```
**********************************************************
  212994 Tue Apr 15 13:19:36 2014
new/usr/src/uts/common/io/ib/mgt/ibdm/ibdm.c
patch fix
**********************************************************
```

```
/*
 * 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.
 */

/*
 * 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 *,
                                    ib_lid_t);
static ibdm_dp_gidinfo_t        *ibdm_create_gid_info(ibdm_port_attr_t *,
                                    ib_gid_t, ib_gid_t);
static ibdm_dp_gidinfo_t        *ibdm_find_gid(ib_guid_t, ib_guid_t);
static int      ibdm_send_ioc_profile(ibdm_dp_gidinfo_t *, uint8_t);
static ibdm_ioc_info_t  *ibdm_update_ioc_gidlist(ibdm_dp_gidinfo_t *, int);
static void     ibdm_saa_event_cb(ibmf_saa_handle_t, ibmf_saa_subnet_event_t,
```

```
128                     ibmf_saa_event_details_t *, void *);
129 static void     ibdm_reprobe_update_port_srv(ibdm_ioc_info_t *,
130     ibdm_dp_gidinfo_t *);
131 static ibdm_dp_gidinfo_t *ibdm_handle_gid_rm(ibdm_dp_gidinfo_t *);
132 static void ibdm_rmfrom_glgid_list(ibdm_dp_gidinfo_t *,
133     ibdm_dp_gidinfo_t *);
134 static void ibdm_addto_gidlist(ibdm_gid_t **, ibdm_gid_t *);
135 static void ibdm_free_gid_list(ibdm_gid_t *);
136 static void ibdm_rescan_gidlist(ib_guid_t *ioc_guid);
137 static void ibdm_notify_newgid_iocs(ibdm_dp_gidinfo_t *);
138 static void ibdm_saa_event_taskq(void *);
139 static void ibdm_free_saa_event_arg(ibdm_saa_event_arg_t *);
140 static void ibdm_get_next_port(ibdm_hca_list_t **,
141     ibdm_port_attr_t **, int);
142 static void ibdm_add_to_gl_gid(ibdm_dp_gidinfo_t *,
143     ibdm_dp_gidinfo_t *);
144 static void ibdm_addto_glhcalist(ibdm_dp_gidinfo_t *,
145     ibdm_hca_list_t *);
146 static void ibdm_delete_glhca_list(ibdm_dp_gidinfo_t *);
147 static void ibdm_saa_handle_new_gid(void *);
148 static void ibdm_reset_all_dgids(ibmf_saa_handle_t);
149 static void ibdm_reset_gidinfo(ibdm_dp_gidinfo_t *);
150 static void ibdm_delete_gidinfo(ibdm_dp_gidinfo_t *);
151 static void ibdm_fill_srv_attr_mod(ib_mad_hdr_t *, ibdm_timeout_cb_args_t *);
152 static void ibdm_bump_transactionID(ibdm_dp_gidinfo_t *);
153 static ibdm_ioc_info_t  *ibdm_handle_prev_iou();
154 static int ibdm_serv_cmp(ibdm_srvents_info_t *, ibdm_srvents_info_t *,
155     int);
156 static ibdm_ioc_info_t *ibdm_get_ioc_info_with_gid(ib_guid_t,
157     ibdm_dp_gidinfo_t **);

159 int     ibdm_dft_timeout        = IBDM_DFT_TIMEOUT;
160 int     ibdm_dft_retry_cnt      = IBDM_DFT_NRETRIES;
161 #ifdef DEBUG
162 int     ibdm_ignore_saa_event = 0;
163 #endif
164 int     ibdm_enumerate_iocs = 0;

166 /* Modload support */
167 static struct modlmisc ibdm_modlmisc    = {
168         &mod_miscops,
169         "InfiniBand Device Manager"
170 };

172 struct modlinkage ibdm_modlinkage = {
173         MODREV_1,
174         (void *)&ibdm_modlmisc,
175         NULL
176 };

178 static ibt_clnt_modinfo_t ibdm_ibt_modinfo = {
179         IBTI_V_CURR,
180         IBT_DM,
181         ibdm_event_hdlr,
182         NULL,
183         "ibdm"
184 };

186 /* Global variables */
187 ibdm_t  ibdm;
188 int     ibdm_taskq_enable = IBDM_ENABLE_TASKQ_HANDLING;
189 char    *ibdm_string = "ibdm";

191 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv",
192     ibdm.ibdm_dp_gidlist_head))
```



```
194 /*
195  * _init
196  *      Loadable module init, called before any other module.
197  *      Initialize mutex
198  *      Register with IBTF
199  */
200 int
201 _init(void)
202 {
203         int             err;

205         IBTF_DPRINTF_L4("ibdm", "\t_init: addr of ibdm %p", &ibdm);

207         if ((err = ibdm_init()) != IBDM_SUCCESS) {
208                 IBTF_DPRINTF_L2("ibdm", "_init: ibdm_init failed 0x%x", err);
209                 (void) ibdm_fini();
210                 return (DDI_FAILURE);
211         }

213         if ((err = mod_install(&ibdm_modlinkage)) != 0) {
214                 IBTF_DPRINTF_L2("ibdm", "_init: mod_install failed 0x%x", err);
215                 (void) ibdm_fini();
216         }
217         return (err);
218 }


221 int
222 _fini(void)
223 {
224         int err;

226         if ((err = ibdm_fini()) != IBDM_SUCCESS) {
227                 IBTF_DPRINTF_L2("ibdm", "_fini: ibdm_fini failed 0x%x", err);
228                 (void) ibdm_init();
229                 return (EBUSY);
230         }

232         if ((err = mod_remove(&ibdm_modlinkage)) != 0) {
233                 IBTF_DPRINTF_L2("ibdm", "_fini: mod_remove failed 0x%x", err);
234                 (void) ibdm_init();
235         }
236         return (err);
237 }


240 int
241 _info(struct modinfo *modinfop)
242 {
243         return (mod_info(&ibdm_modlinkage, modinfop));
244 }


247 /*
248  * ibdm_init():
249  *      Register with IBTF
250  *      Allocate memory for the HCAs
251  *      Allocate minor-nodes for the HCAs
252  */
253 static int
254 ibdm_init(void)
255 {
256         int                     i, hca_count;
257         ib_guid_t               *hca_guids;
258         ibt_status_t            status;
```

```
260         IBTF_DPRINTF_L4("ibdm", "\tibdm_init:");
261         if (!(ibdm.ibdm_state & IBDM_LOCKS_ALLOCED)) {
262                 mutex_init(&ibdm.ibdm_mutex, NULL, MUTEX_DEFAULT, NULL);
263                 mutex_init(&ibdm.ibdm_hl_mutex, NULL, MUTEX_DEFAULT, NULL);
264                 mutex_init(&ibdm.ibdm_ibnex_mutex, NULL, MUTEX_DEFAULT, NULL);
265                 cv_init(&ibdm.ibdm_port_settle_cv, NULL, CV_DRIVER, NULL);
266                 mutex_enter(&ibdm.ibdm_mutex);
267                 ibdm.ibdm_state |= IBDM_LOCKS_ALLOCED;
268         }

270         if (!(ibdm.ibdm_state & IBDM_IBT_ATTACHED)) {
271                 if ((status = ibt_attach(&ibdm_ibt_modinfo, NULL, NULL,
272                     (void *)&ibdm.ibdm_ibt_clnt_hdl)) != IBT_SUCCESS) {
273                         IBTF_DPRINTF_L2("ibdm", "ibdm_init: ibt_attach "
274                             "failed %x", status);
275                         mutex_exit(&ibdm.ibdm_mutex);
276                         return (IBDM_FAILURE);
277                 }

279                 ibdm.ibdm_state |= IBDM_IBT_ATTACHED;
280                 mutex_exit(&ibdm.ibdm_mutex);
281         }


284         if (!(ibdm.ibdm_state & IBDM_HCA_ATTACHED)) {
285                 hca_count = ibt_get_hca_list(&hca_guids);
286                 IBTF_DPRINTF_L4("ibdm", "ibdm_init: num_hcas = %d", hca_count);
287                 for (i = 0; i < hca_count; i++)
288                         (void) ibdm_handle_hca_attach(hca_guids[i]);
289                 if (hca_count)
290                         ibt_free_hca_list(hca_guids, hca_count);

292                 mutex_enter(&ibdm.ibdm_mutex);
293                 ibdm.ibdm_state |= IBDM_HCA_ATTACHED;
294                 mutex_exit(&ibdm.ibdm_mutex);
295         }

297         if (!(ibdm.ibdm_state & IBDM_CVS_ALLOCED)) {
298                 cv_init(&ibdm.ibdm_probe_cv, NULL, CV_DRIVER, NULL);
299                 cv_init(&ibdm.ibdm_busy_cv, NULL, CV_DRIVER, NULL);
300                 mutex_enter(&ibdm.ibdm_mutex);
301                 ibdm.ibdm_state |= IBDM_CVS_ALLOCED;
302                 mutex_exit(&ibdm.ibdm_mutex);
303         }
304         return (IBDM_SUCCESS);
305 }


308 static int
309 ibdm_free_iou_info(ibdm_dp_gidinfo_t *gid_info, ibdm_iou_info_t **ioup)
310 {
311         int                     ii, k, niocs;
312         size_t                  size;
313         ibdm_gid_t              *delete, *head;
314         timeout_id_t            timeout_id;
315         ibdm_ioc_info_t         *ioc;
316         ibdm_iou_info_t         *gl_iou = *ioup;

318         ASSERT(mutex_owned(&gid_info->gl_mutex));
319         if (gl_iou == NULL) {
320                 IBTF_DPRINTF_L4("ibdm", "\tibdm_free_iou_info: No IOU");
321                 return (0);
322         }

324         niocs = gl_iou->iou_info.iou_num_ctrl_slots;
325         IBTF_DPRINTF_L4("ibdm", "\tfree_iou_info: gid_info = %p, niocs %d",
```



```
326             gid_info, niocs);

328         for (ii = 0; ii < niocs; ii++) {
329                 ioc = (ibdm_ioc_info_t *)&gl_iou->iou_ioc_info[ii];

331                 /* handle the case where an ioc_timeout_id is scheduled */
332                 if (ioc->ioc_timeout_id) {
333                         timeout_id = ioc->ioc_timeout_id;
334                         ioc->ioc_timeout_id = 0;
335                         mutex_exit(&gid_info->gl_mutex);
336                         IBTF_DPRINTF_L5("ibdm", "free_iou_info: "
337                             "ioc_timeout_id = 0x%x", timeout_id);
338                         if (untimeout(timeout_id) == -1) {
339                                 IBTF_DPRINTF_L2("ibdm", "free_iou_info: "
340                                     "untimeout ioc_timeout_id failed");
341                                 mutex_enter(&gid_info->gl_mutex);
342                                 return (-1);
343                         }
344                         mutex_enter(&gid_info->gl_mutex);
345                 }

347                 /* handle the case where an ioc_dc_timeout_id is scheduled */
348                 if (ioc->ioc_dc_timeout_id) {
349                         timeout_id = ioc->ioc_dc_timeout_id;
350                         ioc->ioc_dc_timeout_id = 0;
351                         mutex_exit(&gid_info->gl_mutex);
352                         IBTF_DPRINTF_L5("ibdm", "free_iou_info: "
353                             "ioc_dc_timeout_id = 0x%x", timeout_id);
354                         if (untimeout(timeout_id) == -1) {
355                                 IBTF_DPRINTF_L2("ibdm", "free_iou_info: "
356                                     "untimeout ioc_dc_timeout_id failed");
357                                 mutex_enter(&gid_info->gl_mutex);
358                                 return (-1);
359                         }
360                         mutex_enter(&gid_info->gl_mutex);
361                 }

363                 /* handle the case where serv[k].se_timeout_id is scheduled */
364                 for (k = 0; k < ioc->ioc_profile.ioc_service_entries; k++) {
365                         if (ioc->ioc_serv[k].se_timeout_id) {
366                                 timeout_id = ioc->ioc_serv[k].se_timeout_id;
367                                 ioc->ioc_serv[k].se_timeout_id = 0;
368                                 mutex_exit(&gid_info->gl_mutex);
369                                 IBTF_DPRINTF_L5("ibdm", "free_iou_info: "
370                                     "ioc->ioc_serv[%d].se_timeout_id = 0x%x",
371                                     k, timeout_id);
372                                 if (untimeout(timeout_id) == -1) {
373                                         IBTF_DPRINTF_L2("ibdm", "free_iou_info:"
374                                             " untimeout se_timeout_id failed");
375                                         mutex_enter(&gid_info->gl_mutex);
376                                         return (-1);
377                                 }
378                                 mutex_enter(&gid_info->gl_mutex);
379                         }
380                 }

382                 /* delete GID list in IOC */
383                 head = ioc->ioc_gid_list;
384                 while (head) {
385                         IBTF_DPRINTF_L4("ibdm", "\tibdm_free_iou_info: "
386                             "Deleting gid_list struct %p", head);
387                         delete = head;
388                         head = head->gid_next;
389                         kmem_free(delete, sizeof (ibdm_gid_t));
390                 }
391                 ioc->ioc_gid_list = NULL;
```

```
393                 /* delete ioc_serv */
394                 size = ioc->ioc_profile.ioc_service_entries *
395                     sizeof (ibdm_srvents_info_t);
396                 if (ioc->ioc_serv && size) {
397                         kmem_free(ioc->ioc_serv, size);
398                         ioc->ioc_serv = NULL;
399                 }
400         }
401         /*
402          * Clear the IBDM_CISCO_PROBE_DONE flag to get the IO Unit information
403          * via the switch during the probe process.
404          */
405         gid_info->gl_flag &= ~IBDM_CISCO_PROBE_DONE;

407         IBTF_DPRINTF_L4("ibdm", "\tibdm_free_iou_info: deleting IOU & IOC");
408         size = sizeof (ibdm_iou_info_t) + niocs * sizeof (ibdm_ioc_info_t);
409         kmem_free(gl_iou, size);
410         *ioup = NULL;
411         return (0);
412 }


415 /*
416  * ibdm_fini():
417  *      Un-register with IBTF
418  *      De allocate memory for the GID info
419  */
420 static int
421 ibdm_fini()
422 {
423         int                     ii;
424         ibdm_hca_list_t         *hca_list, *temp;
425         ibdm_dp_gidinfo_t       *gid_info, *tmp;
426         ibdm_gid_t              *head, *delete;

428         IBTF_DPRINTF_L4("ibdm", "\tibdm_fini");

430         mutex_enter(&ibdm.ibdm_hl_mutex);
431         if (ibdm.ibdm_state & IBDM_IBT_ATTACHED) {
432                 if (ibt_detach(ibdm.ibdm_ibt_clnt_hdl) != IBT_SUCCESS) {
433                         IBTF_DPRINTF_L2("ibdm", "\t_fini: ibt_detach failed");
434                         mutex_exit(&ibdm.ibdm_hl_mutex);
435                         return (IBDM_FAILURE);
436                 }
437                 ibdm.ibdm_state &= ~IBDM_IBT_ATTACHED;
438                 ibdm.ibdm_ibt_clnt_hdl = NULL;
439         }

441         hca_list = ibdm.ibdm_hca_list_head;
442         IBTF_DPRINTF_L4("ibdm", "\tibdm_fini: nhcas %d", ibdm.ibdm_hca_count);
443         for (ii = 0; ii < ibdm.ibdm_hca_count; ii++) {
444                 temp = hca_list;
445                 hca_list = hca_list->hl_next;
446                 IBTF_DPRINTF_L4("ibdm", "\tibdm_fini: hca %p", temp);
447                 if (ibdm_uninit_hca(temp) != IBDM_SUCCESS) {
448                         IBTF_DPRINTF_L2("ibdm", "\tibdm_fini: "
449                             "uninit_hca %p failed", temp);
450                         mutex_exit(&ibdm.ibdm_hl_mutex);
451                         return (IBDM_FAILURE);
452                 }
453         }
454         mutex_exit(&ibdm.ibdm_hl_mutex);

456         mutex_enter(&ibdm.ibdm_mutex);
457         if (ibdm.ibdm_state & IBDM_HCA_ATTACHED)
```



```
458                 ibdm.ibdm_state &= ~IBDM_HCA_ATTACHED;

460         gid_info = ibdm.ibdm_dp_gidlist_head;
461         while (gid_info) {
462                 mutex_enter(&gid_info->gl_mutex);
463                 (void) ibdm_free_iou_info(gid_info, &gid_info->gl_iou);
464                 mutex_exit(&gid_info->gl_mutex);
465                 ibdm_delete_glhca_list(gid_info);

467                 tmp = gid_info;
468                 gid_info = gid_info->gl_next;
469                 mutex_destroy(&tmp->gl_mutex);
470                 head = tmp->gl_gid;
471                 while (head) {
472                         IBTF_DPRINTF_L4("ibdm",
473                             "\tibdm_fini: Deleting gid structs");
474                         delete = head;
475                         head = head->gid_next;
476                         kmem_free(delete, sizeof (ibdm_gid_t));
477                 }
478                 kmem_free(tmp, sizeof (ibdm_dp_gidinfo_t));
479         }
480         mutex_exit(&ibdm.ibdm_mutex);

482         if (ibdm.ibdm_state & IBDM_LOCKS_ALLOCED) {
483                 ibdm.ibdm_state &= ~IBDM_LOCKS_ALLOCED;
484                 mutex_destroy(&ibdm.ibdm_mutex);
485                 mutex_destroy(&ibdm.ibdm_hl_mutex);
486                 mutex_destroy(&ibdm.ibdm_ibnex_mutex);
487                 cv_destroy(&ibdm.ibdm_port_settle_cv);
488         }
489         if (ibdm.ibdm_state & IBDM_CVS_ALLOCED) {
490                 ibdm.ibdm_state &= ~IBDM_CVS_ALLOCED;
491                 cv_destroy(&ibdm.ibdm_probe_cv);
492                 cv_destroy(&ibdm.ibdm_busy_cv);
493         }
494         return (IBDM_SUCCESS);
495 }


498 /*
499  * ibdm_event_hdlr()
500  *
501  *      IBDM registers  this asynchronous event handler at the time of
502  *      ibt_attach. IBDM support the following async events. For other
503  *      event, simply returns success.
504  *      IBT_HCA_ATTACH_EVENT:
505  *              Retrieves the  information about all the port that are
506  *              present on this HCA,  allocates  the  port  attributes
507  *              structure  and calls IB  nexus  callback  routine with
508  *              the port attributes structure as an input argument.
509  *      IBT_HCA_DETACH_EVENT:
510  *              Retrieves the information about all the ports that are
511  *              present on  this HCA and  calls IB nexus callback with
512  *              port guid as an argument
513  *      IBT_EVENT_PORT_UP:
514  *              Register with IBMF and SA access
515  *              Setup IBMF receive callback routine
516  *      IBT_EVENT_PORT_DOWN:
517  *              Un-Register with IBMF and SA access
518  *              Teardown IBMF receive callback routine
519  */
520 /*ARGSUSED*/
521 static void
522 ibdm_event_hdlr(void *clnt_hdl,
523     ibt_hca_hdl_t hca_hdl, ibt_async_code_t code, ibt_async_event_t *event)
```

```
524 {
525         ibdm_hca_list_t         *hca_list;
526         ibdm_port_attr_t        *port;
527         ibmf_saa_handle_t       port_sa_hdl;

529         IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: async code 0x%x", code);

531         switch (code) {
532         case IBT_HCA_ATTACH_EVENT:      /* New HCA registered with IBTF */
533                 ibdm_handle_hca_attach(event->ev_hca_guid);
534                 break;

536         case IBT_HCA_DETACH_EVENT:      /* HCA unregistered with IBTF */
537                 ibdm_handle_hca_detach(event->ev_hca_guid);
538                 mutex_enter(&ibdm.ibdm_ibnex_mutex);
539                 if (ibdm.ibdm_ibnex_callback != NULL) {
540                         (*ibdm.ibdm_ibnex_callback)((void *)
541                             &event->ev_hca_guid, IBDM_EVENT_HCA_REMOVED);
542                 }
543                 mutex_exit(&ibdm.ibdm_ibnex_mutex);
544                 break;

546         case IBT_EVENT_PORT_UP:
547                 IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: PORT_UP");
548                 mutex_enter(&ibdm.ibdm_hl_mutex);
549                 port = ibdm_get_port_attr(event, &hca_list);
550                 if (port == NULL) {
551                         IBTF_DPRINTF_L2("ibdm",
552                             "\tevent_hdlr: HCA not present");
553                         mutex_exit(&ibdm.ibdm_hl_mutex);
554                         break;
555                 }
556                 ibdm_initialize_port(port);
557                 hca_list->hl_nports_active++;
558                 cv_broadcast(&ibdm.ibdm_port_settle_cv);
559                 mutex_exit(&ibdm.ibdm_hl_mutex);

561                 /* Inform IB nexus driver */
562                 mutex_enter(&ibdm.ibdm_ibnex_mutex);
563                 if (ibdm.ibdm_ibnex_callback != NULL) {
564                         (*ibdm.ibdm_ibnex_callback)((void *)
565                             &event->ev_hca_guid, IBDM_EVENT_PORT_UP);
566                 }
567                 mutex_exit(&ibdm.ibdm_ibnex_mutex);
568                 break;

570         case IBT_ERROR_PORT_DOWN:
571                 IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: PORT_DOWN");
572                 mutex_enter(&ibdm.ibdm_hl_mutex);
573                 port = ibdm_get_port_attr(event, &hca_list);
574                 if (port == NULL) {
575                         IBTF_DPRINTF_L2("ibdm",
576                             "\tevent_hdlr: HCA not present");
577                         mutex_exit(&ibdm.ibdm_hl_mutex);
578                         break;
579                 }
580                 hca_list->hl_nports_active--;
581                 port_sa_hdl = port->pa_sa_hdl;
582                 (void) ibdm_fini_port(port);
583                 port->pa_state = IBT_PORT_DOWN;
584                 cv_broadcast(&ibdm.ibdm_port_settle_cv);
585                 mutex_exit(&ibdm.ibdm_hl_mutex);
586                 ibdm_reset_all_dgids(port_sa_hdl);
587                 break;

589         case IBT_PORT_CHANGE_EVENT:
```



```
590                 IBTF_DPRINTF_L4("ibdm", "\tevent_hdlr: PORT_CHANGE");
591                 if (event->ev_port_flags & IBT_PORT_CHANGE_PKEY)
592                         ibdm_handle_port_change_event(event);
593                 break;

595         default:                /* Ignore all other events/errors */
596                 break;
597         }
598 }

600 static void
601 ibdm_handle_port_change_event(ibt_async_event_t *event)
602 {
603         ibdm_port_attr_t        *port;
604         ibdm_hca_list_t         *hca_list;

606         IBTF_DPRINTF_L2("ibdm", "\tibdm_handle_port_change_event:"
607             " HCA guid  %llx", event->ev_hca_guid);
608         mutex_enter(&ibdm.ibdm_hl_mutex);
609         port = ibdm_get_port_attr(event, &hca_list);
610         if (port == NULL) {
611                 IBTF_DPRINTF_L2("ibdm", "\tevent_hdlr: HCA not present");
612                 mutex_exit(&ibdm.ibdm_hl_mutex);
613                 return;
614         }
615         ibdm_update_port_pkeys(port);
616         cv_broadcast(&ibdm.ibdm_port_settle_cv);
617         mutex_exit(&ibdm.ibdm_hl_mutex);

619         /* Inform IB nexus driver */
620         mutex_enter(&ibdm.ibdm_ibnex_mutex);
621         if (ibdm.ibdm_ibnex_callback != NULL) {
622                 (*ibdm.ibdm_ibnex_callback)((void *)
623                     &event->ev_hca_guid, IBDM_EVENT_PORT_PKEY_CHANGE);
624         }
625         mutex_exit(&ibdm.ibdm_ibnex_mutex);
626 }

628 /*
629  * ibdm_update_port_pkeys()
630  *      Update the pkey table
631  *      Update the port attributes
632  */
633 static void
634 ibdm_update_port_pkeys(ibdm_port_attr_t *port)
635 {
636         uint_t                          nports, size;
637         uint_t                          pkey_idx, opkey_idx;
638         uint16_t                        npkeys;
639         ibt_hca_portinfo_t              *pinfop;
640         ib_pkey_t                       pkey;
641         ibdm_pkey_tbl_t                 *pkey_tbl;
642         ibdm_port_attr_t                newport;

644         IBTF_DPRINTF_L4("ibdm", "\tupdate_port_pkeys:");
645         ASSERT(MUTEX_HELD(&ibdm.ibdm_hl_mutex));

647         /* Check whether the port is active */
648         if (ibt_get_port_state(port->pa_hca_hdl, port->pa_port_num, NULL,
649             NULL) != IBT_SUCCESS)
650                 return;

652         if (ibt_query_hca_ports(port->pa_hca_hdl, port->pa_port_num,
653             &pinfop, &nports, &size) != IBT_SUCCESS) {
654                 /* This should not occur */
655                 port->pa_npkeys = 0;
```

```
656                 port->pa_pkey_tbl = NULL;
657                 return;
658         }

660         npkeys = pinfop->p_pkey_tbl_sz;
661         pkey_tbl = kmem_zalloc(npkeys * sizeof (ibdm_pkey_tbl_t), KM_SLEEP);
662         newport.pa_pkey_tbl = pkey_tbl;
663         newport.pa_ibmf_hdl = port->pa_ibmf_hdl;

665         for (pkey_idx = 0; pkey_idx < npkeys; pkey_idx++) {
666                 pkey = pkey_tbl[pkey_idx].pt_pkey =
667                     pinfop->p_pkey_tbl[pkey_idx];
668                 /*
669                  * Is this pkey present in the current table ?
670                  */
671                 for (opkey_idx = 0; opkey_idx < port->pa_npkeys; opkey_idx++) {
672                         if (pkey == port->pa_pkey_tbl[opkey_idx].pt_pkey) {
673                                 pkey_tbl[pkey_idx].pt_qp_hdl =
674                                     port->pa_pkey_tbl[opkey_idx].pt_qp_hdl;
675                                 port->pa_pkey_tbl[opkey_idx].pt_qp_hdl = NULL;
676                                 break;
677                         }
678                 }

680                 if (opkey_idx == port->pa_npkeys) {
681                         pkey = pkey_tbl[pkey_idx].pt_pkey;
682                         if (IBDM_INVALID_PKEY(pkey)) {
683                                 pkey_tbl[pkey_idx].pt_qp_hdl = NULL;
684                                 continue;
685                         }
686                         ibdm_port_attr_ibmf_init(&newport, pkey, pkey_idx);
687                 }
688         }

690         for (opkey_idx = 0; opkey_idx < port->pa_npkeys; opkey_idx++) {
691                 if (port->pa_pkey_tbl[opkey_idx].pt_qp_hdl != NULL) {
692                         if (ibdm_port_attr_ibmf_fini(port, opkey_idx) !=
693                             IBDM_SUCCESS) {
694                                 IBTF_DPRINTF_L2("ibdm", "\tupdate_port_pkeys: "
695                                     "ibdm_port_attr_ibmf_fini failed for "
696                                     "port pkey 0x%x",
697                                     port->pa_pkey_tbl[opkey_idx].pt_pkey);
698                         }
699                 }
700         }

702         if (port->pa_pkey_tbl != NULL) {
703                 kmem_free(port->pa_pkey_tbl,
704                     port->pa_npkeys * sizeof (ibdm_pkey_tbl_t));
705         }

707         port->pa_npkeys = npkeys;
708         port->pa_pkey_tbl = pkey_tbl;
709         port->pa_sn_prefix = pinfop->p_sgid_tbl[0].gid_prefix;
710         port->pa_state = pinfop->p_linkstate;
711         ibt_free_portinfo(pinfop, size);
712 }

714 /*
715  * ibdm_initialize_port()
716  *      Register with IBMF
717  *      Register with SA access
718  *      Register a receive callback routine with IBMF. IBMF invokes
719  *      this routine whenever a MAD arrives at this port.
720  *      Update the port attributes
721  */
```



```
722 static void
723 ibdm_initialize_port(ibdm_port_attr_t *port)
724 {
725         int                             ii;
726         uint_t                          nports, size;
727         uint_t                          pkey_idx;
728         ib_pkey_t                       pkey;
729         ibt_hca_portinfo_t              *pinfop;
730         ibmf_register_info_t            ibmf_reg;
731         ibmf_saa_subnet_event_args_t    event_args;

733         IBTF_DPRINTF_L4("ibdm", "\tinitialize_port:");
734         ASSERT(MUTEX_HELD(&ibdm.ibdm_hl_mutex));

736         /* Check whether the port is active */
737         if (ibt_get_port_state(port->pa_hca_hdl, port->pa_port_num, NULL,
738             NULL) != IBT_SUCCESS)
739                 return;

741         if (port->pa_sa_hdl != NULL || port->pa_pkey_tbl != NULL)
742                 return;

744         if (ibt_query_hca_ports(port->pa_hca_hdl, port->pa_port_num,
745             &pinfop, &nports, &size) != IBT_SUCCESS) {
746                 /* This should not occur */
747                 port->pa_npkeys         = 0;
748                 port->pa_pkey_tbl       = NULL;
749                 return;
750         }
751         port->pa_sn_prefix = pinfop->p_sgid_tbl[0].gid_prefix;

753         port->pa_state          = pinfop->p_linkstate;
754         port->pa_npkeys         = pinfop->p_pkey_tbl_sz;
755         port->pa_pkey_tbl       = (ibdm_pkey_tbl_t *)kmem_zalloc(
756             port->pa_npkeys * sizeof (ibdm_pkey_tbl_t), KM_SLEEP);

758         for (pkey_idx = 0; pkey_idx < port->pa_npkeys; pkey_idx++)
759                 port->pa_pkey_tbl[pkey_idx].pt_pkey =
760                     pinfop->p_pkey_tbl[pkey_idx];

762         ibt_free_portinfo(pinfop, size);

764         if (ibdm_enumerate_iocs) {
765                 event_args.is_event_callback = ibdm_saa_event_cb;
766                 event_args.is_event_callback_arg = port;
767                 if (ibmf_sa_session_open(port->pa_port_guid, 0, &event_args,
768                     IBMF_VERSION, 0, &port->pa_sa_hdl) != IBMF_SUCCESS) {
769                         IBTF_DPRINTF_L2("ibdm", "\tinitialize_port: "
770                             "sa access registration failed");
771                         (void) ibdm_fini_port(port);
772                         return;
773                 }

775                 ibmf_reg.ir_ci_guid             = port->pa_hca_guid;
776                 ibmf_reg.ir_port_num            = port->pa_port_num;
777                 ibmf_reg.ir_client_class        = DEV_MGT_MANAGER;

779                 if (ibmf_register(&ibmf_reg, IBMF_VERSION, 0, NULL, NULL,
780                     &port->pa_ibmf_hdl, &port->pa_ibmf_caps) != IBMF_SUCCESS) {
781                         IBTF_DPRINTF_L2("ibdm", "\tinitialize_port: "
782                             "IBMF registration failed");
783                         (void) ibdm_fini_port(port);
784                         return;
785                 }

787                 if (ibmf_setup_async_cb(port->pa_ibmf_hdl,
```

```
788                     IBMF_QP_HANDLE_DEFAULT,
789                     ibdm_ibmf_recv_cb, 0, 0) != IBMF_SUCCESS) {
790                         IBTF_DPRINTF_L2("ibdm", "\tinitialize_port: "
791                             "IBMF setup recv cb failed");
792                         (void) ibdm_fini_port(port);
793                         return;
794                 }
795         } else {
796                 port->pa_sa_hdl = NULL;
797                 port->pa_ibmf_hdl = NULL;
798         }
799
800         for (ii = 0; ii < port->pa_npkeys; ii++) {
801                 pkey = port->pa_pkey_tbl[ii].pt_pkey;
802                 if (IBDM_INVALID_PKEY(pkey)) {
803                         port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
804                         continue;
805                 }
806                 ibdm_port_attr_ibmf_init(port, pkey, ii);
807         }
808 }


811 /*
812  * ibdm_port_attr_ibmf_init:
813  *      With IBMF - Alloc QP Handle and Setup Async callback
814  */
815 static void
816 ibdm_port_attr_ibmf_init(ibdm_port_attr_t *port, ib_pkey_t pkey, int ii)
817 {
818         int ret;
819
820         if (ibdm_enumerate_iocs == 0) {
821                 port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
822                 return;
823         }
824
825         if ((ret = ibmf_alloc_qp(port->pa_ibmf_hdl, pkey, IB_GSI_QKEY,
826             IBMF_ALT_QP_MAD_NO_RMPP, &port->pa_pkey_tbl[ii].pt_qp_hdl)) !=
827             IBMF_SUCCESS) {
828                 IBTF_DPRINTF_L2("ibdm", "\tport_attr_ibmf_init: "
829                     "IBMF failed to alloc qp %d", ret);
830                 port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
831                 return;
832         }
833
834         IBTF_DPRINTF_L4("ibdm", "\tport_attr_ibmf_init: QP handle is %p",
835             port->pa_ibmf_hdl);
836
837         if ((ret = ibmf_setup_async_cb(port->pa_ibmf_hdl,
838             port->pa_pkey_tbl[ii].pt_qp_hdl, ibdm_ibmf_recv_cb, 0, 0)) !=
839             IBMF_SUCCESS) {
840                 IBTF_DPRINTF_L2("ibdm", "\tport_attr_ibmf_init: "
841                     "IBMF setup recv cb failed %d", ret);
842                 (void) ibmf_free_qp(port->pa_ibmf_hdl,
843                     &port->pa_pkey_tbl[ii].pt_qp_hdl, 0);
844                 port->pa_pkey_tbl[ii].pt_qp_hdl = NULL;
845         }
846 }


849 /*
850  * ibdm_get_port_attr()
851  *      Get port attributes from HCA guid and port number
852  *      Return pointer to ibdm_port_attr_t on Success
853  *      and NULL on failure
```



```
854  */
855 static ibdm_port_attr_t *
856 ibdm_get_port_attr(ibt_async_event_t *event, ibdm_hca_list_t **retval)
857 {
858         ibdm_hca_list_t         *hca_list;
859         ibdm_port_attr_t        *port_attr;
860         int                     ii;
861
862         IBTF_DPRINTF_L4("ibdm", "\tget_port_attr: port# %d", event->ev_port);
863         ASSERT(MUTEX_HELD(&ibdm.ibdm_hl_mutex));
864         hca_list = ibdm.ibdm_hca_list_head;
865         while (hca_list) {
866                 if (hca_list->hl_hca_guid == event->ev_hca_guid) {
867                         for (ii = 0; ii < hca_list->hl_nports; ii++) {
868                                 port_attr = &hca_list->hl_port_attr[ii];
869                                 if (port_attr->pa_port_num == event->ev_port) {
870                                         *retval = hca_list;
871                                         return (port_attr);
872                                 }
873                         }
874                 }
875                 hca_list = hca_list->hl_next;
876         }
877         return (NULL);
878 }


881 /*
882  * ibdm_update_port_attr()
883  *      Update the port attributes
884  */
885 static void
886 ibdm_update_port_attr(ibdm_port_attr_t *port)
887 {
888         uint_t                  nports, size;
889         uint_t                  pkey_idx;
890         ibt_hca_portinfo_t      *portinfop;
891
892         IBTF_DPRINTF_L4("ibdm", "\tupdate_port_attr: Begin");
893         if (ibt_query_hca_ports(port->pa_hca_hdl,
894             port->pa_port_num, &portinfop, &nports, &size) != IBT_SUCCESS) {
895                 /* This should not occur */
896                 port->pa_npkeys         = 0;
897                 port->pa_pkey_tbl       = NULL;
898                 return;
899         }
900         port->pa_sn_prefix = portinfop->p_sgid_tbl[0].gid_prefix;
901
902         port->pa_state          = portinfop->p_linkstate;
903
904         /*
905          * PKey information in portinfo valid only if port is
906          * ACTIVE. Bail out if not.
907          */
908         if (port->pa_state != IBT_PORT_ACTIVE) {
909                 port->pa_npkeys         = 0;
910                 port->pa_pkey_tbl       = NULL;
911                 ibt_free_portinfo(portinfop, size);
912                 return;
913         }
914
915         port->pa_npkeys         = portinfop->p_pkey_tbl_sz;
916         port->pa_pkey_tbl       = (ibdm_pkey_tbl_t *)kmem_zalloc(
917             port->pa_npkeys * sizeof (ibdm_pkey_tbl_t), KM_SLEEP);
918
919         for (pkey_idx = 0; pkey_idx < port->pa_npkeys; pkey_idx++) {
```

```
920                 port->pa_pkey_tbl[pkey_idx].pt_pkey =
921                     portinfop->p_pkey_tbl[pkey_idx];
922         }
923         ibt_free_portinfo(portinfop, size);
924 }


927 /*
928  * ibdm_handle_hca_attach()
929  */
930 static void
931 ibdm_handle_hca_attach(ib_guid_t hca_guid)
932 {
933         uint_t                  size;
934         uint_t                  ii, nports;
935         ibt_status_t            status;
936         ibt_hca_hdl_t           hca_hdl;
937         ibt_hca_attr_t          *hca_attr;
938         ibdm_hca_list_t         *hca_list, *temp;
939         ibdm_port_attr_t        *port_attr;
940         ibt_hca_portinfo_t      *portinfop;

942         IBTF_DPRINTF_L4("ibdm",
943             "\thandle_hca_attach: hca_guid = 0x%llX", hca_guid);

945         /* open the HCA first */
946         if ((status = ibt_open_hca(ibdm.ibdm_ibt_clnt_hdl, hca_guid,
947             &hca_hdl)) != IBT_SUCCESS) {
948                 IBTF_DPRINTF_L2("ibdm", "\thandle_hca_attach: "
949                     "open_hca failed, status 0x%x", status);
950                 return;
951         }

953         hca_attr = (ibt_hca_attr_t *)
954             kmem_alloc(sizeof (ibt_hca_attr_t), KM_SLEEP);
955         /* ibt_query_hca always returns IBT_SUCCESS */
956         (void) ibt_query_hca(hca_hdl, hca_attr);

958         IBTF_DPRINTF_L4("ibdm", "\tvid: 0x%x, pid: 0x%x, ver: 0x%x,"
959             " #ports: %d", hca_attr->hca_vendor_id, hca_attr->hca_device_id,
960             hca_attr->hca_version_id, hca_attr->hca_nports);

962         if ((status = ibt_query_hca_ports(hca_hdl, 0, &portinfop, &nports,
963             &size)) != IBT_SUCCESS) {
964                 IBTF_DPRINTF_L2("ibdm", "\thandle_hca_attach: "
965                     "ibt_query_hca_ports failed, status 0x%x", status);
966                 kmem_free(hca_attr, sizeof (ibt_hca_attr_t));
967                 (void) ibt_close_hca(hca_hdl);
968                 return;
969         }
970         hca_list = (ibdm_hca_list_t *)
971             kmem_zalloc((sizeof (ibdm_hca_list_t)), KM_SLEEP);
972         hca_list->hl_port_attr = (ibdm_port_attr_t *)kmem_zalloc(
973             (sizeof (ibdm_port_attr_t) * hca_attr->hca_nports), KM_SLEEP);
974         hca_list->hl_hca_guid = hca_attr->hca_node_guid;
975         hca_list->hl_nports = hca_attr->hca_nports;
976         hca_list->hl_attach_time = gethrtime();
 39         hca_list->hl_attach_time = ddi_get_time();
977         hca_list->hl_hca_hdl = hca_hdl;

979         /*
980          * Init a dummy port attribute for the HCA node
981          * This is for Per-HCA Node. Initialize port_attr :
982          *      hca_guid & port_guid -> hca_guid
983          *      npkeys, pkey_tbl is NULL
984          *      port_num, sn_prefix is 0
```



```
985          *      vendorid, product_id, dev_version from HCA
986          *      pa_state is IBT_PORT_ACTIVE
987          */
988         hca_list->hl_hca_port_attr = (ibdm_port_attr_t *)kmem_zalloc(
989             sizeof (ibdm_port_attr_t), KM_SLEEP);
990         port_attr = hca_list->hl_hca_port_attr;
991         port_attr->pa_vendorid  = hca_attr->hca_vendor_id;
992         port_attr->pa_productid = hca_attr->hca_device_id;
993         port_attr->pa_dev_version = hca_attr->hca_version_id;
994         port_attr->pa_hca_guid  = hca_attr->hca_node_guid;
995         port_attr->pa_hca_hdl   = hca_list->hl_hca_hdl;
996         port_attr->pa_port_guid = hca_attr->hca_node_guid;
997         port_attr->pa_state     = IBT_PORT_ACTIVE;


1000         for (ii = 0; ii < nports; ii++) {
1001                 port_attr               = &hca_list->hl_port_attr[ii];
1002                 port_attr->pa_vendorid  = hca_attr->hca_vendor_id;
1003                 port_attr->pa_productid = hca_attr->hca_device_id;
1004                 port_attr->pa_dev_version = hca_attr->hca_version_id;
1005                 port_attr->pa_hca_guid  = hca_attr->hca_node_guid;
1006                 port_attr->pa_hca_hdl   = hca_list->hl_hca_hdl;
1007                 port_attr->pa_port_guid = portinfop[ii].p_sgid_tbl->gid_guid;
1008                 port_attr->pa_sn_prefix = portinfop[ii].p_sgid_tbl->gid_prefix;
1009                 port_attr->pa_port_num  = portinfop[ii].p_port_num;
1010                 port_attr->pa_state     = portinfop[ii].p_linkstate;

1012                 /*
1013                  * Register with IBMF, SA access when the port is in
1014                  * ACTIVE state. Also register a callback routine
1015                  * with IBMF to receive incoming DM MAD's.
1016                  * The IBDM event handler takes care of registration of
1017                  * port which are not active.
1018                  */
1019                 IBTF_DPRINTF_L4("ibdm",
1020                     "\thandle_hca_attach: port guid %llx Port state 0x%x",
1021                     port_attr->pa_port_guid, portinfop[ii].p_linkstate);

1023                 if (portinfop[ii].p_linkstate == IBT_PORT_ACTIVE) {
1024                         mutex_enter(&ibdm.ibdm_hl_mutex);
1025                         hca_list->hl_nports_active++;
1026                         ibdm_initialize_port(port_attr);
1027                         cv_broadcast(&ibdm.ibdm_port_settle_cv);
1028                         mutex_exit(&ibdm.ibdm_hl_mutex);
1029                 }
1030         }
1031         mutex_enter(&ibdm.ibdm_hl_mutex);
1032         for (temp = ibdm.ibdm_hca_list_head; temp; temp = temp->hl_next) {
1033                 if (temp->hl_hca_guid == hca_guid) {
1034                         IBTF_DPRINTF_L2("ibdm", "hca_attach: HCA %llX "
1035                             "already seen by IBDM", hca_guid);
1036                         mutex_exit(&ibdm.ibdm_hl_mutex);
1037                         (void) ibdm_uninit_hca(hca_list);
1038                         return;
1039                 }
1040         }
1041         ibdm.ibdm_hca_count++;
1042         if (ibdm.ibdm_hca_list_head == NULL) {
1043                 ibdm.ibdm_hca_list_head = hca_list;
1044                 ibdm.ibdm_hca_list_tail = hca_list;
1045         } else {
1046                 ibdm.ibdm_hca_list_tail->hl_next = hca_list;
1047                 ibdm.ibdm_hca_list_tail = hca_list;
1048         }
1049         mutex_exit(&ibdm.ibdm_hl_mutex);
1050         mutex_enter(&ibdm.ibdm_ibnex_mutex);
```

```
1051         if (ibdm.ibdm_ibnex_callback != NULL) {
1052                 (*ibdm.ibdm_ibnex_callback)((void *)
1053                     &hca_guid, IBDM_EVENT_HCA_ADDED);
1054         }
1055         mutex_exit(&ibdm.ibdm_ibnex_mutex);

1057         kmem_free(hca_attr, sizeof (ibt_hca_attr_t));
1058         ibt_free_portinfo(portinfop, size);
1059 }
______unchanged_portion_omitted_

4692 /*
4693  * ibdm_get_waittime()
4694  *      Calculates the wait time based on the last HCA attach time
4695  */
4696 static clock_t
4697 ibdm_get_waittime(ib_guid_t hca_guid, time_t dft_wait_sec)
3759 static time_t
3760 ibdm_get_waittime(ib_guid_t hca_guid, int dft_wait)
4698 {
4699         const hrtime_t  dft_wait = dft_wait_sec * NANOSEC;
4700         hrtime_t        temp, wait_time = 0;
4701         clock_t         usecs;
4702         int             i;
3762         int             ii;
3763         time_t          temp, wait_time = 0;
4703         ibdm_hca_list_t *hca;

4705         IBTF_DPRINTF_L4("ibdm", "\tget_waittime hcaguid:%llx"
4706             "\tport settling time %d", hca_guid, dft_wait);

4708         ASSERT(mutex_owned(&ibdm.ibdm_hl_mutex));

4710         hca = ibdm.ibdm_hca_list_head;

4712         for (i = 0; i < ibdm.ibdm_hca_count; i++, hca = hca->hl_next) {
4713                 if (hca->hl_nports == hca->hl_nports_active)
4714                         continue;

4716                 if (hca_guid && (hca_guid != hca->hl_hca_guid))
4717                         continue;

4719                 temp = gethrtime() - hca->hl_attach_time;
4720                 temp = MAX(0, (dft_wait - temp));

4722 #endif /* ! codereview */
4723                 if (hca_guid) {
4724                         wait_time = temp;
3773                 for (ii = 0; ii < ibdm.ibdm_hca_count; ii++) {
3774                         if ((hca_guid == hca->hl_hca_guid) &&
3775                             (hca->hl_nports != hca->hl_nports_active)) {
3776                                 wait_time =
3777                                     ddi_get_time() - hca->hl_attach_time;
3778                                 wait_time = ((wait_time >= dft_wait) ?
3779                                     0 : (dft_wait - wait_time));
4725                         break;
4726                 }

4728                 wait_time = MAX(temp, wait_time);
3782                         hca = hca->hl_next;
3783                 }
3784                 IBTF_DPRINTF_L2("ibdm", "\tget_waittime: wait_time = %ld secs",
3785                     (long)wait_time);
3786                 return (wait_time);
4729         }
```



```
4731         /* convert to microseconds */
4732         usecs = MIN(wait_time, dft_wait) / (NANOSEC / MICROSEC);

4734         IBTF_DPRINTF_L2("ibdm", "\tget_waittime: wait_time = %ld usecs",
4735             (long) usecs);

4737         return (drv_usectohz(usecs));
3789         for (ii = 0; ii < ibdm.ibdm_hca_count; ii++) {
3790                 if (hca->hl_nports != hca->hl_nports_active) {
3791                         temp = ddi_get_time() - hca->hl_attach_time;
3792                         temp = ((temp >= dft_wait) ? 0 : (dft_wait - temp));
3793                         wait_time = (temp > wait_time) ? temp : wait_time;
3794                 }
3795                 hca = hca->hl_next;
3796         }
3797         IBTF_DPRINTF_L2("ibdm", "\tget_waittime: wait_time = %ld secs",
3798             (long)wait_time);
3799         return (wait_time);
4738 }

4740 void
4741 ibdm_ibnex_port_settle_wait(ib_guid_t hca_guid, time_t dft_wait)
3803 ibdm_ibnex_port_settle_wait(ib_guid_t hca_guid, int dft_wait)
4742 {
4743         clock_t wait_time;
3805         time_t wait_time;
3806         clock_t delta;

4745         mutex_enter(&ibdm.ibdm_hl_mutex);

4747         while ((wait_time = ibdm_get_waittime(hca_guid, dft_wait)) > 0)
3810         while ((wait_time = ibdm_get_waittime(hca_guid, dft_wait)) > 0) {
3811                 if (wait_time > dft_wait) {
3812                         IBTF_DPRINTF_L1("ibdm",
3813                             "\tibnex_port_settle_wait: wait_time = %ld secs; "
3814                             "Resetting to %d secs",
3815                             (long)wait_time, dft_wait);
3816                         wait_time = dft_wait;
3817                 }
3818                 delta = drv_usectohz(wait_time * 1000000);
4748                 (void) cv_reltimedwait(&ibdm.ibdm_port_settle_cv,
4749                     &ibdm.ibdm_hl_mutex, wait_time, TR_CLOCK_TICK);
3820                     &ibdm.ibdm_hl_mutex, delta, TR_CLOCK_TICK);
3821         }

4751         mutex_exit(&ibdm.ibdm_hl_mutex);
4752 }
______unchanged_portion_omitted_
```

```
**********************************************************
   12108 Tue Apr 15 13:19:36 2014
new/usr/src/uts/common/sys/ib/mgt/ibdm/ibdm_ibnex.h
patch fix
**********************************************************
______unchanged_portion_omitted_
 207 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv", ibdm_port_attr_s))

 209 /*
 210  * HCA list structure.
 211  */
 212 typedef struct ibdm_hca_list_s {
 213         ibdm_port_attr_t        *hl_port_attr;          /* port attributes */
 214         struct ibdm_hca_list_s  *hl_next;               /* ptr to next list */
 215         ib_guid_t               hl_hca_guid;            /* HCA GUID */
 216         uint32_t                hl_nports;              /* #ports of this HCA */
 217         uint32_t                hl_nports_active;       /* #ports active */
 218         hrtime_t                hl_attach_time;         /* attach time */
 218         time_t                  hl_attach_time;         /* attach time */
 219         ibt_hca_hdl_t           hl_hca_hdl;             /* HCA handle */
 220         ibdm_port_attr_t        *hl_hca_port_attr;      /* Dummy Port Attr */
 221                                                         /* for HCA node */
 222 } ibdm_hca_list_t;
 223 _NOTE(SCHEME_PROTECTS_DATA("Serialized access by cv", ibdm_hca_list_s))

 225 /*
 226  * The DM callback definitions
 227  *
 228  * ibdm_callback_t
 229  *      Pointer to DM callback function
 230  *      IBDM notifies IB nexus of ibdm_event_t using this callback.
 231  * Arguments
 232  *      arg     : The value of "arg" depends on the "event"
 233  *                IBDM_EVENT_CREATE_HCA_NODE      (pointer to HCA GUID)
 234  *                IBDM_EVENT_REMOVE_HCA_NODE      (pointer to HCA GUID)
 235  *                IBDM_EVENT_IOC_PROP_UPDATE      (ibdm_ioc_info_t *)
 236  *
 237  *      event   : ibdm_event_t values
 238  *
 239  * Returns              : None
 240  *
 241  */
 242 typedef void (*ibdm_callback_t)(void *arg, ibdm_events_t event);


 245 /*
 246  * DM interface functions
 247  */

 249 /*
 250  * ibdm_ibnex_register_callback
 251  *      Register the IB nexus IBDM callback routine
 252  *
 253  * Arguments            : IB nexus IBDM callback routine
 254  * Return Values        : None
 255  */
 256 void            ibdm_ibnex_register_callback(ibdm_callback_t cb);

 258 /*
 259  * ibdm_ibnex_unregister_callback
 260  *      Unregister IB nexus DM callback with IBDM
 261  *
 262  * Arguments            : None
 263  * Return Values        : None
 264  */
 265 void            ibdm_ibnex_unregister_callback();
```



```
 268 /*
 269  * PORT devices handling interfaces.
 270  *
 271  * ibdm_ibnex_probe_hcaport
 272  *      Probes the HCA port. If found, returns the port attributes.
 273  *      Caller is responsible for  freeing the memory for the port
 274  *      attribute structure by calling ibdm_ibnex_free_port_attr()
 275  *
 276  * Arguments            : GUID of the HCA and port number
 277  * Return Values        : ibdm_port_attr_t on SUCCESS, NULL on FAILURE.
 278  */
 279 ibdm_port_attr_t *ibdm_ibnex_probe_hcaport(ib_guid_t, uint8_t);

 281 /*
 282  * ibdm_ibnex_get_port_attrs
 283  *      Scans the HCA ports for a matching port_guid. If found,
 284  *      returns the port attributes.
 285  *      Caller is responsible for freeing the memory for the port
 286  *      attribute structure by calling ibdm_ibnex_free_port_attr()
 287  *
 288  * Arguments            : GUID of the port
 289  * Return Values        : ibdm_port_attr_t on SUCCESS, NULL on FAILURE.
 290  */
 291 ibdm_port_attr_t *ibdm_ibnex_get_port_attrs(ib_guid_t);

 293 /*
 294  * ibdm_ibnex_free_port_attr()
 295  *      Deallocates the memory from ibnex_get_dip_from_port_guid() and
 296  *      ibdm_ibnex_get_port_attrs() functions.
 297  */
 298 void            ibdm_ibnex_free_port_attr(ibdm_port_attr_t *);


 301 /*
 302  * IOC devices handling interfaces.
 303  *
 304  * ibdm_ibnex_probe_ioc
 305  *      Probes the  IOC device on the fabric. If found, allocates and
 306  *      returns pointer to the ibdm_ioc_info_t. Caller is responsible
 307  *      to free the memory for the ioc attribute structure by calling
 308  *      ibdm_ibnex_free_ioc_list.
 309  *
 310  * Arguments            :
 311  *      GUID of the IOU and GUID of the IOC
 312  *      reprobe_flag - Set if IOC information has to be reprobed.
 313  * Return Values        : ibdm_ioc_info_t on SUCCESS, NULL on FAILURE.
 314  */
 315 ibdm_ioc_info_t *ibdm_ibnex_probe_ioc(ib_guid_t iou_guid, ib_guid_t ioc_guid,
 316     int reprobe_flag);

 318 /*
 319  * ibdm_ibnex_get_ioc_count
 320  *      Returns number of IOCs currently discovered in the fabric.
 321  * Arguments      : NONE
 322  * Return Values  : number of IOCs seen
 323  */
 324 int     ibdm_ibnex_get_ioc_count(void);

 326 /*
 327  * ibdm_ibnex_get_ioc_list
 328  *      Returns linked list of ibdm_ioc_info_t structures for all the
 329  *      IOCs  present on the fabric. Caller is responsible for freeing
 330  *      the  memory allocated for the ioc  attribute  structure(s) by
 331  *      calling ibdm_ibnex_free_ioc_list().
```

```
 332  *
 333  * Arguments      : list_flag :
 334  *              Get list according to ibdm_ibnex_get_ioclist_mtd_t defination.
 335  * Return Values  : IOC list based containing "ibdm_ioc_info_t"s if
 336  *                      successful, otherwise NULL.
 337  */
 338 ibdm_ioc_info_t *ibdm_ibnex_get_ioc_list(ibdm_ibnex_get_ioclist_mtd_t);

 340 /*
 341  * ibdm_ibnex_get_ioc_info
 342  *      Returns pointer  ibdm_ioc_info_t structures for the request
 343  *      "ioc_guid".  Caller is  responsible to  free the  memory by
 344  *      calling ibdm_ibnex_free_ioc_list() when the return value is
 345  *      not NULL.
 346  *
 347  * Arguments            : GUID of the IOC
 348  * Return Values        : Address of kmem_alloc'ed memory if the IOC exists,
 349  *                        otherwise NULL.
 350  */
 351 ibdm_ioc_info_t *ibdm_ibnex_get_ioc_info(ib_guid_t ioc_guid);

 353 /*
 354  * ibdm_ibnex_free_ioc_list()
 355  *      Deallocates the memory from ibdm_ibnex_probe_ioc(),
 356  *      ibdm_ibnex_get_ioc_list() and ibdm_ibnex_get_ioc_info()
 357  */
 358 void            ibdm_ibnex_free_ioc_list(ibdm_ioc_info_t *);

 360 /*
 361  * HCA handling interfaces.
 362  *
 363  * ibdm_ibnex_get_hca_list
 364  *      Returns linked list of ibdm_hca_list_t structures for all
 365  *      the HCAs present on the fabric. Caller is responsible for
 366  *      freeing the memory for the hca attribute structure(s) by
 367  *      calling ibdm_ibnex_free_hca_list().
 368  *
 369  * Arguments            : "hca" contains pointer to pointer of ibdm_hca_list_t
 370  *                      : "cnt" contains pointer to number of hca's
 371  * Return Values        : None
 372  */
 373 void            ibdm_ibnex_get_hca_list(ibdm_hca_list_t **hca, int *cnt);

 375 /*
 376  * ibdm_ibnex_get_hca_info_by_guid
 377  *      Returns a linked list of ibdm_hca_list_t structure that matches the
 378  *      given argument. The caller is responsible for freeing the memory for
 379  *      the hca attribute structure by calling ibdm_ibnex_free_hca_list().
 380  *
 381  * Arguments            : HCA GUID
 382  * Return Values        : Linked list of ibdm_hca_list_t(s)
 383  */
 384 ibdm_hca_list_t *ibdm_ibnex_get_hca_info_by_guid(ib_guid_t);

 386 /*
 387  * ibdm_ibnex_free_hca_list()
 388  *      Deallocates the memory from ibdm_ibnex_get_hca_list() and
 389  *      ibdm_ibnex_get_hca_info_by_guid() functions.
 390  */
 391 void            ibdm_ibnex_free_hca_list(ibdm_hca_list_t *);

 393 /*
 394  * ibdm_ibnex_update_pkey_tbls
 395  *      Updates the DM P_Key database.
 396  *
 397  * Arguments            : NONE
```



```
 398  * Return Values        : NONE
 399  */
 400 void    ibdm_ibnex_update_pkey_tbls(void);

 402 /*
 403  * ibdm_ibnex_port_settle_wait
 404  *      Wait until the ports come up
 405  *
 406  * Arguments
 407  *      HCA GUID and the maximum wait time since the hca instance attach
 408  */
 409 void    ibdm_ibnex_port_settle_wait(ib_guid_t, time_t);
 409 void    ibdm_ibnex_port_settle_wait(ib_guid_t, int);


 412 #ifdef __cplusplus
 413 }
_____unchanged_portion_omitted_
```