```
**********************************************************
  246182 Mon May  5 11:11:09 2014
new/usr/src/uts/common/io/scsi/adapters/scsi_vhci/scsi_vhci.c
4779 vhci shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2001, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

/*
 * Multiplexed I/O SCSI vHCI implementation
 */

#include <sys/conf.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/scsi/scsi.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/scsi/impl/services.h>
#include <sys/sunmdi.h>
#include <sys/mdi_impldefs.h>
#include <sys/scsi/adapters/scsi_vhci.h>
#include <sys/disp.h>
#include <sys/byteorder.h>

extern uintptr_t scsi_callback_id;
extern ddi_dma_attr_t scsi_alloc_attr;

#ifdef  DEBUG
int     vhci_debug = VHCI_DEBUG_DEFAULT_VAL;
#endif

/* retry for the vhci_do_prout command when a not ready is returned */
int vhci_prout_not_ready_retry = 180;

/*
 * These values are defined to support the internal retry of
 * SCSI packets for better sense code handling.
 */
#define VHCI_CMD_CMPLT  0
```



```
#define VHCI_CMD_RETRY  1
#define VHCI_CMD_ERROR  -1

#define PROPFLAGS (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM)
#define VHCI_SCSI_PERR          0x47
#define VHCI_PGR_ILLEGALOP      -2
#define VHCI_NUM_UPDATE_TASKQ   8
/* changed to 132 to accomodate HDS */

/*
 * Version Macros
 */
#define VHCI_NAME_VERSION       "SCSI VHCI Driver"
char            vhci_version_name[] = VHCI_NAME_VERSION;

int             vhci_first_time = 0;
clock_t         vhci_to_ticks = 0;
int             vhci_init_wait_timeout = VHCI_INIT_WAIT_TIMEOUT;
kcondvar_t      vhci_cv;
kmutex_t        vhci_global_mutex;
void            *vhci_softstate = NULL; /* for soft state */

/*
 * Flag to delay the retry of the reserve command
 */
int             vhci_reserve_delay = 100000;
static int      vhci_path_quiesce_timeout = 60;
static uchar_t  zero_key[MHIOC_RESV_KEY_SIZE];

/* uscsi delay for a TRAN_BUSY */
static int vhci_uscsi_delay = 100000;
static int vhci_uscsi_retry_count = 180;
/* uscsi_restart_sense timeout id in case it needs to get canceled */
static timeout_id_t vhci_restart_timeid = 0;

static int      vhci_bus_config_debug = 0;

/*
 * Bidirectional map of 'target-port' to port id <pid> for support of
 * iostat(1M) '-Xx' and '-Yx' output.
 */
static kmutex_t         vhci_targetmap_mutex;
static uint_t           vhci_targetmap_pid = 1;
static mod_hash_t       *vhci_targetmap_bypid;  /* <pid> -> 'target-port' */
static mod_hash_t       *vhci_targetmap_byport; /* 'target-port' -> <pid> */

/*
 * functions exported by scsi_vhci struct cb_ops
 */
static int vhci_open(dev_t *, int, int, cred_t *);
static int vhci_close(dev_t, int, int, cred_t *);
static int vhci_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

/*
 * functions exported by scsi_vhci struct dev_ops
 */
static int vhci_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int vhci_attach(dev_info_t *, ddi_attach_cmd_t);
static int vhci_detach(dev_info_t *, ddi_detach_cmd_t);

/*
 * functions exported by scsi_vhci scsi_hba_tran_t transport table
 */
static int vhci_scsi_tgt_init(dev_info_t *, dev_info_t *,
    scsi_hba_tran_t *, struct scsi_device *);
static void vhci_scsi_tgt_free(dev_info_t *, dev_info_t *, scsi_hba_tran_t *,
```

```
127     struct scsi_device *);
128 static int vhci_pgr_register_start(scsi_vhci_lun_t *, struct scsi_pkt *);
129 static int vhci_scsi_start(struct scsi_address *, struct scsi_pkt *);
130 static int vhci_scsi_abort(struct scsi_address *, struct scsi_pkt *);
131 static int vhci_scsi_reset(struct scsi_address *, int);
132 static int vhci_scsi_reset_target(struct scsi_address *, int level,
133     uint8_t select_path);
134 static int vhci_scsi_reset_bus(struct scsi_address *);
135 static int vhci_scsi_getcap(struct scsi_address *, char *, int);
136 static int vhci_scsi_setcap(struct scsi_address *, char *, int, int);
137 static int vhci_commoncap(struct scsi_address *, char *, int, int, int);
138 static int vhci_pHCI_cap(struct scsi_address *ap, char *cap, int val, int whom,
139     mdi_pathinfo_t *pip);
140 static struct scsi_pkt *vhci_scsi_init_pkt(struct scsi_address *,
141     struct scsi_pkt *, struct buf *, int, int, int, int, int (*)(), caddr_t);
142 static void vhci_scsi_destroy_pkt(struct scsi_address *, struct scsi_pkt *);
143 static void vhci_scsi_dmafree(struct scsi_address *, struct scsi_pkt *);
144 static void vhci_scsi_sync_pkt(struct scsi_address *, struct scsi_pkt *);
145 static int vhci_scsi_reset_notify(struct scsi_address *, int, void (*)(caddr_t),
146     caddr_t);
147 static int vhci_scsi_get_bus_addr(struct scsi_device *, char *, int);
148 static int vhci_scsi_get_name(struct scsi_device *, char *, int);
149 static int vhci_scsi_bus_power(dev_info_t *, void *, pm_bus_power_op_t,
150     void *, void *);
151 static int vhci_scsi_bus_config(dev_info_t *, uint_t, ddi_bus_config_op_t,
152     void *, dev_info_t **);
153 static int vhci_scsi_bus_unconfig(dev_info_t *, uint_t, ddi_bus_config_op_t,
154     void *);
155 static struct scsi_failover_ops *vhci_dev_fo(dev_info_t *, struct scsi_device *,
156     void **, char **);

158 /*
159  * functions registered with the mpxio framework via mdi_vhci_ops_t
160  */
161 static int vhci_pathinfo_init(dev_info_t *, mdi_pathinfo_t *, int);
162 static int vhci_pathinfo_uninit(dev_info_t *, mdi_pathinfo_t *, int);
163 static int vhci_pathinfo_state_change(dev_info_t *, mdi_pathinfo_t *,
164                 mdi_pathinfo_state_t, uint32_t, int);
165 static int vhci_pathinfo_online(dev_info_t *, mdi_pathinfo_t *, int);
166 static int vhci_pathinfo_offline(dev_info_t *, mdi_pathinfo_t *, int);
167 static int vhci_failover(dev_info_t *, dev_info_t *, int);
168 static void vhci_client_attached(dev_info_t *);
169 static int vhci_is_dev_supported(dev_info_t *, dev_info_t *, void *);

171 static int vhci_ctl(dev_t, int, intptr_t, int, cred_t *, int *);
172 static int vhci_devctl(dev_t, int, intptr_t, int, cred_t *, int *);
173 static int vhci_ioc_get_phci_path(sv_iocdata_t *, caddr_t, int, caddr_t);
174 static int vhci_ioc_get_client_path(sv_iocdata_t *, caddr_t, int, caddr_t);
175 static int vhci_ioc_get_paddr(sv_iocdata_t *, caddr_t, int, caddr_t);
176 static int vhci_ioc_send_client_path(caddr_t, sv_iocdata_t *, int, caddr_t);
177 static void vhci_ioc_devi_to_path(dev_info_t *, caddr_t);
178 static int vhci_get_phci_path_list(dev_info_t *, sv_path_info_t *, uint_t);
179 static int vhci_get_client_path_list(dev_info_t *, sv_path_info_t *, uint_t);
180 static int vhci_get_iocdata(const void *, sv_iocdata_t *, int, caddr_t);
181 static int vhci_get_iocswitchdata(const void *, sv_switch_to_cntlr_iocdata_t *,
182     int, caddr_t);
183 static int vhci_ioc_alloc_pathinfo(sv_path_info_t **, sv_path_info_t **,
184     uint_t, sv_iocdata_t *, int, caddr_t);
185 static void vhci_ioc_free_pathinfo(sv_path_info_t *, sv_path_info_t *, uint_t);
186 static int vhci_ioc_send_pathinfo(sv_path_info_t *, sv_path_info_t *, uint_t,
187     sv_iocdata_t *, int, caddr_t);
188 static int vhci_handle_ext_fo(struct scsi_pkt *, int);
189 static int vhci_efo_watch_cb(caddr_t, struct scsi_watch_result *);
190 static int vhci_quiesce_lun(struct scsi_vhci_lun *);
191 static int vhci_pgr_validate_and_register(scsi_vhci_priv_t *);
192 static void vhci_dispatch_scsi_start(void *);
```



```
193 static void vhci_efo_done(void *);
194 static void vhci_initiate_auto_failback(void *);
195 static void vhci_update_pHCI_pkt(struct vhci_pkt *, struct scsi_pkt *);
196 static int vhci_update_pathinfo(struct scsi_device *, mdi_pathinfo_t *,
197     struct scsi_failover_ops *, scsi_vhci_lun_t *, struct scsi_vhci *);
198 static void vhci_kstat_create_pathinfo(mdi_pathinfo_t *);
199 static int vhci_quiesce_paths(dev_info_t *, dev_info_t *,
200     scsi_vhci_lun_t *, char *, char *);

202 static char *vhci_devnm_to_guid(char *);
203 static int vhci_bind_transport(struct scsi_address *, struct vhci_pkt *,
204     int, int (*func)(caddr_t));
205 static void vhci_intr(struct scsi_pkt *);
206 static int vhci_do_prout(scsi_vhci_priv_t *);
207 static void vhci_run_cmd(void *);
208 static int vhci_do_prin(struct vhci_pkt **);
209 static struct scsi_pkt *vhci_create_retry_pkt(struct vhci_pkt *);
210 static struct vhci_pkt *vhci_sync_retry_pkt(struct vhci_pkt *);
211 static struct scsi_vhci_lun *vhci_lun_lookup(dev_info_t *);
212 static struct scsi_vhci_lun *vhci_lun_lookup_alloc(dev_info_t *, char *, int *);
213 static void vhci_lun_free(struct scsi_vhci_lun *dvlp, struct scsi_device *sd);
214 static int vhci_recovery_reset(scsi_vhci_lun_t *, struct scsi_address *,
215     uint8_t, uint8_t);
216 void vhci_update_pathstates(void *);

218 #ifdef DEBUG
219 static void vhci_print_prin_keys(vhci_prin_readkeys_t *, int);
220 static void vhci_print_cdb(dev_info_t *dip, uint_t level,
221     char *title, uchar_t *cdb);
222 static void vhci_clean_print(dev_info_t *dev, uint_t level,
223     char *title, uchar_t *data, int len);
224 #endif
225 static void vhci_print_prout_keys(scsi_vhci_lun_t *, char *);
226 static void vhci_uscsi_iodone(struct scsi_pkt *pkt);
227 static void vhci_invalidate_mpapi_lu(struct scsi_vhci *, scsi_vhci_lun_t *);

229 /*
230  * MP-API related functions
231  */
232 extern int vhci_mpapi_init(struct scsi_vhci *);
233 extern void vhci_mpapi_add_dev_prod(struct scsi_vhci *, char *);
234 extern int vhci_mpapi_ctl(dev_t, int, intptr_t, int, cred_t *, int *);
235 extern void vhci_update_mpapi_data(struct scsi_vhci *,
236     scsi_vhci_lun_t *, mdi_pathinfo_t *);
237 extern void* vhci_get_mpapi_item(struct scsi_vhci *, mpapi_list_header_t *,
238     uint8_t, void*);
239 extern void vhci_mpapi_set_path_state(dev_info_t *, mdi_pathinfo_t *, int);
240 extern int vhci_mpapi_update_tpg_acc_state_for_lu(struct scsi_vhci *,
241     scsi_vhci_lun_t *);

243 #define VHCI_DMA_MAX_XFER_CAP   INT_MAX

245 #define VHCI_MAX_PGR_RETRIES    3

247 /*
248  * Macros for the device-type mpxio options
249  */
250 #define LOAD_BALANCE_OPTIONS            "load-balance-options"
251 #define LOGICAL_BLOCK_REGION_SIZE       "region-size"
252 #define MPXIO_OPTIONS_LIST              "device-type-mpxio-options-list"
253 #define DEVICE_TYPE_STR                 "device-type"
254 #define isdigit(ch)                     ((ch) >= '0' && (ch) <= '9')

256 static struct cb_ops vhci_cb_ops = {
257         vhci_open,                      /* open */
258         vhci_close,                     /* close */
```

```
259         nodev,                          /* strategy */
260         nodev,                          /* print */
261         nodev,                          /* dump */
262         nodev,                          /* read */
263         nodev,                          /* write */
264         vhci_ioctl,                     /* ioctl */
265         nodev,                          /* devmap */
266         nodev,                          /* mmap */
267         nodev,                          /* segmap */
268         nochpoll,                       /* chpoll */
269         ddi_prop_op,                    /* cb_prop_op */
270         0,                              /* streamtab */
271         D_NEW | D_MP,                   /* cb_flag */
272         CB_REV,                         /* rev */
273         nodev,                          /* aread */
274         nodev                           /* awrite */
275 };

277 static struct dev_ops vhci_ops = {
278         DEVO_REV,
279         0,
280         vhci_getinfo,
281         nulldev,                /* identify */
282         nulldev,                /* probe */
283         vhci_attach,            /* attach and detach are mandatory */
284         vhci_detach,
285         nodev,                  /* reset */
286         &vhci_cb_ops,           /* cb_ops */
287         NULL,                   /* bus_ops */
288         NULL,                   /* power */
289         ddi_quiesce_not_needed, /* quiesce */
290 };

292 extern struct mod_ops mod_driverops;

294 static struct modldrv modldrv = {
295         &mod_driverops,
296         vhci_version_name,      /* module name */
297         &vhci_ops
298 };

300 static struct modlinkage modlinkage = {
301         MODREV_1,
302         &modldrv,
303         NULL
304 };

306 static mdi_vhci_ops_t vhci_opinfo = {
307         MDI_VHCI_OPS_REV,
308         vhci_pathinfo_init,             /* Pathinfo node init callback */
309         vhci_pathinfo_uninit,           /* Pathinfo uninit callback */
310         vhci_pathinfo_state_change,     /* Pathinfo node state change */
311         vhci_failover,                  /* failover callback */
312         vhci_client_attached,           /* client attached callback     */
313         vhci_is_dev_supported           /* is device supported by mdi */
314 };

316 /*
317  * The scsi_failover table defines an ordered set of 'fops' modules supported
318  * by scsi_vhci.  Currently, initialize this table from the 'ddi-forceload'
319  * property specified in scsi_vhci.conf.
320  */
321 static struct scsi_failover {
322         ddi_modhandle_t                 sf_mod;
323         struct scsi_failover_ops        *sf_sfo;
324 } *scsi_failover_table;
```



```
325 static uint_t   scsi_nfailover;

327 int
328 _init(void)
329 {
330         int     rval;

332         /*
333          * Allocate soft state and prepare to do ddi_soft_state_zalloc()
334          * before registering with the transport first.
335          */
336         if ((rval = ddi_soft_state_init(&vhci_softstate,
337             sizeof (struct scsi_vhci), 1)) != 0) {
338                 VHCI_DEBUG(1, (CE_NOTE, NULL,
339                     "!_init:soft state init failed\n"));
340                 return (rval);
341         }

343         if ((rval = scsi_hba_init(&modlinkage)) != 0) {
344                 VHCI_DEBUG(1, (CE_NOTE, NULL,
345                     "!_init: scsi hba init failed\n"));
346                 ddi_soft_state_fini(&vhci_softstate);
347                 return (rval);
348         }

350         mutex_init(&vhci_global_mutex, NULL, MUTEX_DRIVER, NULL);
351         cv_init(&vhci_cv, NULL, CV_DRIVER, NULL);

353         mutex_init(&vhci_targetmap_mutex, NULL, MUTEX_DRIVER, NULL);
354         vhci_targetmap_byport = mod_hash_create_strhash(
355             "vhci_targetmap_byport", 256, mod_hash_null_valdtor);
356         vhci_targetmap_bypid = mod_hash_create_idhash(
357             "vhci_targetmap_bypid", 256, mod_hash_null_valdtor);

359         if ((rval = mod_install(&modlinkage)) != 0) {
360                 VHCI_DEBUG(1, (CE_NOTE, NULL, "!_init: mod_install failed\n"));
361                 if (vhci_targetmap_bypid)
362                         mod_hash_destroy_idhash(vhci_targetmap_bypid);
363                 if (vhci_targetmap_byport)
364                         mod_hash_destroy_strhash(vhci_targetmap_byport);
365                 mutex_destroy(&vhci_targetmap_mutex);
366                 cv_destroy(&vhci_cv);
367                 mutex_destroy(&vhci_global_mutex);
368                 scsi_hba_fini(&modlinkage);
369                 ddi_soft_state_fini(&vhci_softstate);
370         }
371         return (rval);
372 }


375 /*
376  * the system is done with us as a driver, so clean up
377  */
378 int
379 _fini(void)
380 {
381         int rval;

383         /*
384          * don't start cleaning up until we know that the module remove
385          * has worked  -- if this works, then we know that each instance
386          * has successfully been DDI_DETACHed
387          */
388         if ((rval = mod_remove(&modlinkage)) != 0) {
389                 VHCI_DEBUG(4, (CE_NOTE, NULL, "!_fini: mod_remove failed\n"));
390                 return (rval);
```

```
391         }
393         if (vhci_targetmap_bypid)
394                 mod_hash_destroy_idhash(vhci_targetmap_bypid);
395         if (vhci_targetmap_byport)
396                 mod_hash_destroy_strhash(vhci_targetmap_byport);
397         mutex_destroy(&vhci_targetmap_mutex);
398         cv_destroy(&vhci_cv);
399         mutex_destroy(&vhci_global_mutex);
400         scsi_hba_fini(&modlinkage);
401         ddi_soft_state_fini(&vhci_softstate);
403         return (rval);
404 }
406 int
407 _info(struct modinfo *modinfop)
408 {
409         return (mod_info(&modlinkage, modinfop));
410 }
412 /*
413  * Lookup scsi_failover by "short name" of failover module.
414  */
415 struct scsi_failover_ops *
416 vhci_failover_ops_by_name(char *name)
417 {
418         struct scsi_failover    *sf;
420         for (sf = scsi_failover_table; sf->sf_mod; sf++) {
421                 if (sf->sf_sfo == NULL)
422                         continue;
423                 if (strcmp(sf->sf_sfo->sfo_name, name) == 0)
424                         return (sf->sf_sfo);
425         }
426         return (NULL);
427 }
429 /*
430  * Load all scsi_failover_ops 'fops' modules.
431  */
432 static void
433 vhci_failover_modopen(struct scsi_vhci *vhci)
434 {
435         char                    **module;
436         int                     i;
437         struct scsi_failover    *sf;
438         char                    **dt;
439         int                     e;
441         if (scsi_failover_table)
442                 return;
444         /* Get the list of modules from scsi_vhci.conf */
445         if (ddi_prop_lookup_string_array(DDI_DEV_T_ANY,
446             vhci->vhci_dip, DDI_PROP_DONTPASS, "ddi-forceload",
447             &module, &scsi_nfailover) != DDI_PROP_SUCCESS) {
448                 cmn_err(CE_WARN, "scsi_vhci: "
449                     "scsi_vhci.conf is missing 'ddi-forceload'");
450                 return;
451         }
452         if (scsi_nfailover == 0) {
453                 cmn_err(CE_WARN, "scsi_vhci: "
454                     "scsi_vhci.conf has empty 'ddi-forceload'");
455                 ddi_prop_free(module);
456                 return;
```



```
457         }
459         /* allocate failover table based on number of modules */
460         scsi_failover_table = (struct scsi_failover *)
461             kmem_zalloc(sizeof (struct scsi_failover) * (scsi_nfailover + 1),
462             KM_SLEEP);
464         /* loop over modules specified in scsi_vhci.conf and open each module */
465         for (i = 0, sf = scsi_failover_table; i < scsi_nfailover; i++) {
466                 if (module[i] == NULL)
467                         continue;
469                 sf->sf_mod = ddi_modopen(module[i], KRTLD_MODE_FIRST, &e);
470                 if (sf->sf_mod == NULL) {
471                         /*
472                          * A module returns EEXIST if other software is
473                          * supporting the intended function: for example
474                          * the scsi_vhci_f_sum_emc module returns EEXIST
475                          * from _init if EMC powerpath software is installed.
476                          */
477                         if (e != EEXIST)
478                                 cmn_err(CE_WARN, "scsi_vhci: unable to open "
479                                     "module '%s', error %d", module[i], e);
480                         continue;
481                 }
482                 sf->sf_sfo = ddi_modsym(sf->sf_mod,
483                     "scsi_vhci_failover_ops", &e);
484                 if (sf->sf_sfo == NULL) {
485                         cmn_err(CE_WARN, "scsi_vhci: "
486                             "unable to import 'scsi_failover_ops' from '%s', "
487                             "error %d", module[i], e);
488                         (void) ddi_modclose(sf->sf_mod);
489                         sf->sf_mod = NULL;
490                         continue;
491                 }
493                 /* register vid/pid of devices supported with mpapi */
494                 for (dt = sf->sf_sfo->sfo_devices; *dt; dt++)
495                         vhci_mpapi_add_dev_prod(vhci, *dt);
496                 sf++;
497         }
499         /* verify that at least the "well-known" modules were there */
500         if (vhci_failover_ops_by_name(SFO_NAME_SYM) == NULL)
501                 cmn_err(CE_WARN, "scsi_vhci: well-known module \""
502                     SFO_NAME_SYM "\" not defined in scsi_vhci.conf's "
503                     "'ddi-forceload'");
504         if (vhci_failover_ops_by_name(SFO_NAME_TPGS) == NULL)
505                 cmn_err(CE_WARN, "scsi_vhci: well-known module \""
506                     SFO_NAME_TPGS "\" not defined in scsi_vhci.conf's "
507                     "'ddi-forceload'");
509         /* call sfo_init for modules that need it */
510         for (sf = scsi_failover_table; sf->sf_mod; sf++) {
511                 if (sf->sf_sfo && sf->sf_sfo->sfo_init)
512                         sf->sf_sfo->sfo_init();
513         }
515         ddi_prop_free(module);
516 }
518 /*
519  * unload all loaded scsi_failover_ops modules
520  */
521 static void
522 vhci_failover_modclose()
```

```
{
        struct scsi_failover    *sf;

        for (sf = scsi_failover_table; sf->sf_mod; sf++) {
                if ((sf->sf_mod == NULL) || (sf->sf_sfo == NULL))
                        continue;
                (void) ddi_modclose(sf->sf_mod);
                sf->sf_mod = NULL;
                sf->sf_sfo = NULL;
        }

        if (scsi_failover_table && scsi_nfailover)
                kmem_free(scsi_failover_table,
                    sizeof (struct scsi_failover) * (scsi_nfailover + 1));
        scsi_failover_table = NULL;
        scsi_nfailover = 0;
}

/* ARGSUSED */
static int
vhci_open(dev_t *devp, int flag, int otype, cred_t *credp)
{
        struct scsi_vhci        *vhci;

        if (otype != OTYP_CHR) {
                return (EINVAL);
        }

        vhci = ddi_get_soft_state(vhci_softstate, MINOR2INST(getminor(*devp)));
        if (vhci == NULL) {
                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_open: failed ENXIO\n"));
                return (ENXIO);
        }

        mutex_enter(&vhci->vhci_mutex);
        if ((flag & FEXCL) && (vhci->vhci_state & VHCI_STATE_OPEN)) {
                mutex_exit(&vhci->vhci_mutex);
                vhci_log(CE_NOTE, vhci->vhci_dip,
                    "!vhci%d: Already open\n", getminor(*devp));
                return (EBUSY);
        }

        vhci->vhci_state |= VHCI_STATE_OPEN;
        mutex_exit(&vhci->vhci_mutex);
        return (0);
}


/* ARGSUSED */
static int
vhci_close(dev_t dev, int flag, int otype, cred_t *credp)
{
        struct scsi_vhci        *vhci;

        if (otype != OTYP_CHR) {
                return (EINVAL);
        }

        vhci = ddi_get_soft_state(vhci_softstate, MINOR2INST(getminor(dev)));
        if (vhci == NULL) {
                VHCI_DEBUG(1, (CE_NOTE, NULL, "vhci_close: failed ENXIO\n"));
                return (ENXIO);
        }

        mutex_enter(&vhci->vhci_mutex);
        vhci->vhci_state &= ~VHCI_STATE_OPEN;
```



```
        mutex_exit(&vhci->vhci_mutex);

        return (0);
}

/* ARGSUSED */
static int
vhci_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
        cred_t *credp, int *rval)
{
        if (IS_DEVCTL(cmd)) {
                return (vhci_devctl(dev, cmd, data, mode, credp, rval));
        } else if (cmd == MP_CMD) {
                return (vhci_mpapi_ctl(dev, cmd, data, mode, credp, rval));
        } else {
                return (vhci_ctl(dev, cmd, data, mode, credp, rval));
        }
}

/*
 * attach the module
 */
static int
vhci_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int                     rval = DDI_FAILURE;
        int                     scsi_hba_attached = 0;
        int                     vhci_attached = 0;
        int                     mutex_initted = 0;
        int                     instance;
        struct scsi_vhci        *vhci;
        scsi_hba_tran_t         *tran;
        char                    cache_name_buf[64];
        char                    *data;

        VHCI_DEBUG(4, (CE_NOTE, NULL, "vhci_attach: cmd=0x%x\n", cmd));

        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_ATTACH:
                break;

        case DDI_RESUME:
        case DDI_PM_RESUME:
                VHCI_DEBUG(1, (CE_NOTE, NULL, "!vhci_attach: resume not yet"
                    "implemented\n"));
                return (rval);

        default:
                VHCI_DEBUG(1, (CE_NOTE, NULL,
                    "!vhci_attach: unknown ddi command\n"));
                return (rval);
        }

        /*
         * Allocate vhci data structure.
         */
        if (ddi_soft_state_zalloc(vhci_softstate, instance) != DDI_SUCCESS) {
                VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
                    "soft state alloc failed\n"));
                return (DDI_FAILURE);
        }

        if ((vhci = ddi_get_soft_state(vhci_softstate, instance)) == NULL) {
                VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
```

```
655                     "bad soft state\n"));
656                 ddi_soft_state_free(vhci_softstate, instance);
657                 return (DDI_FAILURE);
658         }
659
660         /* Allocate packet cache */
661         (void) snprintf(cache_name_buf, sizeof (cache_name_buf),
662             "vhci%d_cache", instance);
663
664         mutex_init(&vhci->vhci_mutex, NULL, MUTEX_DRIVER, NULL);
665         mutex_initted++;
666
667         /*
668          * Allocate a transport structure
669          */
670         tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);
671         ASSERT(tran != NULL);
672
673         vhci->vhci_tran         = tran;
674         vhci->vhci_dip          = dip;
675         vhci->vhci_instance     = instance;
676
677         tran->tran_hba_private  = vhci;
678         tran->tran_tgt_init     = vhci_scsi_tgt_init;
679         tran->tran_tgt_probe    = NULL;
680         tran->tran_tgt_free     = vhci_scsi_tgt_free;
681
682         tran->tran_start        = vhci_scsi_start;
683         tran->tran_abort        = vhci_scsi_abort;
684         tran->tran_reset        = vhci_scsi_reset;
685         tran->tran_getcap       = vhci_scsi_getcap;
686         tran->tran_setcap       = vhci_scsi_setcap;
687         tran->tran_init_pkt     = vhci_scsi_init_pkt;
688         tran->tran_destroy_pkt  = vhci_scsi_destroy_pkt;
689         tran->tran_dmafree      = vhci_scsi_dmafree;
690         tran->tran_sync_pkt     = vhci_scsi_sync_pkt;
691         tran->tran_reset_notify = vhci_scsi_reset_notify;
692
693         tran->tran_get_bus_addr = vhci_scsi_get_bus_addr;
694         tran->tran_get_name     = vhci_scsi_get_name;
695         tran->tran_bus_reset    = NULL;
696         tran->tran_quiesce      = NULL;
697         tran->tran_unquiesce    = NULL;
698
699         /*
700          * register event notification routines with scsa
701          */
702         tran->tran_get_eventcookie = NULL;
703         tran->tran_add_eventcall = NULL;
704         tran->tran_remove_eventcall = NULL;
705         tran->tran_post_event   = NULL;
706
707         tran->tran_bus_power    = vhci_scsi_bus_power;
708
709         tran->tran_bus_config   = vhci_scsi_bus_config;
710         tran->tran_bus_unconfig = vhci_scsi_bus_unconfig;
711
712         /*
713          * Attach this instance with the mpxio framework
714          */
715         if (mdi_vhci_register(MDI_HCI_CLASS_SCSI, dip, &vhci_opinfo, 0)
716             != MDI_SUCCESS) {
717                 VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
718                     "mdi_vhci_register failed\n"));
719                 goto attach_fail;
720         }
```



```
721         vhci_attached++;
722
723         /*
724          * Attach this instance of the hba.
725          *
726          * Regarding dma attributes: Since scsi_vhci is a virtual scsi HBA
727          * driver, it has nothing to do with DMA. However, when calling
728          * scsi_hba_attach_setup() we need to pass something valid in the
729          * dma attributes parameter. So we just use scsi_alloc_attr.
730          * SCSA itself seems to care only for dma_attr_minxfer and
731          * dma_attr_burstsizes fields of dma attributes structure.
732          * It expects those fileds to be non-zero.
733          */
734         if (scsi_hba_attach_setup(dip, &scsi_alloc_attr, tran,
735             SCSI_HBA_ADDR_COMPLEX) != DDI_SUCCESS) {
736                 VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
737                     "hba attach failed\n"));
738                 goto attach_fail;
739         }
740         scsi_hba_attached++;
741
742         if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
743             INST2DEVCTL(instance), DDI_NT_SCSI_NEXUS, 0) != DDI_SUCCESS) {
744                 VHCI_DEBUG(1, (CE_NOTE, dip, "!vhci_attach:"
745                     " ddi_create_minor_node failed\n"));
746                 goto attach_fail;
747         }
748
749         /*
750          * Set pm-want-child-notification property for
751          * power management of the phci and client
752          */
753         if (ddi_prop_create(DDI_DEV_T_NONE, dip, DDI_PROP_CANSLEEP,
754             "pm-want-child-notification?", NULL, NULL) != DDI_PROP_SUCCESS) {
755                 cmn_err(CE_WARN,
756                     "%s%d fail to create pm-want-child-notification? prop",
757                     ddi_driver_name(dip), ddi_get_instance(dip));
758                 goto attach_fail;
759         }
760
761         vhci->vhci_taskq = taskq_create("vhci_taskq", 1, MINCLSYSPRI, 1, 4, 0);
762         vhci->vhci_update_pathstates_taskq =
763             taskq_create("vhci_update_pathstates", VHCI_NUM_UPDATE_TASKQ,
764             MINCLSYSPRI, 1, 4, 0);
765         ASSERT(vhci->vhci_taskq);
766         ASSERT(vhci->vhci_update_pathstates_taskq);
767
768         /*
769          * Set appropriate configuration flags based on options set in
770          * conf file.
771          */
772         vhci->vhci_conf_flags = 0;
773         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, PROPFLAGS,
774             "auto-failback", &data) == DDI_SUCCESS) {
775                 if (strcmp(data, "enable") == 0)
776                         vhci->vhci_conf_flags |= VHCI_CONF_FLAGS_AUTO_FAILBACK;
777                 ddi_prop_free(data);
778         }
779
780         if (!(vhci->vhci_conf_flags & VHCI_CONF_FLAGS_AUTO_FAILBACK))
781                 vhci_log(CE_NOTE, dip, "!Auto-failback capability "
782                     "disabled through scsi_vhci.conf file.");
783
784         /*
785          * Allocate an mpapi private structure
786          */
```

```
787         vhci->mp_priv = kmem_zalloc(sizeof (mpapi_priv_t), KM_SLEEP);
788         if (vhci_mpapi_init(vhci) != 0) {
789                 VHCI_DEBUG(1, (CE_WARN, NULL, "!vhci_attach: "
790                     "vhci_mpapi_init() failed"));
791         }

793         vhci_failover_modopen(vhci);            /* load failover modules */

795         ddi_report_dev(dip);
796         return (DDI_SUCCESS);

798 attach_fail:
799         if (vhci_attached)
800                 (void) mdi_vhci_unregister(dip, 0);

802         if (scsi_hba_attached)
803                 (void) scsi_hba_detach(dip);

805         if (vhci->vhci_tran)
806                 scsi_hba_tran_free(vhci->vhci_tran);

808         if (mutex_initted) {
809                 mutex_destroy(&vhci->vhci_mutex);
810         }

812         ddi_soft_state_free(vhci_softstate, instance);
813         return (DDI_FAILURE);
814 }


817 /*ARGSUSED*/
818 static int
819 vhci_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
820 {
821         int                     instance = ddi_get_instance(dip);
822         scsi_hba_tran_t         *tran;
823         struct scsi_vhci        *vhci;

825         VHCI_DEBUG(4, (CE_NOTE, NULL, "vhci_detach: cmd=0x%x\n", cmd));

827         if ((tran = ddi_get_driver_private(dip)) == NULL)
828                 return (DDI_FAILURE);

830         vhci = TRAN2HBAPRIVATE(tran);
831         if (!vhci) {
832                 return (DDI_FAILURE);
833         }

835         switch (cmd) {
836         case DDI_DETACH:
837                 break;

839         case DDI_SUSPEND:
840         case DDI_PM_SUSPEND:
841                 VHCI_DEBUG(1, (CE_NOTE, NULL, "!vhci_detach: suspend/pm not yet"
842                     "implemented\n"));
843                 return (DDI_FAILURE);

845         default:
846                 VHCI_DEBUG(1, (CE_NOTE, NULL,
847                     "!vhci_detach: unknown ddi command\n"));
848                 return (DDI_FAILURE);
849         }

851         (void) mdi_vhci_unregister(dip, 0);
852         (void) scsi_hba_detach(dip);
```



```
853         scsi_hba_tran_free(tran);

855         if (ddi_prop_remove(DDI_DEV_T_NONE, dip,
856             "pm-want-child-notification?") != DDI_PROP_SUCCESS) {
857                 cmn_err(CE_WARN,
858                     "%s%d unable to remove prop pm-want_child_notification?",
859                     ddi_driver_name(dip), ddi_get_instance(dip));
860         }
861         if (vhci_restart_timeid != 0) {
862                 (void) untimeout(vhci_restart_timeid);
863         }
864         vhci_restart_timeid = 0;

866         mutex_destroy(&vhci->vhci_mutex);
867         vhci->vhci_dip = NULL;
868         vhci->vhci_tran = NULL;
869         taskq_destroy(vhci->vhci_taskq);
870         taskq_destroy(vhci->vhci_update_pathstates_taskq);
871         ddi_remove_minor_node(dip, NULL);
872         ddi_soft_state_free(vhci_softstate, instance);

874         vhci_failover_modclose();               /* unload failover modules */
875         return (DDI_SUCCESS);
876 }

878 /*
879  * vhci_getinfo()
880  * Given the device number, return the devinfo pointer or the
881  * instance number.
882  * Note: always succeed DDI_INFO_DEVT2INSTANCE, even before attach.
883  */

885 /*ARGSUSED*/
886 static int
887 vhci_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void **result)
888 {
889         struct scsi_vhci        *vhcip;
890         int                     instance = MINOR2INST(getminor((dev_t)arg));

892         switch (cmd) {
893         case DDI_INFO_DEVT2DEVINFO:
894                 vhcip = ddi_get_soft_state(vhci_softstate, instance);
895                 if (vhcip != NULL)
896                         *result = vhcip->vhci_dip;
897                 else {
898                         *result = NULL;
899                         return (DDI_FAILURE);
900                 }
901                 break;

903         case DDI_INFO_DEVT2INSTANCE:
904                 *result = (void *)(uintptr_t)instance;
905                 break;

907         default:
908                 return (DDI_FAILURE);
909         }

911         return (DDI_SUCCESS);
912 }

914 /*ARGSUSED*/
915 static int
916 vhci_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
917         scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
918 {
```

```
        char                    *guid;
        scsi_vhci_lun_t         *vlun;
        struct scsi_vhci        *vhci;
        clock_t                 from_ticks;
        mdi_pathinfo_t          *pip;
        int                     rval;

        ASSERT(hba_dip != NULL);
        ASSERT(tgt_dip != NULL);

        if (ddi_prop_lookup_string(DDI_DEV_T_ANY, tgt_dip, PROPFLAGS,
            MDI_CLIENT_GUID_PROP, &guid) != DDI_SUCCESS) {
                /*
                 * This must be the .conf node without GUID property.
                 * The node under fp already inserts a delay, so we
                 * just return from here. We rely on this delay to have
                 * all dips be posted to the ndi hotplug thread's newdev
                 * list. This is necessary for the deferred attach
                 * mechanism to work and opens() done soon after boot to
                 * succeed.
                 */
                VHCI_DEBUG(4, (CE_WARN, hba_dip, "tgt_init: lun guid "
                    "property failed"));
                return (DDI_NOT_WELL_FORMED);
        }

        if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
                /*
                 * This must be .conf node with the GUID property. We don't
                 * merge property by ndi_merge_node() here  because the
                 * devi_addr_buf of .conf node is "" always according the
                 * implementation of vhci_scsi_get_name_bus_addr().
                 */
                ddi_set_name_addr(tgt_dip, NULL);
                return (DDI_FAILURE);
        }

        vhci = ddi_get_soft_state(vhci_softstate, ddi_get_instance(hba_dip));
        ASSERT(vhci != NULL);

        VHCI_DEBUG(4, (CE_NOTE, hba_dip,
            "!tgt_init: called for %s (instance %d)\n",
            ddi_driver_name(tgt_dip), ddi_get_instance(tgt_dip)));

        vlun = vhci_lun_lookup(tgt_dip);

        mutex_enter(&vhci_global_mutex);

        from_ticks = ddi_get_lbolt();
        if (vhci_to_ticks == 0) {
                vhci_to_ticks = from_ticks +
                    drv_usectohz(vhci_init_wait_timeout);
        }

#if DEBUG
        if (vlun) {
                VHCI_DEBUG(1, (CE_WARN, hba_dip, "tgt_init: "
                    "vhci_scsi_tgt_init: guid %s : found vlun 0x%p "
                    "from_ticks %lx to_ticks %lx",
                    guid, (void *)vlun, from_ticks, vhci_to_ticks));
        } else {
                VHCI_DEBUG(1, (CE_WARN, hba_dip, "tgt_init: "
                    "vhci_scsi_tgt_init: guid %s : vlun not found "
                    "from_ticks %lx to_ticks %lx", guid, from_ticks,
                    vhci_to_ticks));
        }
```



```
#endif

        rval = mdi_select_path(tgt_dip, NULL,
            (MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH), NULL, &pip);
        if (rval == MDI_SUCCESS) {
                mdi_rele_path(pip);
        }

        /*
         * Wait for the following conditions :
         *      1. no vlun available yet
         *      2. no path established
         *      3. timer did not expire
         */
        while ((vlun == NULL) || (mdi_client_get_path_count(tgt_dip) == 0) ||
            (rval != MDI_SUCCESS)) {
                if (vlun && vlun->svl_not_supported) {
                        VHCI_DEBUG(1, (CE_WARN, hba_dip, "tgt_init: "
                            "vlun 0x%p lun guid %s not supported!",
                            (void *)vlun, guid));
                        mutex_exit(&vhci_global_mutex);
                        ddi_prop_free(guid);
                        return (DDI_NOT_WELL_FORMED);
                }
                if ((vhci_first_time == 0) && (from_ticks >= vhci_to_ticks)) {
                        vhci_first_time = 1;
                }
                if (vhci_first_time == 1) {
                        VHCI_DEBUG(1, (CE_WARN, hba_dip, "vhci_scsi_tgt_init: "
                            "no wait for %s. from_tick %lx, to_tick %lx",
                            guid, from_ticks, vhci_to_ticks));
                        mutex_exit(&vhci_global_mutex);
                        ddi_prop_free(guid);
                        return (DDI_NOT_WELL_FORMED);
                }

                if (cv_timedwait(&vhci_cv,
                    &vhci_global_mutex, vhci_to_ticks) == -1) {
                        /* Timed out */
#ifdef DEBUG
                        if (vlun == NULL) {
                                VHCI_DEBUG(1, (CE_WARN, hba_dip,
                                    "tgt_init: no vlun for %s!", guid));
                        } else if (mdi_client_get_path_count(tgt_dip) == 0) {
                                VHCI_DEBUG(1, (CE_WARN, hba_dip,
                                    "tgt_init: client path count is "
                                    "zero for %s!", guid));
                        } else {
                                VHCI_DEBUG(1, (CE_WARN, hba_dip,
                                    "tgt_init: client path not "
                                    "available yet for %s!", guid));
                        }
#endif /* DEBUG */
                        mutex_exit(&vhci_global_mutex);
                        ddi_prop_free(guid);
                        return (DDI_NOT_WELL_FORMED);
                }
                vlun = vhci_lun_lookup(tgt_dip);
                rval = mdi_select_path(tgt_dip, NULL,
                    (MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH),
                    NULL, &pip);
                if (rval == MDI_SUCCESS) {
                        mdi_rele_path(pip);
                }
                from_ticks = ddi_get_lbolt();
        }
```

```
1051         mutex_exit(&vhci_global_mutex);

1053         ASSERT(vlun != NULL);
1054         ddi_prop_free(guid);

1056         scsi_device_hba_private_set(sd, vlun);

1058         return (DDI_SUCCESS);
1059 }

1061 /*ARGSUSED*/
1062 static void
1063 vhci_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1064         scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1065 {
1066         struct scsi_vhci_lun *dvlp;
1067         ASSERT(mdi_client_get_path_count(tgt_dip) <= 0);
1068         dvlp = (struct scsi_vhci_lun *)scsi_device_hba_private_get(sd);
1069         ASSERT(dvlp != NULL);

1071         vhci_lun_free(dvlp, sd);
1072 }

1074 /*
1075  * a PGR register command has started; copy the info we need
1076  */
1077 int
1078 vhci_pgr_register_start(scsi_vhci_lun_t *vlun, struct scsi_pkt *pkt)
1079 {
1080         struct vhci_pkt         *vpkt = TGTPKT2VHCIPKT(pkt);
1081         void                    *addr;

1083         if (!vpkt->vpkt_tgt_init_bp)
1084                 return (TRAN_BADPKT);

1086         addr = bp_mapin_common(vpkt->vpkt_tgt_init_bp,
1087             (vpkt->vpkt_flags & CFLAG_NOWAIT) ? VM_NOSLEEP : VM_SLEEP);
1088         if (addr == NULL)
1089                 return (TRAN_BUSY);

1091         mutex_enter(&vlun->svl_mutex);

1093         vhci_print_prout_keys(vlun, "v_pgr_reg_start: before bcopy:");

1095         bcopy(addr, &vlun->svl_prout, sizeof (vhci_prout_t) -
1096             (2 * MHIOC_RESV_KEY_SIZE*sizeof (char)));
1097         bcopy(pkt->pkt_cdbp, vlun->svl_cdb, sizeof (vlun->svl_cdb));

1099         vhci_print_prout_keys(vlun, "v_pgr_reg_start: after bcopy:");

1101         vlun->svl_time = pkt->pkt_time;
1102         vlun->svl_bcount = vpkt->vpkt_tgt_init_bp->b_bcount;
1103         vlun->svl_first_path = vpkt->vpkt_path;
1104         mutex_exit(&vlun->svl_mutex);
1105         return (0);
1106 }

1108 /*
1109  * Function name : vhci_scsi_start()
1110  *
1111  * Return Values : TRAN_FATAL_ERROR     - vhci has been shutdown
1112  *                                        or other fatal failure
1113  *                                        preventing packet transportation
1114  *                 TRAN_BUSY            - request queue is full
1115  *                 TRAN_ACCEPT          - pkt has been submitted to phci
1116  *                                        (or is held in the waitQ)
```



```
1117  * Description   : Implements SCSA's tran_start() entry point for
1118  *                 packet transport
1119  *
1120  */
1121 static int
1122 vhci_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
1123 {
1124         int                     rval = TRAN_ACCEPT;
1125         int                     instance, held;
1126         struct scsi_vhci        *vhci = ADDR2VHCI(ap);
1127         struct scsi_vhci_lun    *vlun = ADDR2VLUN(ap);
1128         struct vhci_pkt         *vpkt = TGTPKT2VHCIPKT(pkt);
1129         int                     flags = 0;
1130         scsi_vhci_priv_t        *svp, *svp_resrv;
1131         dev_info_t              *cdip;
1132         client_lb_t             lbp;
1133         int                     restore_lbp = 0;
1134         /* set if pkt is SCSI-II RESERVE cmd */
1135         int                     pkt_reserve_cmd = 0;
1136         int                     reserve_failed = 0;
1137         int                     resrv_instance = 0;
1138         mdi_pathinfo_t          *pip;
1139         struct scsi_pkt         *rel_pkt;

1141         ASSERT(vhci != NULL);
1142         ASSERT(vpkt != NULL);
1143         ASSERT(vpkt->vpkt_state != VHCI_PKT_ISSUED);
1144         cdip = ADDR2DIP(ap);

1146         /*
1147          * Block IOs if LUN is held or QUIESCED for IOs.
1148          */
1149         if ((VHCI_LUN_IS_HELD(vlun)) ||
1150             ((vlun->svl_flags & VLUN_QUIESCED_FLG) == VLUN_QUIESCED_FLG)) {
1151                 return (TRAN_BUSY);
1152         }

1154         /*
1155          * vhci_lun needs to be quiesced before SCSI-II RESERVE command
1156          * can be issued.  This may require a cv_timedwait, which is
1157          * dangerous to perform in an interrupt context.  So if this
1158          * is a RESERVE command a taskq is dispatched to service it.
1159          * This taskq shall again call vhci_scsi_start, but we shall be
1160          * sure its not in an interrupt context.
1161          */
1162         if ((pkt->pkt_cdbp[0] == SCMD_RESERVE) ||
1163             (pkt->pkt_cdbp[0] == SCMD_RESERVE_G1)) {
1164                 if (!(vpkt->vpkt_state & VHCI_PKT_THRU_TASKQ)) {
1165                         if (taskq_dispatch(vhci->vhci_taskq,
1166                             vhci_dispatch_scsi_start, (void *) vpkt,
1167                             KM_NOSLEEP)) {
1168                                 return (TRAN_ACCEPT);
1169                         } else {
1170                                 return (TRAN_BUSY);
1171                         }
1172                 }

1174                 /*
1175                  * Here we ensure that simultaneous SCSI-II RESERVE cmds don't
1176                  * get serviced for a lun.
1177                  */
1178                 VHCI_HOLD_LUN(vlun, VH_NOSLEEP, held);
1179                 if (!held) {
1180                         return (TRAN_BUSY);
1181                 } else if ((vlun->svl_flags & VLUN_QUIESCED_FLG) ==
1182                     VLUN_QUIESCED_FLG) {
```

```
1183                        VHCI_RELEASE_LUN(vlun);
1184                        return (TRAN_BUSY);
1185                }

1187                /*
1188                 * To ensure that no IOs occur for this LUN for the duration
1189                 * of this pkt set the VLUN_QUIESCED_FLG.
1190                 * In case this routine needs to exit on error make sure that
1191                 * this flag is cleared.
1192                 */
1193                vlun->svl_flags |= VLUN_QUIESCED_FLG;
1194                pkt_reserve_cmd = 1;

1196                /*
1197                 * if this is a SCSI-II RESERVE command, set load balancing
1198                 * policy to be ALTERNATE PATH to ensure that all subsequent
1199                 * IOs are routed on the same path.  This is because if commands
1200                 * are routed across multiple paths then IOs on paths other than
1201                 * the one on which the RESERVE was executed will get a
1202                 * RESERVATION CONFLICT
1203                 */
1204                lbp = mdi_get_lb_policy(cdip);
1205                if (lbp != LOAD_BALANCE_NONE) {
1206                        if (vhci_quiesce_lun(vlun) != 1) {
1207                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1208                                VHCI_RELEASE_LUN(vlun);
1209                                return (TRAN_FATAL_ERROR);
1210                        }
1211                        vlun->svl_lb_policy_save = lbp;
1212                        if (mdi_set_lb_policy(cdip, LOAD_BALANCE_NONE) !=
1213                            MDI_SUCCESS) {
1214                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1215                                VHCI_RELEASE_LUN(vlun);
1216                                return (TRAN_FATAL_ERROR);
1217                        }
1218                        restore_lbp = 1;
1219                }

1221                VHCI_DEBUG(2, (CE_NOTE, vhci->vhci_dip,
1222                    "!vhci_scsi_start: sending SCSI-2 RESERVE, vlun 0x%p, "
1223                    "svl_resrv_pip 0x%p, svl_flags: %x, lb_policy %x",
1224                    (void *)vlun, (void *)vlun->svl_resrv_pip, vlun->svl_flags,
1225                    mdi_get_lb_policy(cdip)));

1227                /*
1228                 * See comments for VLUN_RESERVE_ACTIVE_FLG in scsi_vhci.h
1229                 * To narrow this window where a reserve command may be sent
1230                 * down an inactive path the path states first need to be
1231                 * updated.  Before calling vhci_update_pathstates reset
1232                 * VLUN_RESERVE_ACTIVE_FLG, just in case it was already set
1233                 * for this lun.  This shall prevent an unnecessary reset
1234                 * from being sent out.  Also remember currently reserved path
1235                 * just for a case the new reservation will go to another path.
1236                 */
1237                if (vlun->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
1238                        resrv_instance = mdi_pi_get_path_instance(
1239                            vlun->svl_resrv_pip);
1240                }
1241                vlun->svl_flags &= ~VLUN_RESERVE_ACTIVE_FLG;
1242                vhci_update_pathstates((void *)vlun);
1243        }

1245        instance = ddi_get_instance(vhci->vhci_dip);

1247        /*
1248         * If the command is PRIN with action of zero, then the cmd
```



```
1249         * is reading PR keys which requires filtering on completion.
1250         * Data cache sync must be guaranteed.
1251         */
1252        if ((pkt->pkt_cdbp[0] == SCMD_PRIN) && (pkt->pkt_cdbp[1] == 0) &&
1253            (vpkt->vpkt_org_vpkt == NULL)) {
1254                vpkt->vpkt_tgt_init_pkt_flags |= PKT_CONSISTENT;
1255        }

1257        /*
1258         * Do not defer bind for PKT_DMA_PARTIAL
1259         */
1260        if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {

1262                /* This is a non pkt_dma_partial case */
1263                if ((rval = vhci_bind_transport(
1264                    ap, vpkt, vpkt->vpkt_tgt_init_pkt_flags, NULL_FUNC))
1265                    != TRAN_ACCEPT) {
1266                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1267                            "!vhci%d %x: failed to bind transport: "
1268                            "vlun 0x%p pkt_reserved %x restore_lbp %x,"
1269                            "lbp %x", instance, rval, (void *)vlun,
1270                            pkt_reserve_cmd, restore_lbp, lbp));
1271                        if (restore_lbp)
1272                                (void) mdi_set_lb_policy(cdip, lbp);
1273                        if (pkt_reserve_cmd)
1274                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1275                        return (rval);
1276                }
1277                VHCI_DEBUG(8, (CE_NOTE, NULL,
1278                    "vhci_scsi_start: v_b_t called 0x%p\n", (void *)vpkt));
1279        }
1280        ASSERT(vpkt->vpkt_hba_pkt != NULL);
1281        ASSERT(vpkt->vpkt_path != NULL);

1283        /*
1284         * This is the chance to adjust the pHCI's pkt and other information
1285         * from target driver's pkt.
1286         */
1287        VHCI_DEBUG(8, (CE_NOTE, vhci->vhci_dip, "vhci_scsi_start vpkt %p\n",
1288            (void *)vpkt));
1289        vhci_update_pHCI_pkt(vpkt, pkt);

1291        if (vlun->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
1292                if (vpkt->vpkt_path != vlun->svl_resrv_pip) {
1293                        VHCI_DEBUG(1, (CE_WARN, vhci->vhci_dip,
1294                            "!vhci_bind: reserve flag set for vlun 0x%p, but, "
1295                            "pktpath 0x%p resrv path 0x%p differ. lb_policy %x",
1296                            (void *)vlun, (void *)vpkt->vpkt_path,
1297                            (void *)vlun->svl_resrv_pip,
1298                            mdi_get_lb_policy(cdip)));
1299                        reserve_failed = 1;
1300                }
1301        }

1303        svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(vpkt->vpkt_path);
1304        if (svp == NULL || reserve_failed) {
1305                if (pkt_reserve_cmd) {
1306                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1307                            "!vhci_bind returned null svp vlun 0x%p",
1308                            (void *)vlun));
1309                        vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1310                        if (restore_lbp)
1311                                (void) mdi_set_lb_policy(cdip, lbp);
1312                }
1313 pkt_cleanup:
1314                if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {
```

```
1315                        scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
1316                        vpkt->vpkt_hba_pkt = NULL;
1317                        if (vpkt->vpkt_path) {
1318                                mdi_rele_path(vpkt->vpkt_path);
1319                                vpkt->vpkt_path = NULL;
1320                        }
1321                }
1322                if ((pkt->pkt_cdbp[0] == SCMD_PROUT) &&
1323                    (((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_REGISTER) ||
1324                    ((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_R_AND_IGNORE))) {
1325                        sema_v(&vlun->svl_pgr_sema);
1326                }
1327                return (TRAN_BUSY);
1328        }

1330        if ((resrv_instance != 0) && (resrv_instance !=
1331            mdi_pi_get_path_instance(vpkt->vpkt_path))) {
1332                /*
1333                 * This is an attempt to reserve vpkt->vpkt_path.  But the
1334                 * previously reserved path referred by resrv_instance might
1335                 * still be reserved.  Hence we will send a release command
1336                 * there in order to avoid a reservation conflict.
1337                 */
1338                VHCI_DEBUG(1, (CE_NOTE, vhci->vhci_dip, "!vhci_scsi_start: "
1339                    "conflicting reservation on another path, vlun 0x%p, "
1340                    "reserved instance %d, new instance: %d, pip: 0x%p",
1341                    (void *)vlun, resrv_instance,
1342                    mdi_pi_get_path_instance(vpkt->vpkt_path),
1343                    (void *)vpkt->vpkt_path));

1345                /*
1346                 * In rare cases, the path referred by resrv_instance could
1347                 * disappear in the meantime. Calling mdi_select_path() below
1348                 * is an attempt to find out if the path still exists. It also
1349                 * ensures that the path will be held when the release is sent.
1350                 */
1351                rval = mdi_select_path(cdip, NULL, MDI_SELECT_PATH_INSTANCE,
1352                    (void *)(intptr_t)resrv_instance, &pip);

1354                if ((rval == MDI_SUCCESS) && (pip != NULL)) {
1355                        svp_resrv = (scsi_vhci_priv_t *)
1356                            mdi_pi_get_vhci_private(pip);
1357                        rel_pkt = scsi_init_pkt(&svp_resrv->svp_psd->sd_address,
1358                            NULL, NULL, CDB_GROUP0,
1359                            sizeof (struct scsi_arq_status), 0, 0, SLEEP_FUNC,
1360                            NULL);

1362                        if (rel_pkt == NULL) {
1363                                char    *p_path;

1365                                /*
1366                                 * This is very unlikely.
1367                                 * scsi_init_pkt(SLEEP_FUNC) does not fail
1368                                 * because of resources. But in theory it could
1369                                 * fail for some other reason. There is not an
1370                                 * easy way how to recover though. Log a warning
1371                                 * and return.
1372                                 */
1373                                p_path = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
1374                                vhci_log(CE_WARN, vhci->vhci_dip, "!Sending "
1375                                    "RELEASE(6) to %s failed, a potential "
1376                                    "reservation conflict ahead.",
1377                                    ddi_pathname(mdi_pi_get_phci(pip), p_path));
1378                                kmem_free(p_path, MAXPATHLEN);

1380                                if (restore_lbp)
```



```
1381                                        (void) mdi_set_lb_policy(cdip, lbp);

1383                                /* no need to check pkt_reserve_cmd here */
1384                                vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1385                                return (TRAN_FATAL_ERROR);
1386                        }

1388                        rel_pkt->pkt_cdbp[0] = SCMD_RELEASE;
1389                        rel_pkt->pkt_time = 60;

1391                        /*
1392                         * Ignore the return value.  If it will fail
1393                         * then most likely it is no longer reserved
1394                         * anyway.
1395                         */
1396                        (void) vhci_do_scsi_cmd(rel_pkt);
1397                        VHCI_DEBUG(1, (CE_NOTE, NULL,
1398                            "!vhci_scsi_start: path 0x%p, issued SCSI-2"
1399                            " RELEASE\n", (void *)pip));
1400                        scsi_destroy_pkt(rel_pkt);
1401                        mdi_rele_path(pip);
1402                }
1403        }

1405        VHCI_INCR_PATH_CMDCOUNT(svp);

1407        /*
1408         * Ensure that no other IOs raced ahead, while a RESERVE cmd was
1409         * QUIESCING the same lun.
1410         */
1411        if ((!pkt_reserve_cmd) &&
1412            ((vlun->svl_flags & VLUN_QUIESCED_FLG) == VLUN_QUIESCED_FLG)) {
1413                VHCI_DECR_PATH_CMDCOUNT(svp);
1414                goto pkt_cleanup;
1415        }

1417        if ((pkt->pkt_cdbp[0] == SCMD_PRIN) ||
1418            (pkt->pkt_cdbp[0] == SCMD_PROUT)) {
1419                /*
1420                 * currently this thread only handles running PGR
1421                 * commands, so don't bother creating it unless
1422                 * something interesting is going to happen (like
1423                 * either a PGR out, or a PGR in with enough space
1424                 * to hold the keys that are getting returned)
1425                 */
1426                mutex_enter(&vlun->svl_mutex);
1427                if (((vlun->svl_flags & VLUN_TASK_D_ALIVE_FLG) == 0) &&
1428                    (pkt->pkt_cdbp[0] == SCMD_PROUT)) {
1429                        vlun->svl_taskq = taskq_create("vlun_pgr_task_daemon",
1430                            1, MINCLSYSPRI, 1, 4, 0);
1431                        vlun->svl_flags |= VLUN_TASK_D_ALIVE_FLG;
1432                }
1433                mutex_exit(&vlun->svl_mutex);
1434                if ((pkt->pkt_cdbp[0] == SCMD_PROUT) &&
1435                    (((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_REGISTER) ||
1436                    ((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_R_AND_IGNORE))) {
1437                        if (rval = vhci_pgr_register_start(vlun, pkt)) {
1438                                /* an error */
1439                                sema_v(&vlun->svl_pgr_sema);
1440                                return (rval);
1441                        }
1442                }
1443        }

1445        /*
1446         * SCSI-II RESERVE cmd is not expected in polled mode.
```

```
1447          * If this changes it needs to be handled for the polled scenario.
1448          */
1449         flags = vpkt->vpkt_hba_pkt->pkt_flags;

1451         /*
1452          * Set the path_instance *before* sending the scsi_pkt down the path
1453          * to mpxio's pHCI so that additional path abstractions at a pHCI
1454          * level (like maybe iSCSI at some point in the future) can update
1455          * the path_instance.
1456          */
1457         if (scsi_pkt_allocated_correctly(vpkt->vpkt_hba_pkt))
1458                 vpkt->vpkt_hba_pkt->pkt_path_instance =
1459                     mdi_pi_get_path_instance(vpkt->vpkt_path);

1461         rval = scsi_transport(vpkt->vpkt_hba_pkt);
1462         if (rval == TRAN_ACCEPT) {
1463                 if (flags & FLAG_NOINTR) {
1464                         struct scsi_pkt *tpkt = vpkt->vpkt_tgt_pkt;
1465                         struct scsi_pkt *pkt = vpkt->vpkt_hba_pkt;

1467                         ASSERT(tpkt != NULL);
1468                         *(tpkt->pkt_scbp) = *(pkt->pkt_scbp);
1469                         tpkt->pkt_resid = pkt->pkt_resid;
1470                         tpkt->pkt_state = pkt->pkt_state;
1471                         tpkt->pkt_statistics = pkt->pkt_statistics;
1472                         tpkt->pkt_reason = pkt->pkt_reason;

1474                         if ((*(pkt->pkt_scbp) == STATUS_CHECK) &&
1475                             (pkt->pkt_state & STATE_ARQ_DONE)) {
1476                                 bcopy(pkt->pkt_scbp, tpkt->pkt_scbp,
1477                                     vpkt->vpkt_tgt_init_scblen);
1478                         }

1480                         VHCI_DECR_PATH_CMDCOUNT(svp);
1481                         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {
1482                                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
1483                                 vpkt->vpkt_hba_pkt = NULL;
1484                                 if (vpkt->vpkt_path) {
1485                                         mdi_rele_path(vpkt->vpkt_path);
1486                                         vpkt->vpkt_path = NULL;
1487                                 }
1488                         }
1489                         /*
1490                          * This path will not automatically retry pkts
1491                          * internally, therefore, vpkt_org_vpkt should
1492                          * never be set.
1493                          */
1494                         ASSERT(vpkt->vpkt_org_vpkt == NULL);
1495                         scsi_hba_pkt_comp(tpkt);
1496                 }
1497                 return (rval);
1498         } else if ((pkt->pkt_cdbp[0] == SCMD_PROUT) &&
1499             (((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_REGISTER) ||
1500             ((pkt->pkt_cdbp[1] & 0x1f) == VHCI_PROUT_R_AND_IGNORE))) {
1501                 /* the command exited with bad status */
1502                 sema_v(&vlun->svl_pgr_sema);
1503         } else if (vpkt->vpkt_tgt_pkt->pkt_cdbp[0] == SCMD_PRIN) {
1504                 /* the command exited with bad status */
1505                 sema_v(&vlun->svl_pgr_sema);
1506         } else if (pkt_reserve_cmd) {
1507                 VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1508                     "!vhci_scsi_start: reserve failed vlun 0x%p",
1509                     (void *)vlun));
1510                 vlun->svl_flags &= ~VLUN_QUIESCED_FLG;
1511                 if (restore_lbp)
1512                         (void) mdi_set_lb_policy(cdip, lbp);
```



```
1513         }

1515         ASSERT(vpkt->vpkt_hba_pkt != NULL);
1516         VHCI_DECR_PATH_CMDCOUNT(svp);

1518         /* Do not destroy phci packet information for PKT_DMA_PARTIAL */
1519         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) == 0) {
1520                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
1521                 vpkt->vpkt_hba_pkt = NULL;
1522                 if (vpkt->vpkt_path) {
1523                         MDI_PI_ERRSTAT(vpkt->vpkt_path, MDI_PI_TRANSERR);
1524                         mdi_rele_path(vpkt->vpkt_path);
1525                         vpkt->vpkt_path = NULL;
1526                 }
1527         }
1528         return (TRAN_BUSY);
1529 }

1531 /*
1532  * Function name : vhci_scsi_reset()
1533  *
1534  * Return Values : 0 - reset failed
1535  *                 1 - reset succeeded
1536  */

1538 /* ARGSUSED */
1539 static int
1540 vhci_scsi_reset(struct scsi_address *ap, int level)
1541 {
1542         int rval = 0;

1544         cmn_err(CE_WARN, "!vhci_scsi_reset 0x%x", level);
1545         if ((level == RESET_TARGET) || (level == RESET_LUN)) {
1546                 return (vhci_scsi_reset_target(ap, level, TRUE));
1547         } else if (level == RESET_ALL) {
1548                 return (vhci_scsi_reset_bus(ap));
1549         }

1551         return (rval);
1552 }

1554 /*
1555  * vhci_recovery_reset:
1556  *      Issues reset to the device
1557  * Input:
1558  *      vlun - vhci lun pointer of the device
1559  *      ap - address of the device
1560  *      select_path:
1561  *              If select_path is FALSE, then the address specified in ap is
1562  *              the path on which reset will be issued.
1563  *              If select_path is TRUE, then path is obtained by calling
1564  *              mdi_select_path.
1565  *
1566  *      recovery_depth:
1567  *              Caller can specify the level of reset.
1568  *              VHCI_DEPTH_LUN -
1569  *                      Issues LUN RESET if device supports lun reset.
1570  *              VHCI_DEPTH_TARGET -
1571  *                      If Lun Reset fails or the device does not support
1572  *                      Lun Reset, issues TARGET RESET
1573  *              VHCI_DEPTH_ALL -
1574  *                      If Lun Reset fails or the device does not support
1575  *                      Lun Reset, issues TARGET RESET.
1576  *                      If TARGET RESET does not succeed, issues Bus Reset.
1577  */
```

```
1579 static int
1580 vhci_recovery_reset(scsi_vhci_lun_t *vlun, struct scsi_address *ap,
1581         uint8_t select_path, uint8_t recovery_depth)
1582 {
1583         int     ret = 0;

1585         ASSERT(ap != NULL);

1587         if (vlun && vlun->svl_support_lun_reset == 1) {
1588                 ret = vhci_scsi_reset_target(ap, RESET_LUN,
1589                     select_path);
1590         }

1592         recovery_depth--;

1594         if ((ret == 0) && recovery_depth) {
1595                 ret = vhci_scsi_reset_target(ap, RESET_TARGET,
1596                     select_path);
1597                 recovery_depth--;
1598         }

1600         if ((ret == 0) && recovery_depth) {
1601                 (void) scsi_reset(ap, RESET_ALL);
1602         }

1604         return (ret);
1605 }

1607 /*
1608  * Note: The scsi_address passed to this routine could be the scsi_address
1609  * for the virtual device or the physical device. No assumptions should be
1610  * made in this routine about the contents of the ap structure.
1611  * Further, note that the child dip would be the dip of the ssd node regardless
1612  * of the scsi_address passed in.
1613  */
1614 static int
1615 vhci_scsi_reset_target(struct scsi_address *ap, int level, uint8_t select_path)
1616 {
1617         dev_info_t              *vdip, *cdip;
1618         mdi_pathinfo_t          *pip = NULL;
1619         mdi_pathinfo_t          *npip = NULL;
1620         int                     rval = -1;
1621         scsi_vhci_priv_t        *svp = NULL;
1622         struct scsi_address     *pap = NULL;
1623         scsi_hba_tran_t         *hba = NULL;
1624         int                     sps;
1625         struct scsi_vhci        *vhci = NULL;

1627         if (select_path != TRUE) {
1628                 ASSERT(ap != NULL);
1629                 if (level == RESET_LUN) {
1630                         hba = ap->a_hba_tran;
1631                         ASSERT(hba != NULL);
1632                         return (hba->tran_reset(ap, RESET_LUN));
1633                 }
1634                 return (scsi_reset(ap, level));
1635         }

1637         cdip = ADDR2DIP(ap);
1638         ASSERT(cdip != NULL);
1639         vdip = ddi_get_parent(cdip);
1640         ASSERT(vdip != NULL);
1641         vhci = ddi_get_soft_state(vhci_softstate, ddi_get_instance(vdip));
1642         ASSERT(vhci != NULL);

1644         rval = mdi_select_path(cdip, NULL, MDI_SELECT_ONLINE_PATH, NULL, &pip);
```



```
1645         if ((rval != MDI_SUCCESS) || (pip == NULL)) {
1646                 VHCI_DEBUG(2, (CE_WARN, NULL, "!vhci_scsi_reset_target: "
1647                     "Unable to get a path, dip 0x%p", (void *)cdip));
1648                 return (0);
1649         }
1650 again:
1651         svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(pip);
1652         if (svp == NULL) {
1653                 VHCI_DEBUG(2, (CE_WARN, NULL, "!vhci_scsi_reset_target: "
1654                     "priv is NULL, pip 0x%p", (void *)pip));
1655                 mdi_rele_path(pip);
1656                 return (0);
1657         }

1659         if (svp->svp_psd == NULL) {
1660                 VHCI_DEBUG(2, (CE_WARN, NULL, "!vhci_scsi_reset_target: "
1661                     "psd is NULL, pip 0x%p, svp 0x%p",
1662                     (void *)pip, (void *)svp));
1663                 mdi_rele_path(pip);
1664                 return (0);
1665         }

1667         pap = &svp->svp_psd->sd_address;
1668         hba = pap->a_hba_tran;

1670         ASSERT(pap != NULL);
1671         ASSERT(hba != NULL);

1673         if (hba->tran_reset != NULL) {
1674                 if (hba->tran_reset(pap, level) == 0) {
1675                         vhci_log(CE_WARN, vdip, "!%s%d: "
1676                             "path %s, reset %d failed",
1677                             ddi_driver_name(cdip), ddi_get_instance(cdip),
1678                             mdi_pi_spathname(pip), level);

1680                         /*
1681                          * Select next path and issue the reset, repeat
1682                          * until all paths are exhausted
1683                          */
1684                         sps = mdi_select_path(cdip, NULL,
1685                             MDI_SELECT_ONLINE_PATH, pip, &npip);
1686                         if ((sps != MDI_SUCCESS) || (npip == NULL)) {
1687                                 mdi_rele_path(pip);
1688                                 return (0);
1689                         }
1690                         mdi_rele_path(pip);
1691                         pip = npip;
1692                         goto again;
1693                 }
1694                 mdi_rele_path(pip);
1695                 mutex_enter(&vhci->vhci_mutex);
1696                 scsi_hba_reset_notify_callback(&vhci->vhci_mutex,
1697                     &vhci->vhci_reset_notify_listf);
1698                 mutex_exit(&vhci->vhci_mutex);
1699                 VHCI_DEBUG(6, (CE_NOTE, NULL, "!vhci_scsi_reset_target: "
1700                     "reset %d sent down pip:%p for cdip:%p\n", level,
1701                     (void *)pip, (void *)cdip));
1702                 return (1);
1703         }
1704         mdi_rele_path(pip);
1705         return (0);
1706 }


1709 /* ARGSUSED */
1710 static int
```

```
vhci_scsi_reset_bus(struct scsi_address *ap)
{
        return (1);
}


/*
 * called by vhci_getcap and vhci_setcap to get and set (respectively)
 * SCSI capabilities
 */
/* ARGSUSED */
static int
vhci_commoncap(struct scsi_address *ap, char *cap,
    int val, int tgtonly, int doset)
{
        struct scsi_vhci                *vhci = ADDR2VHCI(ap);
        struct scsi_vhci_lun            *vlun = ADDR2VLUN(ap);
        int                     cidx;
        int                     rval = 0;

        if (cap == (char *)0) {
                VHCI_DEBUG(3, (CE_WARN, vhci->vhci_dip,
                    "!vhci_commoncap: invalid arg"));
                return (rval);
        }

        if (vlun == NULL) {
                VHCI_DEBUG(3, (CE_WARN, vhci->vhci_dip,
                    "!vhci_commoncap: vlun is null"));
                return (rval);
        }

        if ((cidx = scsi_hba_lookup_capstr(cap)) == -1) {
                return (UNDEFINED);
        }

        /*
         * Process setcap request.
         */
        if (doset) {
                /*
                 * At present, we can only set binary (0/1) values
                 */
                switch (cidx) {
                case SCSI_CAP_ARQ:
                        if (val == 0) {
                                rval = 0;
                        } else {
                                rval = 1;
                        }
                        break;

                case SCSI_CAP_LUN_RESET:
                        if (tgtonly == 0) {
                                VHCI_DEBUG(1, (CE_WARN, vhci->vhci_dip,
                                    "scsi_vhci_setcap: "
                                    "Returning error since whom = 0"));
                                rval = -1;
                                break;
                        }
                        /*
                         * Set the capability accordingly.
                         */
                        mutex_enter(&vlun->svl_mutex);
                        vlun->svl_support_lun_reset = val;
                        rval = val;
```



```
                        mutex_exit(&vlun->svl_mutex);
                        break;

                case SCSI_CAP_SECTOR_SIZE:
                        mutex_enter(&vlun->svl_mutex);
                        vlun->svl_sector_size = val;
                        vlun->svl_setcap_done = 1;
                        mutex_exit(&vlun->svl_mutex);
                        (void) vhci_pHCI_cap(ap, cap, val, tgtonly, NULL);

                        /* Always return success */
                        rval = 1;
                        break;

                default:
                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
                            "!vhci_setcap: unsupported %d", cidx));
                        rval = UNDEFINED;
                        break;
                }

                VHCI_DEBUG(6, (CE_NOTE, vhci->vhci_dip,
                    "!set cap: cap=%s, val/tgtonly/doset/rval = "
                    "0x%x/0x%x/0x%x/%d\n",
                    cap, val, tgtonly, doset, rval));

        } else {
                /*
                 * Process getcap request.
                 */
                switch (cidx) {
                case SCSI_CAP_DMA_MAX:
                        /*
                         * For X86 this capability is caught in scsi_ifgetcap().
                         * XXX Should this be getting the value from the pHCI?
                         */
                        rval = (int)VHCI_DMA_MAX_XFER_CAP;
                        break;

                case SCSI_CAP_INITIATOR_ID:
                        rval = 0x00;
                        break;

                case SCSI_CAP_ARQ:
                case SCSI_CAP_RESET_NOTIFICATION:
                case SCSI_CAP_TAGGED_QING:
                        rval = 1;
                        break;

                case SCSI_CAP_SCSI_VERSION:
                        rval = 3;
                        break;

                case SCSI_CAP_INTERCONNECT_TYPE:
                        rval = INTERCONNECT_FABRIC;
                        break;

                case SCSI_CAP_LUN_RESET:
                        /*
                         * scsi_vhci will always return success for LUN reset.
                         * When request for doing LUN reset comes
                         * through scsi_reset entry point, at that time attempt
                         * will be made to do reset through all the possible
                         * paths.
                         */
                        mutex_enter(&vlun->svl_mutex);
```

```
1843                        rval = vlun->svl_support_lun_reset;
1844                        mutex_exit(&vlun->svl_mutex);
1845                        VHCI_DEBUG(4, (CE_WARN, vhci->vhci_dip,
1846                            "scsi_vhci_getcap:"
1847                            "Getting the Lun reset capability %d", rval));
1848                        break;

1850                case SCSI_CAP_SECTOR_SIZE:
1851                        mutex_enter(&vlun->svl_mutex);
1852                        rval = vlun->svl_sector_size;
1853                        mutex_exit(&vlun->svl_mutex);
1854                        break;

1856                case SCSI_CAP_CDB_LEN:
1857                        rval = VHCI_SCSI_CDB_SIZE;
1858                        break;

1860                case SCSI_CAP_DMA_MAX_ARCH:
1861                        /*
1862                         * For X86 this capability is caught in scsi_ifgetcap().
1863                         * XXX Should this be getting the value from the pHCI?
1864                         */
1865                        rval = 0;
1866                        break;

1868                default:
1869                        VHCI_DEBUG(6, (CE_WARN, vhci->vhci_dip,
1870                            "!vhci_getcap: unsupported %d", cidx));
1871                        rval = UNDEFINED;
1872                        break;
1873                }

1875                VHCI_DEBUG(6, (CE_NOTE, vhci->vhci_dip,
1876                    "!get cap: cap=%s, val/tgtonly/doset/rval = "
1877                    "0x%x/0x%x/0x%x/%d\n",
1878                    cap, val, tgtonly, doset, rval));
1879        }
1880        return (rval);
1881 }


1884 /*
1885  * Function name : vhci_scsi_getcap()
1886  *
1887  */
1888 static int
1889 vhci_scsi_getcap(struct scsi_address *ap, char *cap, int whom)
1890 {
1891        return (vhci_commoncap(ap, cap, 0, whom, 0));
1892 }

1894 static int
1895 vhci_scsi_setcap(struct scsi_address *ap, char *cap, int value, int whom)
1896 {
1897        return (vhci_commoncap(ap, cap, value, whom, 1));
1898 }

1900 /*
1901  * Function name : vhci_scsi_abort()
1902  */
1903 /* ARGSUSED */
1904 static int
1905 vhci_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
1906 {
1907        return (0);
1908 }
```



```
1910 /*
1911  * Function name : vhci_scsi_init_pkt
1912  *
1913  * Return Values : pointer to scsi_pkt, or NULL
1914  */
1915 /* ARGSUSED */
1916 static struct scsi_pkt *
1917 vhci_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
1918        struct buf *bp, int cmdlen, int statuslen, int tgtlen,
1919        int flags, int (*callback)(caddr_t), caddr_t arg)
1920 {
1921        struct scsi_vhci        *vhci = ADDR2VHCI(ap);
1922        struct vhci_pkt         *vpkt;
1923        int                     rval;
1924        int                     newpkt = 0;
1925        struct scsi_pkt         *pktp;


1928        if (pkt == NULL) {
1929                if (cmdlen > VHCI_SCSI_CDB_SIZE) {
1930                        if ((cmdlen != VHCI_SCSI_OSD_CDB_SIZE) ||
1931                            ((flags & VHCI_SCSI_OSD_PKT_FLAGS) !=
1932                            VHCI_SCSI_OSD_PKT_FLAGS)) {
1933                                VHCI_DEBUG(1, (CE_NOTE, NULL,
1934                                    "!init pkt: cdb size not supported\n"));
1935                                return (NULL);
1936                        }
1937                }

1939                pktp = scsi_hba_pkt_alloc(vhci->vhci_dip,
1940                    ap, cmdlen, statuslen, tgtlen, sizeof (*vpkt), callback,
1941                    arg);

1943                if (pktp == NULL) {
1944                        return (NULL);
1945                }

1947                /* Get the vhci's private structure */
1948                vpkt = (struct vhci_pkt *)(pktp->pkt_ha_private);
1949                ASSERT(vpkt);

1951                /* Save the target driver's packet */
1952                vpkt->vpkt_tgt_pkt = pktp;

1954                /*
1955                 * Save pkt_tgt_init_pkt fields if deferred binding
1956                 * is needed or for other purposes.
1957                 */
1958                vpkt->vpkt_tgt_init_pkt_flags = flags;
1959                vpkt->vpkt_flags = (callback == NULL_FUNC) ? CFLAG_NOWAIT : 0;
1960                vpkt->vpkt_state = VHCI_PKT_IDLE;
1961                vpkt->vpkt_tgt_init_cdblen = cmdlen;
1962                vpkt->vpkt_tgt_init_scblen = statuslen;
1963                newpkt = 1;
1964        } else { /* pkt not NULL */
1965                vpkt = pkt->pkt_ha_private;
1966        }

1968        VHCI_DEBUG(8, (CE_NOTE, NULL, "vhci_scsi_init_pkt "
1969            "vpkt %p flags %x\n", (void *)vpkt, flags));

1971        /* Clear any stale error flags */
1972        if (bp) {
1973                bioerror(bp, 0);
1974        }
```

```
1976         vpkt->vpkt_tgt_init_bp = bp;

1978         if (flags & PKT_DMA_PARTIAL) {

1980                 /*
1981                  * Immediate binding is needed.
1982                  * Target driver may not set this flag in next invocation.
1983                  * vhci has to remember this flag was set during first
1984                  * invocation of vhci_scsi_init_pkt.
1985                  */
1986                 vpkt->vpkt_flags |= CFLAG_DMA_PARTIAL;
1987         }

1989         if (vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) {

1991                 /*
1992                  * Re-initialize some of the target driver packet state
1993                  * information.
1994                  */
1995                 vpkt->vpkt_tgt_pkt->pkt_state = 0;
1996                 vpkt->vpkt_tgt_pkt->pkt_statistics = 0;
1997                 vpkt->vpkt_tgt_pkt->pkt_reason = 0;

1999                 /*
2000                  * Binding a vpkt->vpkt_path for this IO at init_time.
2001                  * If an IO error happens later, target driver will clear
2002                  * this vpkt->vpkt_path binding before re-init IO again.
2003                  */
2004                 VHCI_DEBUG(8, (CE_NOTE, NULL,
2005                     "vhci_scsi_init_pkt: calling v_b_t %p, newpkt %d\n",
2006                     (void *)vpkt, newpkt));
2007                 if (pkt && vpkt->vpkt_hba_pkt) {
2008                         VHCI_DEBUG(4, (CE_NOTE, NULL,
2009                             "v_s_i_p calling update_pHCI_pkt resid %ld\n",
2010                             pkt->pkt_resid));
2011                         vhci_update_pHCI_pkt(vpkt, pkt);
2012                 }
2013                 if (callback == SLEEP_FUNC) {
2014                         rval = vhci_bind_transport(
2015                             ap, vpkt, flags, callback);
2016                 } else {
2017                         rval = vhci_bind_transport(
2018                             ap, vpkt, flags, NULL_FUNC);
2019                 }
2020                 VHCI_DEBUG(8, (CE_NOTE, NULL,
2021                     "vhci_scsi_init_pkt: v_b_t called 0x%p rval 0x%x\n",
2022                     (void *)vpkt, rval));
2023                 if (bp) {
2024                         if (rval == TRAN_FATAL_ERROR) {
2025                                 /*
2026                                  * No paths available. Could not bind
2027                                  * any pHCI. Setting EFAULT as a way
2028                                  * to indicate no DMA is mapped.
2029                                  */
2030                                 bioerror(bp, EFAULT);
2031                         } else {
2032                                 /*
2033                                  * Do not indicate any pHCI errors to
2034                                  * target driver otherwise.
2035                                  */
2036                                 bioerror(bp, 0);
2037                         }
2038                 }
2039                 if (rval != TRAN_ACCEPT) {
2040                         VHCI_DEBUG(8, (CE_NOTE, NULL,
```



```
2041                             "vhci_scsi_init_pkt: "
2042                             "v_b_t failed 0x%p newpkt %x\n",
2043                             (void *)vpkt, newpkt));
2044                         if (newpkt) {
2045                                 scsi_hba_pkt_free(ap,
2046                                     vpkt->vpkt_tgt_pkt);
2047                         }
2048                         return (NULL);
2049                 }
2050                 ASSERT(vpkt->vpkt_hba_pkt != NULL);
2051                 ASSERT(vpkt->vpkt_path != NULL);

2053                 /* Update the resid for the target driver */
2054                 vpkt->vpkt_tgt_pkt->pkt_resid =
2055                     vpkt->vpkt_hba_pkt->pkt_resid;
2056         }

2058         return (vpkt->vpkt_tgt_pkt);
2059 }

2061 /*
2062  * Function name : vhci_scsi_destroy_pkt
2063  *
2064  * Return Values : none
2065  */
2066 static void
2067 vhci_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
2068 {
2069         struct vhci_pkt         *vpkt = (struct vhci_pkt *)pkt->pkt_ha_private;

2071         VHCI_DEBUG(8, (CE_NOTE, NULL,
2072             "vhci_scsi_destroy_pkt: vpkt 0x%p\n", (void *)vpkt));

2074         vpkt->vpkt_tgt_init_pkt_flags = 0;
2075         if (vpkt->vpkt_hba_pkt) {
2076                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
2077                 vpkt->vpkt_hba_pkt = NULL;
2078         }
2079         if (vpkt->vpkt_path) {
2080                 mdi_rele_path(vpkt->vpkt_path);
2081                 vpkt->vpkt_path = NULL;
2082         }

2084         ASSERT(vpkt->vpkt_state != VHCI_PKT_ISSUED);
2085         scsi_hba_pkt_free(ap, vpkt->vpkt_tgt_pkt);
2086 }

2088 /*
2089  * Function name : vhci_scsi_dmafree()
2090  *
2091  * Return Values : none
2092  */
2093 /*ARGSUSED*/
2094 static void
2095 vhci_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
2096 {
2097         struct vhci_pkt *vpkt = (struct vhci_pkt *)pkt->pkt_ha_private;

2099         VHCI_DEBUG(6, (CE_NOTE, NULL,
2100             "vhci_scsi_dmafree: vpkt 0x%p\n", (void *)vpkt));

2102         ASSERT(vpkt != NULL);
2103         if (vpkt->vpkt_hba_pkt) {
2104                 scsi_destroy_pkt(vpkt->vpkt_hba_pkt);
2105                 vpkt->vpkt_hba_pkt = NULL;
2106         }
```

```
2107         if (vpkt->vpkt_path) {
2108                 mdi_rele_path(vpkt->vpkt_path);
2109                 vpkt->vpkt_path = NULL;
2110         }
2111 }

2113 /*
2114  * Function name : vhci_scsi_sync_pkt()
2115  *
2116  * Return Values : none
2117  */
2118 /*ARGSUSED*/
2119 static void
2120 vhci_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
2121 {
2122         struct vhci_pkt *vpkt = (struct vhci_pkt *)pkt->pkt_ha_private;

2124         ASSERT(vpkt != NULL);
2125         if (vpkt->vpkt_hba_pkt) {
2126                 scsi_sync_pkt(vpkt->vpkt_hba_pkt);
2127         }
2128 }

2130 /*
2131  * routine for reset notification setup, to register or cancel.
2132  */
2133 static int
2134 vhci_scsi_reset_notify(struct scsi_address *ap, int flag,
2135     void (*callback)(caddr_t), caddr_t arg)
2136 {
2137         struct scsi_vhci *vhci = ADDR2VHCI(ap);
2138         return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
2139             &vhci->vhci_mutex, &vhci->vhci_reset_notify_listf));
2140 }

2142 static int
2143 vhci_scsi_get_name_bus_addr(struct scsi_device *sd,
2144     char *name, int len, int bus_addr)
2145 {
2146         dev_info_t              *cdip;
2147         char                    *guid;
2148         scsi_vhci_lun_t         *vlun;

2150         ASSERT(sd != NULL);
2151         ASSERT(name != NULL);

2153         *name = 0;
2154         cdip = sd->sd_dev;

2156         ASSERT(cdip != NULL);

2158         if (mdi_component_is_client(cdip, NULL) != MDI_SUCCESS)
2159                 return (1);

2161         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, cdip, PROPFLAGS,
2162             MDI_CLIENT_GUID_PROP, &guid) != DDI_SUCCESS)
2163                 return (1);

2165         /*
2166          * Message is "sd# at scsi_vhci0: unit-address <guid>: <bus_addr>".
2167          *      <guid>          bus_addr argument == 0
2168          *      <bus_addr>      bus_addr argument != 0
2169          * Since the <guid> is already provided with unit-address, we just
2170          * provide failover module in <bus_addr> to keep output shorter.
2171          */
2172         vlun = ADDR2VLUN(&sd->sd_address);
```



```
2173         if (bus_addr == 0) {
2174                 /* report the guid:  */
2175                 (void) snprintf(name, len, "g%s", guid);
2176         } else if (vlun && vlun->svl_fops_name) {
2177                 /* report the name of the failover module */
2178                 (void) snprintf(name, len, "%s", vlun->svl_fops_name);
2179         }

2181         ddi_prop_free(guid);
2182         return (1);
2183 }

2185 static int
2186 vhci_scsi_get_bus_addr(struct scsi_device *sd, char *name, int len)
2187 {
2188         return (vhci_scsi_get_name_bus_addr(sd, name, len, 1));
2189 }

2191 static int
2192 vhci_scsi_get_name(struct scsi_device *sd, char *name, int len)
2193 {
2194         return (vhci_scsi_get_name_bus_addr(sd, name, len, 0));
2195 }

2197 /*
2198  * Return a pointer to the guid part of the devnm.
2199  * devnm format is "nodename@busaddr", busaddr format is "gGUID".
2200  */
2201 static char *
2202 vhci_devnm_to_guid(char *devnm)
2203 {
2204         char *cp = devnm;

2206         if (devnm == NULL)
2207                 return (NULL);

2209         while (*cp != '\0' && *cp != '@')
2210                 cp++;
2211         if (*cp == '@' && *(cp + 1) == 'g')
2212                 return (cp + 2);
2213         return (NULL);
2214 }

2216 static int
2217 vhci_bind_transport(struct scsi_address *ap, struct vhci_pkt *vpkt, int flags,
2218     int (*func)(caddr_t))
2219 {
2220         struct scsi_vhci        *vhci = ADDR2VHCI(ap);
2221         dev_info_t              *cdip = ADDR2DIP(ap);
2222         mdi_pathinfo_t          *pip = NULL;
2223         mdi_pathinfo_t          *npip = NULL;
2224         scsi_vhci_priv_t        *svp = NULL;
2225         struct scsi_device      *psd = NULL;
2226         struct scsi_address     *address = NULL;
2227         struct scsi_pkt         *pkt = NULL;
2228         int                     rval = -1;
2229         int                     pgr_sema_held = 0;
2230         int                     held;
2231         int                     mps_flag = MDI_SELECT_ONLINE_PATH;
2232         struct scsi_vhci_lun    *vlun;
  24         time_t                  tnow;
2233         int                     path_instance = 0;

2235         vlun = ADDR2VLUN(ap);
2236         ASSERT(vlun != 0);
```

```
2238         if ((vpkt->vpkt_tgt_pkt->pkt_cdbp[0] == SCMD_PROUT) &&
2239             (((vpkt->vpkt_tgt_pkt->pkt_cdbp[1] & 0x1f) ==
2240             VHCI_PROUT_REGISTER) ||
2241             ((vpkt->vpkt_tgt_pkt->pkt_cdbp[1] & 0x1f) ==
2242             VHCI_PROUT_R_AND_IGNORE))) {
2243                 if (!sema_tryp(&vlun->svl_pgr_sema))
2244                         return (TRAN_BUSY);
2245                 pgr_sema_held = 1;
2246                 if (vlun->svl_first_path != NULL) {
2247                         rval = mdi_select_path(cdip, NULL,
2248                             MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH,
2249                             NULL, &pip);
2250                         if ((rval != MDI_SUCCESS) || (pip == NULL)) {
2251                                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2252                                     "vhci_bind_transport: path select fail\n"));
2253                         } else {
2254                                 npip = pip;
2255                                 do {
2256                                         if (npip == vlun->svl_first_path) {
2257                                                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2258                                                     "vhci_bind_transport: "
2259                                                     "valid first path 0x%p\n",
2260                                                     (void *)
2261                                                     vlun->svl_first_path));
2262                                                 pip = vlun->svl_first_path;
2263                                                 goto bind_path;
2264                                         }
2265                                         pip = npip;
2266                                         rval = mdi_select_path(cdip, NULL,
2267                                             MDI_SELECT_ONLINE_PATH |
2268                                             MDI_SELECT_STANDBY_PATH,
2269                                             pip, &npip);
2270                                         mdi_rele_path(pip);
2271                                 } while ((rval == MDI_SUCCESS) &&
2272                                     (npip != NULL));
2273                         }
2274                 }

2276                 if (vlun->svl_first_path) {
2277                         VHCI_DEBUG(4, (CE_NOTE, NULL,
2278                             "vhci_bind_transport: invalid first path 0x%p\n",
2279                             (void *)vlun->svl_first_path));
2280                         vlun->svl_first_path = NULL;
2281                 }
2282         } else if (vpkt->vpkt_tgt_pkt->pkt_cdbp[0] == SCMD_PRIN) {
2283                 if ((vpkt->vpkt_state & VHCI_PKT_THRU_TASKQ) == 0) {
2284                         if (!sema_tryp(&vlun->svl_pgr_sema))
2285                                 return (TRAN_BUSY);
2286                 }
2287                 pgr_sema_held = 1;
2288         }

2290         /*
2291          * If the path is already bound for PKT_PARTIAL_DMA case,
2292          * try to use the same path.
2293          */
2294         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) && vpkt->vpkt_path) {
2295                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2296                     "vhci_bind_transport: PKT_PARTIAL_DMA "
2297                     "vpkt 0x%p, path 0x%p\n",
2298                     (void *)vpkt, (void *)vpkt->vpkt_path));
2299                 pip = vpkt->vpkt_path;
2300                 goto bind_path;
2301         }

2303         /*
```



```
2304          * Get path_instance. Non-zero with FLAG_PKT_PATH_INSTANCE set
2305          * indicates that mdi_select_path should be called to select a
2306          * specific instance.
2307          *
2308          * NB: Condition pkt_path_instance reference on proper allocation.
2309          */
2310         if ((vpkt->vpkt_tgt_pkt->pkt_flags & FLAG_PKT_PATH_INSTANCE) &&
2311             scsi_pkt_allocated_correctly(vpkt->vpkt_tgt_pkt)) {
2312                 path_instance = vpkt->vpkt_tgt_pkt->pkt_path_instance;
2313         }

2315         /*
2316          * If reservation is active bind the transport directly to the pip
2317          * with the reservation.
2318          */
2319         if (vpkt->vpkt_hba_pkt == NULL) {
2320                 if (vlun->svl_flags & VLUN_RESERVE_ACTIVE_FLG) {
2321                         if (MDI_PI_IS_ONLINE(vlun->svl_resrv_pip)) {
2322                                 pip = vlun->svl_resrv_pip;
2323                                 mdi_hold_path(pip);
2324                                 vlun->svl_waiting_for_activepath = 0;
2325                                 rval = MDI_SUCCESS;
2326                                 goto bind_path;
2327                         } else {
2328                                 if (pgr_sema_held) {
2329                                         sema_v(&vlun->svl_pgr_sema);
2330                                 }
2331                                 return (TRAN_BUSY);
2332                         }
2333                 }
2334 try_again:
2335                 rval = mdi_select_path(cdip, vpkt->vpkt_tgt_init_bp,
2336                     path_instance ? MDI_SELECT_PATH_INSTANCE : 0,
2337                     (void *)(intptr_t)path_instance, &pip);
2338                 if (rval == MDI_BUSY) {
2339                         if (pgr_sema_held) {
2340                                 sema_v(&vlun->svl_pgr_sema);
2341                         }
2342                         return (TRAN_BUSY);
2343                 } else if (rval == MDI_DEVI_ONLINING) {
2344                         /*
2345                          * if we are here then we are in the midst of
2346                          * an attach/probe of the client device.
2347                          * We attempt to bind to ONLINE path if available,
2348                          * else it is OK to bind to a STANDBY path (instead
2349                          * of triggering a failover) because IO associated
2350                          * with attach/probe (eg. INQUIRY, block 0 read)
2351                          * are completed by targets even on passive paths
2352                          * If no ONLINE paths available, it is important
2353                          * to set svl_waiting_for_activepath for two
2354                          * reasons: (1) avoid sense analysis in the
2355                          * "external failure detection" codepath in
2356                          * vhci_intr().  Failure to do so will result in
2357                          * infinite loop (unless an ONLINE path becomes
2358                          * available at some point) (2) avoid
2359                          * unnecessary failover (see "---Waiting For Active
2360                          * Path---" comment below).
2361                          */
2362                         VHCI_DEBUG(1, (CE_NOTE, NULL, "!%p in onlining "
2363                             "state\n", (void *)cdip));
2364                         pip = NULL;
2365                         rval = mdi_select_path(cdip, vpkt->vpkt_tgt_init_bp,
2366                             mps_flag, NULL, &pip);
2367                         if ((rval != MDI_SUCCESS) || (pip == NULL)) {
2368                                 if (vlun->svl_waiting_for_activepath == 0) {
2369                                         vlun->svl_waiting_for_activepath = 1;
```

```
2370                                    vlun->svl_wfa_time = gethrtime();
 162                                    vlun->svl_wfa_time = ddi_get_time();
2371                            }
2372                            mps_flag |= MDI_SELECT_STANDBY_PATH;
2373                            rval = mdi_select_path(cdip,
2374                                vpkt->vpkt_tgt_init_bp,
2375                                mps_flag, NULL, &pip);
2376                            if ((rval != MDI_SUCCESS) || (pip == NULL)) {
2377                                    if (pgr_sema_held) {
2378                                            sema_v(&vlun->svl_pgr_sema);
2379                                    }
2380                                    return (TRAN_FATAL_ERROR);
2381                            }
2382                            goto bind_path;
2383                    }
2384            } else if ((rval == MDI_FAILURE) ||
2385                ((rval == MDI_NOPATH) && (path_instance))) {
2386                    if (pgr_sema_held) {
2387                            sema_v(&vlun->svl_pgr_sema);
2388                    }
2389                    return (TRAN_FATAL_ERROR);
2390            }

2392             if ((pip == NULL) || (rval == MDI_NOPATH)) {
2393                    while (vlun->svl_waiting_for_activepath) {
2394                            /*
2395                             * ---Waiting For Active Path---
2396                             * This device was discovered across a
2397                             * passive path; lets wait for a little
2398                             * bit, hopefully an active path will
2399                             * show up obviating the need for a
2400                             * failover
2401                             */
2402                            if ((gethrtime() - vlun->svl_wfa_time) >=
2403                                (60 * NANOSEC)) {
 194                            tnow = ddi_get_time();
 195                            if (tnow - vlun->svl_wfa_time >= 60) {
2404                                    vlun->svl_waiting_for_activepath = 0;
2405                            } else {
2406                                    drv_usecwait(1000);
2407                                    if (vlun->svl_waiting_for_activepath
2408                                        == 0) {
2409                                            /*
2410                                             * an active path has come
2411                                             * online!
2412                                             */
2413                                            goto try_again;
2414                                    }
2415                            }
2416                    }
2417                    VHCI_HOLD_LUN(vlun, VH_NOSLEEP, held);
2418                    if (!held) {
2419                            VHCI_DEBUG(4, (CE_NOTE, NULL,
2420                                "!Lun not held\n"));
2421                            if (pgr_sema_held) {
2422                                    sema_v(&vlun->svl_pgr_sema);
2423                            }
2424                            return (TRAN_BUSY);
2425                    }
2426                    /*
2427                     * now that the LUN is stable, one last check
2428                     * to make sure no other changes sneaked in
2429                     * (like a path coming online or a
2430                     * failover initiated by another thread)
2431                     */
2432                    pip = NULL;
```



```
2433                            rval = mdi_select_path(cdip, vpkt->vpkt_tgt_init_bp,
2434                                0, NULL, &pip);
2435                            if (pip != NULL) {
2436                                    VHCI_RELEASE_LUN(vlun);
2437                                    vlun->svl_waiting_for_activepath = 0;
2438                                    goto bind_path;
2439                            }

2441                            /*
2442                             * Check if there is an ONLINE path OR a STANDBY path
2443                             * available. If none is available, do not attempt
2444                             * to do a failover, just return a fatal error at this
2445                             * point.
2446                             */
2447                            npip = NULL;
2448                            rval = mdi_select_path(cdip, NULL,
2449                                (MDI_SELECT_ONLINE_PATH | MDI_SELECT_STANDBY_PATH),
2450                                NULL, &npip);
2451                            if ((npip == NULL) || (rval != MDI_SUCCESS)) {
2452                                    /*
2453                                     * No paths available, jus return FATAL error.
2454                                     */
2455                                    VHCI_RELEASE_LUN(vlun);
2456                                    if (pgr_sema_held) {
2457                                            sema_v(&vlun->svl_pgr_sema);
2458                                    }
2459                                    return (TRAN_FATAL_ERROR);
2460                            }
2461                            mdi_rele_path(npip);
2462                            if (!(vpkt->vpkt_state & VHCI_PKT_IN_FAILOVER)) {
2463                                    VHCI_DEBUG(1, (CE_NOTE, NULL, "!invoking "
2464                                        "mdi_failover\n"));
2465                                    rval = mdi_failover(vhci->vhci_dip, cdip,
2466                                        MDI_FAILOVER_ASYNC);
2467                            } else {
2468                                    rval = vlun->svl_failover_status;
2469                            }
2470                            if (rval == MDI_FAILURE) {
2471                                    VHCI_RELEASE_LUN(vlun);
2472                                    if (pgr_sema_held) {
2473                                            sema_v(&vlun->svl_pgr_sema);
2474                                    }
2475                                    return (TRAN_FATAL_ERROR);
2476                            } else if (rval == MDI_BUSY) {
2477                                    VHCI_RELEASE_LUN(vlun);
2478                                    if (pgr_sema_held) {
2479                                            sema_v(&vlun->svl_pgr_sema);
2480                                    }
2481                                    return (TRAN_BUSY);
2482                            } else {
2483                                    if (pgr_sema_held) {
2484                                            sema_v(&vlun->svl_pgr_sema);
2485                                    }
2486                                    vpkt->vpkt_state |= VHCI_PKT_IN_FAILOVER;
2487                                    return (TRAN_BUSY);
2488                            }
2489                    }
2490                    vlun->svl_waiting_for_activepath = 0;
2491 bind_path:
2492                    vpkt->vpkt_path = pip;
2493                    svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(pip);
2494                    ASSERT(svp != NULL);

2496                    psd = svp->svp_psd;
2497                    ASSERT(psd != NULL);
2498                    address = &psd->sd_address;
```

```
2499         } else {
2500                 pkt = vpkt->vpkt_hba_pkt;
2501                 address = &pkt->pkt_address;
2502         }

2504         /* Verify match of specified path_instance and selected path_instance */
2505         ASSERT((path_instance == 0) ||
2506             (path_instance == mdi_pi_get_path_instance(vpkt->vpkt_path)));

2508         /*
2509          * For PKT_PARTIAL_DMA case, call pHCI's scsi_init_pkt whenever
2510          * target driver calls vhci_scsi_init_pkt.
2511          */
2512         if ((vpkt->vpkt_flags & CFLAG_DMA_PARTIAL) &&
2513             vpkt->vpkt_path && vpkt->vpkt_hba_pkt) {
2514                 VHCI_DEBUG(4, (CE_NOTE, NULL,
2515                     "vhci_bind_transport: PKT_PARTIAL_DMA "
2516                     "vpkt 0x%p, path 0x%p hba_pkt 0x%p\n",
2517                     (void *)vpkt, (void *)vpkt->vpkt_path, (void *)pkt));
2518                 pkt = vpkt->vpkt_hba_pkt;
2519                 address = &pkt->pkt_address;
2520         }

2522         if (pkt == NULL || (vpkt->vpkt_flags & CFLAG_DMA_PARTIAL)) {
2523                 pkt = scsi_init_pkt(address, pkt,
2524                     vpkt->vpkt_tgt_init_bp, vpkt->vpkt_tgt_init_cdblen,
2525                     vpkt->vpkt_tgt_init_scblen, 0, flags, func, NULL);

2527                 if (pkt == NULL) {
2528                         VHCI_DEBUG(4, (CE_NOTE, NULL,
2529                             "!bind transport: 0x%p 0x%p 0x%p\n",
2530                             (void *)vhci, (void *)psd, (void *)vpkt));
2531                         if ((vpkt->vpkt_hba_pkt == NULL) && vpkt->vpkt_path) {
2532                                 MDI_PI_ERRSTAT(vpkt->vpkt_path,
2533                                     MDI_PI_TRANSERR);
2534                                 mdi_rele_path(vpkt->vpkt_path);
2535                                 vpkt->vpkt_path = NULL;
2536                         }
2537                         if (pgr_sema_held) {
2538                                 sema_v(&vlun->svl_pgr_sema);
2539                         }
2540                         /*
2541                          * Consider it a fatal error if b_error is
2542                          * set as a result of DMA binding failure
2543                          * vs. a condition of being temporarily out of
2544                          * some resource
2545                          */
2546                         if (vpkt->vpkt_tgt_init_bp == NULL ||
2547                             geterror(vpkt->vpkt_tgt_init_bp))
2548                                 return (TRAN_FATAL_ERROR);
2549                         else
2550                                 return (TRAN_BUSY);
2551                 }
2552         }

2554         pkt->pkt_private = vpkt;
2555         vpkt->vpkt_hba_pkt = pkt;
2556         return (TRAN_ACCEPT);
2557 }
```

*unchanged_portion_omitted_*

```
3571 /*
3572  * two possibilities: (1) failover has completed
3573  * or (2) is in progress; update our path states for
3574  * the former case; for the latter case,
3575  * initiate a scsi_watch request to
```



```
3576  * determine when failover completes - vlun is HELD
3577  * until failover completes; BUSY is returned to upper
3578  * layer in both the cases
3579  */
3580 static int
3581 vhci_handle_ext_fo(struct scsi_pkt *pkt, int fostat)
3582 {
3583         struct vhci_pkt         *vpkt = (struct vhci_pkt *)pkt->pkt_private;
3584         struct scsi_pkt         *tpkt;
3585         scsi_vhci_priv_t        *svp;
3586         scsi_vhci_lun_t         *vlun;
3587         struct scsi_vhci        *vhci;
3588         scsi_vhci_swarg_t       *swarg;
3589         char                    *path;

3591         ASSERT(vpkt != NULL);
3592         tpkt = vpkt->vpkt_tgt_pkt;
3593         ASSERT(tpkt != NULL);
3594         svp = (scsi_vhci_priv_t *)mdi_pi_get_vhci_private(vpkt->vpkt_path);
3595         ASSERT(svp != NULL);
3596         vlun = svp->svp_svl;
3597         ASSERT(vlun != NULL);
3598         ASSERT(VHCI_LUN_IS_HELD(vlun));

3600         vhci = ADDR2VHCI(&tpkt->pkt_address);

3602         if (fostat == SCSI_SENSE_INACTIVE) {
3603                 VHCI_DEBUG(1, (CE_NOTE, NULL, "!Failover "
3604                     "detected for %s; updating path states...\n",
3605                     vlun->svl_lun_wwn));
3606                 /*
3607                  * set the vlun flag to indicate to the task that the target
3608                  * port group needs updating
3609                  */
3610                 vlun->svl_flags |= VLUN_UPDATE_TPG;
3611                 (void) taskq_dispatch(vhci->vhci_update_pathstates_taskq,
3612                     vhci_update_pathstates, (void *)vlun, KM_SLEEP);
3613         } else {
3614                 path = kmem_alloc(MAXPATHLEN, KM_SLEEP);
3615                 vhci_log(CE_NOTE, ddi_get_parent(vlun->svl_dip),
3616                     "!%s (%s%d): Waiting for externally initiated failover "
3617                     "to complete", ddi_pathname(vlun->svl_dip, path),
3618                     ddi_driver_name(vlun->svl_dip),
3619                     ddi_get_instance(vlun->svl_dip));
3620                 kmem_free(path, MAXPATHLEN);
3621                 swarg = kmem_alloc(sizeof (*swarg), KM_NOSLEEP);
3622                 if (swarg == NULL) {
3623                         VHCI_DEBUG(1, (CE_NOTE, NULL, "!vhci_handle_ext_fo: "
3624                             "request packet allocation for %s failed....\n",
3625                             vlun->svl_lun_wwn));
3626                         VHCI_RELEASE_LUN(vlun);
3627                         return (PKT_RETURN);
3628                 }
3629                 swarg->svs_svp = svp;
3630                 swarg->svs_tos = gethrtime();
1422                 swarg->svs_tos = ddi_get_time();
3631                 swarg->svs_pi = vpkt->vpkt_path;
3632                 swarg->svs_release_lun = 0;
3633                 swarg->svs_done = 0;
3634                 /*
3635                  * place a hold on the path...we don't want it to
3636                  * vanish while scsi_watch is in progress
3637                  */
3638                 mdi_hold_path(vpkt->vpkt_path);
3639                 svp->svp_sw_token = scsi_watch_request_submit(svp->svp_psd,
3640                     VHCI_FOWATCH_INTERVAL, SENSE_LENGTH, vhci_efo_watch_cb,
```

```
3641                    (caddr_t)swarg);
3642        }
3643        return (BUSY_RETURN);
3644 }

3646 /*
3647  * vhci_efo_watch_cb:
3648  *      Callback from scsi_watch request to check the failover status.
3649  *      Completion is either due to successful failover or timeout.
3650  *      Upon successful completion, vhci_update_path_states is called.
3651  *      For timeout condition, vhci_efo_done is called.
3652  *      Always returns 0 to scsi_watch to keep retrying till vhci_efo_done
3653  *      terminates this request properly in a separate thread.
3654  */

3656 static int
3657 vhci_efo_watch_cb(caddr_t arg, struct scsi_watch_result *resultp)
3658 {
3659        struct scsi_status              *statusp = resultp->statusp;
3660        uint8_t                         *sensep = (uint8_t *)resultp->sensep;
3661        struct scsi_pkt                 *pkt = resultp->pkt;
3662        scsi_vhci_swarg_t               *swarg;
3663        scsi_vhci_priv_t                *svp;
3664        scsi_vhci_lun_t                 *vlun;
3665        struct scsi_vhci                *vhci;
3666        dev_info_t                      *vdip;
3667        int                             rval, updt_paths;

3669        swarg = (scsi_vhci_swarg_t *)(uintptr_t)arg;
3670        svp = swarg->svs_svp;
3671        if (swarg->svs_done) {
3672                /*
3673                 * Already completed failover or timedout.
3674                 * Waiting for vhci_efo_done to terminate this scsi_watch.
3675                 */
3676                return (0);
3677        }

3679        ASSERT(svp != NULL);
3680        vlun = svp->svp_svl;
3681        ASSERT(vlun != NULL);
3682        ASSERT(VHCI_LUN_IS_HELD(vlun));
3683        vlun->svl_efo_update_path = 0;
3684        vdip = ddi_get_parent(vlun->svl_dip);
3685        vhci = ddi_get_soft_state(vhci_softstate,
3686            ddi_get_instance(vdip));

3688        updt_paths = 0;

3690        if (pkt->pkt_reason != CMD_CMPLT) {
3691                if ((gethrtime() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
1483                if ((ddi_get_time() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
3692                        swarg->svs_release_lun = 1;
3693                        goto done;
3694                }
3695                return (0);
3696        }
3697        if (*((unsigned char *)statusp) == STATUS_CHECK) {
3698                rval = vlun->svl_fops->sfo_analyze_sense(svp->svp_psd, sensep,
3699                    vlun->svl_fops_ctpriv);
3700                switch (rval) {
3701                        /*
3702                         * Only update path states in case path is definitely
3703                         * inactive, or no failover occurred.  For all other
3704                         * check conditions continue pinging.  A unexpected
3705                         * check condition shouldn't cause pinging to complete
```



```
3706                         * prematurely.
3707                         */
3708                        case SCSI_SENSE_INACTIVE:
3709                        case SCSI_SENSE_NOFAILOVER:
3710                                updt_paths = 1;
3711                                break;
3712                        default:
3713                                if ((gethrtime() - swarg->svs_tos)
1505                                if ((ddi_get_time() - swarg->svs_tos)
3714                                    >= VHCI_EXTFO_TIMEOUT) {
3715                                        swarg->svs_release_lun = 1;
3716                                        goto done;
3717                                }
3718                                return (0);
3719                        }
3720                } else if (*((unsigned char *)statusp) ==
3721                    STATUS_RESERVATION_CONFLICT) {
3722                        updt_paths = 1;
3723                } else if ((*((unsigned char *)statusp)) &
3724                    (STATUS_BUSY | STATUS_QFULL)) {
3725                        return (0);
3726                }
3727                if ((*((unsigned char *)statusp) == STATUS_GOOD) ||
3728                    (updt_paths == 1)) {
3729                        /*
3730                         * we got here because we had detected an
3731                         * externally initiated failover; things
3732                         * have settled down now, so let's
3733                         * start up a task to update the
3734                         * path states and target port group
3735                         */
3736                        vlun->svl_efo_update_path = 1;
3737                        swarg->svs_done = 1;
3738                        vlun->svl_swarg = swarg;
3739                        vlun->svl_flags |= VLUN_UPDATE_TPG;
3740                        (void) taskq_dispatch(vhci->vhci_update_pathstates_taskq,
3741                            vhci_update_pathstates, (void *)vlun,
3742                            KM_SLEEP);
3743                        return (0);
3744                }
3745                if ((gethrtime() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
1537                if ((ddi_get_time() - swarg->svs_tos) >= VHCI_EXTFO_TIMEOUT) {
3746                        swarg->svs_release_lun = 1;
3747                        goto done;
3748                }
3749                return (0);
3750 done:
3751                swarg->svs_done = 1;
3752                (void) taskq_dispatch(vhci->vhci_taskq,
3753                    vhci_efo_done, (void *)swarg, KM_SLEEP);
3754                return (0);
3755 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   22616 Mon May  5 11:11:09 2014
new/usr/src/uts/common/sys/scsi/adapters/scsi_vhci.h
4779 vhci shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

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

/*
 * Copyright (c) 2001, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _SYS_SCSI_ADAPTERS_SCSI_VHCI_H
#define _SYS_SCSI_ADAPTERS_SCSI_VHCI_H

/*
 * Multiplexed I/O SCSI vHCI global include
 */
#include <sys/note.h>
#include <sys/taskq.h>
#include <sys/mhd.h>
#include <sys/sunmdi.h>
#include <sys/mdi_impldefs.h>
#include <sys/scsi/adapters/mpapi_impl.h>
#include <sys/scsi/adapters/mpapi_scsi_vhci.h>

#ifdef  __cplusplus
extern "C" {
#endif

#if !defined(_BIT_FIELDS_LTOH) && !defined(_BIT_FIELDS_HTOL)
#error  One of _BIT_FIELDS_LTOH or _BIT_FIELDS_HTOL must be defined
#endif  /* _BIT_FIELDS_LTOH */

#ifdef  _KERNEL

#ifdef  UNDEFINED
#undef  UNDEFINED
#endif
#define UNDEFINED               -1

#define VHCI_STATE_OPEN         0x00000001
```



```
#define VH_SLEEP                0x0
#define VH_NOSLEEP              0x1

/*
 * HBA interface macros
 */

#define TRAN2HBAPRIVATE(tran)   ((struct scsi_vhci *)(tran)->tran_hba_private)
#define VHCI_INIT_WAIT_TIMEOUT  60000000
#define VHCI_FOWATCH_INTERVAL   1000000         /* in usecs */
#define VHCI_EXTFO_TIMEOUT      (3 * 60 * NANOSEC)      /* 3 minutes in nsec */
#define VHCI_EXTFO_TIMEOUT      3*60            /* 3 minutes */

#define SCBP_C(pkt)     ((*(pkt)->pkt_scbp) & STATUS_MASK)

int vhci_do_scsi_cmd(struct scsi_pkt *);
/*PRINTFLIKE3*/
void vhci_log(int, dev_info_t *, const char *, ...);

/*
 * debugging stuff
 */

#ifdef  DEBUG

#ifndef VHCI_DEBUG_DEFAULT_VAL
#define VHCI_DEBUG_DEFAULT_VAL          0
#endif  /* VHCI_DEBUG_DEFAULT_VAL */

extern int vhci_debug;

#include <sys/debug.h>

#define VHCI_DEBUG(level, stmnt) \
            if (vhci_debug >= (level)) vhci_log stmnt

#else   /* !DEBUG */

#define VHCI_DEBUG(level, stmnt)

#endif  /* !DEBUG */



#define VHCI_PKT_PRIV_SIZE              2

#define ADDR2VHCI(ap)   ((struct scsi_vhci *) \
                        ((ap)->a_hba_tran->tran_hba_private))
#define ADDR2VLUN(ap)   (scsi_vhci_lun_t *) \
                        (scsi_device_hba_private_get(scsi_address_device(ap)))
#define ADDR2DIP(ap)    ((dev_info_t *)(scsi_address_device(ap)->sd_dev))

#define HBAPKT2VHCIPKT(pkt) (pkt->pkt_private)
#define TGTPKT2VHCIPKT(pkt) (pkt->pkt_ha_private)
#define VHCIPKT2HBAPKT(pkt) (pkt->pkt_hba_pkt)
#define VHCIPKT2TGTPKT(pkt) (pkt->pkt_tgt_pkt)

#define VHCI_DECR_PATH_CMDCOUNT(svp) { \
        mutex_enter(&(svp)->svp_mutex); \
        (svp)->svp_cmds--; \
        if ((svp)->svp_cmds == 0)  \
                cv_broadcast(&(svp)->svp_cv); \
        mutex_exit(&(svp)->svp_mutex); \
}
```

*unchanged_portion_omitted_*

```
308 typedef struct scsi_vhci_lun {
309         kmutex_t                svl_mutex;
310         kcondvar_t              svl_cv;

312         /*
313          * following three fields are under svl_mutex protection
314          */
315         int                     svl_transient;

317         /*
318          * to prevent unnecessary failover when a device is
319          * is discovered across a passive path and active path
320          * is still comng up
321          */
322         int                     svl_waiting_for_activepath;
323         hrtime_t                svl_wfa_time;
276         time_t                  svl_wfa_time;

325         /*
326          * to keep the failover status in order to return the
327          * failure status to target driver when targer driver
328          * retries the command which originally triggered the
329          * failover.
330          */
331         int                     svl_failover_status;

333         /*
334          * for RESERVE/RELEASE support
335          */
336         client_lb_t             svl_lb_policy_save;

338         /*
339          * Failover ops and ops name selected for the lun.
340          */
341         struct scsi_failover_ops        *svl_fops;
342         char                    *svl_fops_name;

344         void                    *svl_fops_ctpriv;

346         struct scsi_vhci_lun    *svl_hash_next;
347         char                    *svl_lun_wwn;

349         /*
350          * currently active pathclass
351          */
352         char                    *svl_active_pclass;

354         dev_info_t              *svl_dip;
355         uint32_t                svl_flags;      /* protected by svl_mutex */

357         /*
358          * When SCSI-II reservations are active we set the following pip
359          * to point to the path holding the reservation.  As long as
360          * the reservation is active this svl_resrv_pip is bound for the
361          * transport directly.  We bypass calling mdi_select_path to return
362          * a pip.
363          * The following pip is only valid when VLUN_RESERVE_ACTIVE_FLG
364          * is set.  This pip should not be accessed if this flag is reset.
365          */
366         mdi_pathinfo_t  *svl_resrv_pip;

368         /*
369          * following fields are for PGR support
370          */
371         taskq_t                 *svl_taskq;
```



```
372         ksema_t                 svl_pgr_sema;   /* PGR serialization */
373         vhci_prin_readkeys_t    svl_prin;       /* PGR in data */
374         vhci_prout_t            svl_prout;      /* PGR out data */
375         uchar_t                 svl_cdb[CDB_GROUP4];
376         int                     svl_time;       /* pkt_time */
377         uint32_t                svl_bcount;     /* amount of data */
378         int                     svl_pgr_active; /* registrations active */
379         mdi_pathinfo_t          *svl_first_path;

381         /* external failover */
382         int                     svl_efo_update_path;
383         struct  scsi_vhci_swarg *svl_swarg;

385         uint32_t                svl_support_lun_reset; /* Lun reset support */
386         int                     svl_not_supported;
387         int                     svl_xlf_capable; /* XLF implementation */
388         int                     svl_sector_size;
389         int                     svl_setcap_done;
390         uint16_t                svl_fo_support;  /* failover mode */
391 } scsi_vhci_lun_t;
______unchanged_portion_omitted_

463 /*
464  * argument to scsi_watch callback.  Used for processing
465  * externally initiated failovers
466  */
467 typedef struct scsi_vhci_swarg {
468         scsi_vhci_priv_t        *svs_svp;
469         hrtime_t                svs_tos;        /* time of submission */
422         time_t                  svs_tos;        /* time of submission */
470         mdi_pathinfo_t          *svs_pi;        /* pathinfo being "watched" */
471         int                     svs_release_lun;
472         int                     svs_done;
473 } scsi_vhci_swarg_t;
______unchanged_portion_omitted_
```