_info_t *dip);
566 static int hubd_cpr_suspend(hubd_t *hubd);
567 static void hubd_cpr_resume(dev_info_t *dip);
568 static int hubd_restore_state_cb(dev_info_t *dip);
569 static int hubd_check_same_device(hubd_t *hubd, usb_port_t port);

571 static int hubd_init_power_budget(hubd_t *hubd);

573 static ndi_event_definition_t hubd_ndi_event_defs[] = {
574         {USBA_EVENT_TAG_HOT_REMOVAL, DDI_DEVI_REMOVE_EVENT, EPL_KERNEL,
575                                             NDI_EVENT_POST_TO_ALL},
576         {USBA_EVENT_TAG_HOT_INSERTION, DDI_DEVI_INSERT_EVENT, EPL_KERNEL,
577                                             NDI_EVENT_POST_TO_ALL},
578         {USBA_EVENT_TAG_POST_RESUME, USBA_POST_RESUME_EVENT, EPL_KERNEL,
579                                             NDI_EVENT_POST_TO_ALL},
580         {USBA_EVENT_TAG_PRE_SUSPEND, USBA_PRE_SUSPEND_EVENT, EPL_KERNEL,
581                                             NDI_EVENT_POST_TO_ALL}
582 };

584 #define HUBD_N_NDI_EVENTS \
585         (sizeof (hubd_ndi_event_defs) / sizeof (ndi_event_definition_t))

587 static ndi_event_set_t hubd_ndi_events = {
588         NDI_EVENTS_REV1, HUBD_N_NDI_EVENTS, hubd_ndi_event_defs};
```



```
590 /* events received from parent */
591 static usb_event_t hubd_events = {
592         hubd_disconnect_event_cb,
593         hubd_reconnect_event_cb,
594         hubd_pre_suspend_event_cb,
595         hubd_post_resume_event_cb
596 };


599 /*
600  * hubd_get_soft_state() returns the hubd soft state
601  *
602  * WUSB support extends this function to support wire adapter class
603  * devices. The hubd soft state for the wire adapter class device
604  * would be stored in usb_root_hubd field of the usba_device structure,
605  * just as the USB host controller drivers do.
606  */
607 hubd_t *
608 hubd_get_soft_state(dev_info_t *dip)
609 {
610         if (dip == NULL) {

612                 return (NULL);
613         }

615         if (usba_is_root_hub(dip) || usba_is_wa(dip)) {
616                 usba_device_t *usba_device = usba_get_usba_device(dip);

618                 return (usba_device->usb_root_hubd);
619         } else {
620                 int instance = ddi_get_instance(dip);

622                 return (ddi_get_soft_state(hubd_statep, instance));
623         }
624 }


627 /*
628  * PM support functions:
629  */
630 /*ARGSUSED*/
631 static void
632 hubd_pm_busy_component(hubd_t *hubd, dev_info_t *dip, int component)
633 {
634         if (hubd->h_hubpm != NULL) {
635                 hubd->h_hubpm->hubp_busy_pm++;
636                 mutex_exit(HUBD_MUTEX(hubd));
637                 if (pm_busy_component(dip, 0) != DDI_SUCCESS) {
638                         mutex_enter(HUBD_MUTEX(hubd));
639                         hubd->h_hubpm->hubp_busy_pm--;
640                         mutex_exit(HUBD_MUTEX(hubd));
641                 }
642                 mutex_enter(HUBD_MUTEX(hubd));
643                 USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
644                     "hubd_pm_busy_component: %d", hubd->h_hubpm->hubp_busy_pm);
645         }
646 }


649 /*ARGSUSED*/
650 static void
651 hubd_pm_idle_component(hubd_t *hubd, dev_info_t *dip, int component)
652 {
653         if (hubd->h_hubpm != NULL) {
654                 mutex_exit(HUBD_MUTEX(hubd));
```

```
655                 if (pm_idle_component(dip, 0) == DDI_SUCCESS) {
656                         mutex_enter(HUBD_MUTEX(hubd));
657                         ASSERT(hubd->h_hubpm->hubp_busy_pm > 0);
658                         hubd->h_hubpm->hubp_busy_pm--;
659                         mutex_exit(HUBD_MUTEX(hubd));
660                 }
661                 mutex_enter(HUBD_MUTEX(hubd));
662                 USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
663                     "hubd_pm_idle_component: %d", hubd->h_hubpm->hubp_busy_pm);
664         }
665 }


668 /*
669  * track power level changes for children of this instance
670  */
671 static void
672 hubd_set_child_pwrlvl(hubd_t *hubd, usb_port_t port, uint8_t power)
673 {
674         int     old_power, new_power, pwr;
675         usb_port_t      portno;
676         hub_power_t     *hubpm;

678         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
679             "hubd_set_child_pwrlvl: port=%d power=%d",
680             port, power);

682         mutex_enter(HUBD_MUTEX(hubd));
683         hubpm = hubd->h_hubpm;

685         old_power = 0;
686         for (portno = 1; portno <= hubd->h_hub_descr.bNbrPorts; portno++) {
687                 old_power += hubpm->hubp_child_pwrstate[portno];
688         }

690         /* assign the port power */
691         pwr = hubd->h_hubpm->hubp_child_pwrstate[port];
692         hubd->h_hubpm->hubp_child_pwrstate[port] = power;
693         new_power = old_power - pwr + power;

695         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
696             "hubd_set_child_pwrlvl: new_power=%d old_power=%d",
697             new_power, old_power);

699         if ((new_power > 0) && (old_power == 0)) {
700                 /* we have the first child coming out of low power */
701                 (void) hubd_pm_busy_component(hubd, hubd->h_dip, 0);
702         } else if ((new_power == 0) && (old_power > 0)) {
703                 /* we have the last child going to low power */
704                 (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);
705         }
706         mutex_exit(HUBD_MUTEX(hubd));
707 }


710 /*
711  * given a child dip, locate its port number
712  */
713 static usb_port_t
714 hubd_child_dip2port(hubd_t *hubd, dev_info_t *dip)
715 {
716         usb_port_t      port;

718         mutex_enter(HUBD_MUTEX(hubd));
719         for (port = 1; port <= hubd->h_hub_descr.bNbrPorts; port++) {
720                 if (hubd->h_children_dips[port] == dip) {
```



```
722                         break;
723                 }
724         }
725         ASSERT(port <= hubd->h_hub_descr.bNbrPorts);
726         mutex_exit(HUBD_MUTEX(hubd));

728         return (port);
729 }


732 /*
733  * if the hub can be put into low power mode, return success
734  * NOTE: suspend here means going to lower power, not CPR suspend.
735  */
736 static int
737 hubd_can_suspend(hubd_t *hubd)
738 {
739         hub_power_t     *hubpm;
740         int             total_power = 0;
741         usb_port_t      port;

743         hubpm = hubd->h_hubpm;

745         if (DEVI_IS_DETACHING(hubd->h_dip)) {

747                 return (USB_SUCCESS);
748         }

750         /*
751          * Don't go to lower power if haven't been at full power for enough
752          * time to let hotplug thread kickoff.
753          */
754         if (gethrtime() < (hubpm->hubp_time_at_full_power +
 24         if (ddi_get_time() < (hubpm->hubp_time_at_full_power +
755             hubpm->hubp_min_pm_threshold)) {

757                 return (USB_FAILURE);
758         }

760         for (port = 1; (total_power == 0) &&
761             (port <= hubd->h_hub_descr.bNbrPorts); port++) {
762                 total_power += hubpm->hubp_child_pwrstate[port];
763         }

765         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
766             "hubd_can_suspend: %d", total_power);

768         return (total_power ? USB_FAILURE : USB_SUCCESS);
769 }
______unchanged_portion_omitted_


1702 static int
1703 hubd_pwrlvl3(hubd_t *hubd)
1704 {
1705         hub_power_t     *hubpm;
1706         int             rval;

1708         USB_DPRINTF_L2(DPRINT_MASK_PM, hubd->h_log_handle, "hubd_pwrlvl3");

1710         hubpm = hubd->h_hubpm;
1711         switch (hubd->h_dev_state) {
1712         case USB_DEV_PWRED_DOWN:
1713                 ASSERT(hubpm->hubp_current_power == USB_DEV_OS_PWR_OFF);
1714                 if (usba_is_root_hub(hubd->h_dip)) {
```

```
1715                        /* implement global resume here */
1716                        USB_DPRINTF_L2(DPRINT_MASK_PM,
1717                            hubd->h_log_handle,
1718                            "Global Resume: Not Yet Implemented");
1719                }
1720                /* Issue USB D0 command to the device here */
1721                rval = usb_set_device_pwrlvl0(hubd->h_dip);
1722                ASSERT(rval == USB_SUCCESS);
1723                hubd->h_dev_state = USB_DEV_ONLINE;
1724                hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
1725                hubpm->hubp_time_at_full_power = gethrtime();
 995                hubpm->hubp_time_at_full_power = ddi_get_time();
1726                hubd_start_polling(hubd, 0);

1728                /* FALLTHRU */
1729        case USB_DEV_ONLINE:
1730                /* we are already in full power */

1732                /* FALLTHRU */
1733        case USB_DEV_DISCONNECTED:
1734        case USB_DEV_SUSPENDED:
1735                /*
1736                 * PM framework tries to put you in full power
1737                 * during system shutdown. If we are disconnected
1738                 * return success. Also, we should not change state
1739                 * when we are disconnected or suspended or about to
1740                 * transition to that state
1741                 */

1743                return (USB_SUCCESS);
1744        default:
1745                USB_DPRINTF_L2(DPRINT_MASK_PM, hubd->h_log_handle,
1746                    "hubd_pwrlvl3: Illegal dev_state=%d", hubd->h_dev_state);

1748                return (USB_FAILURE);
1749        }
1750 }
```

*unchanged_portion_omitted_*

```
3643 /*
3644  * hubd_hotplug_thread:
3645  *      handles resetting of port, and creating children
3646  *
3647  *      the ports to check are indicated in h_port_change bit mask
3648  * XXX note that one time poll doesn't work on the root hub
3649  */
3650 static void
3651 hubd_hotplug_thread(void *arg)
3652 {
3653        hubd_hotplug_arg_t *hd_arg = (hubd_hotplug_arg_t *)arg;
3654        hubd_t          *hubd = hd_arg->hubd;
3655        boolean_t       attach_flg = hd_arg->hotplug_during_attach;
3656        usb_port_t      port;
3657        uint16_t        nports;
3658        uint16_t        status, change;
3659        hub_power_t     *hubpm;
3660        dev_info_t      *hdip = hubd->h_dip;
3661        dev_info_t      *rh_dip = hubd->h_usba_device->usb_root_hub_dip;
3662        dev_info_t      *child_dip;
3663        boolean_t       online_child = B_FALSE;
3664        boolean_t       offline_child = B_FALSE;
3665        boolean_t       pwrup_child = B_FALSE;
3666        int             prh_circ, rh_circ, chld_circ, circ, old_state;

3668        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
```



```
3669            "hubd_hotplug_thread:  started");

3671        /*
3672         * Before console is init'd, we temporarily block the hotplug
3673         * threads so that BUS_CONFIG_ONE through hubd_bus_config() can be
3674         * processed quickly. This reduces the time needed for vfs_mountroot()
3675         * to mount the root FS from a USB disk. And on SPARC platform,
3676         * in order to load 'consconfig' successfully after OBP is gone,
3677         * we need to check 'modrootloaded' to make sure root filesystem is
3678         * available.
3679         */
3680        while (!modrootloaded || !consconfig_console_is_ready()) {
3681                delay(drv_usectohz(10000));
3682        }

3684        kmem_free(arg, sizeof (hubd_hotplug_arg_t));

3686        /*
3687         * if our bus power entry point is active, process the change
3688         * on the next notification of interrupt pipe
3689         */
3690        mutex_enter(HUBD_MUTEX(hubd));
3691        if (hubd->h_bus_pwr || (hubd->h_hotplug_thread > 1)) {
3692                hubd->h_hotplug_thread--;

3694                /* mark this device as idle */
3695                hubd_pm_idle_component(hubd, hubd->h_dip, 0);
3696                mutex_exit(HUBD_MUTEX(hubd));

3698                USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3699                    "hubd_hotplug_thread: "
3700                    "bus_power in progress/hotplugging undesirable - quit");

3702                return;
3703        }
3704        mutex_exit(HUBD_MUTEX(hubd));

3706        ndi_hold_devi(hdip); /* so we don't race with detach */

3708        mutex_enter(HUBD_MUTEX(hubd));

3710        /* is this the root hub? */
3711        if (hdip == rh_dip) {
3712                if (hubd->h_dev_state == USB_DEV_PWRED_DOWN) {
3713                        hubpm = hubd->h_hubpm;

3715                        /* mark the root hub as full power */
3716                        hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
3717                        hubpm->hubp_time_at_full_power = gethrtime();
2987                        hubpm->hubp_time_at_full_power = ddi_get_time();
3718                        mutex_exit(HUBD_MUTEX(hubd));

3720                        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3721                            "hubd_hotplug_thread: call pm_power_has_changed");

3723                        (void) pm_power_has_changed(hdip, 0,
3724                            USB_DEV_OS_FULL_PWR);

3726                        mutex_enter(HUBD_MUTEX(hubd));
3727                        hubd->h_dev_state = USB_DEV_ONLINE;
3728                }

3730        } else {
3731                USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3732                    "hubd_hotplug_thread: not root hub");
3733        }
```

```
3735         mutex_exit(HUBD_MUTEX(hubd));

3737         /*
3738          * this ensures one hotplug activity per system at a time.
3739          * we enter the parent PCI node to have this serialization.
3740          * this also excludes ioctls and deathrow thread
3741          * (a bit crude but easier to debug)
3742          */
3743         ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
3744         ndi_devi_enter(rh_dip, &rh_circ);

3746         /* exclude other threads */
3747         ndi_devi_enter(hdip, &circ);
3748         mutex_enter(HUBD_MUTEX(hubd));

3750         ASSERT(hubd->h_intr_pipe_state == HUBD_INTR_PIPE_ACTIVE);

3752         nports = hubd->h_hub_descr.bNbrPorts;

3754         hubd_stop_polling(hubd);

3756         while ((hubd->h_dev_state == USB_DEV_ONLINE) &&
3757             (hubd->h_port_change)) {
3758                 /*
3759                  * The 0th bit is the hub status change bit.
3760                  * handle loss of local power here
3761                  */
3762                 if (hubd->h_port_change & HUB_CHANGE_STATUS) {
3763                         USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3764                             "hubd_hotplug_thread: hub status change!");

3766                         /*
3767                          * This should be handled properly.  For now,
3768                          * mask off the bit.
3769                          */
3770                         hubd->h_port_change &= ~HUB_CHANGE_STATUS;

3772                         /*
3773                          * check and ack hub status
3774                          * this causes stall conditions
3775                          * when local power is removed
3776                          */
3777                         (void) hubd_get_hub_status(hubd);
3778                 }

3780                 for (port = 1; port <= nports; port++) {
3781                         usb_port_mask_t port_mask;
3782                         boolean_t was_connected;

3784                         port_mask = 1 << port;
3785                         was_connected =
3786                             (hubd->h_port_state[port] & PORT_STATUS_CCS) &&
3787                             (hubd->h_children_dips[port]);

3789                         USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3790                             "hubd_hotplug_thread: "
3791                             "port %d mask=0x%x change=0x%x connected=0x%x",
3792                             port, port_mask, hubd->h_port_change,
3793                             was_connected);

3795                         /*
3796                          * is this a port connection that changed?
3797                          */
3798                         if ((hubd->h_port_change & port_mask) == 0) {
```



```
3800                                 continue;
3801                         }
3802                         hubd->h_port_change &= ~port_mask;

3804                         /* ack all changes */
3805                         (void) hubd_determine_port_status(hubd, port,
3806                             &status, &change, HUBD_ACK_ALL_CHANGES);

3808                         USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3809                             "handle port %d:\n\t"
3810                             "new status=0x%x change=0x%x was_conn=0x%x ",
3811                             port, status, change, was_connected);

3813                         /* Recover a disabled port */
3814                         if (change & PORT_CHANGE_PESC) {
3815                                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG,
3816                                     hubd->h_log_handle,
3817                                     "port%d Disabled - "
3818                                     "status=0x%x, change=0x%x",
3819                                     port, status, change);

3821                                 /*
3822                                  * if the port was connected and is still
3823                                  * connected, recover the port
3824                                  */
3825                                 if (was_connected && (status &
3826                                     PORT_STATUS_CCS)) {
3827                                         online_child |=
3828                                             (hubd_recover_disabled_port(hubd,
3829                                             port) == USB_SUCCESS);
3830                                 }
3831                         }

3833                         /*
3834                          * Now check what changed on the port
3835                          */
3836                         if ((change & PORT_CHANGE_CSC) || attach_flg) {
3837                                 if ((status & PORT_STATUS_CCS) &&
3838                                     (!was_connected)) {
3839                                         /* new device plugged in */
3840                                         online_child |=
3841                                             (hubd_handle_port_connect(hubd,
3842                                             port) == USB_SUCCESS);

3844                                 } else if ((status & PORT_STATUS_CCS) &&
3845                                     was_connected) {
3846                                         /*
3847                                          * In this case we can never be sure
3848                                          * if the device indeed got hotplugged
3849                                          * or the hub is falsely reporting the
3850                                          * change.
3851                                          */
3852                                         child_dip = hubd->h_children_dips[port];

3854                                         mutex_exit(HUBD_MUTEX(hubd));
3855                                         /*
3856                                          * this ensures we do not race with
3857                                          * other threads which are detaching
3858                                          * the child driver at the same time.
3859                                          */
3860                                         ndi_devi_enter(child_dip, &chld_circ);
3861                                         /*
3862                                          * Now check if the driver remains
3863                                          * attached.
3864                                          */
3865                                         if (i_ddi_devi_attached(child_dip)) {
```

```
3866                                            /*
3867                                             * first post a disconnect event
3868                                             * to the child.
3869                                             */
3870                                            hubd_post_event(hubd, port,
3871                                                USBA_EVENT_TAG_HOT_REMOVAL);
3872                                            mutex_enter(HUBD_MUTEX(hubd));

3874                                            /*
3875                                             * then reset the port and
3876                                             * recover the device
3877                                             */
3878                                            online_child |=
3879                                                (hubd_handle_port_connect(
3880                                                hubd, port) == USB_SUCCESS);

3882                                            mutex_exit(HUBD_MUTEX(hubd));
3883                                    }

3885                                    ndi_devi_exit(child_dip, chld_circ);
3886                                    mutex_enter(HUBD_MUTEX(hubd));
3887                            } else if (was_connected) {
3888                                    /* this is a disconnect */
3889                                    mutex_exit(HUBD_MUTEX(hubd));
3890                                    hubd_post_event(hubd, port,
3891                                        USBA_EVENT_TAG_HOT_REMOVAL);
3892                                    mutex_enter(HUBD_MUTEX(hubd));

3894                                    offline_child = B_TRUE;
3895                            }
3896                    }

3898                    /*
3899                     * Check if any port is coming out of suspend
3900                     */
3901                    if (change & PORT_CHANGE_PSSC) {
3902                            /* a resuming device could have disconnected */
3903                            if (was_connected &&
3904                                hubd->h_children_dips[port]) {

3906                                    /* device on this port resuming */
3907                                    dev_info_t *dip;

3909                                    dip = hubd->h_children_dips[port];

3911                                    /*
3912                                     * Don't raise power on detaching child
3913                                     */
3914                                    if (!DEVI_IS_DETACHING(dip)) {
3915                                            /*
3916                                             * As this child is not
3917                                             * detaching, we set this
3918                                             * flag, causing bus_ctls
3919                                             * to stall detach till
3920                                             * pm_raise_power returns
3921                                             * and flag it for a deferred
3922                                             * raise_power.
3923                                             *
3924                                             * pm_raise_power is deferred
3925                                             * because we need to release
3926                                             * the locks first.
3927                                             */
3928                                            hubd->h_port_state[port] |=
3929                                                HUBD_CHILD_RAISE_POWER;
3930                                            pwrup_child = B_TRUE;
3931                                            mutex_exit(HUBD_MUTEX(hubd));
```



```
3933                                            /*
3934                                             * make sure that child
3935                                             * doesn't disappear
3936                                             */
3937                                            ndi_hold_devi(dip);

3939                                            mutex_enter(HUBD_MUTEX(hubd));
3940                                    }
3941                            }
3942                    }

3944                    /*
3945                     * Check if the port is over-current
3946                     */
3947                    if (change & PORT_CHANGE_OCIC) {
3948                            USB_DPRINTF_L1(DPRINT_MASK_HOTPLUG,
3949                                hubd->h_log_handle,
3950                                "Port%d in over current condition, "
3951                                "please check the attached device to "
3952                                "clear the condition. The system will "
3953                                "try to recover the port, but if not "
3954                                "successful, you need to re-connect "
3955                                "the hub or reboot the system to bring "
3956                                "the port back to work", port);

3958                            if (!(status & PORT_STATUS_PPS)) {
3959                                    /*
3960                                     * Try to enable port power, but
3961                                     * possibly fail. Ignore failure
3962                                     */
3963                                    (void) hubd_enable_port_power(hubd,
3964                                        port);

3966                                    /*
3967                                     * Delay some time to avoid
3968                                     * over-current event to happen
3969                                     * too frequently in some cases
3970                                     */
3971                                    mutex_exit(HUBD_MUTEX(hubd));
3972                                    delay(drv_usectohz(500000));
3973                                    mutex_enter(HUBD_MUTEX(hubd));
3974                            }
3975                    }
3976            }
3977    }

3979    /* release locks so we can do a devfs_clean */
3980    mutex_exit(HUBD_MUTEX(hubd));

3982    /* delete cached dv_node's but drop locks first */
3983    ndi_devi_exit(hdip, circ);
3984    ndi_devi_exit(rh_dip, rh_circ);
3985    ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

3987    (void) devfs_clean(rh_dip, NULL, 0);

3989    /* now check if any children need onlining */
3990    if (online_child) {
3991            USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3992                "hubd_hotplug_thread: onlining children");

3994            (void) ndi_devi_online(hubd->h_dip, 0);
3995    }

3997    /* now check if any disconnected devices need to be cleaned up */
```

```
3998         if (offline_child) {
3999                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
4000                     "hubd_hotplug_thread: scheduling cleanup");
4002                 hubd_schedule_cleanup(hubd->h_usba_device->usb_root_hub_dip);
4003         }
4005         mutex_enter(HUBD_MUTEX(hubd));
4007         /* now raise power on the children that have woken up */
4008         if (pwrup_child) {
4009                 old_state = hubd->h_dev_state;
4010                 hubd->h_dev_state = USB_DEV_HUB_CHILD_PWRLVL;
4011                 for (port = 1; port <= nports; port++) {
4012                         if (hubd->h_port_state[port] & HUBD_CHILD_RAISE_POWER) {
4013                                 dev_info_t *dip = hubd->h_children_dips[port];
4015                                 mutex_exit(HUBD_MUTEX(hubd));
4017                                 /* Get the device to full power */
4018                                 (void) pm_busy_component(dip, 0);
4019                                 (void) pm_raise_power(dip, 0,
4020                                     USB_DEV_OS_FULL_PWR);
4021                                 (void) pm_idle_component(dip, 0);
4023                                 /* release the hold on the child */
4024                                 ndi_rele_devi(dip);
4025                                 mutex_enter(HUBD_MUTEX(hubd));
4026                                 hubd->h_port_state[port] &=
4027                                     ~HUBD_CHILD_RAISE_POWER;
4028                         }
4029                 }
4030                 /*
4031                  * make sure that we don't accidentally
4032                  * over write the disconnect state
4033                  */
4034                 if (hubd->h_dev_state == USB_DEV_HUB_CHILD_PWRLVL) {
4035                         hubd->h_dev_state = old_state;
4036                 }
4037         }
4039         /*
4040          * start polling can immediately kick off read callback
4041          * we need to set the h_hotplug_thread to 0 so that
4042          * the callback is not dropped
4043          *
4044          * if there is device during reset, still stop polling to avoid the
4045          * read callback interrupting the reset, the polling will be started
4046          * in hubd_reset_thread.
4047          */
4048         for (port = 1; port <= MAX_PORTS; port++) {
4049                 if (hubd->h_reset_port[port]) {
4051                         break;
4052                 }
4053         }
4054         if (port > MAX_PORTS) {
4055                 hubd_start_polling(hubd, HUBD_ALWAYS_START_POLLING);
4056         }
4058         /*
4059          * Earlier we would set the h_hotplug_thread = 0 before
4060          * polling was restarted  so that
4061          * if there is any root hub status change interrupt, we can still kick
4062          * off the hotplug thread. This was valid when this interrupt was
4063          * delivered in hardware, and only ONE interrupt would be delivered.
```



```
4064          * Now that we poll on the root hub looking for status change in
4065          * software, this assignment is no longer required.
4066          */
4067         hubd->h_hotplug_thread--;
4069         /* mark this device as idle */
4070         (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);
4072         cv_broadcast(&hubd->h_cv_hotplug_dev);
4074         USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
4075             "hubd_hotplug_thread: exit");
4077         mutex_exit(HUBD_MUTEX(hubd));
4079         ndi_rele_devi(hdip);
4080 }
_____unchanged_portion_omitted_


7258 /*
7259  * Power management
7260  *
7261  * create the pm components required for power management
7262  */
7263 static void
7264 hubd_create_pm_components(dev_info_t *dip, hubd_t *hubd)
7265 {
7266         hub_power_t     *hubpm;
7268         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
7269             "hubd_create_pm_components: Begin");
7271         /* Allocate the state structure */
7272         hubpm = kmem_zalloc(sizeof (hub_power_t), KM_SLEEP);
7274         hubd->h_hubpm = hubpm;
7275         hubpm->hubp_hubd = hubd;
7276         hubpm->hubp_pm_capabilities = 0;
7277         hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
7278         hubpm->hubp_time_at_full_power = gethrtime();
7279         hubpm->hubp_min_pm_threshold = hubdi_min_pm_threshold * NANOSEC;
6548         hubpm->hubp_time_at_full_power = ddi_get_time();
6549         hubpm->hubp_min_pm_threshold = hubdi_min_pm_threshold;
7281         /* alloc memory to save power states of children */
7282         hubpm->hubp_child_pwrstate = (uint8_t *)
7283             kmem_zalloc(MAX_PORTS + 1, KM_SLEEP);
7285         /*
7286          * if the enable remote wakeup fails
7287          * we still want to enable
7288          * parent notification so we can PM the children
7289          */
7290         usb_enable_parent_notification(dip);
7292         if (usb_handle_remote_wakeup(dip,
7293             USB_REMOTE_WAKEUP_ENABLE) == USB_SUCCESS) {
7294                 uint_t          pwr_states;
7296                 USB_DPRINTF_L2(DPRINT_MASK_PM, hubd->h_log_handle,
7297                     "hubd_create_pm_components: "
7298                     "Remote Wakeup Enabled");
7300                 if (usb_create_pm_components(dip, &pwr_states) ==
7301                     USB_SUCCESS) {
```

```
7302                        mutex_enter(HUBD_MUTEX(hubd));
7303                        hubpm->hubp_wakeup_enabled = 1;
7304                        hubpm->hubp_pwr_states = (uint8_t)pwr_states;

7306                        /* we are busy now till end of the attach */
7307                        hubd_pm_busy_component(hubd, dip, 0);
7308                        mutex_exit(HUBD_MUTEX(hubd));

7310                        /* bring the device to full power */
7311                        (void) pm_raise_power(dip, 0,
7312                            USB_DEV_OS_FULL_PWR);
7313                }
7314        }

7316        USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
7317            "hubd_create_pm_components: END");
7318 }
```

*unchanged_portion_omitted_*

```
8647 /*
8648  * hubd_reset_thread:
8649  *      handles the "USB_RESET_LVL_REATTACH" reset of usb device.
8650  *
8651  *      - delete the child (force detaching the device and its children)
8652  *      - reset the corresponding parent hub port
8653  *      - create the child (force re-attaching the device and its children)
8654  */
8655 static void
8656 hubd_reset_thread(void *arg)
8657 {
8658        hubd_reset_arg_t *hd_arg = (hubd_reset_arg_t *)arg;
8659        hubd_t           *hubd = hd_arg->hubd;
8660        uint16_t         reset_port = hd_arg->reset_port;
8661        uint16_t         status, change;
8662        hub_power_t      *hubpm;
8663        dev_info_t       *hdip = hubd->h_dip;
8664        dev_info_t       *rh_dip = hubd->h_usba_device->usb_root_hub_dip;
8665        dev_info_t       *child_dip;
8666        boolean_t        online_child = B_FALSE;
8667        int              prh_circ, rh_circ, circ, devinst;
8668        char             *devname;
8669        int              i = 0;
8670        int              rval = USB_FAILURE;

8672        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8673            "hubd_reset_thread:  started, hubd_reset_port = 0x%x", reset_port);

8675        kmem_free(arg, sizeof (hubd_reset_arg_t));

8677        mutex_enter(HUBD_MUTEX(hubd));

8679        child_dip = hubd->h_children_dips[reset_port];
8680        ASSERT(child_dip != NULL);

8682        devname = (char *)ddi_driver_name(child_dip);
8683        devinst = ddi_get_instance(child_dip);

8685        /* if our bus power entry point is active, quit the reset */
8686        if (hubd->h_bus_pwr) {
8687                USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8688                    "%s%d is under bus power management, cannot be reset. "
8689                    "Please disconnect and reconnect this device.",
8690                    devname, devinst);

8692                goto Fail;
8693        }
```



```
8695        if (hubd_wait_for_hotplug_exit(hubd) == USB_FAILURE) {
8696                /* we got woken up because of a timeout */
8697                USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG,
8698                    hubd->h_log_handle, "Time out when resetting the device"
8699                    " %s%d. Please disconnect and reconnect this device.",
8700                    devname, devinst);

8702                goto Fail;
8703        }

8705        hubd->h_hotplug_thread++;

8707        /* is this the root hub? */
8708        if ((hdip == rh_dip) &&
8709            (hubd->h_dev_state == USB_DEV_PWRED_DOWN)) {
8710                hubpm = hubd->h_hubpm;

8712                /* mark the root hub as full power */
8713                hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
8714                hubpm->hubp_time_at_full_power = gethrtime();
7984                hubpm->hubp_time_at_full_power = ddi_get_time();
8715                mutex_exit(HUBD_MUTEX(hubd));

8717                USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8718                    "hubd_reset_thread: call pm_power_has_changed");

8720                (void) pm_power_has_changed(hdip, 0,
8721                    USB_DEV_OS_FULL_PWR);

8723                mutex_enter(HUBD_MUTEX(hubd));
8724                hubd->h_dev_state = USB_DEV_ONLINE;
8725        }

8727        mutex_exit(HUBD_MUTEX(hubd));

8729        /*
8730         * this ensures one reset activity per system at a time.
8731         * we enter the parent PCI node to have this serialization.
8732         * this also excludes ioctls and deathrow thread
8733         */
8734        ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
8735        ndi_devi_enter(rh_dip, &rh_circ);

8737        /* exclude other threads */
8738        ndi_devi_enter(hdip, &circ);
8739        mutex_enter(HUBD_MUTEX(hubd));

8741        /*
8742         * We need to make sure that the child is still online for a hotplug
8743         * thread could have inserted which detached the child.
8744         */
8745        if (hubd->h_children_dips[reset_port]) {
8746                mutex_exit(HUBD_MUTEX(hubd));
8747                /* First disconnect the device */
8748                hubd_post_event(hubd, reset_port, USBA_EVENT_TAG_HOT_REMOVAL);

8750                /* delete cached dv_node's but drop locks first */
8751                ndi_devi_exit(hdip, circ);
8752                ndi_devi_exit(rh_dip, rh_circ);
8753                ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

8755                (void) devfs_clean(rh_dip, NULL, DV_CLEAN_FORCE);

8757                /*
8758                 * workaround only for storage device. When it's able to force
```

```
8759                 * detach a driver, this code can be removed safely.
8760                 *
8761                 * If we're to reset storage device and the device is used, we
8762                 * will wait at most extra 20s for applications to exit and
8763                 * close the device. This is especially useful for HAL-based
8764                 * applications.
8765                 */
8766                if ((strcmp(devname, "scsa2usb") == 0) &&
8767                    DEVI(child_dip)->devi_ref != 0) {
8768                        while (i++ < hubdi_reset_delay) {
8769                                mutex_enter(HUBD_MUTEX(hubd));
8770                                rval = hubd_delete_child(hubd, reset_port,
8771                                    NDI_DEVI_REMOVE, B_FALSE);
8772                                mutex_exit(HUBD_MUTEX(hubd));
8773                                if (rval == USB_SUCCESS)
8774                                        break;

8776                                delay(drv_usectohz(1000000)); /* 1s */
8777                        }
8778                }

8780                ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
8781                ndi_devi_enter(rh_dip, &rh_circ);
8782                ndi_devi_enter(hdip, &circ);

8784                mutex_enter(HUBD_MUTEX(hubd));

8786                /* Then force detaching the device */
8787                if ((rval != USB_SUCCESS) && (hubd_delete_child(hubd,
8788                    reset_port, NDI_DEVI_REMOVE, B_FALSE) != USB_SUCCESS)) {
8789                        USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8790                            "%s%d cannot be reset due to other applications "
8791                            "are using it, please first close these "
8792                            "applications, then disconnect and reconnect"
8793                            "the device.", devname, devinst);

8795                        mutex_exit(HUBD_MUTEX(hubd));
8796                        /* post a re-connect event */
8797                        hubd_post_event(hubd, reset_port,
8798                            USBA_EVENT_TAG_HOT_INSERTION);
8799                        mutex_enter(HUBD_MUTEX(hubd));
8800                } else {
8801                        (void) hubd_determine_port_status(hubd, reset_port,
8802                            &status, &change, HUBD_ACK_ALL_CHANGES);

8804                        /* Reset the parent hubd port and create new child */
8805                        if (status & PORT_STATUS_CCS) {
8806                                online_child |= (hubd_handle_port_connect(hubd,
8807                                    reset_port) == USB_SUCCESS);
8808                        }
8809                }
8810        }

8812        /* release locks so we can do a devfs_clean */
8813        mutex_exit(HUBD_MUTEX(hubd));

8815        /* delete cached dv_node's but drop locks first */
8816        ndi_devi_exit(hdip, circ);
8817        ndi_devi_exit(rh_dip, rh_circ);
8818        ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

8820        (void) devfs_clean(rh_dip, NULL, 0);

8822        /* now check if any children need onlining */
8823        if (online_child) {
8824                USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
```



```
8825                    "hubd_reset_thread: onlining children");

8827                (void) ndi_devi_online(hubd->h_dip, 0);
8828        }

8830        mutex_enter(HUBD_MUTEX(hubd));

8832        /* allow hotplug thread now */
8833        hubd->h_hotplug_thread--;
8834 Fail:
8835        hubd_start_polling(hubd, 0);

8837        /* mark this device as idle */
8838        (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);

8840        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8841            "hubd_reset_thread: exit, %d", hubd->h_hotplug_thread);

8843        hubd->h_reset_port[reset_port] = B_FALSE;

8845        mutex_exit(HUBD_MUTEX(hubd));

8847        ndi_rele_devi(hdip);
8848 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    1694 Mon May  5 14:29:46 2014
new/usr/src/uts/common/sys/fibre-channel/fca/emlxs/emlxs_device.h
4786 emlxs shouldn't abuse ddi_get_time(9f)
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Emulex.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _EMLXS_DEVICE_H
#define _EMLXS_DEVICE_H

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * This is the global device driver control structure
 */

#ifndef EMLXS_HBA_T
typedef struct emlxs_hba emlxs_hba_t;
#endif

/* This structure must match the one in ./mdb/msgblib.c */
typedef struct emlxs_device
{
        uint32_t hba_count;
        emlxs_hba_t *hba[MAX_FC_BRDS];
        kmutex_t lock;

        hrtime_t drv_timestamp;
        time_t drv_timestamp;
        clock_t log_timestamp;
        emlxs_msg_log_t *log[MAX_FC_BRDS];

#ifdef DUMP_SUPPORT
        emlxs_file_t *dump_txtfile[MAX_FC_BRDS];
        emlxs_file_t *dump_dmpfile[MAX_FC_BRDS];
        emlxs_file_t *dump_ceefile[MAX_FC_BRDS];
```



```
#endif /* DUMP_SUPPORT */

} emlxs_device_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   24349 Mon May  5 14:29:46 2014
new/usr/src/uts/common/sys/fibre-channel/fca/emlxs/emlxs_dhchap.h
4786 emlxs shouldn't abuse ddi_get_time(9f)
**********************************************************
```

```
/*
 * 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 Emulex.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _EMLXS_DHCHAP_H
#define _EMLXS_DHCHAP_H

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef DHCHAP_SUPPORT
#include <sys/random.h>


/* emlxs_auth_cfg_t */
#define PASSWORD_TYPE_ASCII     1
#define PASSWORD_TYPE_BINARY    2
#define PASSWORD_TYPE_IGNORE    3

#define AUTH_MODE_DISABLED      1
#define AUTH_MODE_ACTIVE        2
#define AUTH_MODE_PASSIVE       3

#define ELX_DHCHAP              0x01    /* Only one currently supported */
#define ELX_FCAP                0x02
#define ELX_FCPAP               0x03
#define ELX_KERBEROS            0x04

#define ELX_MD5                 0x01
#define ELX_SHA1                0x02

#define ELX_GROUP_NULL          0x01
#define ELX_GROUP_1024          0x02
#define ELX_GROUP_1280          0x03
```



```
#define ELX_GROUP_1536          0x04
#define ELX_GROUP_2048          0x05


/* AUTH_ELS Code */
#define ELS_CMD_AUTH_CODE       0x90

/* AUTH_ELS Flags */

/* state ? */
#define AUTH_FINISH             0xFF
#define AUTH_ABORT              0xFE

/* auth_msg code for DHCHAP */
#define AUTH_REJECT             0x0A
#define AUTH_NEGOTIATE          0x0B
#define AUTH_DONE               0x0C
#define DHCHAP_CHALLENGE        0x10
#define DHCHAP_REPLY            0x11
#define DHCHAP_SUCCESS          0x12

/* BIG ENDIAN and LITTLE ENDIAN */

/* authentication protocol identifiers */
#ifdef EMLXS_BIG_ENDIAN

#define AUTH_DHCHAP             0x00000001
#define AUTH_FCAP               0x00000002
#define AUTH_FCPAP              0x00000003
#define AUTH_KERBEROS           0x00000004

#define HASH_LIST_TAG           0x0001
#define DHGID_LIST_TAG          0x0002

/* hash function identifiers */
#define AUTH_SHA1               0x00000006
#define AUTH_MD5                0x00000005

/* DHCHAP group ids */
#define GROUP_NULL              0x00000000
#define GROUP_1024              0x00000001
#define GROUP_1280              0x00000002
#define GROUP_1536              0x00000003
#define GROUP_2048              0x00000004

/* Tran_id Mask */
#define AUTH_TRAN_ID_MASK       0x000000FF

#endif  /* EMLXS_BIG_ENDIAN */

#ifdef EMLXS_LITTLE_ENDIAN

#define AUTH_DHCHAP             0x01000000
#define AUTH_FCAP               0x02000000
#define AUTH_FCPAP              0x03000000
#define AUTH_KERBEROS           0x04000000

#define HASH_LIST_TAG           0x0100
#define DHGID_LIST_TAG          0x0200

/* hash function identifiers */
#define AUTH_SHA1               0x06000000
#define AUTH_MD5                0x05000000

/* DHCHAP group ids */
#define GROUP_NULL              0x00000000
```

```
128 #define GROUP_1024              0x01000000
129 #define GROUP_1280              0x02000000
130 #define GROUP_1536              0x03000000
131 #define GROUP_2048              0x04000000

133 /* Tran_id Mask */
134 #define AUTH_TRAN_ID_MASK       0xFF000000

136 #endif  /* EMLXS_LITTLE_ENDIAN */

138 /* hash funcs hash length in byte */
139 #define SHA1_LEN                0x00000014      /* 20 bytes */
140 #define MD5_LEN                 0x00000010      /* 16 bytes */

142 #define HBA_SECURITY                    0x20

144 /* AUTH_Reject Reason Codes */
145 #define AUTHRJT_FAILURE                 0x01
146 #define AUTHRJT_LOGIC_ERR               0x02

148 /* LS_RJT Reason Codes for AUTH_ELS */
149 #define LSRJT_AUTH_REQUIRED             0x03
150 #define LSRJT_AUTH_LOGICAL_BSY          0x05
151 #define LSRJT_AUTH_ELS_NOT_SUPPORTED    0x0B
152 #define LSRJT_AUTH_NOT_LOGGED_IN        0x09

154 /* AUTH_Reject Reason Code Explanations */
155 #define AUTHEXP_MECH_UNUSABLE           0x01 /* AUTHRJT_LOGIC_ERR */
156 #define AUTHEXP_DHGROUP_UNUSABLE        0x02 /* AUTHRJT_LOGIC_ERR */
157 #define AUTHEXP_HASHFUNC_UNUSABLE       0x03 /* AUTHRJT_LOGIC_ERR */
158 #define AUTHEXP_AUTHTRAN_STARTED        0x04 /* AUTHRJT_LOGIC_ERR */
159 #define AUTHEXP_AUTH_FAILED             0x05 /* AUTHRJT_FAILURE */
160 #define AUTHEXP_BAD_PAYLOAD             0x06 /* AUTHRJT_FAILURE */
161 #define AUTHEXP_BAD_PROTOCOL            0x07 /* AUTHRJT_FAILURE */
162 #define AUTHEXP_RESTART_AUTH            0x08 /* AUTHRJT_LOGIC_ERR */
163 #define AUTHEXP_CONCAT_UNSUPP           0x09 /* AUTHRJT_LOGIC_ERR */
164 #define AUTHEXP_BAD_PROTOVERS           0x0A /* AUTHRJT_LOGIC_ERR */

166 /* LS_RJT Reason Code Explanations for AUTH_ELS */
167 #define LSEXP_AUTH_REQUIRED             0x48
168 #define LSEXP_AUTH_ELS_NOT_SUPPORTED    0x2C
169 #define LSEXP_AUTH_ELS_NOT_LOGGED_IN    0x1E
170 #define LSEXP_AUTH_LOGICAL_BUSY         0x00


173 #define MAX_AUTH_MSA_SIZE 1024

175 #define MAX_AUTH_PID    0x4     /* Max auth proto identifier list */

177 /* parameter tag */
178 #define HASH_LIST       0x0001
179 #define DHG_ID_LIST     0x0002

181 /* name tag from Table 13 v1.8 pp 30 */
182 #ifdef EMLXS_BIG_ENDIAN
183 #define AUTH_NAME_ID            0x0001
184 #define AUTH_NAME_LEN           0x0008
185 #define AUTH_PROTO_NUM          0x00000001
186 #define AUTH_NULL_PARA_LEN      0x00000028
187 #endif  /* EMLXS_BIG_ENDIAN */

189 #ifdef EMLXS_LITTLE_ENDIAN
190 #define AUTH_NAME_ID            0x0100
191 #define AUTH_NAME_LEN           0x0800
192 #define AUTH_PROTO_NUM          0x01000000
193 #define AUTH_NULL_PARA_LEN      0x28000000
```



```
194 #endif  /* EMLXS_LITTLE_ENDIAN */

196 /* name tag from Table 103 v 1.8 pp 123 */
197 #define AUTH_NODE_NAME          0x0002
198 #define AUTH_PORT_NAME          0x0003

200 /*
201  * Sysevent support
202  */
203 /* ddi_log_sysevent() vendors */
204 #define DDI_VENDOR_EMLX         "EMLXS"

206 /* Class */
207 #define EC_EMLXS        "EC_emlxs"

209 /* Subclass */
210 #define ESC_EMLXS_01    "ESC_emlxs_issue_auth_negotiate"
211 #define ESC_EMLXS_02    "ESC_emlxs_cmpl_auth_negotiate_issue"

213 #define ESC_EMLXS_03    "ESC_emlxs_rcv_auth_msg_auth_negotiate_issue"
214 #define ESC_EMLXS_04    "ESC_emlxs_cmpl_auth_msg_auth_negotiate_issue"

216 #define ESC_EMLXS_05    "ESC_emlxs_rcv_auth_msg_unmapped_node"
217 #define ESC_EMLXS_06    "ESC_emlxs_issue_dhchap_challenge"
218 #define ESC_EMLXS_07    "ESC_emlxs_cmpl_dhchap_challenge_issue"

220 #define ESC_EMLXS_08    "ESC_emlxs_rcv_auth_msg_dhchap_challenge_cmpl_wait4next"

222 #define ESC_EMLXS_09    "ESC_emlxs_rcv_auth_msg_auth_negotiate_rcv"
223 #define ESC_EMLXS_10    "ESC_emlxs_cmpl_auth_msg_auth_negotiate_rcv"

225 #define ESC_EMLXS_11    "ESC_emlxs_cmpl_cmpl_dhchap_reply_issue"
226 #define ESC_EMLXS_12    "ESC_emlxs_cmpl_dhchap_reply_issue"
227 #define ESC_EMLXS_13    "ESC_emlxs_cmpl_auth_msg_dhchap_reply_issue"

229 #define ESC_EMLXS_14    "ESC_emlxs_cmpl_auth_msg_auth_negotiate_cmpl_wait4next"

231 #define ESC_EMLXS_15    "ESC_emlxs_issue_dhchap_success"

233 #define ESC_EMLXS_16    "ESC_emlxs_rcv_auth_msg_dhchap_challenge_issue"
234 #define ESC_EMLXS_17    "ESC_emlxs_cmpl_auth_msg_dhchap_challenge_issue"

236 #define ESC_EMLXS_18    "ESC_emlxs_rcv_auth_msg_dhchap_reply_issue"

238 #define ESC_EMLXS_19 \
239         "ESC_emlxs_cmpl_auth_msg_dhchap_challenge_cmpl_wait4next"

241 #define ESC_EMLXS_20    "ESC_emlxs_rcv_auth_msg_dhchap_reply_cmpl_wait4next"
242 #define ESC_EMLXS_21    "ESC_emlxs_cmpl_dhchap_success_issue"
243 #define ESC_EMLXS_22    "ESC_emlxs_cmpl_auth_msg_dhchap_success_issue"

245 #define ESC_EMLXS_23    "ESC_emlxs_cmpl_auth_msg_dhchap_reply_cmpl_wait4next"

247 #define ESC_EMLXS_24    "ESC_emlxs_rcv_auth_msg_dhchap_success_issue_wait4next"
248 #define ESC_EMLXS_25    "ESC_emlxs_cmpl_auth_msg_dhchap_success_issue_wait4next"

250 #define ESC_EMLXS_26    "ESC_emlxs_rcv_auth_msg_dhchap_success_cmpl_wait4next"
251 #define ESC_EMLXS_27    "ESC_emlxs_cmpl_auth_msg_dhchap_success_cmpl_wait4next"

253 #define ESC_EMLXS_28    "ESC_emlxs_issue_auth_reject"
254 #define ESC_EMLXS_29    "ESC_emlxs_cmpl_auth_reject_issue"

256 #define ESC_EMLXS_30    "ESC_emlxs_rcv_auth_msg_npr_node"

258 #define ESC_EMLXS_31    "ESC_emlxs_dhc_reauth_timeout"
```

```
#define ESC_EMLXS_32    "ESC_emlxs_dhc_authrsp_timeout"

#define ESC_EMLXS_33    "ESC_emlxs_ioctl_auth_setcfg"
#define ESC_EMLXS_34    "ESC_emlxs_ioctl_auth_setpwd"
#define ESC_EMLXS_35    "ESC_emlxs_ioctl_auth_delcfg"
#define ESC_EMLXS_36    "ESC_emlxs_ioctl_auth_delpwd"


/* From HBAnyware dfc lib FC-SP */
typedef struct emlxs_auth_cfg
{
        NAME_TYPE               local_entity;   /* host wwpn (NPIV support) */
        NAME_TYPE               remote_entity;  /* switch or target wwpn */
        uint32_t                authentication_timeout;
        uint32_t                authentication_mode;
        uint32_t                bidirectional:1;
        uint32_t                reserved:31;
        uint32_t                authentication_type_priority[4];
        uint32_t                hash_priority[4];
        uint32_t                dh_group_priority[8];
        uint32_t                reauthenticate_time_interval;

        dfc_auth_status_t       auth_status;
        uint32_t                auth_time;
        time_t                  auth_time;
        struct emlxs_node       *node;

        struct emlxs_auth_cfg   *prev;
        struct emlxs_auth_cfg   *next;
} emlxs_auth_cfg_t;
______unchanged_portion_omitted_


/*
 * emlxs_port_dhc struct to be used by emlxs_port_t in emlxs_fc.h
 *
 * This structure contains all the data used by DHCHAP.
 * They are from EMLXSHBA_t in emlxs driver.
 *
 */
typedef struct emlxs_port_dhc
{

        int32_t                 state;
#define ELX_FABRIC_STATE_UNKNOWN        0x00
#define ELX_FABRIC_AUTH_DISABLED        0x01
#define ELX_FABRIC_AUTH_FAILED          0x02
#define ELX_FABRIC_AUTH_SUCCESS         0x03
#define ELX_FABRIC_IN_AUTH              0x04
#define ELX_FABRIC_IN_REAUTH            0x05

        dfc_auth_status_t       auth_status;  /* Fabric auth status */
        uint32_t                auth_time;
        time_t                  auth_time;

} emlxs_port_dhc_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   57703 Mon May  5 14:29:46 2014
new/usr/src/uts/common/sys/fibre-channel/fca/emlxs/emlxs_fc.h
4786 emlxs shouldn't abuse ddi_get_time(9f)
**********************************************************
/*
 * 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 Emulex.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _EMLXS_FC_H
#define _EMLXS_FC_H

#ifdef  __cplusplus
extern "C" {
#endif

typedef struct emlxs_buf
{
        fc_packet_t             *pkt;           /* scsi_pkt reference */
        struct emlxs_port       *port;          /* pointer to port */
        void                    *bmp;           /* Save the buffer pointer */
                                                /* list for later use. */
        struct emlxs_buf        *fc_fwd;        /* Use it by chip_Q */
        struct emlxs_buf        *fc_bkwd;       /* Use it by chip_Q */
        struct emlxs_buf        *next;          /* Use it when the iodone */
        struct emlxs_node       *node;
        void                    *channel;       /* Save channel and used by */
                                                /* abort */
        struct emlxs_buf        *fpkt;          /* Flush pkt pointer */
        struct XRIobj           *xrip;          /* Exchange resource */
        IOCBQ                   iocbq;
        kmutex_t                mtx;
        uint32_t                pkt_flags;
        uint32_t                iotag;          /* iotag for this cmd */
        uint32_t                ticks;          /* save the timeout ticks */
                                                /* for the fc_packet_t */
        uint32_t                abort_attempts;
        uint32_t                lun;
#define EMLXS_LUN_NONE          0xFFFFFFFF
```



```
        uint32_t                class;          /* Save class and used by */
                                                /* abort */
        uint32_t                ucmd;           /* Unsolicted command that */
                                                /* this packet is responding */
                                                /* to, if any */
        int32_t                 flush_count;    /* Valid only in flush pkts */
        uint32_t                did;

#ifdef SFCT_SUPPORT
        kmutex_t                fct_mtx;
        fc_packet_t             *fct_pkt;
        fct_cmd_t               *fct_cmd;

        uint8_t                 fct_type;

#define EMLXS_FCT_ELS_CMD               0x01    /* Unsolicted */
#define EMLXS_FCT_ELS_REQ               0x02    /* Solicited */
#define EMLXS_FCT_ELS_RSP               0x04
#define EMLXS_FCT_CT_REQ                0x08    /* Solicited */
#define EMLXS_FCT_FCP_CMD               0x10    /* Unsolicted */
#define EMLXS_FCT_FCP_DATA              0x20
#define EMLXS_FCT_FCP_STATUS            0x40


        uint8_t                 fct_flags;

#define EMLXS_FCT_SEND_STATUS           0x01
#define EMLXS_FCT_ABORT_INP             0x02
#define EMLXS_FCT_IO_INP                0x04
#define EMLXS_FCT_PLOGI_RECEIVED        0x10
#define EMLXS_FCT_REGISTERED            0x20

        uint16_t                fct_state;

#define EMLXS_FCT_FCP_CMD_RECEIVED      1
#define EMLXS_FCT_ELS_CMD_RECEIVED      2
#define EMLXS_FCT_CMD_POSTED            3
#define EMLXS_FCT_CMD_WAITQ             4
#define EMLXS_FCT_SEND_CMD_RSP          5
#define EMLXS_FCT_SEND_ELS_RSP          6
#define EMLXS_FCT_SEND_ELS_REQ          7
#define EMLXS_FCT_SEND_CT_REQ           8
#define EMLXS_FCT_RSP_PENDING           9
#define EMLXS_FCT_REQ_PENDING           10
#define EMLXS_FCT_REG_PENDING           11
#define EMLXS_FCT_REG_COMPLETE          12
#define EMLXS_FCT_OWNED                 13
#define EMLXS_FCT_SEND_FCP_DATA         14
#define EMLXS_FCT_SEND_FCP_STATUS       15
#define EMLXS_FCT_DATA_PENDING          16
#define EMLXS_FCT_STATUS_PENDING        17
#define EMLXS_FCT_PKT_COMPLETE          18
#define EMLXS_FCT_PKT_FCPRSP_COMPLETE   19
#define EMLXS_FCT_PKT_ELSRSP_COMPLETE   20
#define EMLXS_FCT_PKT_ELSCMD_COMPLETE   21
#define EMLXS_FCT_PKT_CTCMD_COMPLETE    22
#define EMLXS_FCT_REQ_COMPLETE          23
#define EMLXS_FCT_CLOSE_PENDING         24
#define EMLXS_FCT_ABORT_PENDING         25
#define EMLXS_FCT_ABORT_DONE            26
#define EMLXS_FCT_IO_DONE               27

#define EMLXS_FCT_IOCB_ISSUED           256 /* For tracing only */
#define EMLXS_FCT_IOCB_COMPLETE         257 /* For tracing only */

        stmf_data_buf_t         *fct_buf;
```

```
#endif /* SFCT_SUPPORT */

#ifdef SAN_DIAG_SUPPORT
        hrtime_t                sd_start_time;
#endif
} emlxs_buf_t;



#ifdef FCT_IO_TRACE
#define EMLXS_FCT_STATE_CHG(_fct_cmd, _cmd_sbp, _state) \
        (_cmd_sbp)->fct_state = _state;                 \
        emlxs_fct_io_trace((_cmd_sbp)->port, _fct_cmd, _state)
#else
/* define to set fct_state */
#define EMLXS_FCT_STATE_CHG(_fct_cmd, _cmd_sbp, _state) \
        (_cmd_sbp)->fct_state = _state
#endif /* FCT_IO_TRACE */


/* pkt_flags */
#define PACKET_IN_COMPLETION    0x00000001
#define PACKET_IN_TXQ           0x00000002
#define PACKET_IN_CHIPQ         0x00000004
#define PACKET_IN_DONEQ         0x00000008

#define PACKET_FCP_RESET        0x00000030
#define PACKET_FCP_TGT_RESET    0x00000010
#define PACKET_FCP_LUN_RESET    0x00000020
#define PACKET_POLLED           0x00000040

#ifdef EMLXS_I386
#define PACKET_FCP_SWAPPED      0x00000100
#define PACKET_ELS_SWAPPED      0x00000200
#define PACKET_CT_SWAPPED       0x00000400
#define PACKET_CSP_SWAPPED      0x00000800
#endif  /* EMLXS_I386 */

#define PACKET_STALE            0x00001000

#define PACKET_IN_TIMEOUT       0x00010000
#define PACKET_IN_FLUSH         0x00020000
#define PACKET_IN_ABORT         0x00040000
#define PACKET_XRI_CLOSED       0x00080000 /* An XRI abort/close was issued */

#define PACKET_CHIP_COMP        0x00100000
#define PACKET_COMPLETED        0x00200000
#define PACKET_ULP_OWNED        0x00400000

#define PACKET_STATE_VALID      0x01000000
#define PACKET_FCP_RSP_VALID    0x02000000
#define PACKET_ELS_RSP_VALID    0x04000000
#define PACKET_CT_RSP_VALID     0x08000000

#define PACKET_DELAY_REQUIRED   0x10000000
#define PACKET_ALLOCATED        0x40000000
#define PACKET_VALID            0x80000000


#define STALE_PACKET            ((emlxs_buf_t *)0xFFFFFFFF)


/*
 * From fc_error.h pkt_reason (except for state = NPORT_RJT, FABRIC_RJT,
 * NPORT_BSY, FABRIC_BSY, LS_RJT, BA_RJT, FS_RJT)
```



```
 *
 * FCA unique error codes can begin after FC_REASON_FCA_UNIQUE.
 * Each FCA defines its own set with values greater >= 0x7F
 */
#define FC_REASON_FCA_DEFINED   0x100


/*
 * Device VPD save area
 */

typedef struct emlxs_vpd
{
        uint32_t        biuRev;
        uint32_t        smRev;
        uint32_t        smFwRev;
        uint32_t        endecRev;
        uint16_t        rBit;
        uint8_t         fcphHigh;
        uint8_t         fcphLow;
        uint8_t         feaLevelHigh;
        uint8_t         feaLevelLow;

        uint32_t        postKernRev;
        char            postKernName[32];

        uint32_t        opFwRev;
        char            opFwName[32];
        char            opFwLabel[32];

        uint32_t        sli1FwRev;
        char            sli1FwName[32];
        char            sli1FwLabel[32];

        uint32_t        sli2FwRev;
        char            sli2FwName[32];
        char            sli2FwLabel[32];

        uint32_t        sli3FwRev;
        char            sli3FwName[32];
        char            sli3FwLabel[32];

        uint32_t        sli4FwRev;
        char            sli4FwName[32];
        char            sli4FwLabel[32];

        char            fw_version[32];
        char            fw_label[32];

        char            fcode_version[32];
        char            boot_version[32];

        char            serial_num[32];
        char            part_num[32];
        char            port_num[20];
        char            eng_change[32];
        char            manufacturer[80];
        char            model[80];
        char            model_desc[256];
        char            prog_types[256];
        char            id[80];

        uint32_t        port_index;
        uint16_t        link_speed;
} emlxs_vpd_t;
```

```
typedef struct emlxs_queue
{
        void            *q_first;       /* queue first element */
        void            *q_last;        /* queue last element */
        uint16_t        q_cnt;  /* current length of queue */
        uint16_t        q_max;  /* max length queue can get */
} emlxs_queue_t;
typedef emlxs_queue_t Q;



/*
 * This structure is used when allocating a buffer pool.
 * Note: this should be identical to gasket buf_info (fldl.h).
 */
typedef struct emlxs_buf_info
{
        int32_t         size;   /* Specifies the number of bytes to allocate. */
        int32_t         align;  /* The desired address boundary. */

        int32_t         flags;

#define FC_MBUF_DMA             0x01    /* blocks are for DMA */
#define FC_MBUF_PHYSONLY        0x02    /* For malloc - map a given virtual */
                                        /* address to physical address (skip */
                                        /* the malloc). */
                                        /* For free - just unmap the given */
                                        /* physical address (skip the free). */
#define FC_MBUF_IOCTL           0x04    /* called from dfc_ioctl */
#define FC_MBUF_UNLOCK          0x08    /* called with driver unlocked */
#define FC_MBUF_SNGLSG          0x10    /* allocate a single contiguous */
                                        /* physical memory */
#define FC_MBUF_DMA32           0x20

        uint64_t        phys;           /* specifies physical buffer pointer */
        void            *virt;          /* specifies virtual buffer pointer */
        void            *data_handle;
        void            *dma_handle;
} emlxs_buf_info_t;
typedef emlxs_buf_info_t MBUF_INFO;


#define EMLXS_MAX_HBQ           16      /* Max HBQs handled by firmware */
#define EMLXS_ELS_HBQ_ID        0
#define EMLXS_IP_HBQ_ID         1
#define EMLXS_CT_HBQ_ID         2
#define EMLXS_FCT_HBQ_ID        3

#ifdef SFCT_SUPPORT
#define EMLXS_NUM_HBQ           4       /* Number of HBQs supported by driver */
#else
#define EMLXS_NUM_HBQ           3       /* Number of HBQs supported by driver */
#endif /* SFCT_SUPPORT */


/*
 * An IO Channel is a object that comprises a xmit/cmpl
 * path for IOs.
 * For SLI3, an IO path maps to a ring (cmd/rsp)
 * For SLI4, an IO path map to a queue pair (WQ/CQ)
 */
typedef struct emlxs_channel
{
        struct emlxs_hba *hba;                  /* ptr to hba for channel */
        void            *iopath;                /* ptr to SLI3/4 io path */
```



```
        kmutex_t        rsp_lock;
        IOCBQ           *rsp_head;      /* deferred completion head */
        IOCBQ           *rsp_tail;      /* deferred completion tail */
        emlxs_thread_t  intr_thread;


        uint16_t        channelno;
        uint16_t        chan_flag;

#define EMLXS_NEEDS_TRIGGER 1

        /* Protected by EMLXS_TX_CHANNEL_LOCK */
        emlxs_queue_t   nodeq;                  /* Node service queue */

        kmutex_t        channel_cmd_lock;
        uint32_t        timeout;

        /* Channel command counters */
        uint32_t        ulpSendCmd;
        uint32_t        ulpCmplCmd;
        uint32_t        hbaSendCmd;
        uint32_t        hbaCmplCmd;
        uint32_t        hbaSendCmd_sbp;
        uint32_t        hbaCmplCmd_sbp;

} emlxs_channel_t;
typedef emlxs_channel_t CHANNEL;

/*
 * Should be able to handle max number of io paths for a
 * SLI4 HBA (EMLXS_MAX_WQS) or for a SLI3 HBA (MAX_RINGS)
 */
#define MAX_CHANNEL EMLXS_MSI_MAX_INTRS


/* Structure used to access adapter rings */
typedef struct emlxs_ring
{
        void            *fc_cmdringaddr;        /* virtual offset for cmd */
                                                /* rings */
        void            *fc_rspringaddr;        /* virtual offset for rsp */
                                                /* rings */

        void            *fc_mpon;               /* index ptr for match */
                                                /* structure */
        void            *fc_mpoff;              /* index ptr for match */
                                                /* structure */
        struct emlxs_hba *hba;                  /* ptr to hba for ring */

        uint8_t         fc_numCiocb;            /* number of command iocb's */
                                                /* per ring */
        uint8_t         fc_numRiocb;            /* number of response iocb's */
                                                /* per ring */
        uint8_t         fc_rspidx;              /* current index in response */
                                                /* ring */
        uint8_t         fc_cmdidx;              /* current index in command */
                                                /* ring */
        uint8_t         fc_port_rspidx;
        uint8_t         fc_port_cmdidx;
        uint8_t         ringno;

        uint16_t        fc_missbufcnt;          /* buf cnt we need to repost */
        CHANNEL         *channelp;
```

```
392 } emlxs_ring_t;
393 typedef emlxs_ring_t RING;


396 #ifdef SAN_DIAG_SUPPORT
397 /*
398  * Although right now it's just 1 field, SAN Diag anticipates that this
399  * structure will grow in the future.
400  */
401 typedef struct sd_timestat_level0 {
402         int             count;
403 } sd_timestat_level0_t;
404 #endif

406 typedef struct emlxs_node
407 {
408         struct emlxs_node       *nlp_list_next;
409         struct emlxs_node       *nlp_list_prev;

411         NAME_TYPE               nlp_portname;   /* port name */
412         NAME_TYPE               nlp_nodename;   /* node name */

414         uint32_t                nlp_DID;        /* fibre channel D_ID */
415         uint32_t                nlp_oldDID;

417         uint16_t                nlp_Rpi;        /* login id returned by */
418                                                 /* REG_LOGIN */
419         uint16_t                nlp_Xri;        /* login id returned by */
420                                                 /* REG_LOGIN */

422         uint8_t                 nlp_fcp_info;   /* Remote class info */

424         /* nlp_fcp_info */
425 #define NLP_FCP_TGT_DEVICE      0x10    /* FCP TGT device */
426 #define NLP_FCP_INI_DEVICE      0x20    /* FCP Initiator device */
427 #define NLP_FCP_2_DEVICE        0x40    /* FCP-2 TGT device */
428 #define NLP_EMLX_VPORT          0x80    /* Virtual port */

430         uint32_t                nlp_force_rscn;
431         uint32_t                nlp_tag;        /* Tag used by port_offline */
432         uint32_t                flag;

434 #define NODE_POOL_ALLOCATED     0x00000001

436         SERV_PARM               sparm;

438         /* Protected by EMLXS_TX_CHANNEL_LOCK */
439         uint32_t                nlp_active;     /* Node active flag */
440         uint32_t                nlp_base;
441         uint32_t                nlp_flag[MAX_CHANNEL];  /* Node level channel */
442                                                         /* flags */

444         /* nlp_flag */
445 #define NLP_CLOSED              0x1
446 #define NLP_OFFLINE             0x2
447 #define NLP_RPI_XRI             0x4

449         uint32_t                nlp_tics[MAX_CHANNEL];  /* gate timeout */
450         emlxs_queue_t           nlp_tx[MAX_CHANNEL];    /* Transmit Q head */
451         emlxs_queue_t           nlp_ptx[MAX_CHANNEL];   /* Priority transmit */
452                                                         /* Queue head */
453         void                    *nlp_next[MAX_CHANNEL]; /* Service Request */
454                                                         /* Queue pointer used */
455                                                         /* when node needs */
456                                                         /* servicing */
457 #ifdef DHCHAP_SUPPORT
```



```
458         emlxs_node_dhc_t        node_dhc;
459 #endif  /* DHCHAP_SUPPORT */

461 #ifdef SAN_DIAG_SUPPORT
462         sd_timestat_level0_t    sd_dev_bucket[SD_IO_LATENCY_MAX_BUCKETS];
463 #endif

465         struct RPIobj           *rpip;  /* SLI4 only */
466 #define EMLXS_NODE_TO_RPI(_p, _n)       \
467         ((_n)?((_n->rpip)?_n->rpip:emlxs_rpi_find(_p, _n->nlp_Rpi)):NULL)

469 } emlxs_node_t;
470 typedef emlxs_node_t NODELIST;



474 #define NADDR_LEN       6       /* MAC network address length */
475 typedef struct emlxs_fcip_nethdr
476 {
477         NAME_TYPE       fc_destname;    /* destination port name */
478         NAME_TYPE       fc_srcname;     /* source port name */
479 } emlxs_fcip_nethdr_t;
480 typedef emlxs_fcip_nethdr_t NETHDR;


483 #define MEM_NLP         0       /* memory segment to hold node list entries */
484 #define MEM_IOCB        1       /* memory segment to hold iocb commands */
485 #define MEM_MBOX        2       /* memory segment to hold mailbox cmds  */
486 #define MEM_BPL         3       /* and to hold buffer ptr lists - SLI2   */
487 #define MEM_BUF         4       /* memory segment to hold buffer data   */
488 #define MEM_ELSBUF      4       /* memory segment to hold buffer data   */
489 #define MEM_IPBUF       5       /* memory segment to hold IP buffer data */
490 #define MEM_CTBUF       6       /* memory segment to hold CT buffer data */
491 #define MEM_FCTBUF      7       /* memory segment to hold FCT buffer data */

493 #ifdef SFCT_SUPPORT
494 #define FC_MAX_SEG      8
495 #else
496 #define FC_MAX_SEG      7
497 #endif /* SFCT_SUPPORT */


500 /* A BPL entry is 12 bytes. Subtract 2 for command and response buffers */
501 #define BPL_TO_SGLLEN(_bpl)   ((_bpl/12)-2)
502 #define MEM_BPL_SIZE            1024  /* Default size */

504 /* A SGL entry is 16 bytes. Subtract 2 for command and response buffers */
505 #define SGL_TO_SGLLEN(_sgl)   ((_sgl/16)-2)
506 #define MEM_SGL_SIZE            4096  /* Default size */

508 #ifdef EMLXS_I386
509 #define EMLXS_SGLLEN            BPL_TO_SGLLEN(MEM_BPL_SIZE)
510 #else   /* EMLXS_SPARC */
511 #define EMLXS_SGLLEN            1
512 #endif  /* EMLXS_I386 */

514 #define MEM_BUF_SIZE            1024
515 #define MEM_BUF_COUNT           64

517 #define MEM_ELSBUF_SIZE         MEM_BUF_SIZE
518 #define MEM_ELSBUF_COUNT        hba->max_nodes
519 #define MEM_IPBUF_SIZE          65535
520 #define MEM_IPBUF_COUNT         60
521 #define MEM_CTBUF_SIZE          MAX_CT_PAYLOAD  /* (1024*320) */
522 #define MEM_CTBUF_COUNT         8
523 #define MEM_FCTBUF_SIZE         65535
```

```
524 #define MEM_FCTBUF_COUNT        128

526 typedef struct emlxs_memseg
527 {
528         void                    *fc_memget_ptr;
529         void                    *fc_memget_end;
530         void                    *fc_memput_ptr;
531         void                    *fc_memput_end;

533         void                    *fc_memstart_virt;      /* beginning address */
534                                                         /* of memory block */
535         uint64_t                fc_memstart_phys;       /* beginning address */
536                                                         /* of memory block */
537         ddi_dma_handle_t        fc_mem_dma_handle;
538         ddi_acc_handle_t        fc_mem_dat_handle;
539         uint32_t                fc_total_memsize;
540         uint32_t                fc_memsize;             /* size of mem blks */
541         uint32_t                fc_numblks;             /* no of mem blks */
542         uint32_t                fc_memget_cnt;          /* no of mem get blks */
543         uint32_t                fc_memput_cnt;          /* no of mem put blks */
544         uint32_t                fc_memflag;  /* emlxs_buf_info_t FLAGS */
545         uint32_t                fc_reserved; /* used with priority flag */
546         uint32_t                fc_memalign;
547         uint32_t                fc_memtag;
548         char                    fc_label[32];

550 } emlxs_memseg_t;
551 typedef emlxs_memseg_t MEMSEG;


554 /* Board stat counters */
555 typedef struct emlxs_stats
556 {
557         uint32_t        LinkUp;
558         uint32_t        LinkDown;
559         uint32_t        LinkEvent;
560         uint32_t        LinkMultiEvent;

562         uint32_t        MboxIssued;
563         uint32_t        MboxCompleted;  /* MboxError + MbxGood */
564         uint32_t        MboxGood;
565         uint32_t        MboxError;
566         uint32_t        MboxBusy;
567         uint32_t        MboxInvalid;

569         uint32_t        IocbIssued[MAX_CHANNEL];
570         uint32_t        IocbReceived[MAX_CHANNEL];
571         uint32_t        IocbTxPut[MAX_CHANNEL];
572         uint32_t        IocbTxGet[MAX_CHANNEL];
573         uint32_t        IocbRingFull[MAX_CHANNEL];
574         uint32_t        IocbThrottled;

576         uint32_t        IntrEvent[8];

578         uint32_t        FcpIssued;
579         uint32_t        FcpCompleted;   /* FcpGood + FcpError */
580         uint32_t        FcpGood;
581         uint32_t        FcpError;

583         uint32_t        FcpEvent;       /* FcpStray + FcpCompleted */
584         uint32_t        FcpStray;
585 #ifdef SFCT_SUPPORT
586         uint32_t        FctRingEvent;
587         uint32_t        FctRingError;
588         uint32_t        FctRingDropped;
589 #endif /* SFCT_SUPPORT */
```



```
591         uint32_t        ElsEvent;       /* ElsStray + ElsCmplt (cmd + rsp) */
592         uint32_t        ElsStray;

594         uint32_t        ElsCmdIssued;
595         uint32_t        ElsCmdCompleted;        /* ElsCmdGood + ElsCmdError */
596         uint32_t        ElsCmdGood;
597         uint32_t        ElsCmdError;

599         uint32_t        ElsRspIssued;
600         uint32_t        ElsRspCompleted;

602         uint32_t        ElsRcvEvent;    /* ElsRcvErr + ElsRcvDrop + ElsCmdRcv */
603         uint32_t        ElsRcvError;
604         uint32_t        ElsRcvDropped;
605         uint32_t        ElsCmdReceived; /* ElsRscnRcv + ElsPlogiRcv + ... */
606         uint32_t        ElsRscnReceived;
607         uint32_t        ElsFlogiReceived;
608         uint32_t        ElsPlogiReceived;
609         uint32_t        ElsPrliReceived;
610         uint32_t        ElsPrloReceived;
611         uint32_t        ElsLogoReceived;
612         uint32_t        ElsAdiscReceived;
613         uint32_t        ElsAuthReceived;
614         uint32_t        ElsGenReceived;

616         uint32_t        CtEvent;        /* CtStray + CtCompleted (cmd + rsp) */
617         uint32_t        CtStray;

619         uint32_t        CtCmdIssued;
620         uint32_t        CtCmdCompleted; /* CtCmdGood + CtCmdError */
621         uint32_t        CtCmdGood;
622         uint32_t        CtCmdError;

624         uint32_t        CtRspIssued;
625         uint32_t        CtRspCompleted;

627         uint32_t        CtRcvEvent;     /* CtRcvError + CtRcvDrop + CtCmdRcvd */
628         uint32_t        CtRcvError;
629         uint32_t        CtRcvDropped;
630         uint32_t        CtCmdReceived;

632         uint32_t        IpEvent;        /* IpStray + IpSeqCmpl + IpBcastCmpl */
633         uint32_t        IpStray;

635         uint32_t        IpSeqIssued;
636         uint32_t        IpSeqCompleted; /* IpSeqGood + IpSeqError */
637         uint32_t        IpSeqGood;
638         uint32_t        IpSeqError;

640         uint32_t        IpBcastIssued;
641         uint32_t        IpBcastCompleted;       /* IpBcastGood + IpBcastError */
642         uint32_t        IpBcastGood;
643         uint32_t        IpBcastError;

645         uint32_t        IpRcvEvent;     /* IpDrop + IpSeqRcv + IpBcastRcv */
646         uint32_t        IpDropped;
647         uint32_t        IpSeqReceived;
648         uint32_t        IpBcastReceived;

650         uint32_t        IpUbPosted;
651         uint32_t        ElsUbPosted;
652         uint32_t        CtUbPosted;
653 #ifdef SFCT_SUPPORT
654         uint32_t        FctUbPosted;
655 #endif /* SFCT_SUPPORT */
```

```
657         uint32_t        ResetTime;      /* Time of last reset */

659         uint32_t        ElsTestReceived;
660         uint32_t        ElsEstcReceived;
661         uint32_t        ElsFarprReceived;
662         uint32_t        ElsEchoReceived;
663         uint32_t        ElsRlsReceived;
664         uint32_t        ElsRtvReceived;

666 } emlxs_stats_t;


669 #define FC_MAX_ADPTMSG   (8*28) /* max size of a msg from adapter */

671 #define EMLXS_NUM_THREADS       8
672 #define EMLXS_MIN_TASKS         8
673 #define EMLXS_MAX_TASKS         8

675 #define EMLXS_NUM_HASH_QUES     32
676 #define EMLXS_DID_HASH(x)       ((x) & (EMLXS_NUM_HASH_QUES - 1))


679 /* pkt_tran_flag */
680 #define FC_TRAN_COMPLETED       0x8000


683 typedef struct emlxs_dfc_event
684 {
685         uint32_t        pid;
686         uint32_t        event;
687         uint32_t        last_id;

689         void            *dataout;
690         uint32_t        size;
691         uint32_t        mode;
692 } emlxs_dfc_event_t;


695 typedef struct emlxs_hba_event
696 {
697         uint32_t        last_id;
698         uint32_t        new;
699         uint32_t        missed;
700 } emlxs_hba_event_t;


703 #ifdef SFCT_SUPPORT

705 #define TGTPORTSTAT                     port->fct_stat

707 /*
708  * FctP2IOXcnt will count IOs by their fcpDL. Counters
709  * are for buckets of various power of 2 sizes.
710  * Bucket 0  <  512  > 0
711  * Bucket 1  >= 512  < 1024
712  * Bucket 2  >= 1024 < 2048
713  * Bucket 3  >= 2048 < 4096
714  * Bucket 4  >= 4096 < 8192
715  * Bucket 5  >= 8192 < 16K
716  * Bucket 6  >= 16K  < 32K
717  * Bucket 7  >= 32K  < 64K
718  * Bucket 8  >= 64K  < 128K
719  * Bucket 9  >= 128K < 256K
720  * Bucket 10 >= 256K < 512K
721  * Bucket 11 >= 512K < 1MB
```



```
722  * Bucket 12 >= 1MB  < 2MB
723  * Bucket 13 >= 2MB  < 4MB
724  * Bucket 14 >= 4MB  < 8MB
725  * Bucket 15 >= 8MB
726  */
727 #define MAX_TGTPORT_IOCNT  16


730 /*
731  * These routines will bump the right counter, based on
732  * the size of the IO inputed, with the least number of
733  * comparisions.  A max of 5 comparisions is only needed
734  * to classify the IO in one of 16 ranges. A binary search
735  * to locate the high bit in the size is used.
736  */
737 #define EMLXS_BUMP_RDIOCTR(port, cnt) \
738 { \
739         /* Use binary search to find the first high bit */ \
740         if (cnt & 0xffff0000) { \
741                 if (cnt & 0xff800000) { \
742                         TGTPORTSTAT.FctP2IORcnt[15]++; \
743                 } \
744                 else { \
745                         /* It must be 0x007f0000 */ \
746                         if (cnt & 0x00700000) { \
747                                 if (cnt & 0x00400000) { \
748                                         TGTPORTSTAT.FctP2IORcnt[14]++; \
749                                 } \
750                                 else { \
751                                         /* it must be 0x00300000 */ \
752                                         if (cnt & 0x00200000) { \
753                                                 TGTPORTSTAT.FctP2IORcnt[13]++; \
754                                         } \
755                                         else { \
756                                                 /* It must be 0x00100000 */ \
757                                                 TGTPORTSTAT.FctP2IORcnt[12]++; \
758                                         } \
759                                 } \
760                         } \
761                         else { \
762                                 /* It must be 0x000f0000 */ \
763                                 if (cnt & 0x000c0000) { \
764                                         if (cnt & 0x00080000) { \
765                                                 TGTPORTSTAT.FctP2IORcnt[11]++; \
766                                         } \
767                                         else { \
768                                                 /* It must be 0x00040000 */ \
769                                                 TGTPORTSTAT.FctP2IORcnt[10]++; \
770                                         } \
771                                 } \
772                                 else { \
773                                         /* It must be 0x00030000 */ \
774                                         if (cnt & 0x00020000) { \
775                                                 TGTPORTSTAT.FctP2IORcnt[9]++; \
776                                         } \
777                                         else { \
778                                                 /* It must be 0x00010000 */ \
779                                                 TGTPORTSTAT.FctP2IORcnt[8]++; \
780                                         } \
781                                 } \
782                         } \
783                 } \
784         } \
785         else { \
786                 if (cnt & 0x0000fe00) { \
787                         if (cnt & 0x0000f000) { \
```

```
788                                 if (cnt & 0x0000c000) { \
789                                         if (cnt & 0x00008000) { \
790                                                 TGTPORTSTAT.FctP2IORcnt[7]++; \
791                                         } \
792                                         else { \
793                                                 /* It must be 0x00004000 */ \
794                                                 TGTPORTSTAT.FctP2IORcnt[6]++; \
795                                         } \
796                                 } \
797                                 else { \
798                                         /* It must be 0x00000300 */ \
799                                         if (cnt & 0x00000200) { \
800                                                 TGTPORTSTAT.FctP2IORcnt[5]++; \
801                                         } \
802                                         else { \
803                                                 /* It must be 0x00000100 */ \
804                                                 TGTPORTSTAT.FctP2IORcnt[4]++; \
805                                         } \
806                                 } \
807                         } \
808                         else { \
809                                 /* It must be 0x00000e00 */ \
810                                 if (cnt & 0x00000800) { \
811                                         TGTPORTSTAT.FctP2IORcnt[3]++; \
812                                 } \
813                                 else { \
814                                         /* It must be 0x00000600 */ \
815                                         if (cnt & 0x00000400) { \
816                                                 TGTPORTSTAT.FctP2IORcnt[2]++; \
817                                         } \
818                                         else { \
819                                                 /* It must be 0x00000200 */ \
820                                                 TGTPORTSTAT.FctP2IORcnt[1]++; \
821                                         } \
822                                 } \
823                         } \
824                 } \
825                 else { \
826                         /* It must be 0x000001ff */ \
827                         TGTPORTSTAT.FctP2IORcnt[0]++; \
828                 } \
829         } \
830 }


833 #define EMLXS_BUMP_WRIOCTR(port, cnt) \
834 { \
835 /* Use binary search to find the first high bit */ \
836         if (cnt & 0xffff0000) { \
837                 if (cnt & 0xff800000) { \
838                         TGTPORTSTAT.FctP2IOWcnt[15]++; \
839                 } \
840                 else { \
841                         /* It must be 0x007f0000 */ \
842                         if (cnt & 0x00700000) { \
843                                 if (cnt & 0x00400000) { \
844                                         TGTPORTSTAT.FctP2IOWcnt[14]++; \
845                                 } \
846                                 else { \
847                                         /* It must be 0x00300000 */ \
848                                         if (cnt & 0x00200000) { \
849                                                 TGTPORTSTAT.FctP2IOWcnt[13]++; \
850                                         } \
851                                         else { \
852                                                 /* It must be 0x00100000 */ \
853                                                 TGTPORTSTAT.FctP2IOWcnt[12]++; \
```



```
854                                         } \
855                                 } \
856                         } \
857                         else { \
858                                 /* It must be 0x000f0000 */ \
859                                 if (cnt & 0x000c0000) { \
860                                         if (cnt & 0x00080000) { \
861                                                 TGTPORTSTAT.FctP2IOWcnt[11]++; \
862                                         } \
863                                         else { \
864                                                 /* it must be 0x00040000 */ \
865                                                 TGTPORTSTAT.FctP2IOWcnt[10]++; \
866                                         } \
867                                 } \
868                                 else { \
869                                         /* It must be 0x00030000 */ \
870                                         if (cnt & 0x00020000) { \
871                                                 TGTPORTSTAT.FctP2IOWcnt[9]++; \
872                                         } \
873                                         else { \
874                                                 /* It must be 0x00010000 */ \
875                                                 TGTPORTSTAT.FctP2IOWcnt[8]++; \
876                                         } \
877                                 } \
878                         } \
879                 } \
880         } \
881         else { \
882                 if (cnt & 0x0000fe00) { \
883                         if (cnt & 0x0000f000) { \
884                                 if (cnt & 0x0000c000) { \
885                                         if (cnt & 0x00008000) { \
886                                                 TGTPORTSTAT.FctP2IOWcnt[7]++; \
887                                         } \
888                                         else { \
889                                                 /* It must be 0x00004000 */ \
890                                                 TGTPORTSTAT.FctP2IOWcnt[6]++; \
891                                         } \
892                                 } \
893                                 else { \
894                                         /* It must be 0x00000300 */ \
895                                         if (cnt & 0x00000200) { \
896                                                 TGTPORTSTAT.FctP2IOWcnt[5]++; \
897                                         } \
898                                         else { \
899                                                 /* It must be 0x00000100 */ \
900                                                 TGTPORTSTAT.FctP2IOWcnt[4]++; \
901                                         } \
902                                 } \
903                         } \
904                         else { \
905                                 /* It must be 0x00000e00 */ \
906                                 if (cnt & 0x00000800) { \
907                                         TGTPORTSTAT.FctP2IOWcnt[3]++; \
908                                 } \
909                                 else { \
910                                         /* It must be 0x00000600 */ \
911                                         if (cnt & 0x00000400) { \
912                                                 TGTPORTSTAT.FctP2IOWcnt[2]++; \
913                                         } \
914                                         else { \
915                                                 /* It must be 0x00000200 */ \
916                                                 TGTPORTSTAT.FctP2IOWcnt[1]++; \
917                                         } \
918                                 } \
919                         } \
```

```
920                 } \
921                 else { \
922                         /* It must be 0x000001ff */ \
923                         TGTPORTSTAT.FctP2IOWcnt[0]++; \
924                 } \
925         } \
926 }

928 typedef struct emlxs_tgtport_stat
929 {
930         /* IO counters */
931         uint64_t        FctP2IOWcnt[MAX_TGTPORT_IOCNT]; /* Writes */
932         uint64_t        FctP2IORcnt[MAX_TGTPORT_IOCNT]; /* Reads  */
933         uint64_t        FctIOCmdCnt;                    /* Other, ie TUR */
934         uint64_t        FctCmdReceived;                 /* total IOs */
935         uint64_t        FctReadBytes;                   /* total read bytes */
936         uint64_t        FctWriteBytes;                  /* total write bytes */

938         /* IOCB handling counters */
939         uint64_t        FctEvent;       /* FctStray + FctCompleted */
940         uint64_t        FctCompleted;   /* FctCmplGood + FctCmplError */
941         uint64_t        FctCmplGood;

943         uint32_t        FctCmplError;
944         uint32_t        FctStray;

946         /* Fct event counters */
947         uint32_t        FctRcvDropped;
948         uint32_t        FctOverQDepth;
949         uint32_t        FctOutstandingIO;
950         uint32_t        FctFailedPortRegister;
951         uint32_t        FctPortRegister;
952         uint32_t        FctPortDeregister;

954         uint32_t        FctAbortSent;
955         uint32_t        FctNoBuffer;
956         uint32_t        FctScsiStatusErr;
957         uint32_t        FctScsiQfullErr;
958         uint32_t        FctScsiResidOver;
959         uint32_t        FctScsiResidUnder;
960         uint32_t        FctScsiSenseErr;

962         uint32_t        FctFiller1;
963 } emlxs_tgtport_stat_t;

965 #ifdef FCT_IO_TRACE
966 #define MAX_IO_TRACE    67
967 typedef struct emlxs_iotrace
968 {
969         fct_cmd_t       *fct_cmd;
970         uint32_t        xri;
971         uint8_t         marker;  /* 0xff */
972         uint8_t         trc[MAX_IO_TRACE]; /* trc[0] = index */
973 } emlxs_iotrace_t;
974 #endif /* FCT_IO_TRACE */
975 #endif /* SFCT_SUPPORT */


978 #include <emlxs_fcf.h>

980 /*
981  *     Port Information Data Structure
982  */

984 typedef struct emlxs_port
985 {
```



```
986         struct emlxs_hba        *hba;

988         /* Virtual port management */
989         struct VPIobj           VPIobj;
990         uint32_t                vpi;

992         uint32_t                flag;
993 #define EMLXS_PORT_ENABLE               0x00000001
994 #define EMLXS_PORT_BOUND                0x00000002

996 #define EMLXS_PORT_REG_VPI              0x00010000 /* SLI3 */
997 #define EMLXS_PORT_REG_VPI_CMPL         0x00020000 /* SLI3 */

999 #define EMLXS_PORT_IP_UP                0x00000010
1000 #define EMLXS_PORT_CONFIG               0x00000020
1001 #define EMLXS_PORT_RESTRICTED           0x00000040 /* Restrict logins */
1002 #define EMLXS_PORT_FLOGI_CMPL           0x00000080

1004 #define EMLXS_PORT_RESET_MASK           0x0000FFFF /* Flags to keep */
1005                                                    /* across hard reset */
1006 #define EMLXS_PORT_LINKDOWN_MASK        0xFFFFFF7F /* Flags to keep */
1007                                                    /* across link reset */

1009         uint32_t                options;
1010 #define EMLXS_OPT_RESTRICT              0x00000001 /* Force restricted */
1011                                                    /* logins */
1012 #define EMLXS_OPT_UNRESTRICT            0x00000002 /* Force Unrestricted */
1013                                                    /* logins */
1014 #define EMLXS_OPT_RESTRICT_MASK         0x00000003


1017         /* FC world wide names */
1018         NAME_TYPE               wwnn;
1019         NAME_TYPE               wwpn;
1020         char                    snn[256];
1021         char                    spn[256];

1023         /* Common service paramters */
1024         SERV_PARM               sparam;
1025         SERV_PARM               fabric_sparam;
1026         SERV_PARM               prev_fabric_sparam;

1028         /* fc_id management */
1029         uint32_t                did;
1030         uint32_t                prev_did;

1032         /* support FC_PORT_GET_P2P_INFO only */
1033         uint32_t                rdid;

1035         /* FC_AL management */
1036         uint8_t                 lip_type;
1037         uint8_t                 alpa_map[128];

1039         /* Node management */
1040         emlxs_node_t            node_base;
1041         uint32_t                node_count;
1042         krwlock_t               node_rwlock;
1043         emlxs_node_t            *node_table[EMLXS_NUM_HASH_QUES];

1045         /* Polled packet management */
1046         kcondvar_t              pkt_lock_cv;    /* pkt polling */
1047         kmutex_t                pkt_lock;       /* pkt polling */

1049         /* ULP */
1050         uint32_t                ulp_statec;
1051         void                    (*ulp_statec_cb) ();    /* Port state change */
```

```
1052                                                    /* callback routine */
1053         void                    (*ulp_unsol_cb) ();     /* unsolicited event */
1054                                                    /* callback routine */
1055         opaque_t                ulp_handle;

1057         /* ULP unsolicited buffers */
1058         kmutex_t                ub_lock;
1059         uint32_t                ub_count;
1060         emlxs_unsol_buf_t       *ub_pool;
1061         uint32_t                ub_post[MAX_CHANNEL];
1062         uint32_t                ub_timer;

1064         emlxs_ub_priv_t         *ub_wait_head;  /* Unsolicited IO received */
1065                                                 /* before link up */
1066         emlxs_ub_priv_t         *ub_wait_tail;  /* Unsolicited IO received */
1067                                                 /* before link up */


1070 #ifdef DHCHAP_SUPPORT
1071         emlxs_port_dhc_t        port_dhc;
1072 #endif  /* DHCHAP_SUPPORT */

1074         uint16_t                ini_mode;
1075         uint16_t                tgt_mode;

1077 #ifdef SFCT_SUPPORT

1079 #define FCT_BUF_COUNT_512               256
1080 #define FCT_BUF_COUNT_8K                128
1081 #define FCT_BUF_COUNT_64K               64
1082 #define FCT_BUF_COUNT_128K              64
1083 #define FCT_MAX_BUCKETS                 16
1084 #define FCT_DMEM_MAX_BUF_SIZE           131072   /* 128K */
1085 #define FCT_DMEM_MAX_BUF_SEGMENT        8388608  /* 8M */

1087         struct emlxs_fct_dmem_bucket dmem_bucket[FCT_MAX_BUCKETS];

1089         char                    cfd_name[24];
1090         stmf_port_provider_t    *port_provider;
1091         fct_local_port_t        *fct_port;
1092         uint32_t                fct_flags;

1094 #define FCT_STATE_PORT_ONLINE           0x00000001
1095 #define FCT_STATE_NOT_ACKED             0x00000002
1096 #define FCT_STATE_LINK_UP               0x00000010
1097 #define FCT_STATE_LINK_UP_ACKED         0x00000020

1099         emlxs_tgtport_stat_t    fct_stat;

1101         /* Used to save fct_cmd for deferred unsol ELS commands, except FLOGI */
1102         emlxs_buf_t             *fct_wait_head;
1103         emlxs_buf_t             *fct_wait_tail;

1105         /* Used to save context for deferred unsol FLOGIs */
1106         fct_flogi_xchg_t        fx;

1108 #ifdef FCT_IO_TRACE
1109         emlxs_iotrace_t         *iotrace;
1110         uint16_t                iotrace_cnt;
1111         uint16_t                iotrace_index;
1112         kmutex_t                iotrace_mtx;
1113 #endif /* FCT_IO_TRACE */

1115 #endif /* SFCT_SUPPORT */

1117 #ifdef SAN_DIAG_SUPPORT
```



```
1118         uint8_t                 sd_io_latency_state;
1119 #define SD_INVALID      0x00
1120 #define SD_COLLECTING   0x01
1121 #define SD_STOPPED      0x02

1123         /* SD event management list */
1124         uint32_t                sd_event_mask;   /* bit-mask */
1125         emlxs_dfc_event_t       sd_events[MAX_DFC_EVENTS];
1126 #endif

1128 } emlxs_port_t;


1131 /* Host Attn reg */
1132 #define FC_HA_REG(_hba)         ((volatile uint32_t *) \
1133                                     ((_hba)->sli.sli3.ha_reg_addr))

1135 /* Chip Attn reg */
1136 #define FC_CA_REG(_hba)         ((volatile uint32_t *) \
1137                                     ((_hba)->sli.sli3.ca_reg_addr))

1139 /* Host Status reg */
1140 #define FC_HS_REG(_hba)         ((volatile uint32_t *) \
1141                                     ((_hba)->sli.sli3.hs_reg_addr))

1143 /* Host Cntl reg */
1144 #define FC_HC_REG(_hba)         ((volatile uint32_t *) \
1145                                     ((_hba)->sli.sli3.hc_reg_addr))

1147 /* BIU Configuration reg */
1148 #define FC_BC_REG(_hba)         ((volatile uint32_t *) \
1149                                     ((_hba)->sli.sli3.bc_reg_addr))

1151 /* Used by SBUS adapter */
1152 /* TITAN Cntl reg */
1153 #define FC_SHC_REG(_hba)        ((volatile uint32_t *) \
1154                                     ((_hba)->sli.sli3.shc_reg_addr))

1156 /* TITAN Status reg */
1157 #define FC_SHS_REG(_hba)        ((volatile uint32_t *) \
1158                                     ((_hba)->sli.sli3.shs_reg_addr))

1160 /* TITAN Update reg */
1161 #define FC_SHU_REG(_hba)        ((volatile uint32_t *) \
1162                                     ((_hba)->sli.sli3.shu_reg_addr))

1164 /* MPU Semaphore reg */
1165 #define FC_SEMA_REG(_hba)       ((volatile uint32_t *)\
1166                                     ((_hba)->sli.sli4.MPUEPSemaphore_reg_addr))

1168 /* Bootstrap Mailbox Doorbell reg */
1169 #define FC_MBDB_REG(_hba)       ((volatile uint32_t *) \
1170                                     ((_hba)->sli.sli4.MBDB_reg_addr))

1172 /* MQ Doorbell reg */
1173 #define FC_MQDB_REG(_hba)       ((volatile uint32_t *) \
1174                                     ((_hba)->sli.sli4.MQDB_reg_addr))

1176 /* CQ Doorbell reg */
1177 #define FC_CQDB_REG(_hba)       ((volatile uint32_t *) \
1178                                     ((_hba)->sli.sli4.CQDB_reg_addr))

1180 /* WQ Doorbell reg */
1181 #define FC_WQDB_REG(_hba)       ((volatile uint32_t *) \
1182                                     ((_hba)->sli.sli4.WQDB_reg_addr))
```

```
/* RQ Doorbell reg */
#define FC_RQDB_REG(_hba)       ((volatile uint32_t *) \
                                    ((_hba)->sli.sli4.RQDB_reg_addr))


#define FC_SLIM2_MAILBOX(_hba)  ((MAILBOX *)(_hba)->sli.sli3.slim2.virt)

#define FC_SLIM1_MAILBOX(_hba)  ((MAILBOX *)(_hba)->sli.sli3.slim_addr)

#define FC_MAILBOX(_hba)        (((_hba)->flag & FC_SLIM2_MODE) ? \
        FC_SLIM2_MAILBOX(_hba) : FC_SLIM1_MAILBOX(_hba))

#define WRITE_CSR_REG(_hba, _regp, _value) ddi_put32(\
        (_hba)->sli.sli3.csr_acc_handle, (uint32_t *)(_regp), \
        (uint32_t)(_value))

#define READ_CSR_REG(_hba, _regp) ddi_get32(\
        (_hba)->sli.sli3.csr_acc_handle, (uint32_t *)(_regp))

#define WRITE_SLIM_ADDR(_hba, _regp, _value) ddi_put32(\
        (_hba)->sli.sli3.slim_acc_handle, (uint32_t *)(_regp), \
        (uint32_t)(_value))

#define READ_SLIM_ADDR(_hba, _regp) ddi_get32(\
        (_hba)->sli.sli3.slim_acc_handle, (uint32_t *)(_regp))

#define WRITE_SLIM_COPY(_hba, _bufp, _slimp, _wcnt) ddi_rep_put32(\
        (_hba)->sli.sli3.slim_acc_handle, (uint32_t *)(_bufp), \
        (uint32_t *)(_slimp), (_wcnt), DDI_DEV_AUTOINCR)

#define READ_SLIM_COPY(_hba, _bufp, _slimp, _wcnt) ddi_rep_get32(\
        (_hba)->sli.sli3.slim_acc_handle, (uint32_t *)(_bufp), \
        (uint32_t *)(_slimp), (_wcnt), DDI_DEV_AUTOINCR)

/* Used by SBUS adapter */
#define WRITE_SBUS_CSR_REG(_hba, _regp, _value) ddi_put32(\
        (_hba)->sli.sli3.sbus_csr_handle, (uint32_t *)(_regp), \
        (uint32_t)(_value))

#define READ_SBUS_CSR_REG(_hba, _regp) ddi_get32(\
        (_hba)->sli.sli3.sbus_csr_handle, (uint32_t *)(_regp))

#define SBUS_WRITE_FLASH_COPY(_hba, _offset, _value) ddi_put8(\
        (_hba)->sli.sli3.sbus_flash_acc_handle, \
        (uint8_t *)((volatile uint8_t *)(_hba)->sli.sli3.sbus_flash_addr + \
        (_offset)), (uint8_t)(_value))

#define SBUS_READ_FLASH_COPY(_hba, _offset) ddi_get8(\
        (_hba)->sli.sli3.sbus_flash_acc_handle, \
        (uint8_t *)((volatile uint8_t *)(_hba)->sli.sli3.sbus_flash_addr + \
        (_offset)))

/* SLI4 registers */
#define WRITE_BAR1_REG(_hba, _regp, _value) ddi_put32(\
        (_hba)->sli.sli4.bar1_acc_handle, (uint32_t *)(_regp), \
        (uint32_t)(_value))

#define READ_BAR1_REG(_hba, _regp) ddi_get32(\
        (_hba)->sli.sli4.bar1_acc_handle, (uint32_t *)(_regp))

#define WRITE_BAR2_REG(_hba, _regp, _value) ddi_put32(\
        (_hba)->sli.sli4.bar2_acc_handle, (uint32_t *)(_regp), \
        (uint32_t)(_value))

#define READ_BAR2_REG(_hba, _regp) ddi_get32(\
        (_hba)->sli.sli4.bar2_acc_handle, (uint32_t *)(_regp))
```



```


#define EMLXS_STATE_CHANGE(_hba, _state)\
{                                                                       \
        mutex_enter(&EMLXS_PORT_LOCK);                                  \
        EMLXS_STATE_CHANGE_LOCKED((_hba), (_state));                    \
        mutex_exit(&EMLXS_PORT_LOCK);                                   \
}

/* Used when EMLXS_PORT_LOCK is already held */
#define EMLXS_STATE_CHANGE_LOCKED(_hba, _state)                 \
{                                                                       \
        if ((_hba)->state != (_state))                                  \
        {                                                               \
                uint32_t _st = _state;                                  \
                EMLXS_MSGF(EMLXS_CONTEXT,                               \
                        &emlxs_state_msg, "%s --> %s",                  \
                        emlxs_ffstate_xlate((_hba)->state),             \
                        emlxs_ffstate_xlate(_state));                   \
                        (_hba)->state = (_state);                       \
                if ((_st) == FC_ERROR)                                  \
                {                                                       \
                        (_hba)->flag |= FC_HARDWARE_ERROR;              \
                }                                                       \
        }                                                               \
}

#ifdef FMA_SUPPORT
#define EMLXS_CHK_ACC_HANDLE(_hba, _acc) \
        if (emlxs_fm_check_acc_handle(_hba, _acc) != DDI_FM_OK) { \
                EMLXS_MSGF(EMLXS_CONTEXT, \
                    &emlxs_invalid_access_handle_msg, NULL); \
        }
#endif  /* FMA_SUPPORT */

/*
 * This is the HBA control area for the adapter
 */

#ifdef MODSYM_SUPPORT

typedef struct emlxs_modsym
{
        ddi_modhandle_t  mod_fctl;      /* For Leadville */

        /* Leadville (fctl) */
        int             (*fc_fca_attach)(dev_info_t *, fc_fca_tran_t *);
        int             (*fc_fca_detach)(dev_info_t *);
        int             (*fc_fca_init)(struct dev_ops *);

#ifdef SFCT_SUPPORT
        uint32_t        fct_modopen;
        uint32_t        reserved;  /* Padding for alignment */

        ddi_modhandle_t  mod_fct;       /* For Comstar */
        ddi_modhandle_t  mod_stmf;      /* For Comstar */

        /* Comstar (fct) */
        void*   (*fct_alloc)(fct_struct_id_t, int, int);
        void    (*fct_free)(void *);
        void*   (*fct_scsi_task_alloc)(void *, uint16_t, uint32_t, uint8_t *,
                        uint16_t, uint16_t);
        int     (*fct_register_local_port)(fct_local_port_t *);
        void    (*fct_deregister_local_port)(fct_local_port_t *);
        void    (*fct_handle_event)(fct_local_port_t *, int, uint32_t, caddr_t);
        void    (*fct_post_rcvd_cmd)(fct_cmd_t *, stmf_data_buf_t *);
```

```
1316         void    (*fct_ctl)(void *, int, void *);
1317         void    (*fct_queue_cmd_for_termination)(fct_cmd_t *, fct_status_t);
1318         void    (*fct_send_response_done)(fct_cmd_t *, fct_status_t, uint32_t);
1319         void    (*fct_send_cmd_done)(fct_cmd_t *, fct_status_t, uint32_t);
1320         void    (*fct_scsi_data_xfer_done)(fct_cmd_t *, stmf_data_buf_t *,
1321                         uint32_t);
1322         fct_status_t    (*fct_port_shutdown)
1323                                 (fct_local_port_t *, uint32_t, char *);
1324         fct_status_t    (*fct_port_initialize)
1325                                 (fct_local_port_t *, uint32_t, char *);
1326         void            (*fct_cmd_fca_aborted)
1327                                 (fct_cmd_t *, fct_status_t, int);
1328         fct_status_t    (*fct_handle_rcvd_flogi)
1329                                 (fct_local_port_t *, fct_flogi_xchg_t *);

1331         /* Comstar (stmf) */
1332         void*  (*stmf_alloc)(stmf_struct_id_t, int, int);
1333         void   (*stmf_free)(void *);
1334         void    (*stmf_deregister_port_provider) (stmf_port_provider_t *);
1335         int     (*stmf_register_port_provider) (stmf_port_provider_t *);
1336 #endif /* SFCT_SUPPORT */
1337 } emlxs_modsym_t;
1338 extern emlxs_modsym_t emlxs_modsym;

1340 #define MODSYM(_f)      emlxs_modsym._f

1342 #else

1344 #define MODSYM(_f)      _f

1346 #endif /* MODSYM_SUPPORT */



1350 typedef struct RPIHdrTmplate
1351 {
1352         uint32_t        Word[16];  /* 64 bytes */
1353 } RPIHdrTmplate_t;


1356 typedef struct EQ_DESC
1357 {
1358         uint16_t        host_index;
1359         uint16_t        max_index;
1360         uint16_t        qid;
1361         uint16_t        msix_vector;
1362         kmutex_t        lastwq_lock;
1363         uint16_t        lastwq;
1364         MBUF_INFO       addr;
1365 } EQ_DESC_t;


1368 typedef struct CQ_DESC
1369 {
1370         uint16_t        host_index;
1371         uint16_t        max_index;
1372         uint16_t        qid;
1373         uint16_t        eqid;
1374         uint16_t        type;
1375 #define EMLXS_CQ_TYPE_GROUP1    1  /* associated with a MQ and async events */
1376 #define EMLXS_CQ_TYPE_GROUP2    2  /* associated with a WQ and RQ */
1377         uint16_t        rsvd;

1379         MBUF_INFO       addr;
1380         CHANNEL         *channelp; /* ptr to CHANNEL associated with CQ */
```



```
1382 } CQ_DESC_t;


1385 typedef struct WQ_DESC
1386 {
1387         uint16_t        host_index;
1388         uint16_t        max_index;
1389         uint16_t        port_index;
1390         uint16_t        release_depth;
1391 #define WQE_RELEASE_DEPTH       (8 * EMLXS_NUM_WQ_PAGES)
1392         uint16_t        qid;
1393         uint16_t        cqid;
1394         MBUF_INFO       addr;
1395 } WQ_DESC_t;


1398 typedef struct RQ_DESC
1399 {
1400         uint16_t        host_index;
1401         uint16_t        max_index;
1402         uint16_t        qid;
1403         uint16_t        cqid;

1405         MBUF_INFO       addr;
1406         MBUF_INFO       rqb[RQ_DEPTH];

1408         kmutex_t        lock;

1410 } RQ_DESC_t;


1413 typedef struct RXQ_DESC
1414 {
1415         kmutex_t        lock;
1416         emlxs_queue_t   active;

1418 } RXQ_DESC_t;


1421 typedef struct MQ_DESC
1422 {
1423         uint16_t        host_index;
1424         uint16_t        max_index;
1425         uint16_t        qid;
1426         uint16_t        cqid;
1427         MBUF_INFO       addr;
1428 } MQ_DESC_t;


1431 /* Define the number of queues the driver will be using */
1432 #define EMLXS_MAX_EQS   EMLXS_MSI_MAX_INTRS
1433 #define EMLXS_MAX_WQS   EMLXS_MSI_MAX_INTRS
1434 #define EMLXS_MAX_RQS   2       /* ONLY 1 pair is allowed */
1435 #define EMLXS_MAX_MQS   1

1437 /* One CQ for each WQ & (RQ pair) plus one for the MQ */
1438 #define EMLXS_MAX_CQS   (EMLXS_MAX_WQS + (EMLXS_MAX_RQS/2) + 1)

1440 /* The First CQ created is ALWAYS for mbox / event handling */
1441 #define EMLXS_CQ_MBOX           0

1443 /* The Second CQ created is ALWAYS for unsol rcv handling */
1444 /* At this time we are allowing ONLY 1 pair of RQs */
1445 #define EMLXS_CQ_RCV            1

1447 /* The remaining CQs are for WQ completions */
```

```
1448 #define EMLXS_CQ_OFFSET_WQ      2


1451 /* FCFI RQ Configuration */
1452 #define EMLXS_FCFI_RQ0_INDEX    0
1453 #define EMLXS_FCFI_RQ0_RMASK    0 /* match all */
1454 #define EMLXS_FCFI_RQ0_RCTL     0 /* match all */
1455 #define EMLXS_FCFI_RQ0_TMASK    0 /* match all */
1456 #define EMLXS_FCFI_RQ0_TYPE     0 /* match all */

1458 /* Define the maximum value for a Queue Id */
1459 #define EMLXS_MAX_EQ_IDS        256
1460 #define EMLXS_MAX_CQ_IDS        1024
1461 #define EMLXS_MAX_WQ_IDS        1024
1462 #define EMLXS_MAX_RQ_IDS        4

1464 #define EMLXS_RXQ_ELS           0
1465 #define EMLXS_RXQ_CT            1
1466 #define EMLXS_MAX_RXQS          2

1468 #define PCI_CONFIG_SIZE   0x80

1470 typedef struct emlxs_sli3
1471 {
1472         /* SLIM management */
1473         MATCHMAP        slim2;

1475         /* HBQ management */
1476         uint32_t        hbq_count;      /* Total number of HBQs */
1477                                         /* configured */
1478         HBQ_INIT_t      hbq_table[EMLXS_NUM_HBQ];

1480         /* Adapter memory management */
1481         caddr_t         csr_addr;
1482         caddr_t         slim_addr;
1483         ddi_acc_handle_t csr_acc_handle;
1484         ddi_acc_handle_t slim_acc_handle;

1486         /* SBUS adapter management */
1487         caddr_t         sbus_flash_addr;        /* Virt addr of R/W */
1488                                                 /* Flash */
1489         caddr_t         sbus_core_addr;         /* Virt addr of TITAN */
1490                                                 /* CORE */
1491         caddr_t         sbus_csr_addr;          /* Virt addr of TITAN */
1492                                                 /* CSR */
1493         ddi_acc_handle_t sbus_flash_acc_handle;
1494         ddi_acc_handle_t sbus_core_acc_handle;
1495         ddi_acc_handle_t sbus_csr_handle;

1497         /* SLI 2/3 Adapter register management */
1498         uint32_t        *bc_reg_addr;   /* virtual offset for BIU */
1499                                         /* config reg */
1500         uint32_t        *ha_reg_addr;   /* virtual offset for host */
1501                                         /* attn reg */
1502         uint32_t        *hc_reg_addr;   /* virtual offset for host */
1503                                         /* ctl reg */
1504         uint32_t        *ca_reg_addr;   /* virtual offset for FF */
1505                                         /* attn reg */
1506         uint32_t        *hs_reg_addr;   /* virtual offset for */
1507                                         /* status reg */
1508         uint32_t        *shc_reg_addr;  /* virtual offset for SBUS */
1509                                         /* Ctrl reg */
1510         uint32_t        *shs_reg_addr;  /* virtual offset for SBUS */
1511                                         /* Status reg */
1512         uint32_t        *shu_reg_addr;  /* virtual offset for SBUS */
1513                                         /* Update reg */
```



```
1514         uint16_t        hgp_ring_offset;
1515         uint16_t        hgp_hbq_offset;
1516         uint16_t        iocb_cmd_size;
1517         uint16_t        iocb_rsp_size;
1518         uint32_t        hc_copy;        /* local copy of HC register */

1520         /* Ring management */
1521         uint32_t        ring_count;
1522         emlxs_ring_t    ring[MAX_RINGS];
1523         kmutex_t        ring_cmd_lock[MAX_RINGS];
1524         uint8_t         ring_masks[4];  /* number of masks/rings used */
1525         uint8_t         ring_rval[6];
1526         uint8_t         ring_rmask[6];
1527         uint8_t         ring_tval[6];
1528         uint8_t         ring_tmask[6];

1530         /* Protected by EMLXS_FCTAB_LOCK */
1531 #ifdef EMLXS_SPARC
1532         MEMSEG          fcp_bpl_seg;
1533         MATCHMAP        **fcp_bpl_table; /* iotag table for */
1534                                         /* bpl buffers */
1535 #endif  /* EMLXS_SPARC */
1536         uint32_t        mem_bpl_size;
1537 } emlxs_sli3_t;

1539 typedef struct emlxs_sli4
1540 {
1541         MATCHMAP        bootstrapmb;
1542         caddr_t         bar1_addr;
1543         caddr_t         bar2_addr;
1544         ddi_acc_handle_t bar1_acc_handle;
1545         ddi_acc_handle_t bar2_acc_handle;

1547         /* SLI4 Adapter register management */
1548         uint32_t        *MPUEPSemaphore_reg_addr;
1549         uint32_t        *MBDB_reg_addr;

1551         uint32_t        *CQDB_reg_addr;
1552         uint32_t        *MQDB_reg_addr;
1553         uint32_t        *WQDB_reg_addr;
1554         uint32_t        *RQDB_reg_addr;

1556         uint32_t        flag;
1557 #define EMLXS_SLI4_INTR_ENABLED         0x00000001
1558 #define EMLXS_SLI4_HW_ERROR             0x00000002
1559 #define EMLXS_SLI4_DOWN_LINK            0x00000004

1561         uint16_t        XRICount;
1562         uint16_t        XRIBase;
1563         uint16_t        RPICount;
1564         uint16_t        RPIBase;
1565         uint16_t        VPICount;
1566         uint16_t        VPIBase;
1567         uint16_t        VFICount;
1568         uint16_t        VFIBase;
1569         uint16_t        FCFICount;

1571         kmutex_t        fcf_lock;
1572         FCFTable_t      fcftab;
1573         VFIobj_t        *VFI_table;

1575         /* Save Config Region 23 info */
1576         tlv_fcoe_t      cfgFCOE;
1577         tlv_fcfconnectlist_t    cfgFCF;

1579         MBUF_INFO       slim2;
```

```
1580         MBUF_INFO       dump_region;
1581 #define EMLXS_DUMP_REGION_SIZE  1024

1583         RPIobj_t        *RPIp;
1584         MBUF_INFO       HeaderTmplate;
1585         XRIobj_t        *XRIp;

1587         /* Double linked list for available XRIs */
1588         XRIobj_t        *XRIfree_f;
1589         XRIobj_t        *XRIfree_b;
1590         uint32_t        xrif_count;
1591         uint32_t        mem_sgl_size;

1593         /* Double linked list for XRIs in use */
1594         XRIobj_t        *XRIinuse_f;
1595         XRIobj_t        *XRIinuse_b;
1596         uint32_t        xria_count;

1598         kmutex_t        que_lock[EMLXS_MAX_WQS];
1599         EQ_DESC_t       eq[EMLXS_MAX_EQS];
1600         CQ_DESC_t       cq[EMLXS_MAX_CQS];
1601         WQ_DESC_t       wq[EMLXS_MAX_WQS];
1602         RQ_DESC_t       rq[EMLXS_MAX_RQS];
1603         MQ_DESC_t       mq;

1605         /* Used to map a queue ID to a queue DESC_t */
1606         uint16_t        eq_map[EMLXS_MAX_EQ_IDS];
1607         uint16_t        cq_map[EMLXS_MAX_CQ_IDS];
1608         uint16_t        wq_map[EMLXS_MAX_WQ_IDS];
1609         uint16_t        rq_map[EMLXS_MAX_RQ_IDS];

1611         RXQ_DESC_t      rxq[EMLXS_MAX_RXQS];

1613         uint32_t        ue_mask_lo;
1614         uint32_t        ue_mask_hi;

1616 } emlxs_sli4_t;


1619 typedef struct emlxs_sli_api
1620 {
1621         int             (*sli_map_hdw)();
1622         void            (*sli_unmap_hdw)();
1623         int32_t         (*sli_online)();
1624         void            (*sli_offline)();
1625         uint32_t        (*sli_hba_reset)();
1626         void            (*sli_hba_kill)();
1627         void            (*sli_issue_iocb_cmd)();
1628         uint32_t        (*sli_issue_mbox_cmd)();
1629         uint32_t        (*sli_prep_fct_iocb)();
1630         uint32_t        (*sli_prep_fcp_iocb)();
1631         uint32_t        (*sli_prep_ip_iocb)();
1632         uint32_t        (*sli_prep_els_iocb)();
1633         uint32_t        (*sli_prep_ct_iocb)();
1634         void            (*sli_poll_intr)();
1635         int32_t         (*sli_intx_intr)();
1636         uint32_t        (*sli_msi_intr)();
1637         void            (*sli_disable_intr)();
1638         void            (*sli_timer)();
1639         void            (*sli_poll_erratt)();

1641 } emlxs_sli_api_t;


1644 typedef struct emlxs_hba
1645 {
```



```
1646         dev_info_t      *dip;
1647         int32_t         emlxinst;
1648         int32_t         ddiinst;
1649         uint8_t         pci_function_number;
1650         uint8_t         pci_device_number;
1651         uint8_t         pci_bus_number;
1652         uint8_t         pci_cap_offset[PCI_CAP_MAX_PTR];

1654 #ifdef FMA_SUPPORT
1655         int32_t         fm_caps;        /* FMA capabilities */
1656 #endif  /* FMA_SUPPORT */
1657         fc_fca_tran_t   *fca_tran;

1659         /* DMA attributes */
1660         ddi_dma_attr_t  dma_attr;
1661         ddi_dma_attr_t  dma_attr_ro;
1662         ddi_dma_attr_t  dma_attr_1sg;
1663         ddi_dma_attr_t  dma_attr_fcip_rsp;

1665         /* HBA Info */
1666         emlxs_model_t   model_info;
1667         emlxs_vpd_t     vpd;    /* vital product data */
1668         NAME_TYPE       wwnn;
1669         NAME_TYPE       wwpn;
1670         char            snn[256];
1671         char            spn[256];
1672         PROG_ID         load_list[MAX_LOAD_ENTRY];
1673         WAKE_UP_PARMS   wakeup_parms;
1674         uint32_t        max_nodes;
1675         uint32_t        io_throttle;
1676         uint32_t        io_active;
1677         uint32_t        bus_type;
1678 #define PCI_FC          0
1679 #define SBUS_FC         1

1681         /* Link management */
1682         uint32_t        link_event_tag;
1683         uint8_t         topology;
1684         uint8_t         linkspeed;
1685         uint16_t        qos_linkspeed;
1686         uint32_t        linkup_wait_flag;
1687         kcondvar_t      linkup_lock_cv;
1688         kmutex_t        linkup_lock;

1690         /* Memory Pool management */
1691         emlxs_memseg_t  memseg[FC_MAX_SEG];     /* memory for buffer */
1692                                                         /* structures */
1693         kmutex_t        memget_lock;    /* locks all memory pools get */
1694         kmutex_t        memput_lock;    /* locks all memory pools put */

1696         /* Fibre Channel Service Parameters */
1697         SERV_PARM       sparam;
1698         uint32_t        fc_edtov;       /* E_D_TOV timer value */
1699         uint32_t        fc_arbtov;      /* ARB_TOV timer value */
1700         uint32_t        fc_ratov;       /* R_A_TOV timer value */
1701         uint32_t        fc_rttov;       /* R_T_TOV timer value */
1702         uint32_t        fc_altov;       /* AL_TOV timer value */
1703         uint32_t        fc_crtov;       /* C_R_TOV timer value */
1704         uint32_t        fc_citov;       /* C_I_TOV timer value */

1706         /* Adapter State management */
1707         int32_t         state;
1708 #define FC_ERROR                0x01    /* Adapter shutdown */
1709 #define FC_KILLED               0x02    /* Adapter interlocked/killed */
1710 #define FC_WARM_START           0x03    /* Adapter reset, but not restarted */
1711 #define FC_INIT_START           0x10    /* Adapter restarted */
```

```
#define FC_INIT_NVPARAMS        0x11
#define FC_INIT_REV             0x12
#define FC_INIT_CFGPORT         0x13
#define FC_INIT_CFGRING         0x14
#define FC_INIT_INITLINK        0x15
#define FC_LINK_DOWN            0x20
#define FC_LINK_DOWN_PERSIST    0x21
#define FC_LINK_UP              0x30
#define FC_CLEAR_LA             0x31
#define FC_READY                0x40

        uint32_t        flag;
#define FC_ONLINING_MODE        0x00000001
#define FC_ONLINE_MODE          0x00000002
#define FC_OFFLINING_MODE       0x00000004
#define FC_OFFLINE_MODE         0x00000008

#define FC_NPIV_ENABLED         0x00000010      /* NPIV enabled on adapter    */
#define FC_NPIV_SUPPORTED       0x00000020      /* NPIV supported on fabric   */
#define FC_NPIV_UNSUPPORTED     0x00000040      /* NPIV unsupported on fabric */
#define FC_NPIV_LINKUP          0x00000100      /* NPIV enabled, supported, */
                                                /* and link is ready */
#define FC_NPIV_DELAY_REQUIRED  0x00000200      /* Delay issuing FLOGI/FDISC */
                                                /* and NameServer cmds */

#define FC_BOOTSTRAPMB_INIT     0x00000400
#define FC_FIP_SUPPORTED        0x00000800      /* FIP supported */

#define FC_FABRIC_ATTACHED      0x00001000
#define FC_PT_TO_PT             0x00002000
#define FC_BYPASSED_MODE        0x00004000
#define FC_MENLO_MODE           0x00008000      /* Menlo maintenance mode */

#define FC_DUMP_SAFE            0x00010000      /* Safe to DUMP */
#define FC_DUMP_ACTIVE          0x00020000      /* DUMP in progress */
#define FC_NEW_FABRIC           0x00040000

#define FC_SLIM2_MODE           0x00100000      /* SLIM in host memory */
#define FC_INTERLOCKED          0x00200000
#define FC_HBQ_ENABLED          0x00400000
#define FC_ASYNC_EVENTS         0x00800000

#define FC_ILB_MODE             0x01000000
#define FC_ELB_MODE             0x02000000
#define FC_LOOPBACK_MODE        0x03000000      /* Loopback Mode Mask */
#define FC_DUMP                 0x04000000      /* DUMP in progress */
#define FC_SHUTDOWN             0x08000000      /* SHUTDOWN in progress */

#define FC_OVERTEMP_EVENT       0x10000000      /* FC_ERROR reason: */
                                                /* over temperature event */
#define FC_MBOX_TIMEOUT         0x20000000      /* FC_ERROR reason: */
                                                /* mailbox timeout event */
#define FC_DMA_CHECK_ERROR      0x40000000      /* Shared memory (slim,..) */
                                                /* DMA handle went bad */
#define FC_HARDWARE_ERROR       0x80000000      /* FC_ERROR state triggered */

#define FC_RESET_MASK           0x00030C1F      /* Bits to protect during */
                                                /* a hard reset */
#define FC_LINKDOWN_MASK        0xFFF30C1F      /* Bits to protect during */
                                                /* a linkdown */

        uint32_t fw_timer;
        uint32_t fw_flag;
#define FW_UPDATE_NEEDED        0x00000001
#define FW_UPDATE_KERNEL        0x00000002
```



```
        uint32_t temperature;                   /* Last reported temperature */

        /* SBUS adapter management */
        caddr_t         sbus_pci_addr;          /* Virt addr of TITAN */
                                                /* pci config */
        ddi_acc_handle_t sbus_pci_handle;

        /* PCI BUS adapter management */
        caddr_t         pci_addr;
        ddi_acc_handle_t pci_acc_handle;

        uint32_t        sli_mode;
#define EMLXS_HBA_SLI1_MODE     1
#define EMLXS_HBA_SLI2_MODE     2
#define EMLXS_HBA_SLI3_MODE     3
#define EMLXS_HBA_SLI4_MODE     4

        /* SLI private data */
        union {
                emlxs_sli3_t sli3;
                emlxs_sli4_t sli4;
        } sli;

        /* SLI API entry point routines */
        emlxs_sli_api_t sli_api;

        uint32_t        io_poll_count;  /* Number of poll commands */
                                        /* in progress */

        /* IO Completion management */
        uint32_t        iodone_count;   /* Number of IO's on done Q */
        /* Protected by EMLXS_PORT_LOCK  */
        emlxs_buf_t     *iodone_list;   /* fc_packet being deferred */
        emlxs_buf_t     *iodone_tail;   /* fc_packet being deferred */
        emlxs_thread_t  iodone_thread;
        emlxs_thread_t  *spawn_thread_head;
        emlxs_thread_t  *spawn_thread_tail;
        kmutex_t        spawn_lock;
        uint32_t        spawn_open;

        /* IO Channel management */
        int32_t         chan_count;
        emlxs_channel_t chan[MAX_CHANNEL];
        kmutex_t        channel_tx_lock;
        uint8_t         channel_fcp;    /* Default channel to use for FCP IO */
#define CHANNEL_FCT channel_fcp
        uint8_t         channel_ip;     /* Default channel to use for IP IO */
        uint8_t         channel_els;    /* Default channel to use for ELS IO */
        uint8_t         channel_ct;     /* Default channel to use for CT IO */

        /* IOTag management */
        emlxs_buf_t     **fc_table;     /* sc_buf pointers indexed by */
                                        /* iotag */
        uint16_t        fc_iotag;       /* used to identify I/Os */
        uint16_t        fc_oor_iotag;   /* OutOfRange (fc_table) iotags */
                                        /* typically used for Abort/close */
#define EMLXS_MAX_ABORT_TAG     0x7fff
        uint16_t        max_iotag;      /* ALL IOCBs except aborts */
        kmutex_t        iotag_lock;
        uint32_t        io_count;               /* No of IO holding */
                                                /* regular iotag */
        uint32_t        channel_tx_count;       /* No of IO on tx Q */

        /* Mailbox Management */
        uint32_t        mbox_queue_flag;
        emlxs_queue_t   mbox_queue;
```

```
1844         void            *mbox_mqe;      /* active mbox mqe */
1845         void            *mbox_mbq;      /* active MAILBOXQ */
1846         kcondvar_t      mbox_lock_cv;   /* MBX_SLEEP */
1847         kmutex_t        mbox_lock;      /* MBX_SLEEP */
1848         uint32_t        mbox_timer;

1850         /* Interrupt management */
1851         void            *intr_arg;
1852         uint32_t        intr_unclaimed;
1853         uint32_t        intr_autoClear;
1854         uint32_t        intr_flags;
1855 #define EMLXS_INTX_INITED       0x0001
1856 #define EMLXS_INTX_ADDED        0x0002
1857 #define EMLXS_MSI_ENABLED       0x0010
1858 #define EMLXS_MSI_INITED        0x0020
1859 #define EMLXS_MSI_ADDED         0x0040
1860 #define EMLXS_INTR_INITED       (EMLXS_INTX_INITED|EMLXS_MSI_INITED)
1861 #define EMLXS_INTR_ADDED        (EMLXS_INTX_ADDED|EMLXS_MSI_ADDED)

1863 #ifdef MSI_SUPPORT
1864         ddi_intr_handle_t *intr_htable;
1865         uint32_t        *intr_pri;
1866         int32_t         *intr_cap;
1867         uint32_t        intr_count;
1868         uint32_t        intr_type;
1869         uint32_t        intr_cond;
1870         uint32_t        intr_map[EMLXS_MSI_MAX_INTRS];
1871         uint32_t        intr_mask;

1873         kmutex_t        msiid_lock; /* for last_msiid */
1874         int             last_msiid;

1876         kmutex_t        intr_lock[EMLXS_MSI_MAX_INTRS];
1877         int                     chan2msi[MAX_CHANNEL];
1878                                 /* Index is the channel id */
1879         int                     msi2chan[EMLXS_MSI_MAX_INTRS];
1880                                 /* Index is the MSX-X msg id */
1881 #endif  /* MSI_SUPPORT */

1883         uint32_t        heartbeat_timer;
1884         uint32_t        heartbeat_flag;
1885         uint32_t        heartbeat_active;

1887         /* IOCTL management */
1888         kmutex_t        ioctl_lock;
1889         uint32_t        ioctl_flags;
1890 #define EMLXS_OPEN              0x00000001
1891 #define EMLXS_OPEN_EXCLUSIVE    0x00000002

1893         /* Timer management */
1894         kcondvar_t      timer_lock_cv;
1895         kmutex_t        timer_lock;
1896         timeout_id_t    timer_id;
1897         uint32_t        timer_tics;
1898         uint32_t        timer_flags;
1899 #define EMLXS_TIMER_STARTED     0x0000001
1900 #define EMLXS_TIMER_BUSY        0x0000002
1901 #define EMLXS_TIMER_KILL        0x0000004
1902 #define EMLXS_TIMER_ENDED       0x0000008

1904         /* Misc Timers */
1905         uint32_t        linkup_timer;
1906         uint32_t        discovery_timer;
1907         uint32_t        pkt_timer;

1909         /* Power Management */
```



```
1910         uint32_t        pm_state;
1911         /* pm_state */
1912 #define EMLXS_PM_IN_ATTACH      0x00000001
1913 #define EMLXS_PM_IN_DETACH      0x00000002
1914 #define EMLXS_PM_IN_SOL_CB      0x00000010
1915 #define EMLXS_PM_IN_UNSOL_CB    0x00000020
1916 #define EMLXS_PM_IN_LINK_RESET  0x00000100
1917 #define EMLXS_PM_IN_HARD_RESET  0x00000200
1918 #define EMLXS_PM_SUSPENDED      0x01000000

1920         uint32_t        pm_level;
1921         /* pm_level */
1922 #define EMLXS_PM_ADAPTER_DOWN   0
1923 #define EMLXS_PM_ADAPTER_UP     1

1925         uint32_t        pm_busy;
1926         kmutex_t        pm_lock;
1927         uint8_t         pm_config[PCI_CONFIG_SIZE];
1928 #ifdef IDLE_TIMER
1929         uint32_t        pm_idle_timer;
1930         uint32_t        pm_active;      /* Only used by timer */
1931 #endif  /* IDLE_TIMER */

1933         /* Loopback management */
1934         uint32_t        loopback_tics;
1935         void            *loopback_pkt;

1937         /* Event management */
1938         emlxs_event_queue_t event_queue;
1939         uint32_t        event_mask;
1940         uint32_t        event_timer;
1941         emlxs_dfc_event_t dfc_event[MAX_DFC_EVENTS];
1942         emlxs_hba_event_t hba_event;

1944         /* Parameter management */
1945         emlxs_config_t  config[NUM_CFG_PARAM];

1947         /* Driver stat management */
1948         kstat_t         *kstat;
1949         emlxs_stats_t   stats;

1951         /* Log management */
1952         emlxs_msg_log_t log;

1954         /* Port managment */
1955         uint32_t        vpi_base;
1956         uint32_t        vpi_max;
1957         uint32_t        vpi_high;
1958         uint32_t        num_of_ports;

1960         kmutex_t        port_lock;      /* locks port, nodes, rings */
1961         emlxs_port_t    port[MAX_VPORTS + 1];   /* port specific info */
1962                                                 /* Last one is for */
1963                                                 /* NPIV ready test */

1965 #ifdef DHCHAP_SUPPORT
1966         kmutex_t        dhc_lock;
1967         kmutex_t        auth_lock;
1968         emlxs_auth_cfg_t        auth_cfg;       /* Default auth_cfg. */
1969                                                 /* Points to list of entries. */
1970                                                 /* Protected by auth_lock */
1971         uint32_t        auth_cfg_count;
1972         emlxs_auth_key_t        auth_key;       /* Default auth_key. */
1973                                                 /* Points to list of entries. */
1974                                                 /* Protected by auth_lock */
1975         uint32_t        auth_key_count;
```

```
1976         uint32_t        rdn_flag;
1977 #endif  /* DHCHAP_SUPPORT */

1979         uint16_t        ini_mode;
1980         uint16_t        tgt_mode;

1982 #ifdef TEST_SUPPORT
1983         uint32_t        underrun_counter;
1984 #endif /* TEST_SUPPORT */

1986 #ifdef MODFW_SUPPORT
1987         ddi_modhandle_t fw_modhandle;
1988 #endif /* MODFW_SUPPORT */

1990 #ifdef DUMP_SUPPORT
1991         emlxs_file_t    dump_txtfile;
1992         emlxs_file_t    dump_dmpfile;
1993         emlxs_file_t    dump_ceefile;
1994         kmutex_t        dump_lock;
1995 #define EMLXS_DUMP_LOCK         hba->dump_lock
1996 #define EMLXS_TXT_FILE          1
1997 #define EMLXS_DMP_FILE          2
1998 #define EMLXS_CEE_FILE          3

2000 #define EMLXS_DRV_DUMP          0
2001 #define EMLXS_TEMP_DUMP         1
2002 #define EMLXS_USER_DUMP         2

2004 #endif /* DUMP_SUPPORT */

2006 } emlxs_hba_t;

2008 #define EMLXS_SLI_MAP_HDW               (hba->sli_api.sli_map_hdw)
2009 #define EMLXS_SLI_UNMAP_HDW             (hba->sli_api.sli_unmap_hdw)
2010 #define EMLXS_SLI_ONLINE                (hba->sli_api.sli_online)
2011 #define EMLXS_SLI_OFFLINE               (hba->sli_api.sli_offline)
2012 #define EMLXS_SLI_HBA_RESET             (hba->sli_api.sli_hba_reset)
2013 #define EMLXS_SLI_HBA_KILL              (hba->sli_api.sli_hba_kill)
2014 #define EMLXS_SLI_ISSUE_IOCB_CMD        (hba->sli_api.sli_issue_iocb_cmd)
2015 #define EMLXS_SLI_ISSUE_MBOX_CMD        (hba->sli_api.sli_issue_mbox_cmd)
2016 #define EMLXS_SLI_PREP_FCT_IOCB         (hba->sli_api.sli_prep_fct_iocb)
2017 #define EMLXS_SLI_PREP_FCP_IOCB         (hba->sli_api.sli_prep_fcp_iocb)
2018 #define EMLXS_SLI_PREP_IP_IOCB          (hba->sli_api.sli_prep_ip_iocb)
2019 #define EMLXS_SLI_PREP_ELS_IOCB         (hba->sli_api.sli_prep_els_iocb)
2020 #define EMLXS_SLI_PREP_CT_IOCB          (hba->sli_api.sli_prep_ct_iocb)
2021 #define EMLXS_SLI_POLL_INTR             (hba->sli_api.sli_poll_intr)
2022 #define EMLXS_SLI_INTX_INTR             (hba->sli_api.sli_intx_intr)
2023 #define EMLXS_SLI_MSI_INTR              (hba->sli_api.sli_msi_intr)
2024 #define EMLXS_SLI_DISABLE_INTR          (hba->sli_api.sli_disable_intr)
2025 #define EMLXS_SLI_TIMER                 (hba->sli_api.sli_timer)
2026 #define EMLXS_SLI_POLL_ERRATT           (hba->sli_api.sli_poll_erratt)

2028 #define EMLXS_HBA_T  1  /* flag emlxs_hba_t is already typedefed */

2030 #ifdef MSI_SUPPORT
2031 #define EMLXS_INTR_INIT(_hba, _m)               emlxs_msi_init(_hba, _m)
2032 #define EMLXS_INTR_UNINIT(_hba)                 emlxs_msi_uninit(_hba)
2033 #define EMLXS_INTR_ADD(_hba)                    emlxs_msi_add(_hba)
2034 #define EMLXS_INTR_REMOVE(_hba)                 emlxs_msi_remove(_hba)
2035 #else
2036 #define EMLXS_INTR_INIT(_hba, _m)               emlxs_intx_init(_hba, _m)
2037 #define EMLXS_INTR_UNINIT(_hba)                 emlxs_intx_uninit(_hba)
2038 #define EMLXS_INTR_ADD(_hba)                    emlxs_intx_add(_hba)
2039 #define EMLXS_INTR_REMOVE(_hba)                 emlxs_intx_remove(_hba)
2040 #endif  /* MSI_SUPPORT */
```



```
2043 /* Power Management Component */
2044 #define EMLXS_PM_ADAPTER        0


2047 #define DRV_TIME        (uint32_t)((gethrtime() - emlxs_device.drv_timestamp) /
  26 #define DRV_TIME        (uint32_t)(ddi_get_time() - emlxs_device.drv_timestamp)

2049 #define HBA                     port->hba
2050 #define PPORT                   hba->port[0]
2051 #define VPORT(x)                hba->port[x]
2052 #define EMLXS_TIMER_LOCK        hba->timer_lock
2053 #define VPD                     hba->vpd
2054 #define CFG                     hba->config[0]
2055 #define LOG                     hba->log
2056 #define EVENTQ                  hba->event_queue
2057 #define EMLXS_MBOX_LOCK         hba->mbox_lock
2058 #define EMLXS_MBOX_CV           hba->mbox_lock_cv
2059 #define EMLXS_LINKUP_LOCK       hba->linkup_lock
2060 #define EMLXS_LINKUP_CV         hba->linkup_lock_cv
2061 #define EMLXS_TX_CHANNEL_LOCK   hba->channel_tx_lock    /* ring txq lock */
2062 #define EMLXS_MEMGET_LOCK       hba->memget_lock        /* mempool get lock */
2063 #define EMLXS_MEMPUT_LOCK       hba->memput_lock        /* mempool put lock */
2064 #define EMLXS_IOCTL_LOCK        hba->ioctl_lock         /* ioctl lock */
2065 #define EMLXS_SPAWN_LOCK        hba->spawn_lock         /* spawn lock */
2066 #define EMLXS_PM_LOCK           hba->pm_lock            /* pm lock */
2067 #define HBASTATS                hba->stats
2068 #define EMLXS_CMD_RING_LOCK(n)  hba->sli.sli3.ring_cmd_lock[n]

2070 #define EMLXS_QUE_LOCK(n)       hba->sli.sli4.que_lock[n]
2071 #define EMLXS_MSIID_LOCK        hba->msiid_lock

2073 #define EMLXS_FCTAB_LOCK        hba->iotag_lock

2075 #define EMLXS_FCF_LOCK          hba->sli.sli4.fcf_lock

2077 #define EMLXS_PORT_LOCK         hba->port_lock          /* locks ports, */
2078                                                         /* nodes, rings */
2079 #define EMLXS_INTR_LOCK(_id)    hba->intr_lock[_id]     /* locks intr threads */

2081 #define EMLXS_PKT_LOCK          port->pkt_lock          /* used for pkt */
2082                                                         /* polling */
2083 #define EMLXS_PKT_CV            port->pkt_lock_cv       /* Used for pkt */
2084                                                         /* polling */
2085 #define EMLXS_UB_LOCK           port->ub_lock           /* locks unsolicited */
2086                                                         /* buffer pool */

2088 /* These SWAPs will swap on any platform */
2089 #define SWAP32_BUFFER(_b, _c)           emlxs_swap32_buffer(_b, _c)
2090 #define SWAP32_BCOPY(_s, _d, _c)        emlxs_swap32_bcopy(_s, _d, _c)

2092 #define SWAP64(_x)      ((((uint64_t)(_x) & 0xFF)<<56) | \
2093                          (((uint64_t)(_x) & 0xFF00)<<40) | \
2094                          (((uint64_t)(_x) & 0xFF0000)<<24) | \
2095                          (((uint64_t)(_x) & 0xFF000000)<<8) | \
2096                          (((uint64_t)(_x) & 0xFF00000000)>>8) | \
2097                          (((uint64_t)(_x) & 0xFF0000000000)>>24) | \
2098                          (((uint64_t)(_x) & 0xFF000000000000)>>40) | \
2099                          (((uint64_t)(_x) & 0xFF00000000000000)>>56))

2101 #define SWAP32(_x)      ((((uint32_t)(_x) & 0xFF)<<24) | \
2102                          (((uint32_t)(_x) & 0xFF00)<<8) | \
2103                          (((uint32_t)(_x) & 0xFF0000)>>8) | \
2104                          (((uint32_t)(_x) & 0xFF000000)>>24))

2106 #define SWAP16(_x)      ((((uint16_t)(_x) & 0xFF)<<8) | \
```

```
2107                            (((uint16_t)(_x) & 0xFF00)>>8))

2109 #define SWAP24_LO(_x)   ((((uint32_t)(_x) & 0xFF)<<16) | \
2110                            ((uint32_t)(_x) & 0xFF00FF00) | \
2111                            (((uint32_t)(_x) & 0x00FF0000)>>16))

2113 #define SWAP24_HI(_x)   (((uint32_t)(_x) & 0x00FF00FF) | \
2114                            (((uint32_t)(_x) & 0x0000FF00)<<16) | \
2115                            (((uint32_t)(_x) & 0xFF000000)>>16))

2117 /* These LE_SWAPs will only swap on a LE platform */
2118 #ifdef EMLXS_LITTLE_ENDIAN
2119 #define LE_SWAP32_BUFFER(_b, _c)        SWAP32_BUFFER(_b, _c)
2120 #define LE_SWAP32_BCOPY(_s, _d, _c)     SWAP32_BCOPY(_s, _d, _c)
2121 #define LE_SWAP64(_x)                   SWAP64(_x)
2122 #define LE_SWAP32(_x)                   SWAP32(_x)
2123 #define LE_SWAP16(_x)                   SWAP16(_x)
2124 #define LE_SWAP24_LO(_x)                SWAP24_LO(X)
2125 #define LE_SWAP24_HI(_x)                SWAP24_HI(X)

2127 #if (EMLXS_MODREVX == EMLXS_MODREV2X)
2128 #undef  LE_SWAP24_LO
2129 #define LE_SWAP24_LO(_x)                (_x)
2130 #undef  LE_SWAP24_HI
2131 #define LE_SWAP24_HI(_x)                (_x)
2132 #endif  /* EMLXS_MODREV2X */

2134 #else /* BIG ENDIAN */
2135 #define LE_SWAP32_BUFFER(_b, _c)
2136 #define LE_SWAP32_BCOPY(_s, _d, _c)     bcopy(_s, _d, _c)
2137 #define LE_SWAP64(_x)                   (_x)
2138 #define LE_SWAP32(_x)                   (_x)
2139 #define LE_SWAP16(_x)                   (_x)
2140 #define LE_SWAP24_LO(_x)                (_x)
2141 #define LE_SWAP24_HI(_x)                (_x)
2142 #endif /* EMLXS_LITTLE_ENDIAN */

2144 /* These BE_SWAPs will only swap on a BE platform */
2145 #ifdef EMLXS_BIG_ENDIAN
2146 #define BE_SWAP32_BUFFER(_b, _c)        SWAP32_BUFFER(_b, _c)
2147 #define BE_SWAP32_BCOPY(_s, _d, _c)     SWAP32_BCOPY(_s, _d, _c)
2148 #define BE_SWAP64(_x)                   SWAP64(_x)
2149 #define BE_SWAP32(_x)                   SWAP32(_x)
2150 #define BE_SWAP16(_x)                   SWAP16(_x)
2151 #else /* LITTLE ENDIAN */
2152 #define BE_SWAP32_BUFFER(_b, _c)
2153 #define BE_SWAP32_BCOPY(_s, _d, _c)     bcopy(_s, _d, _c)
2154 #define BE_SWAP64(_x)                   (_x)
2155 #define BE_SWAP32(_x)                   (_x)
2156 #define BE_SWAP16(_x)                   (_x)
2157 #endif /* EMLXS_BIG_ENDIAN */

2159 #ifdef  __cplusplus
2160 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   12174 Mon May  5 14:29:46 2014
new/usr/src/uts/common/sys/ib/mgt/ibdm/ibdm_ibnex.h
4777 ibdm shouldn't abuse ddi_get_time(9f)
Reviewed by: Rob Gittins <rob.gittins@nexenta.com>
Reviewed by: Albert Lee <albert.lee@nexenta.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _SYS_IB_MGT_IBDM_IBDM_IBNEX_H
#define _SYS_IB_MGT_IBDM_IBDM_IBNEX_H

/*
 * This file contains the definitions of private interfaces
 * and data structures used between IB nexus and IBDM.
 */

#include <sys/ib/ibtl/ibti_common.h>
#include <sys/ib/mgt/ibmf/ibmf.h>
#include <sys/ib/mgt/ib_dm_attr.h>

#ifdef __cplusplus
extern "C" {
#endif

/* DM return status codes from private interfaces */
typedef enum ibdm_status_e {
        IBDM_SUCCESS = 0,
        IBDM_FAILURE = 1
} ibdm_status_t;

/*
 * IBDM events that are passed to IB nexus driver
 * NOTE: These are different from ibt_async_code_t
 */
typedef enum ibdm_events_e {
        IBDM_EVENT_HCA_ADDED,
        IBDM_EVENT_HCA_REMOVED,
```



```
        IBDM_EVENT_IOC_PROP_UPDATE,
        IBDM_EVENT_PORT_UP,
        IBDM_EVENT_PORT_PKEY_CHANGE
} ibdm_events_t;

/*
 * Flags for ibdm_ibnex_get_ioc_list.
 * The flags determine the functioning of ibdm_ibnex_get_ioc_list.
 *
 *      IBDM_IBNEX_NORMAL_PROBE
 *              Sweep fabric and discover new GIDs only
 *              This value should be same as IBNEX_PROBE_ALLOWED_FLAG
 *      IBDM_IBNEX_DONOT_PROBE
 *              Do not probe, just get the current ioc_list.
 *              This value should be same as IBNEX_DONOT_PROBE_FLAG
 *      IBDM_IBNEX_REPROBE_ALL
 *              Sweep fabric, discover new GIDs. For GIDs
 *              discovered before, reprobe the IOCs on it.
 */
typedef enum ibdm_ibnex_get_ioclist_mtd_e {
        IBDM_IBNEX_NORMAL_PROBE,
        IBDM_IBNEX_DONOT_PROBE,
        IBDM_IBNEX_REPROBE_ALL
} ibdm_ibnex_get_ioclist_mtd_t;


/*
 * Private data structure called from IBDM timeout handler
 */
typedef struct ibdm_timeout_cb_args_s {
        struct ibdm_dp_gidinfo_s        *cb_gid_info;
        int                             cb_req_type;
        int                             cb_ioc_num;             /* IOC# */
        int                             cb_retry_count;
        int                             cb_srvents_start;
        int                             cb_srvents_end;
} ibdm_timeout_cb_args_t;

/*
 * Service entry structure
 */
typedef struct ibdm_srvents_info_s {
        int                             se_state;
        ib_dm_srv_t                     se_attr;
        timeout_id_t                    se_timeout_id;  /* IBDM specific */
        ibdm_timeout_cb_args_t          se_cb_args;
} ibdm_srvents_info_t;

/* values for "se_state" */
#define IBDM_SE_VALID                   0x1
#define IBDM_SE_INVALID                 0x0


/* I/O Controller information */
typedef struct ibdm_ioc_info_s {
        ib_dm_ioc_ctrl_profile_t        ioc_profile;
        int                             ioc_state;
        ibdm_srvents_info_t             *ioc_serv;
        struct ibdm_gid_s               *ioc_gid_list;
        uint_t                          ioc_nportgids;
        ib_guid_t                       ioc_iou_guid;
        timeout_id_t                    ioc_timeout_id;
        timeout_id_t                    ioc_dc_timeout_id;
        boolean_t                       ioc_dc_valid;
        boolean_t                       ioc_iou_dc_valid;
        ibdm_timeout_cb_args_t          ioc_cb_args;
```

```
125         ibdm_timeout_cb_args_t          ioc_dc_cb_args;
126         ib_guid_t                       ioc_nodeguid;
127         uint16_t                        ioc_diagcode;
128         uint16_t                        ioc_iou_diagcode;
129         uint16_t                        ioc_diagdeviceid;
130         struct ibdm_iou_info_s          *ioc_iou_info;
131         struct ibdm_ioc_info_s          *ioc_next;

133         /* Previous fields for reprobe */
134         ibdm_srvents_info_t             *ioc_prev_serv;
135         struct ibdm_gid_s               *ioc_prev_gid_list;
136         uint8_t                         ioc_prev_serv_cnt;
137         uint_t                          ioc_prev_nportgids;

139         /* Flag indicating which IOC info has changed */
140         ibt_prop_update_payload_t       ioc_info_updated;

142         /*
143          * List of HCAs through which IOC is accessible
144          * This field will be initialized in ibdm_ibnex_probe_ioc
145          * and ibdm_get_ioc_list for all IOCs in the fabric.
146          *
147          * HCAs could have been added or deleted from the list,
148          * on calls to ibdm_ibnex_get_ioc_list & ibdm_ibnex_probe_ioc.
149          *
150          * Updates to HCAs in the list will be reported by
151          * IBDM_EVENT_HCA_DOWN and IBDM_EVENT_IOC_HCA_UNREACHABLE events
152          * in the IBDM<->IBDM callback.
153          *
154          * IOC not visible to the host system(because all HCAs cannot
155          * reach the IOC) will be reported in the same manner as TCA
156          * ports getting to 0 (using IOC_PROP_UPDATE event).
157          */
158         struct ibdm_hca_list_s                  *ioc_hca_list;

160 } ibdm_ioc_info_t;
161 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv",
162         ibdm_ioc_info_s::ioc_next))
163 _NOTE(SCHEME_PROTECTS_DATA("Unique per copy of ibdm_ioc_info_t",
164         ibdm_ioc_info_s::ioc_info_updated))
165 _NOTE(DATA_READABLE_WITHOUT_LOCK(ibdm_ioc_info_s::ioc_dc_valid))

167 /* values for "ioc_state */
168 #define IBDM_IOC_STATE_PROBE_SUCCESS    0x0
169 #define IBDM_IOC_STATE_PROBE_INVALID    0x1
170 #define IBDM_IOC_STATE_PROBE_FAILED     0x2
171 #define IBDM_IOC_STATE_REPROBE_PROGRESS 0x4

173 /* I/O Unit Information */
174 typedef struct ibdm_iou_info_s {
175         ib_dm_io_unitinfo_t     iou_info;
176         ibdm_ioc_info_t         *iou_ioc_info;
177         ib_guid_t               iou_guid;
178         boolean_t               iou_dc_valid;
179         uint16_t                iou_diagcode;
180         int                     iou_niocs_probe_in_progress;
181 } ibdm_iou_info_t;


184 /* P_Key table related info */
185 typedef struct ibdm_pkey_tbl_s {
186         ib_pkey_t               pt_pkey;                /* P_Key value */
187         ibmf_qp_handle_t        pt_qp_hdl;              /* QP handle */
188 } ibdm_pkey_tbl_t;
189 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv", ibdm_pkey_tbl_s))
```



```
192 /*
193  * Port Attributes structure
194  */
195 typedef struct ibdm_port_attr_s {
196         ibdm_pkey_tbl_t         *pa_pkey_tbl;
197         ib_guid_t               pa_hca_guid;
198         ib_guid_t               pa_port_guid;
199         uint16_t                pa_npkeys;
200         ibmf_handle_t           pa_ibmf_hdl;
201         ib_sn_prefix_t          pa_sn_prefix;
202         uint16_t                pa_port_num;
203         uint32_t                pa_vendorid;
204         uint32_t                pa_productid;
205         uint32_t                pa_dev_version;
206         ibt_port_state_t        pa_state;
207         ibmf_saa_handle_t       pa_sa_hdl;
208         ibmf_impl_caps_t        pa_ibmf_caps;
209         ibt_hca_hdl_t           pa_hca_hdl;
210 } ibdm_port_attr_t;
211 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv", ibdm_port_attr_s))

213 /*
214  * HCA list structure.
215  */
216 typedef struct ibdm_hca_list_s {
217         ibdm_port_attr_t        *hl_port_attr;          /* port attributes */
218         struct ibdm_hca_list_s  *hl_next;               /* ptr to next list */
219         ib_guid_t               hl_hca_guid;            /* HCA GUID */
220         uint32_t                hl_nports;              /* #ports of this HCA */
221         uint32_t                hl_nports_active;       /* #ports active */
222         hrtime_t                hl_attach_time;         /* attach time */
 25         time_t                  hl_attach_time;         /* attach time */
223         ibt_hca_hdl_t           hl_hca_hdl;             /* HCA handle */
224         ibdm_port_attr_t        *hl_hca_port_attr;      /* Dummy Port Attr */
225                                                         /* for HCA node */
226 } ibdm_hca_list_t;
______unchanged_portion_omitted_
```

```
***********************************************************
   14701 Mon May  5 14:29:47 2014
new/usr/src/uts/common/sys/idm/idm_impl.h
4780 comstar iSCSI target shouldn't abuse ddi_get_time(9f)
Reviewed by: Eric Diven <eric.diven@delphix.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
***********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _IDM_IMPL_H_
#define _IDM_IMPL_H_

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/avl.h>
#include <sys/socket_impl.h>
#include <sys/taskq_impl.h>

/*
 * IDM lock order:
 *
 * idm_taskid_table_lock, idm_task_t.idt_mutex
 */

#define CF_LOGIN_READY          0x00000001
#define CF_INITIAL_LOGIN        0x00000002
#define CF_ERROR                0x80000000

typedef enum {
        CONN_TYPE_INI = 1,
        CONN_TYPE_TGT
} idm_conn_type_t;
_____unchanged_portion_omitted_

#define PDU_MAX_IOVLEN  12
#define IDM_PDU_MAGIC   0x49504455      /* "IPDU" */

typedef struct idm_pdu_s {
```



```
        uint32_t        isp_magic;      /* "IPDU" */

        /*
         * Internal - Order is vital.  idm_tx_link *must* be the second
         * element in this structure for proper TX PDU ordering.
         */
        list_node_t     idm_tx_link;

        list_node_t     isp_client_lnd;

        idm_conn_t      *isp_ic;        /* Must be set */
        iscsi_hdr_t     *isp_hdr;
        uint_t          isp_hdrlen;
        uint8_t         *isp_data;
        uint_t          isp_datalen;

        /* Transport header */
        void            *isp_transport_hdr;
        uint32_t        isp_transport_hdrlen;
        void            *isp_transport_private;

        /*
         * isp_data is used for sending SCSI status, NOP, text, scsi and
         * non-scsi data. Data is received using isp_iov and isp_iovlen
         * to support data over multiple buffers.
         */
        void            *isp_private;
        idm_pdu_cb_t    *isp_callback;
        idm_status_t    isp_status;

        /*
         * The following four elements are only used in
         * idm_sorecv_scsidata() currently.
         */
        struct iovec    isp_iov[PDU_MAX_IOVLEN];
        int             isp_iovlen;
        idm_buf_t       *isp_sorx_buf;

        /* Implementation data for idm_pdu_alloc and sorx PDU cache */
        uint32_t        isp_flags;
        uint_t          isp_hdrbuflen;
        uint_t          isp_databuflen;
        hrtime_t        isp_queue_time;
        time_t          isp_queue_time;

        /* Taskq dispatching state for deferred PDU */
        taskq_ent_t     isp_tqent;
} idm_pdu_t;
_____unchanged_portion_omitted_
```

```
**********************************************************
   22616 Mon May  5 14:29:47 2014
new/usr/src/uts/common/sys/scsi/adapters/scsi_vhci.h
4779 vhci shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

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

/*
 * Copyright (c) 2001, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _SYS_SCSI_ADAPTERS_SCSI_VHCI_H
#define _SYS_SCSI_ADAPTERS_SCSI_VHCI_H

/*
 * Multiplexed I/O SCSI vHCI global include
 */
#include <sys/note.h>
#include <sys/taskq.h>
#include <sys/mhd.h>
#include <sys/sunmdi.h>
#include <sys/mdi_impldefs.h>
#include <sys/scsi/adapters/mpapi_impl.h>
#include <sys/scsi/adapters/mpapi_scsi_vhci.h>

#ifdef  __cplusplus
extern "C" {
#endif

#if !defined(_BIT_FIELDS_LTOH) && !defined(_BIT_FIELDS_HTOL)
#error  One of _BIT_FIELDS_LTOH or _BIT_FIELDS_HTOL must be defined
#endif  /* _BIT_FIELDS_LTOH */

#ifdef  _KERNEL

#ifdef  UNDEFINED
#undef  UNDEFINED
#endif
#define UNDEFINED               -1

#define VHCI_STATE_OPEN         0x00000001
```



```
#define VH_SLEEP                0x0
#define VH_NOSLEEP              0x1

/*
 * HBA interface macros
 */

#define TRAN2HBAPRIVATE(tran)   ((struct scsi_vhci *)(tran)->tran_hba_private)
#define VHCI_INIT_WAIT_TIMEOUT  60000000
#define VHCI_FOWATCH_INTERVAL   1000000         /* in usecs */
#define VHCI_EXTFO_TIMEOUT      (3 * 60 * NANOSEC)      /* 3 minutes in nsec */
#define VHCI_EXTFO_TIMEOUT      3*60            /* 3 minutes */

#define SCBP_C(pkt)     ((*(pkt)->pkt_scbp) & STATUS_MASK)

int vhci_do_scsi_cmd(struct scsi_pkt *);
/*PRINTFLIKE3*/
void vhci_log(int, dev_info_t *, const char *, ...);

/*
 * debugging stuff
 */

#ifdef  DEBUG

#ifndef VHCI_DEBUG_DEFAULT_VAL
#define VHCI_DEBUG_DEFAULT_VAL          0
#endif  /* VHCI_DEBUG_DEFAULT_VAL */

extern int vhci_debug;

#include <sys/debug.h>

#define VHCI_DEBUG(level, stmnt) \
            if (vhci_debug >= (level)) vhci_log stmnt

#else   /* !DEBUG */

#define VHCI_DEBUG(level, stmnt)

#endif  /* !DEBUG */



#define VHCI_PKT_PRIV_SIZE              2

#define ADDR2VHCI(ap)   ((struct scsi_vhci *) \
                        ((ap)->a_hba_tran->tran_hba_private))
#define ADDR2VLUN(ap)   (scsi_vhci_lun_t *) \
                        (scsi_device_hba_private_get(scsi_address_device(ap)))
#define ADDR2DIP(ap)    ((dev_info_t *)(scsi_address_device(ap)->sd_dev))

#define HBAPKT2VHCIPKT(pkt) (pkt->pkt_private)
#define TGTPKT2VHCIPKT(pkt) (pkt->pkt_ha_private)
#define VHCIPKT2HBAPKT(pkt) (pkt->pkt_hba_pkt)
#define VHCIPKT2TGTPKT(pkt) (pkt->pkt_tgt_pkt)

#define VHCI_DECR_PATH_CMDCOUNT(svp) { \
        mutex_enter(&(svp)->svp_mutex); \
        (svp)->svp_cmds--; \
        if ((svp)->svp_cmds == 0)  \
                cv_broadcast(&(svp)->svp_cv); \
        mutex_exit(&(svp)->svp_mutex); \
}
```

*unchanged_portion_omitted_*

```
308 typedef struct scsi_vhci_lun {
309         kmutex_t                svl_mutex;
310         kcondvar_t              svl_cv;

312         /*
313          * following three fields are under svl_mutex protection
314          */
315         int                     svl_transient;

317         /*
318          * to prevent unnecessary failover when a device is
319          * is discovered across a passive path and active path
320          * is still comng up
321          */
322         int                     svl_waiting_for_activepath;
323         hrtime_t                svl_wfa_time;
276         time_t                  svl_wfa_time;

325         /*
326          * to keep the failover status in order to return the
327          * failure status to target driver when targer driver
328          * retries the command which originally triggered the
329          * failover.
330          */
331         int                     svl_failover_status;

333         /*
334          * for RESERVE/RELEASE support
335          */
336         client_lb_t             svl_lb_policy_save;

338         /*
339          * Failover ops and ops name selected for the lun.
340          */
341         struct scsi_failover_ops        *svl_fops;
342         char                    *svl_fops_name;

344         void                    *svl_fops_ctpriv;

346         struct scsi_vhci_lun    *svl_hash_next;
347         char                    *svl_lun_wwn;

349         /*
350          * currently active pathclass
351          */
352         char                    *svl_active_pclass;

354         dev_info_t              *svl_dip;
355         uint32_t                svl_flags;      /* protected by svl_mutex */

357         /*
358          * When SCSI-II reservations are active we set the following pip
359          * to point to the path holding the reservation.  As long as
360          * the reservation is active this svl_resrv_pip is bound for the
361          * transport directly.  We bypass calling mdi_select_path to return
362          * a pip.
363          * The following pip is only valid when VLUN_RESERVE_ACTIVE_FLG
364          * is set.  This pip should not be accessed if this flag is reset.
365          */
366         mdi_pathinfo_t  *svl_resrv_pip;

368         /*
369          * following fields are for PGR support
370          */
371         taskq_t                 *svl_taskq;
```



```
372         ksema_t                 svl_pgr_sema;   /* PGR serialization */
373         vhci_prin_readkeys_t    svl_prin;       /* PGR in data */
374         vhci_prout_t            svl_prout;      /* PGR out data */
375         uchar_t                 svl_cdb[CDB_GROUP4];
376         int                     svl_time;       /* pkt_time */
377         uint32_t                svl_bcount;     /* amount of data */
378         int                     svl_pgr_active; /* registrations active */
379         mdi_pathinfo_t          *svl_first_path;

381         /* external failover */
382         int                     svl_efo_update_path;
383         struct  scsi_vhci_swarg *svl_swarg;

385         uint32_t                svl_support_lun_reset; /* Lun reset support */
386         int                     svl_not_supported;
387         int                     svl_xlf_capable; /* XLF implementation */
388         int                     svl_sector_size;
389         int                     svl_setcap_done;
390         uint16_t                svl_fo_support;  /* failover mode */
391 } scsi_vhci_lun_t;
______unchanged_portion_omitted_

463 /*
464  * argument to scsi_watch callback.  Used for processing
465  * externally initiated failovers
466  */
467 typedef struct scsi_vhci_swarg {
468         scsi_vhci_priv_t        *svs_svp;
469         hrtime_t                svs_tos;        /* time of submission */
422         time_t                  svs_tos;        /* time of submission */
470         mdi_pathinfo_t          *svs_pi;        /* pathinfo being "watched" */
471         int                     svs_release_lun;
472         int                     svs_done;
473 } scsi_vhci_swarg_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   75245 Mon May  5 14:29:47 2014
new/usr/src/uts/common/sys/scsi/targets/sddef.h
4781 sd shouldn't abuse ddi_get_time(9f)
Reviewed by: Richard Elling <richard.elling@gmail.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1990, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2011 cyril.galibern@opensvc.com
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#ifndef _SYS_SCSI_TARGETS_SDDEF_H
#define _SYS_SCSI_TARGETS_SDDEF_H

#include <sys/dktp/fdisk.h>
#include <sys/note.h>
#include <sys/mhd.h>
#include <sys/cmlb.h>

#ifdef  __cplusplus
extern "C" {
#endif


#if defined(_KERNEL) || defined(_KMEMUSER)


#define SD_SUCCESS              0
#define SD_FAILURE              (-1)

#if defined(TRUE)
#undef TRUE
#endif

#if defined(FALSE)
#undef FALSE
#endif

#define TRUE                    1
#define FALSE                   0
```



```
#if defined(VERBOSE)
#undef VERBOSE
#endif

#if defined(SILENT)
#undef SILENT
#endif


/*
 *  Fault Injection Flag for Inclusion of Code
 *
 *  This should only be defined when SDDEBUG is defined
 * #if DEBUG || lint
 * #define      SD_FAULT_INJECTION
 * #endif
 */

#if DEBUG || lint
#define SD_FAULT_INJECTION
#endif
#define VERBOSE                 1
#define SILENT                  0

/*
 * Structures for recording whether a device is fully open or closed.
 * Assumptions:
 *
 *      + There are only 8 (sparc) or 16 (x86) disk slices possible.
 *      + BLK, MNT, CHR, SWP don't change in some future release!
 */

#if defined(_SUNOS_VTOC_8)

#define SDUNIT_SHIFT    3
#define SDPART_MASK     7
#define NSDMAP          NDKMAP

#elif defined(_SUNOS_VTOC_16)

/*
 * XXX - NSDMAP has multiple definitions, one more in cmlb_impl.h
 * If they are coalesced into one, this definition will follow suit.
 * FDISK partitions - 4 primary and MAX_EXT_PARTS number of Extended
 * Partitions.
 */
#define FDISK_PARTS             (FD_NUMPART + MAX_EXT_PARTS)

#define SDUNIT_SHIFT    6
#define SDPART_MASK     63
#define NSDMAP          (NDKMAP + FDISK_PARTS + 1)

#else
#error "No VTOC format defined."
#endif


#define SDUNIT(dev)     (getminor((dev)) >> SDUNIT_SHIFT)
#define SDPART(dev)     (getminor((dev)) &  SDPART_MASK)

/*
 * maximum number of partitions the driver keeps track of; with
 * EFI this can be larger than the number of partitions accessible
 * through the minor nodes.  It won't be used for keeping track
 * of open counts, partition kstats, etc.
 */
```

```
#define MAXPART         (NSDMAP + 1)

/*
 * Macro to retrieve the DDI instance number from the given buf struct.
 * The instance number is encoded in the minor device number.
 */
#define SD_GET_INSTANCE_FROM_BUF(bp)                                    \
        (getminor((bp)->b_edev) >> SDUNIT_SHIFT)



struct ocinfo {
        /*
         * Types BLK, MNT, CHR, SWP,
         * assumed to be types 0-3.
         */
        uint64_t  lyr_open[NSDMAP];
        uint64_t  reg_open[OTYPCNT - 1];
};

#define OCSIZE  sizeof (struct ocinfo)

union ocmap {
        uchar_t chkd[OCSIZE];
        struct ocinfo rinfo;
};

#define lyropen rinfo.lyr_open
#define regopen rinfo.reg_open


#define SD_CDB_GROUP0           0
#define SD_CDB_GROUP1           1
#define SD_CDB_GROUP5           2
#define SD_CDB_GROUP4           3

struct sd_cdbinfo {
        uchar_t  sc_grpcode;    /* CDB group code */
        uchar_t  sc_grpmask;    /* CDB group code mask (for cmd opcode) */
        uint64_t sc_maxlba;     /* Maximum logical block addr. supported */
        uint32_t sc_maxlen;     /* Maximum transfer length supported */
};



/*
 * The following declaration are for Non-512 byte block support for the
 * removable devices. (ex - DVD RAM, MO).
 * wm_state: This is an enumeration for the different states for
 * manipalating write range list during the read-modify-write-operation.
 */
typedef enum {
        SD_WM_CHK_LIST,         /* Check list for overlapping writes */
        SD_WM_WAIT_MAP,         /* Wait for an overlapping I/O to complete */
        SD_WM_LOCK_RANGE,       /* Lock the range of lba to be written */
        SD_WM_DONE              /* I/O complete */
} wm_state;

/*
 * sd_w_map: Every write I/O will get one w_map allocated for it which will tell
 * the range on the media which is being written for that request.
 */
struct sd_w_map {
        uint_t          wm_start;       /* Write start location */
        uint_t          wm_end;         /* Write end location */
        ushort_t        wm_flags;       /* State of the wmap */
```



```
        ushort_t        wm_wanted_count; /* # of threads waiting for region */
        void            *wm_private;    /* Used to store bp->b_private */
        struct buf      *wm_bufp;       /* to store buf pointer */
        struct sd_w_map *wm_next;       /* Forward pointed to sd_w_map */
        struct sd_w_map *wm_prev;       /* Back pointer to sd_w_map */
        kcondvar_t      wm_avail;       /* Sleep on this, while not available */
};

_NOTE(MUTEX_PROTECTS_DATA(scsi_device::sd_mutex, sd_w_map::wm_flags))


/*
 * This is the struct for the layer-private data area for the
 * mapblocksize layer.
 */

struct sd_mapblocksize_info {
        void            *mbs_oprivate;  /* saved value of xb_private */
        struct buf      *mbs_orig_bp;   /* ptr to original bp */
        struct sd_w_map *mbs_wmp;       /* ptr to write-map struct for RMW */
        ssize_t         mbs_copy_offset;
        int             mbs_layer_index;        /* chain index for RMW */
};

_NOTE(SCHEME_PROTECTS_DATA("unshared data", sd_mapblocksize_info))


/*
 * sd_lun: The main data structure for a scsi logical unit.
 * Stored as the softstate structure for each device.
 */

struct sd_lun {

        /* Back ptr to the SCSA scsi_device struct for this LUN */
        struct scsi_device      *un_sd;

        /*
         * Support for Auto-Request sense capability
         */
        struct buf      *un_rqs_bp;     /* ptr to request sense bp */
        struct scsi_pkt *un_rqs_pktp;   /* ptr to request sense scsi_pkt */
        int     un_sense_isbusy;        /* Busy flag for RQS buf */

        /*
         * These specify the layering chains to use with this instance. These
         * are initialized according to the values in the sd_chain_index_map[]
         * array. See the description of sd_chain_index_map[] for details.
         */
        int un_buf_chain_type;
        int un_uscsi_chain_type;
        int un_direct_chain_type;
        int un_priority_chain_type;

        /* Head & tail ptrs to the queue of bufs awaiting transport */
        struct buf      *un_waitq_headp;
        struct buf      *un_waitq_tailp;

        /* Ptr to the buf currently being retried (NULL if none) */
        struct buf      *un_retry_bp;

        /* This tracks the last kstat update for the un_retry_bp buf */
        void            (*un_retry_statp)(kstat_io_t *);

        void            *un_xbuf_attr;  /* xbuf attribute struct */
```

```
259         /* System logical block size, in bytes. (defaults to DEV_BSIZE.) */
260         uint32_t        un_sys_blocksize;
261
262         /* The size of a logical block on the target, in bytes. */
263         uint32_t        un_tgt_blocksize;
264
265         /* The size of a physical block on the target, in bytes. */
266         uint32_t        un_phy_blocksize;
267
268         /*
269          * The number of logical blocks on the target. This is adjusted
270          * to be in terms of the block size specified by un_sys_blocksize
271          * (ie, the system block size).
272          */
273         uint64_t        un_blockcount;
274
275         /*
276          * Various configuration data
277          */
278         uchar_t un_ctype;               /* Controller type */
279         char    *un_node_type;          /* minor node type */
280         uchar_t un_interconnect_type;   /* Interconnect for underlying HBA */
281
282         uint_t  un_notready_retry_count; /* Per disk notready retry count */
283         uint_t  un_busy_retry_count;    /* Per disk BUSY retry count */
284
285         uint_t  un_retry_count;         /* Per disk retry count */
286         uint_t  un_victim_retry_count;  /* Per disk victim retry count */
287
288         /* (4356701, 4367306) */
289         uint_t  un_reset_retry_count; /* max io retries before issuing reset */
290         ushort_t un_reserve_release_time; /* reservation release timeout */
291
292         uchar_t un_reservation_type;    /* SCSI-3 or SCSI-2 */
293         uint_t  un_max_xfer_size;       /* Maximum DMA transfer size */
294         int     un_partial_dma_supported;
295         int     un_buf_breakup_supported;
296
297         int     un_mincdb;              /* Smallest CDB to use */
298         int     un_maxcdb;              /* Largest CDB to use */
299         int     un_max_hba_cdb;         /* Largest CDB supported by HBA */
300         int     un_status_len;
301         int     un_pkt_flags;
302
303         /*
304          * Note: un_uscsi_timeout is a "mirror" of un_cmd_timeout, adjusted
305          * for ISCD().  Any updates to un_cmd_timeout MUST be reflected
306          * in un_uscsi_timeout as well!
307          */
308         ushort_t un_cmd_timeout;        /* Timeout for completion */
309         ushort_t un_uscsi_timeout;      /* Timeout for USCSI completion */
310         ushort_t un_busy_timeout;       /* Timeout for busy retry */
311
312         /*
313          * Info on current states, statuses, etc. (Updated frequently)
314          */
315         uchar_t un_state;               /* current state */
316         uchar_t un_last_state;          /* last state */
317         uchar_t un_last_pkt_reason;     /* used to suppress multiple msgs */
318         int     un_tagflags;            /* Pkt Flags for Tagged Queueing  */
319         short   un_resvd_status;        /* Reservation Status */
320         ulong_t un_detach_count;        /* !0 if executing detach routine */
321         ulong_t un_layer_count;         /* Current total # of layered opens */
322         ulong_t un_opens_in_progress;   /* Current # of threads in sdopen */
```



```
324         ksema_t un_semoclose;           /* serialize opens/closes */
325
326         /*
327          * Control & status info for command throttling
328          */
329         long    un_ncmds_in_driver;     /* number of cmds in driver */
330         short   un_ncmds_in_transport;  /* number of cmds in transport */
331         short   un_throttle;            /* max #cmds allowed in transport */
332         short   un_saved_throttle;      /* saved value of un_throttle */
333         short   un_busy_throttle;       /* saved un_throttle for BUSY */
334         short   un_min_throttle;        /* min value of un_throttle */
335         timeout_id_t    un_reset_throttle_timeid; /* timeout(9F) handle */
336
337         /*
338          * Multi-host (clustering) support
339          */
340         opaque_t        un_mhd_token;           /* scsi watch request */
341         timeout_id_t    un_resvd_timeid;        /* for resvd recover */
342
343         /* Event callback resources (photon) */
344         ddi_eventcookie_t un_insert_event;      /* insert event */
345         ddi_callback_id_t un_insert_cb_id;      /* insert callback */
346         ddi_eventcookie_t un_remove_event;      /* remove event */
347         ddi_callback_id_t un_remove_cb_id;      /* remove callback */
348
349         uint_t          un_start_stop_cycle_page;       /* Saves start/stop */
350                                                         /* cycle page */
351         timeout_id_t    un_dcvb_timeid;         /* dlyd cv broadcast */
352
353         /*
354          * Data structures for open counts, partition info, VTOC,
355          * stats, and other such bookkeeping info.
356          */
357         union   ocmap   un_ocmap;               /* open partition map */
358         struct  kstat   *un_pstats[NSDMAP];     /* partition statistics */
359         struct  kstat   *un_stats;              /* disk statistics */
360         kstat_t         *un_errstats;           /* for error statistics */
361         uint64_t        un_exclopen;            /* exclusive open bitmask */
362         ddi_devid_t     un_devid;               /* device id */
363         uint_t          un_vpd_page_mask;       /* Supported VPD pages */
364
365         /*
366          * Bit fields for various configuration/state/status info.
367          * Comments indicate the condition if the value of the
368          * variable is TRUE (nonzero).
369          */
370         uint32_t
371             un_f_arq_enabled            :1,     /* Auto request sense is */
372                                                 /* currently enabled */
373             un_f_blockcount_is_valid    :1,     /* The un_blockcount */
374                                                 /* value is currently valid */
375             un_f_tgt_blocksize_is_valid :1,     /* The un_tgt_blocksize */
376                                                 /* value is currently valid */
377             un_f_allow_bus_device_reset :1,     /* Driver may issue a BDR as */
378                                                 /* a part of error recovery. */
379             un_f_is_fibre               :1,     /* The device supports fibre */
380                                                 /* channel */
381             un_f_sync_cache_supported   :1,     /* sync cache cmd supported */
382                                                 /* supported */
383             un_f_format_in_progress     :1,     /* The device is currently */
384                                                 /* executing a FORMAT cmd. */
385             un_f_opt_queueing           :1,     /* Enable Command Queuing to */
386                                                 /* Host Adapter */
387             un_f_opt_fab_devid          :1,     /* Disk has no valid/unique */
388                                                 /* serial number.  */
389             un_f_opt_disable_cache      :1,     /* Read/Write disk cache is */
```

```
390                                             /* disabled.  */
391             un_f_cfg_is_atapi           :1,     /* This is an ATAPI device.  */
392             un_f_write_cache_enabled    :1,     /* device return success on */
393                                             /* writes before transfer to */
394                                             /* physical media complete */
395             un_f_cfg_playmsf_bcd        :1,     /* Play Audio, BCD params. */
396             un_f_cfg_readsub_bcd        :1,     /* READ SUBCHANNEL BCD resp. */
397             un_f_cfg_read_toc_trk_bcd   :1,     /* track # is BCD */
398             un_f_cfg_read_toc_addr_bcd  :1,     /* address is BCD */
399             un_f_cfg_no_read_header     :1,     /* READ HEADER not supported */
400             un_f_cfg_read_cd_xd4        :1,     /* READ CD opcode is 0xd4 */
401             un_f_mmc_cap                :1,     /* Device is MMC compliant */
402             un_f_mmc_writable_media     :1,     /* writable media in device */
403             un_f_dvdram_writable_device :1,     /* DVDRAM device is writable */
404             un_f_cfg_cdda               :1,     /* READ CDDA supported */
405             un_f_cfg_tur_check          :1,     /* verify un_ncmds before tur */

407             un_f_use_adaptive_throttle  :1,     /* enable/disable adaptive */
408                                             /* throttling */
409             un_f_pm_is_enabled          :1,     /* PM is enabled on this */
410                                             /* instance */
411             un_f_watcht_stopped         :1,     /* media watch thread flag */
412             un_f_pkstats_enabled        :1,     /* Flag to determine if */
413                                             /* partition kstats are */
414                                             /* enabled. */
415             un_f_disksort_disabled      :1,     /* Flag to disable disksort */
416             un_f_lun_reset_enabled      :1,     /* Set if target supports */
417                                             /* SCSI Logical Unit Reset */
418             un_f_doorlock_supported     :1,     /* Device supports Doorlock */
419             un_f_start_stop_supported   :1,     /* device has motor */
420             un_f_reserved1              :1;

422         uint32_t
423             un_f_mboot_supported        :1,     /* mboot supported */
424             un_f_is_hotpluggable        :1,     /* hotpluggable */
425             un_f_has_removable_media    :1,     /* has removable media */
426             un_f_non_devbsize_supported :1,     /* non-512 blocksize */
427             un_f_devid_supported        :1,     /* device ID supported */
428             un_f_eject_media_supported  :1,     /* media can be ejected */
429             un_f_chk_wp_open            :1,     /* check if write-protected */
430                                             /* when being opened */
431             un_f_descr_format_supported :1,     /* support descriptor format */
432                                             /* for sense data */
433             un_f_check_start_stop       :1,     /* needs to check if */
434                                             /* START-STOP command is */
435                                             /* supported by hardware */
436                                             /* before issuing it */
437             un_f_monitor_media_state    :1,     /* need a watch thread to */
438                                             /* monitor device state */
439             un_f_attach_spinup          :1,     /* spin up once the */
440                                             /* device is attached */
441             un_f_log_sense_supported    :1,     /* support log sense */
442             un_f_pm_supported           :1,     /* support power-management */
443             un_f_cfg_is_lsi             :1,     /* Is LSI device, */
444                                             /* default to NO */
445             un_f_wcc_inprog             :1,     /* write cache change in */
446                                             /* progress */
447             un_f_ejecting               :1,     /* media is ejecting */
448             un_f_suppress_cache_flush   :1,     /* supress flush on */
449                                             /* write cache */
450             un_f_sync_nv_supported      :1,     /* SYNC_NV */
451                                             /* bit is supported */
452             un_f_sync_cache_required    :1,     /* flag to check if */
453                                             /* SYNC CACHE needs to be */
454                                             /* sent in sdclose */
455             un_f_devid_transport_defined :1,    /* devid defined by transport */
```



```
456             un_f_rmw_type                :2,    /* RMW type */
457             un_f_power_condition_disabled :1,   /* power condition disabled */
458                                                 /* through sd configuration */
459             un_f_power_condition_supported :1,  /* support power condition */
460                                                 /* field by hardware */
461             un_f_pm_log_sense_smart     :1,     /* log sense support SMART */
462                                                 /* feature attribute */
463             un_f_is_solid_state         :1,     /* has solid state media */
464             un_f_mmc_gesn_polling       :1,     /* use GET EVENT STATUS */
465                                                 /* NOTIFICATION for polling */
466             un_f_enable_rmw             :1,     /* Force RMW in sd driver */
467             un_f_expnevent              :1,
468             un_f_reserved               :3;

470         /* Ptr to table of strings for ASC/ASCQ error message printing */
471         struct scsi_asq_key_strings     *un_additional_codes;

473         /*
474          * Power Management support.
475          *
476          * un_pm_mutex protects, un_pm_count, un_pm_timeid, un_pm_busy,
477          * un_pm_busy_cv, and un_pm_idle_timeid.
478          * It's not required that SD_MUTEX be acquired before acquiring
479          * un_pm_mutex, however if they must both be held
480          * then acquire SD_MUTEX first.
481          *
482          * un_pm_count is used to indicate PM state as follows:
483          *      less than 0 the device is powered down,
484          *      transition from 0 ==> 1, mark the device as busy via DDI
485          *      transition from 1 ==> 0, mark the device as idle via DDI
486          */
487         kmutex_t        un_pm_mutex;
488         int             un_pm_count;            /* indicates pm state */
489         timeout_id_t    un_pm_timeid;           /* timeout id for pm */
490         uint_t          un_pm_busy;
491         kcondvar_t      un_pm_busy_cv;
492         short           un_power_level;         /* Power Level */
493         uchar_t         un_save_state;
494         kcondvar_t      un_suspend_cv;          /* power management */
495         kcondvar_t      un_disk_busy_cv;        /* wait for IO completion */

497         /* Resources used for media change callback support */
498         kcondvar_t      un_state_cv;            /* Cond Var on mediastate */
499         enum dkio_state un_mediastate;          /* current media state */
500         enum dkio_state un_specified_mediastate; /* expected state */
501         opaque_t        un_swr_token;           /* scsi_watch request token */

503         /* Non-512 byte block support */
504         struct kmem_cache *un_wm_cache; /* fast alloc in non-512 write case */
505         uint_t          un_rmw_count;   /* count of read-modify-writes */
506         struct sd_w_map *un_wm;         /* head of sd_w_map chain */
507         uint64_t        un_rmw_incre_count;     /* count I/O */
508         timeout_id_t    un_rmw_msg_timeid;      /* for RMW message control */

510         /* For timeout callback to issue a START STOP UNIT command */
511         timeout_id_t    un_startstop_timeid;

513         /* Timeout callback handle for SD_PATH_DIRECT_PRIORITY cmd restarts */
514         timeout_id_t    un_direct_priority_timeid;

516         /* TRAN_FATAL_ERROR count. Cleared by TRAN_ACCEPT from scsi_transport */
517         ulong_t         un_tran_fatal_count;

519         timeout_id_t    un_retry_timeid;

521         hrtime_t        un_pm_idle_time;
```

```
 26          time_t          un_pm_idle_time;
522          timeout_id_t    un_pm_idle_timeid;

524          /*
525           * Count to determine if a Sonoma controller is in the process of
526           * failing over, and how many I/O's are failed with the 05/94/01
527           * sense code.
528           */
529          uint_t          un_sonoma_failure_count;

531          /*
532           * Support for failfast operation.
533           */
534          struct buf      *un_failfast_bp;
535          struct buf      *un_failfast_headp;
536          struct buf      *un_failfast_tailp;
537          uint32_t        un_failfast_state;
538          /* Callback routine active counter */
539          short           un_in_callback;

541          kcondvar_t      un_wcc_cv;      /* synchronize changes to */
542                                          /* un_f_write_cache_enabled */

544 #ifdef SD_FAULT_INJECTION
545          /* SD Fault Injection */
546 #define SD_FI_MAX_BUF 65536
547 #define SD_FI_MAX_ERROR 1024
548          kmutex_t                        un_fi_mutex;
549          uint_t                          sd_fi_buf_len;
550          char                            sd_fi_log[SD_FI_MAX_BUF];
551          struct sd_fi_pkt        *sd_fi_fifo_pkt[SD_FI_MAX_ERROR];
552          struct sd_fi_xb         *sd_fi_fifo_xb[SD_FI_MAX_ERROR];
553          struct sd_fi_un         *sd_fi_fifo_un[SD_FI_MAX_ERROR];
554          struct sd_fi_arq        *sd_fi_fifo_arq[SD_FI_MAX_ERROR];
555          uint_t                          sd_fi_fifo_start;
556          uint_t                          sd_fi_fifo_end;
557          uint_t                          sd_injection_mask;

559 #endif

561          cmlb_handle_t   un_cmlbhandle;

563          /*
564           * Pointer to internal struct sd_fm_internal in which
565           * will pass necessary information for FMA ereport posting.
566           */
567          void            *un_fm_private;
568 };
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   12448 Mon May  5 14:29:47 2014
new/usr/src/uts/common/sys/usb/hubd/hubdvar.h
4782 usba shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _SYS_USB_HUBDVAR_H
#define _SYS_USB_HUBDVAR_H


#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/sunndi.h>
#include <sys/ndi_impldefs.h>
#include <sys/usb/usba/usba_types.h>
#include <sys/callb.h>

/*
 * HUB USB device state management :
 *
 *                          CHILD_PWRLVL---1>--------+
 *                              ^                    |
 *                              8                    |
 *                              |                    |
 *                              9                    |
 *                              v                    |
 *      PWRED_DWN---<3----4>--ONLINE---<2-----1>-DISCONNECTED
 *          |                 |  ^                 |  |
 *          |                 |  10                |  |
 *          |                 |  |                 |  |
 *          |                 |  RECOVER-<2-------+  |
 *          |                 |  ^                   |
 *          |                5  6                   |
 *          |                v  |                   |
```



```
 *          +----5>----------SUSPENDED----<5----7>----+
 *
 *      1 = Device Unplug
 *      2 = Original Device reconnected and after hub driver restores its own
 *          device state.
 *      3 = Device idles for time T & transitions to low power state
 *      4 = Remote wakeup by device OR Application kicking off IO to device
 *      5 = Notification to save state prior to DDI_SUSPEND
 *      6 = Notification to restore state after DDI_RESUME with correct device
 *          and after hub driver restores its own device state.
 *      7 = Notification to restore state after DDI_RESUME with device
 *          disconnected or a wrong device
 *      8 = Hub detect child doing remote wakeup and request the PM
 *          framework to bring it to full power
 *      9 = PM framework has compeleted call power entry point of the child
 *          and bus ctls of hub
 *     10 = Restoring states of its children i.e. set addrs & config.
 *
 */

#define HUBD_INITIAL_SOFT_SPACE 4

typedef struct hub_power_struct {
        void            *hubp_hubd;     /* points back to hubd_t */

        uint8_t         hubp_wakeup_enabled;    /* remote wakeup enabled? */

        /* this is the bit mask of the power states that device has */
        uint8_t         hubp_pwr_states;

        int             hubp_busy_pm;   /* device busy accounting */

        /* wakeup and power transition capabilities of an interface */
        uint8_t         hubp_pm_capabilities;

        uint8_t         hubp_current_power;     /* current power level */

        hrtime_t        hubp_time_at_full_power;        /* timestamp 0->3 */
        time_t          hubp_time_at_full_power;        /* timestamp 0->3 */

        hrtime_t        hubp_min_pm_threshold;          /* in nanoseconds */
        uint8_t         hubp_min_pm_threshold;          /* in seconds */

        /* power state of all children are tracked here */
        uint8_t         *hubp_child_pwrstate;

        /* pm-components properties are stored here */
        char            *hubp_pmcomp[5];

        usba_cfg_pwr_descr_t    hubp_confpwr_descr; /* config pwr descr */
} hub_power_t;
______unchanged_portion_omitted_
```