```
**********************************************************
   35199 Wed Jun 27 10:40:34 2012
new/usr/src/uts/common/io/comstar/port/pppt/pppt.c
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

 113 pppt_global_t pppt_global;

 115 int pppt_logging = 0;

 117 static int pppt_enable_svc(void);

 119 static void pppt_disable_svc(void);

 121 static int pppt_task_avl_compare(const void *tgt1, const void *tgt2);

 123 static stmf_data_buf_t *pppt_dbuf_alloc(scsi_task_t *task,
 124     uint32_t size, uint32_t *pminsize, uint32_t flags);

 126 static void pppt_dbuf_free(stmf_dbuf_store_t *ds, stmf_data_buf_t *dbuf);

 128 static void pppt_sess_destroy_task(void *ps_void);

 130 static void pppt_task_sent_status(pppt_task_t *ptask);

 132 static pppt_status_t pppt_task_try_abort(pppt_task_t *ptask);

 134 static pppt_status_t pppt_task_hold(pppt_task_t *ptask);

 134 static void pppt_task_rele(pppt_task_t *ptask);

 136 static void pppt_task_update_state(pppt_task_t *ptask,
 137     pppt_task_state_t new_state);

 139 /*
 140  * Lock order:  global --> target --> session --> task
 141  */

 143 int
 144 _init(void)
 145 {
 146         int rc;

 148         mutex_init(&pppt_global.global_lock, NULL, MUTEX_DEFAULT, NULL);
 149         mutex_init(&pppt_global.global_door_lock, NULL, MUTEX_DEFAULT, NULL);
 150         pppt_global.global_svc_state = PSS_DETACHED;

 152         if ((rc = mod_install(&modlinkage)) != 0) {
 153                 mutex_destroy(&pppt_global.global_door_lock);
 154                 mutex_destroy(&pppt_global.global_lock);
 155                 return (rc);
 156         }

 158         return (rc);
 159 }
______unchanged_portion_omitted_

 796 void
 797 pppt_lport_task_free(scsi_task_t *task)
 798 {
 799         pppt_task_t *ptask = task->task_port_private;
 800         pppt_sess_t *ps = ptask->pt_sess;

 802         pppt_task_rele(ptask);
 804         pppt_task_free(ptask);
```



```
 803         pppt_sess_rele(ps);
 804 }
______unchanged_portion_omitted_

1198 pppt_task_t *
1199 pppt_task_alloc(void)
1200 {
1201         pppt_task_t     *ptask;
1202         pppt_buf_t      *immed_pbuf;

1204         ptask = kmem_alloc(sizeof (pppt_task_t) + sizeof (pppt_buf_t) +
1205             sizeof (stmf_data_buf_t), KM_NOSLEEP);
1206         if (ptask != NULL) {
1207                 ptask->pt_state = PTS_INIT;
1208                 ptask->pt_read_buf = NULL;
1209                 ptask->pt_read_xfer_msgid = 0;
1210                 ptask->pt_refcnt = 0;
1212                 cv_init(&ptask->pt_cv, NULL, CV_DRIVER, NULL);
1211                 mutex_init(&ptask->pt_mutex, NULL, MUTEX_DRIVER, NULL);
1212                 immed_pbuf = (pppt_buf_t *)(ptask + 1);
1213                 bzero(immed_pbuf, sizeof (*immed_pbuf));
1214                 immed_pbuf->pbuf_is_immed = B_TRUE;
1215                 immed_pbuf->pbuf_stmf_buf = (stmf_data_buf_t *)(immed_pbuf + 1);

1217                 bzero(immed_pbuf->pbuf_stmf_buf, sizeof (stmf_data_buf_t));
1218                 immed_pbuf->pbuf_stmf_buf->db_port_private = immed_pbuf;
1219                 immed_pbuf->pbuf_stmf_buf->db_sglist_length = 1;
1220                 immed_pbuf->pbuf_stmf_buf->db_flags = DB_DIRECTION_FROM_RPORT |
1221                     DB_DONT_CACHE;
1222                 ptask->pt_immed_data = immed_pbuf;
1223         }

1225         return (ptask);

1227 }

1229 void
1230 pppt_task_free(pppt_task_t *ptask)
1231 {
1232         mutex_enter(&ptask->pt_mutex);
1233         ASSERT(ptask->pt_refcnt == 0);
1234         mutex_destroy(&ptask->pt_mutex);
1236         cv_destroy(&ptask->pt_cv);
1235         kmem_free(ptask, sizeof (pppt_task_t) + sizeof (pppt_buf_t) +
1236             sizeof (stmf_data_buf_t));
1237 }

1239 pppt_status_t
1240 pppt_task_start(pppt_task_t *ptask)
1241 {
1242         avl_index_t             where;

1244         ASSERT(ptask->pt_state == PTS_INIT);

1246         mutex_enter(&ptask->pt_sess->ps_mutex);
1247         mutex_enter(&ptask->pt_mutex);
1248         if (avl_find(&ptask->pt_sess->ps_task_list, ptask, &where) == NULL) {
1249                 pppt_task_update_state(ptask, PTS_ACTIVE);
1250                 /* Manually increment refcnt, sincd we hold the mutex... */
1251                 ptask->pt_refcnt++;
1252                 avl_insert(&ptask->pt_sess->ps_task_list, ptask, where);
1253                 mutex_exit(&ptask->pt_mutex);
1254                 mutex_exit(&ptask->pt_sess->ps_mutex);
1255                 return (PPPT_STATUS_SUCCESS);
1256         }
1257         mutex_exit(&ptask->pt_mutex);
```

```
1258         mutex_exit(&ptask->pt_sess->ps_mutex);

1260         return (PPPT_STATUS_FAIL);
1261 }

1263 pppt_status_t
1264 pppt_task_done(pppt_task_t *ptask)
1265 {
1266         pppt_status_t   pppt_status = PPPT_STATUS_SUCCESS;
1267         boolean_t       remove = B_FALSE;

1269         mutex_enter(&ptask->pt_mutex);

1271         switch (ptask->pt_state) {
1272         case PTS_ACTIVE:
1273                 remove = B_TRUE;
1274                 pppt_task_update_state(ptask, PTS_DONE);
1275                 break;
1276         case PTS_ABORTED:
1277                 pppt_status = PPPT_STATUS_ABORTED;
1278                 break;
1279         case PTS_DONE:
1280                 /* Repeat calls are OK.  Do nothing, return success */
1281                 break;
1282         default:
1283                 ASSERT(0);
1284         }

1286         mutex_exit(&ptask->pt_mutex);

1288         if (remove) {
1289                 mutex_enter(&ptask->pt_sess->ps_mutex);
1290                 avl_remove(&ptask->pt_sess->ps_task_list, ptask);
1291                 mutex_exit(&ptask->pt_sess->ps_mutex);
1292                 /* Out of the AVL tree, so drop a reference. */
1293                 pppt_task_rele(ptask);
1294         }

1296         return (pppt_status);
1297 }
```

_____*unchanged_portion_omitted_*

```
1371 static pppt_status_t
1372 pppt_task_try_abort(pppt_task_t *ptask)
1373 {
1374         boolean_t       remove = B_FALSE;
1375         pppt_status_t   pppt_status = PPPT_STATUS_SUCCESS;

1377         mutex_enter(&ptask->pt_mutex);

1379         switch (ptask->pt_state) {
1380         case PTS_ACTIVE:
1381                 remove = B_TRUE;
1382                 pppt_task_update_state(ptask, PTS_ABORTED);
1383                 break;
1384         case PTS_DONE:
1385                 pppt_status = PPPT_STATUS_DONE;
1386                 break;
1387         case PTS_SENT_STATUS:
1388                 /*
1389                  * Already removed so leave remove set to B_FALSE
1390                  * and leave status set to PPPT_STATUS_SUCCESS.
1391                  */
1392                 pppt_task_update_state(ptask, PTS_ABORTED);
1393                 break;
1394         case PTS_ABORTED:
```



```
1395                 break;
1396         default:
1397                 ASSERT(0);
1398         }

1400         mutex_exit(&ptask->pt_mutex);

1402         if (remove) {
1403                 mutex_enter(&ptask->pt_sess->ps_mutex);
1404                 avl_remove(&ptask->pt_sess->ps_task_list, ptask);
1405                 mutex_exit(&ptask->pt_sess->ps_mutex);
1406                 /* Out of the AVL tree, so drop a reference. */
1407                 pppt_task_rele(ptask);
1408         }

1410         return (pppt_status);
1411 }

1413 pppt_status_t
1409 static pppt_status_t
1414 pppt_task_hold(pppt_task_t *ptask)
1415 {
1416         pppt_status_t   pppt_status = PPPT_STATUS_SUCCESS;

1418         mutex_enter(&ptask->pt_mutex);
1419         if (ptask->pt_state == PTS_ACTIVE) {
1420                 ptask->pt_refcnt++;
1421         } else {
1422                 pppt_status = PPPT_STATUS_FAIL;
1423         }
1424         mutex_exit(&ptask->pt_mutex);

1426         return (pppt_status);
1427 }

1429 static void
1430 pppt_task_rele(pppt_task_t *ptask)
1431 {
1432         boolean_t freeit;

1434         mutex_enter(&ptask->pt_mutex);
1435         ptask->pt_refcnt--;
1436         freeit = (ptask->pt_refcnt == 0);
1430         cv_signal(&ptask->pt_cv);
1437         mutex_exit(&ptask->pt_mutex);
1438         if (freeit)
1439                 pppt_task_free(ptask);
1440 }

1442 static void
1443 pppt_task_update_state(pppt_task_t *ptask,
1444     pppt_task_state_t new_state)
1445 {
1446         PPPT_LOG(CE_NOTE, "task %p %d -> %d", (void *)ptask,
1447             ptask->pt_state, new_state);

1449         ASSERT(mutex_owned(&ptask->pt_mutex));
1450         ptask->pt_state = new_state;
1443         cv_signal(&ptask->pt_cv);
1451 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    7460 Wed Jun 27 10:40:35 2012
new/usr/src/uts/common/io/comstar/port/pppt/pppt.h
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

 166 typedef struct {
 167         pppt_sess_t             *pt_sess;
 168         avl_node_t              pt_sess_ln;
 169         int                     pt_refcnt;
 170         kmutex_t                pt_mutex;
 171         kcondvar_t              pt_cv;
 171         stmf_ic_msgid_t         pt_task_id;
 172         uint8_t                 pt_lun_id[16];
 173         pppt_task_state_t       pt_state;
 174         scsi_task_t             *pt_stmf_task;
 175         pppt_buf_t              *pt_immed_data;
 176         pppt_buf_t              *pt_read_buf;
 177         stmf_ic_msgid_t         pt_read_xfer_msgid;
 178 } pppt_task_t;
______unchanged_portion_omitted_

 231 extern pppt_global_t pppt_global;

 233 stmf_status_t pppt_lport_xfer_data(scsi_task_t *task, stmf_data_buf_t *dbuf,
 234     uint32_t ioflags);

 236 void pppt_xfer_read_complete(pppt_task_t *pppt_task, stmf_status_t status);

 238 stmf_status_t pppt_lport_send_status(scsi_task_t *task, uint32_t ioflags);

 240 void pppt_lport_task_free(scsi_task_t *task);

 242 stmf_status_t pppt_lport_abort(stmf_local_port_t *lport, int abort_cmd,
 243     void *arg, uint32_t flags);

 245 void pppt_lport_ctl(stmf_local_port_t *lport, int cmd, void *arg);

 247 pppt_sess_t *pppt_sess_lookup_locked(uint64_t session_id,
 248     scsi_devid_desc_t *lport_devid,
 249     stmf_remote_port_t *rport);

 251 pppt_sess_t *pppt_sess_lookup_by_id_locked(uint64_t session_id);

 253 pppt_sess_t *pppt_sess_lookup_create(scsi_devid_desc_t *lport_devid,
 254     scsi_devid_desc_t *rport_devid, stmf_remote_port_t *rport,
 255     uint64_t session_id, stmf_status_t *statusp);

 257 void pppt_sess_rele(pppt_sess_t *sks);

 259 void pppt_sess_rele_locked(pppt_sess_t *sks);

 261 void pppt_sess_close_locked(pppt_sess_t *ps);

 263 int pppt_sess_avl_compare_by_id(const void *void_sess1,
 264     const void *void_sess2);

 266 int pppt_sess_avl_compare_by_name(const void *void_sess1,
 267     const void *void_sess2);

 269 pppt_task_t *pppt_task_alloc(void);

 271 void pppt_task_free(pppt_task_t *ptask);

 273 pppt_status_t pppt_task_start(pppt_task_t *ptask);
```



```
 275 pppt_status_t pppt_task_done(pppt_task_t *ptask);

 277 pppt_task_t *pppt_task_lookup(stmf_ic_msgid_t msgid);

 279 void pppt_msg_rx(stmf_ic_msg_t *msg);

 281 void pppt_msg_tx_status(stmf_ic_msg_t *orig_msg, stmf_status_t status);

 283 pppt_tgt_t *pppt_tgt_lookup(scsi_devid_desc_t *tgt_devid);

 285 pppt_tgt_t *pppt_tgt_lookup_locked(scsi_devid_desc_t *tgt_devid);

 287 pppt_tgt_t *pppt_tgt_create(stmf_ic_reg_port_msg_t *reg_port,
 288     stmf_status_t *errcode);

 290 void pppt_tgt_async_delete(pppt_tgt_t *tgt);

 292 void pppt_tgt_destroy(pppt_tgt_t *tgt);

 294 int pppt_tgt_avl_compare(const void *void_tgt1, const void *void_tgt2);

 296 void pppt_tgt_sm_ctl(stmf_local_port_t *lport, int cmd, void *arg);

 298 pppt_status_t pppt_task_hold(pppt_task_t *);

 300 #ifdef  __cplusplus
 301 }
______unchanged_portion_omitted_
```

```
**********************************************************
   10291 Wed Jun 27 10:40:36 2012
new/usr/src/uts/common/io/comstar/port/pppt/pppt_msg.c
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

 246 static void
 247 pppt_msg_scsi_cmd(stmf_ic_msg_t *msg)
 248 {
 249         pppt_sess_t                     *pppt_sess;
 250         pppt_buf_t                      *pbuf;
 251         stmf_ic_scsi_cmd_msg_t          *scmd;
 252         pppt_task_t                     *ptask;
 253         scsi_task_t                     *task;
 254         pppt_status_t                   pppt_status;
 255         stmf_local_port_t               *lport;
 256         stmf_scsi_session_t             *stmf_sess;
 257         stmf_status_t                   stmf_status;

 259         /*
 260          * Get a task context
 261          */
 262         ptask = pppt_task_alloc();
 263         if (ptask == NULL) {
 264                 /*
 265                  * We must be very low on memory.  Just free the message
 266                  * and let the command timeout.
 267                  */
 268                 stmf_ic_msg_free(msg);
 269                 PPPT_INC_STAT(es_scmd_ptask_alloc_fail);
 270                 return;
 271         }

 273         scmd = msg->icm_msg;

 275         /*
 276          * Session are created implicitly on the first use of an
 277          * IT nexus
 278          */
 279         pppt_sess = pppt_sess_lookup_create(scmd->icsc_tgt_devid,
 280             scmd->icsc_ini_devid, scmd->icsc_rport,
 281             scmd->icsc_session_id, &stmf_status);
 282         if (pppt_sess == NULL) {
 283                 pppt_task_free(ptask);
 284                 pppt_msg_tx_status(msg, stmf_status);
 285                 stmf_ic_msg_free(msg);
 286                 PPPT_INC_STAT(es_scmd_sess_create_fail);
 287                 return;
 288         }

 290         ptask->pt_sess = pppt_sess;
 291         ptask->pt_task_id = scmd->icsc_task_msgid;
 292         stmf_sess = pppt_sess->ps_stmf_sess;
 293         lport = stmf_sess->ss_lport;

 295         /*
 296          * Add task to our internal task set.
 297          */
 298         pppt_status = pppt_task_start(ptask);

 300         if (pppt_status != 0) {
 301                 /* Release hold from pppt_sess_lookup_create() */
 302                 PPPT_LOG(CE_WARN, "Duplicate taskid from remote node 0x%llx",
 303                     (longlong_t)scmd->icsc_task_msgid);
 304                 pppt_task_free(ptask);
```



```
 305                 pppt_sess_rele(pppt_sess);
 306                 pppt_msg_tx_status(msg, STMF_ALREADY);
 307                 stmf_ic_msg_free(msg);
 308                 PPPT_INC_STAT(es_scmd_dup_task_count);
 309                 return;
 310         }

 312         /*
 313          * Allocate STMF task context
 314          */
 315         ptask->pt_stmf_task = stmf_task_alloc(lport, stmf_sess,
 316             scmd->icsc_task_lun_no,
 317             scmd->icsc_task_cdb_length, 0);
 318         if (ptask->pt_stmf_task == NULL) {
 319                 (void) pppt_task_done(ptask);
 320                 ASSERT(ptask->pt_refcnt == 0);
 321                 pppt_task_free(ptask);
 322                 pppt_sess_rele(pppt_sess);
 323                 pppt_msg_tx_status(msg, STMF_ALLOC_FAILURE);
 324                 stmf_ic_msg_free(msg);
 325                 PPPT_INC_STAT(es_scmd_stask_alloc_fail);
 326                 return;
 327         }

 329         task = ptask->pt_stmf_task;
 330         /* task_port_private reference is a real reference. */
 331         (void) pppt_task_hold(ptask);
 332         task->task_port_private = ptask;
 333         task->task_flags = scmd->icsc_task_flags;
 334         task->task_additional_flags = 0;
 335         task->task_priority = 0;

 337         /*
 338          * Set task->task_mgmt_function to TM_NONE for a normal SCSI task
 339          * or one of these values for a task management command:
 340          *
 341          * TM_ABORT_TASK ***
 342          * TM_ABORT_TASK_SET
 343          * TM_CLEAR_ACA
 344          * TM_CLEAR_TASK_SET
 345          * TM_LUN_RESET
 346          * TM_TARGET_WARM_RESET
 347          * TM_TARGET_COLD_RESET
 348          *
 349          * *** Note that STMF does not currently support TM_ABORT_TASK so
 350          * port providers must implement this command on their own
 351          * (e.g. lookup the desired task and call stmf_abort).
 352          */
 353         task->task_mgmt_function = scmd->icsc_task_mgmt_function;

 355         task->task_max_nbufs = 1; /* Don't allow parallel xfers */
 356         task->task_cmd_seq_no = msg->icm_msgid;
 357         task->task_expected_xfer_length =
 358             scmd->icsc_task_expected_xfer_length;

 360         if (scmd->icsc_task_cdb_length) {
 361                 bcopy(scmd->icsc_task_cdb, task->task_cdb,
 362                     scmd->icsc_task_cdb_length);
 363         }
 364         bcopy(scmd->icsc_lun_id, ptask->pt_lun_id, 16);

 366         if (scmd->icsc_immed_data_len) {
 367                 pbuf = ptask->pt_immed_data;
 368                 pbuf->pbuf_immed_msg = msg;
 369                 pbuf->pbuf_stmf_buf->db_data_size = scmd->icsc_immed_data_len;
 370                 pbuf->pbuf_stmf_buf->db_buf_size = scmd->icsc_immed_data_len;
```

```
371                 pbuf->pbuf_stmf_buf->db_relative_offset = 0;
372                 pbuf->pbuf_stmf_buf->db_sglist[0].seg_length =
373                     scmd->icsc_immed_data_len;
374                 pbuf->pbuf_stmf_buf->db_sglist[0].seg_addr =
375                     scmd->icsc_immed_data;

377                 stmf_post_task(task, pbuf->pbuf_stmf_buf);
378         } else {
379                 stmf_post_task(task, NULL);
380                 stmf_ic_msg_free(msg);
381         }
382 }
```

_____***unchanged_portion_omitted_***