```
**********************************************************
  213056 Mon May  5 11:11:00 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_
```

```
**********************************************************
   12174 Mon May  5 11:11:01 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_
```