```
**********************************************************
  215707 Mon Jul 28 07:45:19 2014
new/usr/src/uts/common/io/comstar/stmf/stmf.c
5046 comstar: use the correct type instead of casting all the time
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2012, Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

#include <sys/conf.h>
#include <sys/file.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/scsi/scsi.h>
#include <sys/scsi/generic/persist.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/disp.h>
#include <sys/byteorder.h>
#include <sys/atomic.h>
#include <sys/ethernet.h>
#include <sys/sdt.h>
#include <sys/nvpair.h>
#include <sys/zone.h>
#include <sys/id_space.h>

#include <sys/stmf.h>
#include <sys/lpif.h>
#include <sys/portif.h>
#include <sys/stmf_ioctl.h>
#include <sys/pppt_ic_if.h>

#include "stmf_impl.h"
#include "lun_map.h"
#include "stmf_state.h"
#include "stmf_stats.h"

/*
 * Lock order:
 * stmf_state_lock --> ilport_lock/iss_lockp --> ilu_task_lock
 */
```



```
static uint64_t stmf_session_counter = 0;
static uint16_t stmf_rtpid_counter = 0;
/* start messages at 1 */
static uint64_t stmf_proxy_msg_id = 1;
#define MSG_ID_TM_BIT   0x8000000000000000
#define ALIGNED_TO_8BYTE_BOUNDARY(i)    (((i) + 7) & ~7)

/*
 * When stmf_io_deadman_enabled is set to B_TRUE, we check that finishing up
 * I/O operations on an offlining LU doesn't take longer than stmf_io_deadman
 * seconds. If it does, we trigger a panic to inform the user of hung I/O
 * blocking us for too long.
 */
boolean_t stmf_io_deadman_enabled = B_TRUE;
int stmf_io_deadman = 1000;                     /* seconds */

struct stmf_svc_clocks;

static int stmf_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int stmf_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static int stmf_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
        void **result);
static int stmf_open(dev_t *devp, int flag, int otype, cred_t *credp);
static int stmf_close(dev_t dev, int flag, int otype, cred_t *credp);
static int stmf_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
        cred_t *credp, int *rval);
static int stmf_get_stmf_state(stmf_state_desc_t *std);
static int stmf_set_stmf_state(stmf_state_desc_t *std);
static void stmf_abort_task_offline(scsi_task_t *task, int offline_lu,
    char *info);
static int stmf_set_alua_state(stmf_alua_state_desc_t *alua_state);
static void stmf_get_alua_state(stmf_alua_state_desc_t *alua_state);

static void stmf_task_audit(stmf_i_scsi_task_t *itask,
    task_audit_event_t te, uint32_t cmd_or_iof, stmf_data_buf_t *dbuf);

static boolean_t stmf_base16_str_to_binary(char *c, int dplen, uint8_t *dp);
static char stmf_ctoi(char c);
stmf_xfer_data_t *stmf_prepare_tpgs_data(uint8_t ilu_alua);
void stmf_svc_init();
stmf_status_t stmf_svc_fini();
void stmf_svc(void *arg);
static void stmf_wait_ilu_tasks_finish(stmf_i_lu_t *ilu);
void stmf_svc_queue(int cmd, void *obj, stmf_state_change_info_t *info);
static void stmf_svc_kill_obj_requests(void *obj);
static void stmf_svc_timeout(struct stmf_svc_clocks *);
void stmf_check_freetask();
void stmf_abort_target_reset(scsi_task_t *task);
stmf_status_t stmf_lun_reset_poll(stmf_lu_t *lu, struct scsi_task *task,
                                                        int target_reset);
void stmf_target_reset_poll(struct scsi_task *task);
void stmf_handle_lun_reset(scsi_task_t *task);
void stmf_handle_target_reset(scsi_task_t *task);
void stmf_xd_to_dbuf(stmf_data_buf_t *dbuf, int set_rel_off);
int stmf_load_ppd_ioctl(stmf_ppioctl_data_t *ppi, uint64_t *ppi_token,
    uint32_t *err_ret);
int stmf_delete_ppd_ioctl(stmf_ppioctl_data_t *ppi);
int stmf_get_ppd_ioctl(stmf_ppioctl_data_t *ppi, stmf_ppioctl_data_t *ppi_out,
    uint32_t *err_ret);
void stmf_delete_ppd(stmf_pp_data_t *ppd);
void stmf_delete_all_ppds();
void stmf_trace_clear();
void stmf_worker_init();
stmf_status_t stmf_worker_fini();
void stmf_worker_mgmt();
```

```
128 void stmf_worker_task(void *arg);
129 static void stmf_task_lu_free(scsi_task_t *task, stmf_i_scsi_session_t *iss);
130 static stmf_status_t stmf_ic_lu_reg(stmf_ic_reg_dereg_lun_msg_t *msg,
131     uint32_t type);
132 static stmf_status_t stmf_ic_lu_dereg(stmf_ic_reg_dereg_lun_msg_t *msg);
133 static stmf_status_t stmf_ic_rx_scsi_status(stmf_ic_scsi_status_msg_t *msg);
134 static stmf_status_t stmf_ic_rx_status(stmf_ic_status_msg_t *msg);
135 static stmf_status_t stmf_ic_rx_scsi_data(stmf_ic_scsi_data_msg_t *msg);
136 void stmf_task_lu_killall(stmf_lu_t *lu, scsi_task_t *tm_task, stmf_status_t s);

138 /* pppt modhandle */
139 ddi_modhandle_t pppt_mod;

141 /* pppt modload imported functions */
142 stmf_ic_reg_port_msg_alloc_func_t ic_reg_port_msg_alloc;
143 stmf_ic_dereg_port_msg_alloc_func_t ic_dereg_port_msg_alloc;
144 stmf_ic_reg_lun_msg_alloc_func_t ic_reg_lun_msg_alloc;
145 stmf_ic_dereg_lun_msg_alloc_func_t ic_dereg_lun_msg_alloc;
146 stmf_ic_lun_active_msg_alloc_func_t ic_lun_active_msg_alloc;
147 stmf_ic_scsi_cmd_msg_alloc_func_t ic_scsi_cmd_msg_alloc;
148 stmf_ic_scsi_data_xfer_done_msg_alloc_func_t ic_scsi_data_xfer_done_msg_alloc;
149 stmf_ic_session_create_msg_alloc_func_t ic_session_reg_msg_alloc;
150 stmf_ic_session_destroy_msg_alloc_func_t ic_session_dereg_msg_alloc;
151 stmf_ic_tx_msg_func_t ic_tx_msg;
152 stmf_ic_msg_free_func_t ic_msg_free;

154 static void stmf_itl_task_start(stmf_i_scsi_task_t *itask);
155 static void stmf_itl_lu_new_task(stmf_i_scsi_task_t *itask);
156 static void stmf_itl_task_done(stmf_i_scsi_task_t *itask);

158 static void stmf_lport_xfer_start(stmf_i_scsi_task_t *itask,
159     stmf_data_buf_t *dbuf);
160 static void stmf_lport_xfer_done(stmf_i_scsi_task_t *itask,
161     stmf_data_buf_t *dbuf);

163 static void stmf_update_kstat_lu_q(scsi_task_t *, void());
164 static void stmf_update_kstat_lport_q(scsi_task_t *, void());
165 static void stmf_update_kstat_lu_io(scsi_task_t *, stmf_data_buf_t *);
166 static void stmf_update_kstat_lport_io(scsi_task_t *, stmf_data_buf_t *);

168 static int stmf_irport_compare(const void *void_irport1,
169     const void *void_irport2);
170 static stmf_i_remote_port_t *stmf_irport_create(scsi_devid_desc_t *rport_devid);
171 static void stmf_irport_destroy(stmf_i_remote_port_t *irport);
172 static stmf_i_remote_port_t *stmf_irport_register(
173     scsi_devid_desc_t *rport_devid);
174 static stmf_i_remote_port_t *stmf_irport_lookup_locked(
175     scsi_devid_desc_t *rport_devid);
176 static void stmf_irport_deregister(stmf_i_remote_port_t *irport);

178 extern struct mod_ops mod_driverops;

180 /* =====[ Tunables ]===== */
181 /* Internal tracing */
182 volatile int    stmf_trace_on = 1;
183 volatile int    stmf_trace_buf_size = (1 * 1024 * 1024);
184 /*
185  * The reason default task timeout is 75 is because we want the
186  * host to timeout 1st and mostly host timeout is 60 seconds.
187  */
188 volatile int    stmf_default_task_timeout = 75;
189 /*
190  * Setting this to one means, you are responsible for config load and keeping
191  * things in sync with persistent database.
192  */
193 volatile int    stmf_allow_modunload = 0;
```



```
195 volatile int stmf_max_nworkers = 256;
196 volatile int stmf_min_nworkers = 4;
197 volatile int stmf_worker_scale_down_delay = 20;

199 /* === [ Debugging and fault injection ] === */
200 #ifdef  DEBUG
201 volatile uint32_t stmf_drop_task_counter = 0;
202 volatile uint32_t stmf_drop_buf_counter = 0;
201 volatile int stmf_drop_task_counter = 0;
202 volatile int stmf_drop_buf_counter = 0;

204 #endif

206 stmf_state_t            stmf_state;
207 static stmf_lu_t        *dlun0;

209 static uint8_t stmf_first_zero[] =
210         { 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 0xff };
211 static uint8_t stmf_first_one[] =
212         { 0xff, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0 };

214 static kmutex_t trace_buf_lock;
215 static int      trace_buf_size;
216 static int      trace_buf_curndx;
217 caddr_t stmf_trace_buf;

219 static enum {
220         STMF_WORKERS_DISABLED = 0,
221         STMF_WORKERS_ENABLING,
222         STMF_WORKERS_ENABLED
223 } stmf_workers_state = STMF_WORKERS_DISABLED;
______unchanged_portion_omitted_


4577 /*
4578  * ++++++++++++++ ABORT LOGIC ++++++++++++++++++++
4579  * Once ITASK_BEING_ABORTED is set, ITASK_KNOWN_TO_LU can be reset already
4580  * i.e. before ITASK_BEING_ABORTED being set. But if it was not, it cannot
4581  * be reset until the LU explicitly calls stmf_task_lu_aborted(). Of course
4582  * the LU will make this call only if we call the LU's abort entry point.
4583  * we will only call that entry point if ITASK_KNOWN_TO_LU was set.
4584  *
4585  * Same logic applies for the port.
4586  *
4587  * Also ITASK_BEING_ABORTED will not be allowed to set if both KNOWN_TO_LU
4588  * and KNOWN_TO_TGT_PORT are reset.
4589  *
4590  * ++++++++++++++++++++++++++++++++++++++++++++++++
4591  */

4593 stmf_status_t
4594 stmf_xfer_data(scsi_task_t *task, stmf_data_buf_t *dbuf, uint32_t ioflags)
4595 {
4596         stmf_status_t ret = STMF_SUCCESS;

4598         stmf_i_scsi_task_t *itask =
4599             (stmf_i_scsi_task_t *)task->task_stmf_private;

4601         stmf_task_audit(itask, TE_XFER_START, ioflags, dbuf);

4603         if (ioflags & STMF_IOF_LU_DONE) {
4604                 uint32_t new, old;
4605                 do {
4606                         new = old = itask->itask_flags;
4607                         if (new & ITASK_BEING_ABORTED)
```

```
4608                                return (STMF_ABORTED);
4609                        new &= ~ITASK_KNOWN_TO_LU;
4610                } while (atomic_cas_32(&itask->itask_flags, old, new) != old);
4611        }
4612        if (itask->itask_flags & ITASK_BEING_ABORTED)
4613                return (STMF_ABORTED);
4614 #ifdef  DEBUG
4615        if (!(ioflags & STMF_IOF_STATS_ONLY) && stmf_drop_buf_counter > 0) {
4616                if (atomic_dec_32_nv(&stmf_drop_buf_counter) == 1)
4616                atomic_dec_32_nv((uint32_t *)&stmf_drop_buf_counter) ==
4617                    1)
4617                        return (STMF_SUCCESS);
4618        }
4619 #endif

4621        stmf_update_kstat_lu_io(task, dbuf);
4622        stmf_update_kstat_lport_io(task, dbuf);
4623        stmf_lport_xfer_start(itask, dbuf);
4624        if (ioflags & STMF_IOF_STATS_ONLY) {
4625                stmf_lport_xfer_done(itask, dbuf);
4626                return (STMF_SUCCESS);
4627        }

4629        dbuf->db_flags |= DB_LPORT_XFER_ACTIVE;
4630        ret = task->task_lport->lport_xfer_data(task, dbuf, ioflags);

4632        /*
4633         * Port provider may have already called the buffer callback in
4634         * which case dbuf->db_xfer_start_timestamp will be 0.
4635         */
4636        if (ret != STMF_SUCCESS) {
4637                dbuf->db_flags &= ~DB_LPORT_XFER_ACTIVE;
4638                if (dbuf->db_xfer_start_timestamp != 0)
4639                        stmf_lport_xfer_done(itask, dbuf);
4640        }

4642        return (ret);
4643 }
```

*unchanged_portion_omitted_*

```
6183 void
6184 stmf_worker_task(void *arg)
6185 {
6186        stmf_worker_t *w;
6187        stmf_i_scsi_session_t *iss;
6188        scsi_task_t *task;
6189        stmf_i_scsi_task_t *itask;
6190        stmf_data_buf_t *dbuf;
6191        stmf_lu_t *lu;
6192        clock_t wait_timer = 0;
6193        clock_t wait_ticks, wait_delta = 0;
6194        uint32_t old, new;
6195        uint8_t curcmd;
6196        uint8_t abort_free;
6197        uint8_t wait_queue;
6198        uint8_t dec_qdepth;

6200        w = (stmf_worker_t *)arg;
6201        wait_ticks = drv_usectohz(10000);

6203        DTRACE_PROBE1(worker__create, stmf_worker_t, w);
6204        mutex_enter(&w->worker_lock);
6205        w->worker_flags |= STMF_WORKER_STARTED | STMF_WORKER_ACTIVE;
6206 stmf_worker_loop:;
6207        if ((w->worker_ref_count == 0) &&
6208            (w->worker_flags & STMF_WORKER_TERMINATE)) {
```



```
6209                w->worker_flags &= ~(STMF_WORKER_STARTED |
6210                    STMF_WORKER_ACTIVE | STMF_WORKER_TERMINATE);
6211                w->worker_tid = NULL;
6212                mutex_exit(&w->worker_lock);
6213                DTRACE_PROBE1(worker__destroy, stmf_worker_t, w);
6214                thread_exit();
6215        }
6216        /* CONSTCOND */
6217        while (1) {
6218                dec_qdepth = 0;
6219                if (wait_timer && (ddi_get_lbolt() >= wait_timer)) {
6220                        wait_timer = 0;
6221                        wait_delta = 0;
6222                        if (w->worker_wait_head) {
6223                                ASSERT(w->worker_wait_tail);
6224                                if (w->worker_task_head == NULL)
6225                                        w->worker_task_head =
6226                                            w->worker_wait_head;
6227                                else
6228                                        w->worker_task_tail->itask_worker_next =
6229                                            w->worker_wait_head;
6230                                w->worker_task_tail = w->worker_wait_tail;
6231                                w->worker_wait_head = w->worker_wait_tail =
6232                                    NULL;
6233                        }
6234                }
6235                if ((itask = w->worker_task_head) == NULL) {
6236                        break;
6237                }
6238                task = itask->itask_task;
6239                DTRACE_PROBE2(worker__active, stmf_worker_t, w,
6240                    scsi_task_t *, task);
6241                w->worker_task_head = itask->itask_worker_next;
6242                if (w->worker_task_head == NULL)
6243                        w->worker_task_tail = NULL;

6245                wait_queue = 0;
6246                abort_free = 0;
6247                if (itask->itask_ncmds > 0) {
6248                        curcmd = itask->itask_cmd_stack[itask->itask_ncmds - 1];
6249                } else {
6250                        ASSERT(itask->itask_flags & ITASK_BEING_ABORTED);
6251                }
6252                do {
6253                        old = itask->itask_flags;
6254                        if (old & ITASK_BEING_ABORTED) {
6255                                itask->itask_ncmds = 1;
6256                                curcmd = itask->itask_cmd_stack[0] =
6257                                    ITASK_CMD_ABORT;
6258                                goto out_itask_flag_loop;
6259                        } else if ((curcmd & ITASK_CMD_MASK) ==
6260                            ITASK_CMD_NEW_TASK) {
6261                                /*
6262                                 * set ITASK_KSTAT_IN_RUNQ, this flag
6263                                 * will not reset until task completed
6264                                 */
6265                                new = old | ITASK_KNOWN_TO_LU |
6266                                    ITASK_KSTAT_IN_RUNQ;
6267                        } else {
6268                                goto out_itask_flag_loop;
6269                        }
6270                } while (atomic_cas_32(&itask->itask_flags, old, new) != old);

6272 out_itask_flag_loop:

6274                /*
```

```
6275                 * Decide if this task needs to go to a queue and/or if
6276                 * we can decrement the itask_cmd_stack.
6277                 */
6278                if (curcmd == ITASK_CMD_ABORT) {
6279                        if (itask->itask_flags & (ITASK_KNOWN_TO_LU |
6280                            ITASK_KNOWN_TO_TGT_PORT)) {
6281                                wait_queue = 1;
6282                        } else {
6283                                abort_free = 1;
6284                        }
6285                } else if ((curcmd & ITASK_CMD_POLL) &&
6286                    (itask->itask_poll_timeout > ddi_get_lbolt())) {
6287                        wait_queue = 1;
6288                }

6290                if (wait_queue) {
6291                        itask->itask_worker_next = NULL;
6292                        if (w->worker_wait_tail) {
6293                                w->worker_wait_tail->itask_worker_next = itask;
6294                        } else {
6295                                w->worker_wait_head = itask;
6296                        }
6297                        w->worker_wait_tail = itask;
6298                        if (wait_timer == 0) {
6299                                wait_timer = ddi_get_lbolt() + wait_ticks;
6300                                wait_delta = wait_ticks;
6301                        }
6302                } else if ((--(itask->itask_ncmds)) != 0) {
6303                        itask->itask_worker_next = NULL;
6304                        if (w->worker_task_tail) {
6305                                w->worker_task_tail->itask_worker_next = itask;
6306                        } else {
6307                                w->worker_task_head = itask;
6308                        }
6309                        w->worker_task_tail = itask;
6310                } else {
6311                        atomic_and_32(&itask->itask_flags,
6312                            ~ITASK_IN_WORKER_QUEUE);
6313                        /*
6314                         * This is where the queue depth should go down by
6315                         * one but we delay that on purpose to account for
6316                         * the call into the provider. The actual decrement
6317                         * happens after the worker has done its job.
6318                         */
6319                        dec_qdepth = 1;
6320                        itask->itask_waitq_time +=
6321                            gethrtime() - itask->itask_waitq_enter_timestamp;
6322                }

6324                /* We made it here means we are going to call LU */
6325                if ((itask->itask_flags & ITASK_DEFAULT_HANDLING) == 0)
6326                        lu = task->task_lu;
6327                else
6328                        lu = dlun0;
6329                dbuf = itask->itask_dbufs[ITASK_CMD_BUF_NDX(curcmd)];
6330                mutex_exit(&w->worker_lock);
6331                curcmd &= ITASK_CMD_MASK;
6332                stmf_task_audit(itask, TE_PROCESS_CMD, curcmd, dbuf);
6333                switch (curcmd) {
6334                case ITASK_CMD_NEW_TASK:
6335                        iss = (stmf_i_scsi_session_t *)
6336                            task->task_session->ss_stmf_private;
6337                        stmf_itl_lu_new_task(itask);
6338                        if (iss->iss_flags & ISS_LUN_INVENTORY_CHANGED) {
6339                                if (stmf_handle_cmd_during_ic(itask))
6340                                        break;
```



```
6341                        }
6342 #ifdef  DEBUG
6343                        if (stmf_drop_task_counter > 0) {
6344                                if (atomic_dec_32_nv(&stmf_drop_task_counter) ==
6345                                if (atomic_dec_32_nv((uint32_t *)&stmf_drop_task
6346                                    1) {
6345                                        break;
6346                        }
6349                        }
6347 #endif
6348                        DTRACE_PROBE1(scsi__task__start, scsi_task_t *, task);
6349                        lu->lu_new_task(task, dbuf);
6350                        break;
6351                case ITASK_CMD_DATA_XFER_DONE:
6352                        lu->lu_dbuf_xfer_done(task, dbuf);
6353                        break;
6354                case ITASK_CMD_STATUS_DONE:
6355                        lu->lu_send_status_done(task);
6356                        break;
6357                case ITASK_CMD_ABORT:
6358                        if (abort_free) {
6359                                stmf_task_free(task);
6360                        } else {
6361                                stmf_do_task_abort(task);
6362                        }
6363                        break;
6364                case ITASK_CMD_POLL_LU:
6365                        if (!wait_queue) {
6366                                lu->lu_task_poll(task);
6367                        }
6368                        break;
6369                case ITASK_CMD_POLL_LPORT:
6370                        if (!wait_queue)
6371                                task->task_lport->lport_task_poll(task);
6372                        break;
6373                case ITASK_CMD_SEND_STATUS:
6374                /* case ITASK_CMD_XFER_DATA: */
6375                        break;
6376                }
6377                mutex_enter(&w->worker_lock);
6378                if (dec_qdepth) {
6379                        w->worker_queue_depth--;
6380                }
6381        }
6382        if ((w->worker_flags & STMF_WORKER_TERMINATE) && (wait_timer == 0)) {
6383                if (w->worker_ref_count == 0)
6384                        goto stmf_worker_loop;
6385                else {
6386                        wait_timer = ddi_get_lbolt() + 1;
6387                        wait_delta = 1;
6388                }
6389        }
6390        w->worker_flags &= ~STMF_WORKER_ACTIVE;
6391        if (wait_timer) {
6392                DTRACE_PROBE1(worker__timed__sleep, stmf_worker_t, w);
6393                (void) cv_reltimedwait(&w->worker_cv, &w->worker_lock,
6394                    wait_delta, TR_CLOCK_TICK);
6395        } else {
6396                DTRACE_PROBE1(worker__sleep, stmf_worker_t, w);
6397                cv_wait(&w->worker_cv, &w->worker_lock);
6398        }
6399        DTRACE_PROBE1(worker__wakeup, stmf_worker_t, w);
6400        w->worker_flags |= STMF_WORKER_ACTIVE;
6401        goto stmf_worker_loop;
6402 }
```

_____unchanged_portion_omitted_