SCSI Target Mode Framework

Version 1.2

September 2007

Table of Contents

1 Introduction

The SCSI Target Mode Framework(stmf) helps developers to create software to use a solaris host as a SCSI Target. The SCSI Target subsystem can be divided into the following layers

Logical Unit Providers

These are kernel modules which implement the functionality of a SCSI Logical Unit. They interface with stmf through the LU Provider interface. The functionality implemented by LU Providers is transport protocol independent.

SCSI Target Mode Framework

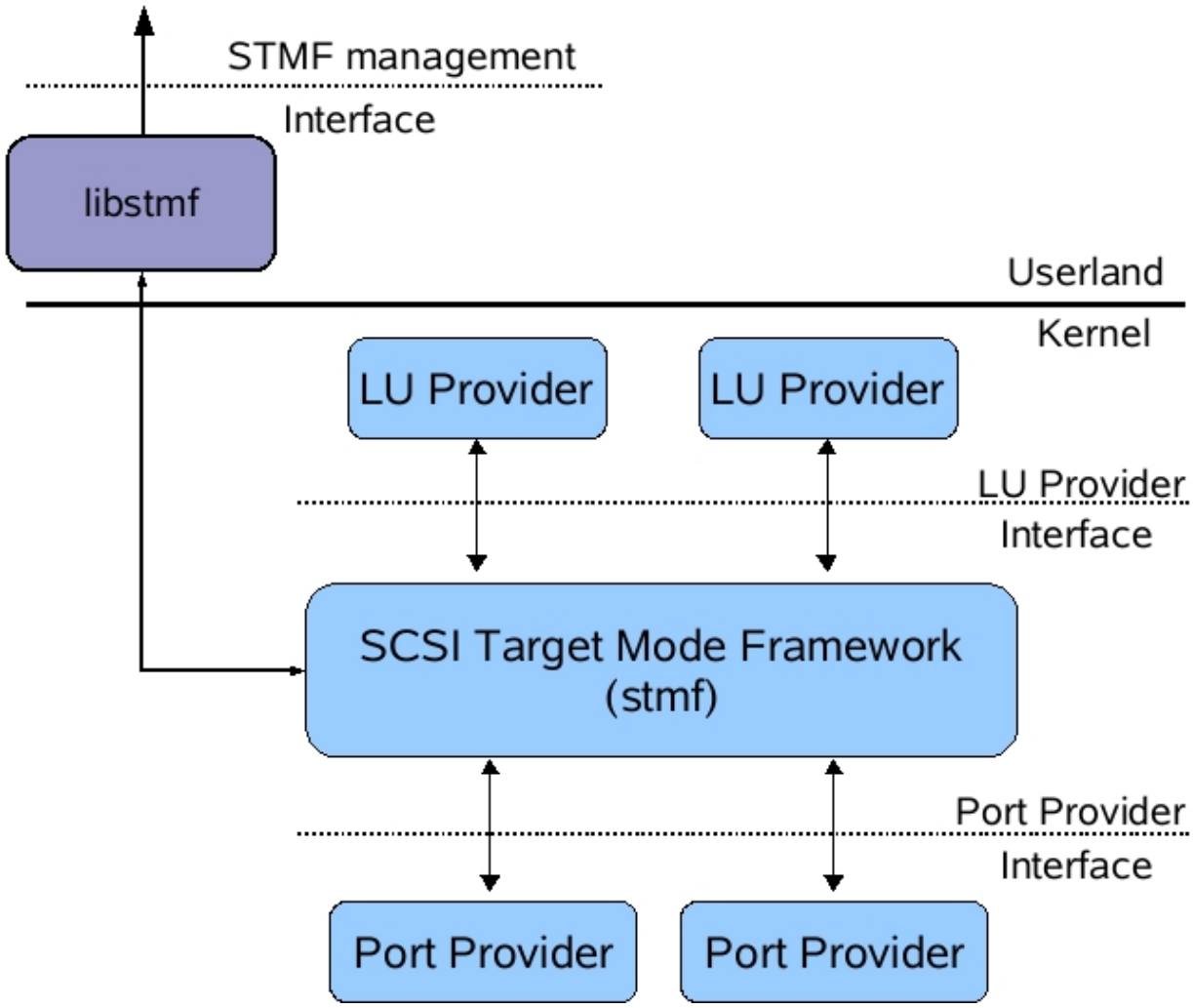
The framework itself performs a number of different tasks.

- Manages and keeps track of LU Providers and Target Port Providers.
- Manages LUN Mappings for an Initiator and I_T sessions.
- Manages context and resources for SCSI command execution.
- Handles abnormal termination of commands and cleanup.

Port Providers

These are kernel modules which implement the protocol for SCSI command transport e.g. Fibre channel, iSCSI, SAS, iSER etc. and provide local ports to be used as targets. The port providers only focus on the transport protocol and dont need to worry about SCSI command functionality of the Logical units exposed through the port.

1.1 High level view of the framework



2 Terms

The terms that are used in this specification are defined in this section.

Term	Definition
STMF, Framework	SCSI Target Mode Framework.
LU Provider	The kernel subsystem that exports Logical Units to STMF
LU, Logical Unit	The software that implements the functionality of a Logical Unit.
Port Provider	The kernel subsystem that exports Local Ports to STMF which implement the functionality of a SCSI target.
LPORT, Port	The software that implements the functionality of a SCSI Target.
Remote Port	An entity talking to the LPORT i.e. The initiator.
Transport	A subsystem for sending and receiving SCSI tasks. It consists of the LPORT and related support functions from STMF.

3 Constants and types

3.1 stmf_status_t

Return status from the interface entry points.

Format

```
typedef uint64_t    stmf_status_t;
```

Values

STMF_SUCCESS

The operation was successful.

STMF_FAILURE

A generic failure code to indicate that the operation failed in the stmf layer.

STMF_BUSY

The operation cannot be performed at this time but can be retried at a later time.

STMF_NOT_FOUND

The Requested object was not found.

STMF_INVALID_ARG

Atleast one of the arguments is not valid

STMF_LUN_TAKEN

The Logical Unit Number is already in use.

STMF_ABORTED

The operation was aborted

STMF_ABORT_SUCCESS

The request to abort has completed successfully.

STMF_TARGET_FAILURE

A generic error code to indicate a target level failure. Target port providers can use STMF_FSC() macro to define more error codes.

STMF_LU_FAILURE

A gereric error code to indicate a LU level failure. LU Providers can use STMF_FSC() macro to define more error codes.

STMF_ALLOC_FAILURE

Allocation of an object or memory failed.

STMF_ALREADY

The requested action is already in progress or has already completed

Macros

STMF_FSC(x)

Defines a failure sub code. e.g. a target port can define failure codes as follows

```
#define FCT_TRAN_ERROR   (STMF_TARGET_FAILURE | STMF_FSC(3))
```

STMF_GET_FSC(x)

Returns the failure subcode.

3.2 stmf_struct_id_t

IDs for different structures allocated and freed by the framework.

Format

```
typedef enum stmf_struct_id {
        STMF_STRUCT_LU_PROVIDER = 1,
        STMF_STRUCT_PORT_PROVIDER,
        STMF_STRUCT_STMF_LOCAL_PORT,
        STMF_STRUCT_STMF_LU,
        STMF_STRUCT_SCSI_SESSION,
        STMF_STRUCT_SCSI_TASK,
        STMF_STRUCT_DATA_BUF
} stmf_struct_id_t;
```

Fields

STMF_STRUCT_LU_PROVIDER

Identifier to represent data type stmf_lu_provider_t

STMF_STRUCT_PORT_PROVIDER

Identifier to represent data type stmf_port_provider_t

STMF_STRUCT_STMF_LOCAL_PORT

Identifier to represent data type stmf_local_port_t

STMF_STRUCT_STMF_LU

Identifier to represent data type stmf_lu_t

STMF_STRUCT_SCSI_SESSION

Identifier to represent data type stmf_scsi_session_t

STMF_STRUCT_SCSI_TASK

Identifier to represent data type scsi_task_t

STMF_STRUCT_DATA_BUF

Odentifier to represent data type stmf_data_buf_t

Notes

All the data structures that are share between a provide module and stmf are divided into three structures. A framework specific structure, shared structure and module specific structure. Memory for all the three structures is allocated by the framework. The shared structure maintains two pointers in the beginning of the structure. The first pointer points to the framework specific structure and the second pointer is initialized to the module specific

structure. Providers use stmf_alloc() and stmf_free() interfaces to allocate and free these structures. These are described in the interface section below.

3.3 stmf_lu_provider_t

Structure representing the LU Provider.

Format

```
typedef struct stmf_lu_provider {
        void                    *lp_stmf_private;
        void                    *lp_private;
        uint32_t                lp_lpif_rev;
        int                     lp_instance;
        char                    *lp_name;
} stmf_lu_provider_t;
```

Fields

lp_lpif_rev

This is set by the LU provider to the LU Provider rev. supported by the provider. Current rev value is LPIF_REV_1

lp_instance

Instance number of the LU provider. Can be initialized to zero in case the provider does not have an instance assigned.

lp_name

The provider sets this value to a NULL terminated string representing the name of the provider. This string is used by management applications.

Notes

The stmf_lu_provider_t represents the kernel module that exports zero or more SCSI logical units to the target mode framework. This structure allows the provider to register with and validate the framework without exporting any logical units.

3.4 stmf_port_provider_t

Structure representing the Port Provider.

Format

```
typedef struct stmf_port_provider {
        void                    *pp_stmf_private;
        void                    *pp_provider_private;

        uint32_t                pp_portif_rev;
        int                     pp_instance;
        char                    *pp_name;
} stmf_port_provider_t;
```

Fields

pp_portif_rev

This is set by the port provider to the Port Provider rev. supported by the provider. Current rev value is PORTIF_REV_1

pp_instance

Instance number of the provider. Can be initialized to zero if the provider does not have an instance assigned.

pp_name

The provider sets this value to a NULL terminated string representing the name of the provider. This string is used by management applications.

Notes

The stmf_port_provider_t represents the kernel module that exports zero or more SCSI ports to the target mode framework. This structure allows the provider to register with and validate the framework without exporting any ports.

3.5 scsi_devid_desc_t

Device identification descriptor for both LUs and LPORTs as per SPC3.

Format

```
typedef struct scsi_devid_desc {
#ifdef  _BIT_FIELDS_HTOL
        uint8_t         protocol_id:4,
                        code_set:4;
        uint8_t         piv:1,
                        rsvd1:1,
                        association:2,
                        ident_type:4;
#else
        uint8_t         code_set:4,
                        protocol_id:4;
        uint8_t         ident_type:4,
                        association:2,
                        rsvd1:1,
                        piv:1;
#endif
        uint8_t         rsvd2;
        uint8_t         ident_length;
        uint8_t         ident[1];
} scsi_devid_desc_t;
```

Notes

scsi_devid_desc_t represents device identification descriptor as defined in SPC3 section 7.6.3. The device identification descriptor can be used to represent both Logical units and targets.

The LUs can have multiple device identification descriptors and report them in the INQUIRY VPD page 0x83. However when registering with stmf, they should use the ident_type of ID_TYPE_NAA (3) and NAA type of 6 (GUID).

3.6 stmf_data_buf_t

Structure to abstract out a data buffer and the transfer associated with it.

Format

```
typedef struct tgt_data_buf {
        void            *db_stmf_private;
        void            *db_port_private;
        void            *db_lu_private;
        uint32_t        db_buf_size;
        uint32_t        db_data_size;
        uint32_t        db_relative_offset;
        uint16_t        db_flags;
        uint16_t        db_sglist_length;
        stmf_status_t   db_xfer_status;
        uint8_t         db_handle;
        struct stmf_sglist_ent {
               uint32_t       seg_length;
               uint8_t        *seg_addr;
        }               db_sglist[1];
} stmf_data_buf_t;
```

Fields

db_lu_private

Pointer for the LU to store any buffer specific information.

db_buf_size

Total size in bytes for the entire buffer.

db_data_size

Size in bytes of the data contained within this buffer.

db_relative offset

Relative offset of this buffer in the data transfer phase of the SCSI command.

db_flags

This is a bit field with following bit values defined:

DB_DIRECTION_TO_RPORT

Transmit data to remote port.

DB_DIRECTION_FROM_RPORT

Receive data from remote port.

DB_SEND_STATUS_GOOD

This flag is set to request transfer of SCSI status good with no residual counts along with the data.

DB_STATUS_GOOD_SENT

This flag is set by the transport layer to indicate that SCSI good status has been successfully sent and no additional status transfer completion will be posted.

db_sglist_length

Number of buffers (scatter gather list) which makeup the entire data space of this buffer.

db_xfer_status

Transfer completion status for this buffer.

db_handle

Handle assigned to the buffer by the framework within the context of a SCSI task. This is used by the framework to track buffers a given SCSI task has.

db_sglist

The scatter gather list describing this buffer. The structure defines the first element but during allocation the port can request bigger size for the structure and hence can have multiple elements in the scatter gather list. The db_sglist has following members:

seg_length - Length of the segment.

seg_addr - Kernel address of the segment.

3.7 stmf_scsi_session_t

Structure representing a I_T nexus established when a remote port performs a SCSI login into a LPORT.

Format

```
typedef struct stmf_scsi_session {
        void                    *ss_stmf_private;
        void                    *ss_port_private;

        struct scsi_devid_desc  *ss_rport_id;
        struct stmf_local_port  *ss_lport;
} stmf_scsi_session_t;
```

Fields

ss_rport_id

Remote port identifier. Defined as per SPC3.

ss_lport

Local port maintaining this session.

Notes

A stmf_scsi_session_t represents an I_T nexus. All the sessions are tracked on a stmf_local_port_t basis.

3.8 scsi_task_t

Structure used to manage the execution of a SCSI task.

Format

```
typedef struct scsi_task {
        void            *task_stmf_private;
        void            *task_port_private;

        void            *task_lu_private;
        struct stmf_scsi_session *task_session;
        struct stmf_local_port *task_lport;
        struct stmf_lu  *task_lu;

        uint8_t         task_lun_no[8];
        uint8_t         task_flags;
        uint8_t         task_priority;
        uint8_t         task_mgmt_function;
        uint8_t         task_cur_nbufs;
        uint16_t        task_additional_flags;
        uint8_t         task_cur_nbufs;
        uint32_t        task_cmd_seq_no;
        uint32_t        task_expected_xfer_length;
        uint8_t         task_cdb[16];
        uint32_t        task_cmd_xfer_length;
        uint32_t        task_nbytes_transferred;

        stmf_status_t   task_completion_status;
        uint32_t        task_resid;
        uint8_t         task_status_ctrl;
        uint8_t         task_scsi_status;
        uint16_t        task_sense_length;
        uint8_t         *task_sense_data;

        /* Misc. task data */
        void            *task_extended_cmd;
} scsi_task_t;
```

Fields

task_lu_private

Pointer for Logical Unit Emulation code to store per task data.

task_session

Pointer to session structure representing the I_T nexus.

task_lport

Pointer to local port which has received this task.

task_lu

Pointer to the logical unit processing the task.

task_lun_no

8 byte LUN received with the task.

task_flags

This is a bit field with the following values

TF_INITIAL_BURST

The task received an initial burst of data.

TF_READ_DATA

The data direction is towards the remote port.

TF_WRITE_DATA

The data direction is towards the local port.

TF_ATTR_MASK

This is a mask to obtain the SCSI Task attributes as defined by SAM. The different attribute values are:

TF_ATTR_UNTAGGED

TF_ATTR_SIMPLE_QUEUE

TF_ATTR_ORDERED_QUEUE

TF_ATTR_HEAD_OF_QUEUE

TF_ATTR_ACA

task_priority

Task priority as defined by SAM-3

task_mgmt_function

Task management function as defined under SAM-3. This can have following values. A lot of these values were choosen to be in sync. with the iSCSI spec also.

TM_NONE

TM_ABORT_TASK

TM_ABORT_TASK_SET

TM_CLEAR_ACA

TM_CLEAR_TASK_SET

TM_LUN_RESET

TM_TARGET_WARM_RESET

TM_TARGET_COLD_RESET

TM_TASK_REASSIGN

task_cur_nbufs

Number of parallel data buffers currently in use by the task.

task_additional_flags

This is a bit field to defined additional attributes beyond what is defined by SAM. Currently the only attribute defined is TASK_AF_ENABLE_COMP_CONF to enable fibre channel completion confirmations.

task_max_nbufs

This field is set by the framework to indicate maximum number of parallel data buffers the task can use. LPORT should initialize this field to maximum number of parallel data buffers it can handle or STMF_BUFS_MAX if it has no such limits. STMF will adjust this field based on its own limits before passing this task on to the LU.

task_cmd_seq_no

Command sequence number. This is a 32 bit integer to cover most of the cases. It is defined as 32 bit in iSCSI spec but only a 8 bit integer in Fibre channel (FCP) spec.

task_expected_xfer_length

This field contains the expected data transfer size as requested by the initiator as a part of the transport protocol.

task_cdb

The SCSI CDB. This field contains the initial 16 bytes, which covers the most common case. 'task_extended_cmd' field will be used in case the CDB extends beyond 16 bytes.

task_cmd_xfer_length

This field in initialized by the LU to the data transfer length as per the command. This field indicates the actual amount of data the LU should transfer based on the command and independent of the 'task_expected_xfer_length'. This is used to calculate the residual count and 'underrun' or 'overrun' conditions by the framework.

task_nbytes_transferred

This field is initialized to the actual number of bytes which were transferred. This field is kept updated by the LU.

task_completion_status

This field indicates how the task completed. The only time task_completion_status is not STMF_SUCCESS, is when the task is aborted due to some reason.

task_resid

The residual count indicating the amount of data not transferred as compared with task_expected_xfer_length. This value could indicate an overrun or underrun depending upon the task_status_ctrl value.

task_status_ctrl

This field contains additional information about task SCSI status. It can have the following values:

TASK_SCTRL_OVER

The LU was supposed to transfer more data than what was indicated by task_expected_xfer_length.

TASK_SCTRL_UNDER

The LU transferred less data than what was indicated by task_expected_xfer_length.

task_scsi_status

SCSI status value to be sent to the remote port.

task_sense_length

Length of the sense data, if any, pointed to by task_sense_data.

task_sense_data

The sense information. The SCSI status must be 2 (CHECK CONDITION) in this case.

The buffer pointed to by this pointer is no longer needed (copied by transport) after the call has been made to send SCSI status.

task_extended_cmd

Pointer for defining extensions to scsi_task_t. None are defined at this point.

3.9 stmf_state_change_info_t

Information about the state change request through a (*lu_ctl)() or (*lport_ctl)() entry point.

Format

```
typedef struct stmf_state_change_info {
        uint64_t        st_rflags;
        char            *st_additional_info;
} stmf_state_change_info_t;
```

Fields

st_rflags

Bit flags describing the reason for this change. Defined flags are:

STMF_RFLAG_USER_REQUEST

The state change was initiated by the user.

STMF_RFLAG_FATAL_ERROR

The state change was initiated due to a fatal error.

STMF_RFLAG_STAY_OFFLINE

Unless requested b user, an online request should not be initiated after the offline has completed.

STMF_RFLAG_RESET

Online the LU or LPORT after offline (Reset).

STMF_RFLAG_COLLECT_DEBUG_DUMP

Collect debug dump as a part of offline.

st_additional_info

A text string describing why the state change was requested. Can also be NULL. When stmf_state_change_info_t is passed to stmf_ctl(), the framework makes a local copy of the entire structure including st_additional_info (if any). So the caller can allocate this string on the stack if needed.

3.10 stmf_state_change_status_t

Completion status of a state change request.

Format

```
typedef struct stmf_change_status {
        stmf_status_t   st_completion_status;
        char            *st_additional_info;
} stmf_change_status_t;
```

Fields

st_completion_status

Completion status of the change request.

st_additional_info

A text string describing the completion status. Can also be NULL.

4 LU entry points

LU exports entry points through the stmf_lu_t structure described below.

Format

```
typedef struct stmf_lu {
        void                    *lu_stmf_private;
        void                    *lu_provider_private;

        struct scsi_devid_desc  *lu_id;
        struct stmf_lu_provider *lu_lp;

        stmf_status_t           (*lu_task_alloc)(
            struct scsi_task *task);
        void                    (*lu_new_task)(
            struct scsi_task *task, struct tgt_data_buf
            *initial_dbuf);
        void                    (*lu_dbuf_xfer_done)(
            struct scsi_task *task, struct tgt_data_buf
            *dbuf);
        void                    (*lu_send_status_done)(
            struct scsi_task *task);
        void                    (*lu_task_free)(
            struct scsi_task *task);
        stmf_status_t           (*lu_abort)(
            struct stmf_lu *lu, int abort_cmd, void *arg,
            uint32_t flags);
        void                    (*lu_ctl)(
            struct stmf_lu *lu, int cmd, void *arg);
} stmf_lu_t;
```

Fields

lu_id

Device identification descriptor for the logical unit. This descriptor has to have the ident_type of ID_TYPE_NAA (3) and NAA type of 6 (GUID).

lu_lp

Pointer to the LU provider structure for this Logical Unit.

4.1 lu_task_alloc

This enatry point is called whenever the framework is allocating a new scsi_task_t for this logical unit. The purpose of this entry point is to allocate per task logical unit specific data and initialize the task_lu_private member of the scsi_task_t with that data.

Format

```
stmf_status_t (*lu_task_alloc)(struct scsi_task *task);
```

Parameters

task

Pointer to the scsi task structure for which the LU needs to do allocations.

Return value

STMF_SUCCESS

The operation completed successfully.

STMF_ALLOC_FAILURE

The LU could not allocate task private data. This will result in stmf aborting the task.

Notes

The LU frees this data when the framework calls the lu_task_free() entry point. The tasks are cached by the framework. Hence the framework might not call lu_task_alloc or lu_task_free for every task. This optimization saves the LU from allocating a per task structure every time.

4.2 lu_new_task

This entry point is called whenever a new SCSI task is received by the framework for this Logical Unit.

Format

```
void (*lu_new_task)(sruct scsi_task *task, struct
                            tgt_data_buf *initial_dbuf);
```

Parameters

task

Pointer to the SCSI task struct allocated by the framework.

initial_dbuf

The transport may pass on a data buffer with the task. In case the transport receives an initial data burst with the command, task_flags field will be used to indicate that there is data in this buffer other wise the buffer is empty and is passed by the transport to save a call to allocate buffer. The initial_dbuf pointer may be NULL.

4.3 lu_dbuf_xfer_done

This entry point is called by the framework whenever a data transfer operation completes for a SCSI task.

Format

```
void (*lu_dbuf_xfer_done)(struct scsi_task *task,
                          struct stmf_data_buf *dbuf);
```

Parameters

task

scsi_task_t associated with this transfer.

dbuf

stmf_data_buf_t for which the transfer has completed.

Notes

If the logical unit has already indicated that it is done processing this task, then this entry point may not be called by the framework. Logical unit should be able to handle this call even after telling the framework that it is done processing a task.

If the buffer transfer request also requested a transfer of scsi STATUS GOOD and the transport was able to successfully send both buffer and status as a part of the same request, the db_flags bit DB_STATUS_GOOD_SENT will be set. Otherwise the framework will call the lu_send_status_done() entry point.

4.4 lu_send_status_done

This entry point is called when the framework completes the transfer of SCSI status.

Format

```
void (*lu_send_status_done)(scsi_task_t *task);
```

Parameters

task

The SCSI task for which the status transfer has completed.

Notes

The task_completion_status member of the scsi_task structure contains the status of the status transfer request.

This entry point may not be called if the LU has already indicated to the framework that it is done with this task.

4.5 lu_task_free

This entry point is called when the framework is about to free the SCSI task structure.

Format

```
void (*lu_task_free)(scsi_task_t *task);
```

Parameters

task

The SCSI task which is being freed.

Notes

This entry point is only called after the LU has indicated to the framework that it is done processing the task. The only purpose of this entry point is for the LU to free its per task structure. The LU should have released any other resource it has for the task before indicating to the framework that it is done processing the task.

The framework might not call this entry point after a task is done to keep the LU's per task structure cached and the next call to lu_new_task() entry point may reuse the this structure. In this case the task_lu_private member of the scsi_task_t will not be NULL.

4.6 lu_abort

This entry point is called when the framework wants the LU to abort processing of a task.

Format

```
stmf_status_t (*lu_abort)(struct stmf_lu *lu, int abort_cmd
                          void *arg, uint32_t flags);
```

Parameters

lu

Pointer to the logical unit structure.

abort_cmd

The abort operation being requested by the framework. Currently defined operations are:

STMF_LU_ABORT_TASK

Abort the task pointed to by the 'arg' parameter.

arg

abort_cmd specific arguments.

flags

Any additional conditions modifying the abort behavior. Currently no flags are defined.

Return values

The LU should return one of the following values in response to this call.

STMF_SUCESS

The LU has accepted the abort request and will abort the task asynchronously.

STMF_BUSY

The LU cannot accept this request at this time. The framework should retry again after some time.

STMF_NOT_FOUND

The LU is not aware of the task. This is not considered a failure by the framework.

STMF_ABORTED

The LU has completed the request synchronously.

Notes

If the abort operation is being performed asynchronously the LU must call stmf_task_lu_aborted() to indicate completion of the abort operation.

4.7 lu_ctl

Logical Unit control entry point.

Format

```
void (*lu_ctl)(struct stmf_lu *lu, int cmd,
                                        void *arg);
```

Parameters

lu

Pointer to the logical unit structure itself.

cmd

Control command. Currently defined values are:

STMF_CMD_LU_ONLINE

Bring the logical unit online. arg (3rd argument to the entry point) points to stmf_state_change_info_t.

STMF_CMD_LU_OFFLINE

Bring the LU offline. arg points to stmf_state_change_info_t

STMF_ACK_LU_ONLINE_COMPLETE

An acknowledgement from the framework that everythi ng that is needed to bring this LU online has been done. This is sent in response to a call from the LU to stmf_ctl() interface with a CMD calue of STMF_CMD_LU_ONLINE_COMPLETE. arg is a pointer to the stmf_change_status_t, passed in the stmf_ctl() call.

STMF_ACK_LU_OFFLINE_COMPLETE

An acknowledgement from the framework that the framework has removed all references to this LU. After this call the LU can deregister from the framework and even unload itself if needed.

arg

Argument specific to the cmd.

Notes

- When a LU is offlined, it does not have to worry about terminating existing commands as the framework takes care of doing that before calling the (*lu_ctl)() entry point. As soon as the LU has done its internal shutdown (e.g. close any file descriptors etc.), it should call stmf_ctl(STMF_CMD_LU_OFFLINE_COMPLETE, lu, ...).
- After the LU been offlined, it must return STMF_ABORT_SUCCESS from the (*lu_abort)() entry point if that entry point is called by the framework. This entry point will not be called after the framework ACKs the offline.

5 LPORT entry points

A LPORT exports entry points through the stmf_local_port_t as described below.

Format

```
typedef struct stmf_local_port {
  void                          *lport_stmf_private;
  void                          *lport_port_private;

  struct scsi_devid_desc        *lport_id;
  struct stmf_port_provider     *lport_pp;
  stmf_data_buf_t *             (*lport_alloc_data_buf)(
       struct stmf_local_port *lport, uint32_t size,
                      uint32_t *pminsize, uint32_t flags);
  void                           (*lport_free_data_buf)(
       struct stmf_local_port *lport, stmf_data_buf_t
                                      *dbuf);
  stmf_status_t                  (*lport_xfer_data)(
       struct scsi_task *task, struct tgt_data_buf *dbuf,
                                      uint32_t ioflags);
  stmf_status_t                  (*lport_send_status)(
       struct scsi_task *task, uint32_t ioflags);
  void                           (*lport_task_free)(
                                 struct scsi_task *task);
  stmf_status_t                  (*lport_abort)(
       struct stmf_local_port *lport, int abort_cmd,
                                 void *arg, uint32_t flags);
  void                           (*lport_ctl)(
    struct stmf_local_port *lport, int cmd, void *arg);
} stmf_local_port_t;
```

Fields

lport_id

Device identification descriptor for the LPORT.

lport_pp

Pointer to the port provider structure for this LPORT.

5.1 lport_alloc_data_buf

This entry point is called when the framework needs a buffer for data transfer.

Format

```
stmf_data_buf_t *(*lport_alloc_data_buf)(
  struct stmf_local_port *lport, uint32_t size,
                        uint32_t *pminsize, uint32_t flags);
```

Parameters

lport

Pointer to the local port structure.

size

Size in bytes of the buffer needed.

pminsiz

Pointer to a uint32_t containing the minimum buffer size needed by the caller.

flags

Additional modifier flags for the allocation. Currently no flags are defined.

Return Values

This entry point returns pointer to a data buffer of type stmf_data_buf_t. The return value can be NULL if the data buffer cannot be allocated in which case pminsize will be updated to indicate the minimum size of buffer that can be allocated. The LPORT can initialize the uint32_t pointed to by pminsize to zero to indicate that it cannot allocate any buffer size.

5.2 lport_free_data_buf

The framework calls this entry point to free a data buffer previously allocated via lport_alloc_data_buf() entry point.

Format

```
void (*lport_free_data_buf)(struct stmf_local_port *lport,
                                stmf_data_buf_t *dbuf);
```

Parameters

lport

Pointer to the local port structure.

dbuf

Pointer to the data buf to be freed.

Notes

The framework may cache the buffers and hence not call free for a buffer for a long time.

5.3 lport_xfer_data

This entry point is called by the framework when data needs to be transported to or from a data buffer for a SCSI task.

Format

```
stmf_status_t (*lport_xfer_data)(struct scsi_task *task,
     struct tgt_data_buf *dbuf, uint32_t ioflags);
```

Parameters

task

Pointer to the scsi task structure responsible for data transfer.

dbuf

Pointer to the data buffer which is the source or destination of the data depending upon the direction.

ioflags

Flags to control the I/O flow. None are defined at this point.

Return values

STMF_SUCCESS

The port has accepted the request to transfer data.

STMF_BUSY

The port cannot accept the request at this time. The framework should try later.

Any other return value will result in framework aborting this task.

5.4 lport_send_status

This entry point is called by the framework to send scsi status to the initiator.

Format

```
stmf_status_t (*lport_send_status)(struct scsi_task *task,
                                   uint32_t ioflags);
```

Parameters

task

task for which the scsi status is to be sent

ioflags

Flags to control the I/O flow. None are defined at this point.

Return values

STMF_SUCCESS

The port has accepted the request to transfer status.

STMF_BUSY

The port cannot accept the request at this time. The framework should try later.

Any other return value will result in framework aborting this task.

Notes

task_scsi_status member of scsi_task_t contains the scsi status value to be sent. task_status_ctrl provides the information about and underrun or overrun that might have occurred during the execution of this task in which case task_resid contains any residual count. If the task had a check condition then task sense length will be nonzero and task_sense_data will point to a buffer with the sense data information. The port should copy the sense data before returning from this call.

5.5 lport_task_free

This entry point is called when the framework is about to free the SCSI task structure.

Format

```
void (*lport_task_free)(scsi_task_t *task);
```

Parameters

task

The SCSI task which is being freed.

Notes

This entry point is only called after the target port has indicated to the framework that it is done processing the task. The only purpose of this entry point is for the port to free its per task structure. The LU should have released any other resource it has for the task before indicating to the framework that it is done processing the task.

5.6 lport_abort

This entry point is called by the framework to abort processing of a task.

Format

```
stmf_status_t (*lport_abort)(struct stmf_local_port_t *lport,
          int abort_cmd, void *arg, uint32_t flags);
```

Parameters

lport

Pointer to the local port structure.

abort_cmd

The abort operation being requested by the framework. Currently defined operations are:

STMF_TGT_PORT_ABORT_TASK

Abort the task pointed to by the 'arg' parameter.

arg

abort_cmd specific arguments.

flags

Any additional conditions modifying the abort behavior. Currently no flags are defined.

Return values

The target port should return one of the following values in response to this call.

STMF_SUCESS

The target port has accepted the abort request and will abort the task asynchronously.

STMF_BUSY

The target cannot accept this request at this time. The framework should retry again after some time.

STMF_NOT_FOUND

The target is not aware of the task. This is not considered a failure by the framework.

STMF_ABORTED

The target port has completed the request synchronously.

Notes

If the abort operation is being performed asynchronously the target port must call stmf_task_tgt_port_aborted() to indicate completion of the abort operation.

5.7 lport_ctl

Local port control entry point.

Format

```
void (*lport_ctl)(struct stmf_local_port *lport, int cmd,
                                        void *arg);
```

Parameters

lport

Pointer to the stmf_local_port structure itself.

cmd

Control command. Currently defined values are:

STMF_CMD_LPORT_ONLINE

Bring the local port online. arg (3rd argument to the entry point) points to stmf_state_change_info_t.

STMF_CMD_LPORT_OFFLINE

Bring the local port offline. arg points to stmf_state_change_info_t

STMF_ACK_LPORT_ONLINE_COMPLETE

An acknowledgement from the framework that everythi ng that is needed to bring this local port online has been done. This is sent in response to a call from the LPORT to stmf_ctl() interface with a CMD calue of STMF_CMD_LPORT_ONLINE_COMPLETE. arg is a pointer to the stmf_change_status_t, passed in the stmf_ctl() call.

STMF_ACK_LPORT_OFFLINE_COMPLETE

An acknowledgement from the framework that the framework has removed all references to this LPORT. After this call the LPORT can deregister from the framework and even unload itself if needed.

arg

Argument specific to the cmd.

Notes

- When a LPORT is offlined, the LPORT must cleanup all of its internal commands before calling the stmf_ctl(STMF_CMD_LPORT_OFFLINE_COMPLETE, lport, ..); For SCSI commands, the LPORT must call stmf_queue_cmd_for_termination(task) and wait for the command to get terminated.
- After the port has been offlined, it must return STMF_ABORT_SUCCESS from the (*lport_abort)() entry point if that entry point is called by the framework.

6 STMF Interfaces

These interfaces are provided by stmf to lports and lus and their respective providers.

6.1 stmf_alloc

This interface is used to allocate shared data structures.

Format

```
void * stmf_alloc(stmf_struct_id_t struct_id, int
                                  additional_size, int flags);
```

Parameters

struct_id

Identifier for the data structure to be allocated. See section 3.2 for details on stmf_struct_id_t

additional_size

Size of the caller private data. The 2nd void pointer in the structure will be initialized to point to this block.

flags

Flags affecting the allocation. Currently defined values are:

AF_FORCE_NOSLEEP

This flag is force the allocation to use KM_NOSLEEP while doing kmem_zalloc(9F).

Return values

The function returns a pointer to the allocated structure. In case of failure the function returns NULL.

Notes

Every data structure allocated through stmf alloc() starts with two void pointers. The 1st pointer points to the stmf private data and the 2nd pointer points to the caller private data.

The function uses KM_NOSLEEP if called from interrupt context. Otherwise it uses KM_SLEEP to do the allocations. Callers can override KM_SLEEP behavior by specifying the flag AF_FORCE_NOSLEEP.

Caller should use stmf_free() to free any data structures allocated via stmf_alloc.

6.2 stmf_free

This interface is used to free any data structures which were allocated using stmf_alloc()

Format

```
void stmf_free(void *ptr);
```

Parameters

ptr

Pointer to the memory block which needs to be freed.

6.3 stmf_register_lu_provider

This interface is called by the LU provider to register itself with the framework.

Format

```
stmf_status_t stmf_register_lu_provider(
                                stmf_lu_provider_t *lp);
```

Parameters

lp

Pointer to the LU Provider structure allocated vis stmf_alloc().

Return values

The function return STMF_SUCCESS upon success.

6.4 stmf_deregister_lu_provider

This interface is called by the LU providers to de register themselves from the framework.

Format

```
stmf_status_t stmf_deregister_lu_provider(
                            stmf_lu_provider_t *lp)
```

Parameters

lp

Pointer to the LU provider structure.

Return Values

STMF_SUCCESS

The deregister operation was successful.

STMF_BUSY

The deregister operation was not successful because the provider has Logical units registered with the framework.

Notes

The call to deregister LU provider will not succeed until all the LUs this provider has have been deregistered first.

6.5 stmf_register_port_provider

This interface is called by the Port Provider to register itself with the framework.

Format

```
stmf_status_t stmf_register_port_provider(
                                    stmf_port_provider_t *pp);
```

Parameters

pp

Pointer to the Port Provider structure allocated vis stmf_alloc().

Return values

The function return STMF_SUCCESS upon success.

6.6 stmf_deregister_port_provider

This interface is called by the Port Providers to deregister themselves from the framework.

Format

```
stmf_status_t stmf_deregister_port_provider(
                            stmf_port_provider_t *pp)
```

Parameters

pp

Pointer to the Target Port Provider structure.

Return Values

STMF_SUCCESS

The deregister operation was successful.

STMF_BUSY

The deregister operation was not successful because the provider has LPORTs registered with the framework.

Notes

The call to deregister Target Port Provider will not succeed until all the LPORTs this provider has have been deregistered first.

6.7 stmf_register_lu

This function is called by LU providers to register a LU with the framework.

Format

```
stmf_status_t stmf_register_lu(stmf_lu_t *lu);
```

Parameters

lu

Pointer to the logical unit structure to be registered.

Return values

The function returns STMF SUCCESS if the registration was successful.

6.8 stmf_deregister_lu

This function is called by LU provider to deregister a LU from the framework.

Format

```
stmf_status_t stmf_deregister_lu(stmf_lu_t *lu);
```

Parameters

lu

Pointer to the LU structure to deregister.

Return Values

The function returns STMF_SUCCESS if the deregisteration process succeeds.

Notes

If there are pending I/Os on this LU the function will fail with a return code of STMF_BUSY.

6.9 stmf_register_local_port

This function is called by Port Providers to register a local port with the framework.

Format

```
stmf_status_t stmf_register_local_port(
                    stmf_local_port_t *lport);
```

Parameters

lport

Pointer to the local port structure to be registered.

Return values

The function returns STMF SUCCESS if the registration was successful.

6.10 stmf_deregister_local_port

This function is called by Port Provider to deregister a local port from the framework.

Format

```
stmf_status_t stmf_deregister_local_port(
                           stmf_local_port_t *lport);
```

Parameters

lport

Pointer to the local port structure to deregister.

Return Values

The function returns STMF_SUCCESS if the deregisteration process succeeds.

Notes

If there are pending I/Os on this port the function will fail with a return code of STMF_BUSY.

6.11 stmf_register_scsi_session

This function is called by LPORTs to register a SCSI session representing a I_T nexus.

Format

```
stmf_status_t stmf_register_scsi_session(
     stmf_local_port_t *lport, stmf_scsi_session_t *ss);
```

Parameters

lport

Pointer to local port structure receiving the login from initiator.

ss

Pointer to a SCSI session structure allocated and initialized by the LPORT.

Return Values

The function returns STMF_SUCCESS upon a successful registration of the session.

Notes

The framework will perform LUN mapping for this initiator as a part of the registration.

6.12 stmf_deregister_scsi_session

This function is called by LPORT to deregister a SCSI session and remove the I_T nexus.

Format

```
void stmf_deregister_scsi_session(
     stmf_local_port_t *lport, stmf_scsi_session_t *ss);
```

Parameters

lport

Pointer to the local port to which the session belongs.

ss

Pointer to the SCSI session to be deregistered.

Notes

The LPORT should terminate all the exchanges before calling stmf_deregister_scsi_session.

6.13 stmf_alloc_dbuf

This interface is used by both LUs and LPORTs to allocate a data buffer for a SCSI task.

Format

```
stmf_data_buf_t * stmf_alloc_dbuf(scsi_task_t *task,
          uint32_t size, uint32_t *pminsize, uint32_t flags);
```

Parameters

task

SCSI task for which the buffer is needed.

size

Data size in bytes needed for transfer.

pminsiz

Pointer to a uint32_t containing minimum buffer size in bytes needed.

flags

Flags to modify the allocation. None defined at this time.

Return Values

This function returns a pointer to the allocated data buffer. If the allocation process fails, the return value is NULL in which case pminsize will be initialized to the size which the framework can allocate. A value of zero in pminsize indicates that the framework cannot allocate any buffer.

Notes

The returned data buffer may be of a smaller size than what is requested but will not be less than *pminsize. If the request size is very large, the LU should use smaller buffer sizes to do multiple transfers instead of allocating one big buffer.

6.14 stmf_free_dbuf

This interface is used by both LUs and LPORTs to free a data buffers previously allocated via stmf_alloc_dbuf().

Format

```
void stmf_free_dbuf(scsi_task_t *task,
                            stmf_data_buf_t *dbuf);
```

Parameters

task

SCSI task to which the data buffer currently belongs.

dbuf

Pointer to the data buffer to be freed.

Notes

The framework frees all the data buffers a task has when the task is freed.

6.15 stmf_handle_to_buf

Given a SCSI task and a handle, this call returns the associated stmf_data_buf_t.

Format

```
stmf_data_buf_t *stmf_handle_to_buf(scsi_task_t *task,
                                    uint8_t h);
```

Parameters

task

The SCSI task holding the required data buffer.

h

Handle to the data buffer for the task.

Return values

This function returns the associated stmf_data_buf_t or NULL if the data buffer cannot be found.

6.16 stmf_task_alloc

This function is called by LPORT to allocate a scsi_task_t data structure upon receiving a new scsi task from an initiator.

Format

```
struct scsi_task *stmf_task_alloc(
  struct stmf_local_port *lport, stmf_scsi_session_t *ss,
                              uint8_t *lun)
```

Parameters

lport

Pointer to the stmf_local_port structure receiving the task.

ss

Pointer to a previously registered scsi session representing this I_T nexus.

lun

8 byte LUN received from the initiator.

Return value

This function allocated a new scsi_task_t structure and returns a pointer to it. It returns NULL in case any allocation error happens or the session is not valid.

6.17 stmf_post_task

This function is called by the LPORT to dispatch a new scsi_task_t to the framework.

Format

```
void stmf_post_task(scsi_task_t *task,
                            stmf_data_buf_t *dbuf);
```

Parameters

task

Pointer to the task structure to be submitted.

dbuf

An initial data buffer. This may contain an initial burst from the initiator or just an empty buffer provided by LPORT to the LU to avoid an extra allocation. This pointer can also be NULL in which case the LU will allocate a data buffer if it needs one.

6.18 stmf_xfer_data

This function is called by LU to start a data transfer. The direction of the transfer can be from the initiator or towards the initiator as determined by the db_flags field of the stmf_data_buf_t.

Format

```
stmf_status_t stmf_xfer_data(scsi_task_t *task,
                stmf_data_buf_t *dbuf, uint32_t ioflags);
```

Parameters

task

Pointer to scsi_task_t representing the task for which this data transfer is being done.

dbuf

The data buffer for the transfer.

ioflags

Flags to indicate LU's state. Currently defines values are:

0	- No change in LU's state.
STMF_IOF_LU_DONE	- The LU is done with this task.

Return values

STMF_SUCCESS	- Transfer was started successfully
STMF_ABORTED	- The task has been aborted.

6.19 stmf_data_xfer_done

This function is called by the LPORT to indicate completion or failure or a data transfer operation started by stmf_xfer_data().

Format

```
void stmf_data_xfer_done(scsi_task_t *task,
                         stmf_data_buf_t *dbuf, uint32_t iof);
```

Parameters

task

Pointer to SCSI task to which the data transfer belongs.

dbuf

The data buffer for the transfer.

iof

Flags to indicate LPORT's state. Currently defined values are:

0	- No change in LPORT's state.
STMF_IOF_ LPORT_DONE	- LPORT is done with this task.

Notes

The status of the transfer is stored in the db_xfer_status member of the dbuf.

6.20 stmf_send_scsi_status

This function is called by LU to transfer SCSI status.

Format

```
stmf_status_t stmf_send_scsi_status(scsi_task_t *task,
                                    uint32_t ioflags);
```

Parameters

tsak

Pointer to scsi_task_t for which the status is to be sent. The scsi_task_t structure contains the status information to be sent.

ioflags

Flags to indicate LU's state. Currently defines values are:

0	- No change in LU's state.
STMF_IOF_LU_DONE	- The LU is done with this task.

Return values

STMF_SUCCESS	- Transfer was started successfully.
STMF_ABORTED	- The task has been aborted.

Notes

See definition of scsi_task_t in section 3.8 for details on how to setup status values.

6.21 stmf_send_status_done

This function is called by LPORT when it has finished transferring SCSI status.

Format

```
void stmf_send_status_done(scsi_task_t *task,
                           stmf_status_t s, uint32_t iof);
```

Parameters

task

Pointer to the scsi_task_t to which this SCSI status belongs.

s

Completion status of the command.

iof

Flags indicating the LPORT state. This is same as defined in section 6.19

6.22 stmf_task_lu_done

This function is called by LU to indicate that it has finished processing of a SCSI task.

Format

```
void stmf_task_lu_done(scsi_task_t *task)
```

Parameters

task

Pointer to the scsi_task_t structure which LU has finished processing.

Notes

Normally the LU will not reach this stage i.e. need to call this function because it would have already indicated to the framework that it is done with the task. The only time LU will need to call this function if it needs confirmation of the SCSI status transfer.

LU should not free any memory associated with the task (pointed to by task_lu_private) and should wait for the framework to call lu_task_free entry point.

6.23 stmf_queue_task_for_termination

This function can be called by either LU or LPORT to queue a scsi_task_t for abnormal termination.

Format

```
void stmf_queue_task_for_termination(scsi_task_t *task,
                        stmf_status_t s);
```

Parameters

task

Pointer to scsi_task_t which needs to be termination.

s

Status indicating the reason for termination.

Notes

Either LU or LPORT or both can call this function for a task as long as they have not indicated to the framework that they are done with the processing of a task. This function will result in the framework calling the abort entry point of the LU or LPORT, if it has not already indicated that it is done with the processing of the task. It is possible that the framework might call the abort entry point even after the LU or LPORT has indicated that it is done processing the task but it will never be called after the *_task_free() entry point has been called.

6.24 stmf_task_lu_aborted

This function is called by the LU to indicate the completion of abort, after the framework has called the lu_abort entry point of the LU.

Format

```
void stmf_task_lu_aborted(scsi_task_t *task,
                          stmf_status_t s, uint32_t iof);
```

Parameters

task

Pointer to the scsi_task_t for which lu_abort() was called.

s

The status of the framework's abort request. The LU can return STMF_ABORTED or STMF_NOT_FOUND to indicate a successful completion of the abort. Any other status will result in framework offlining the LU.

iof

State of the LU after completion of abort. The LU should set this value to STMF_IOF_LU_DONE if the abort was successful.

Notes

If the framework calls lu_abort() entry point of the LU, it will not free the task until the LU responds back with stmf_task_lu_aborted().

6.25 stmf_task_lport_aborted

This function is called by the LPORT to indicate the completion of abort, after the framework has called the lport_abort entry point of the LPORT.

Format

```
void stmf_task_lport_aborted(scsi_task_t *task,
                    stmf_status_t s, uint32_t iof);
```

Parameters

task

Pointer to the scsi_task_t for which lport_abort() was called.

s

The status of the framework's abort request. The LPORT can return STMF_ABORTED or STMF_NOT_FOUND to indicate a successful completion of the abort. Any other status will result in framework offlining the LPORT.

iof

State of the LPORT after completion of abort. The LPORT should set this value to STMF_IOF_LPORT_DONE if the abort was successful.

Notes

If the framework calls lport_abort() entry point of the LPORT, it will not free the task until the LPORT responds back with stmf_task_lport_aborted().

6.26 stmf_ctl

STMF interface to drive various control operations on LUs and LPORTs.

Format

```
stmf_status_t stmf_ctl(int cmd, void *obj, void *arg);
```

Parameters

cmd

Control command. The following cmds are currently defined.

STMF_CMD_LU_ONLINE

Initiate the online of the stmf_lu_t pointed to by obj. arg points to stmf_state_change_info_t structure.

STMF_CMD_LU_OFFLINE

Initiate offline of the stmf_lu_t pointed to by obj. Again arg points to stmf_state_change_info_t structure.

STMF_CMD_LU_ONLINE_COMPLETE

A stmf_lu_t, pointed to by obj, has completed the online request. The result of the completion is provided by the structure stmf_change_status_t which is pointed to by arg.

STMF_CMD_LPORT_ONLINE

Initiate the online of the stmf_local_port_t pointed to by obj. arg points to stmf_state_change_info_t structure.

STMF_CMD_LPORT_OFFLINE

Initiate offline of the stmf_local_port_t pointed to by obj. Again arg points to stmf_state_change_info_t structure.

STMF_CMD_LPORT_ONLINE_COMPLETE

A stmf_local_port_t, pointed to by obj, has completed the online request. The result of the completion is provided by the structure stmf_change_status_t which is pointed to by arg.

obj

Depends upon the cmd. See the description of cmd above.

arg

Depends upon the cmd. See the description of cmd above.

Return values

STMF_SUCCESS	- The ctl cmd was accepted successfully.
STMF_ALREADY	- The object is already in the requested state.
STMF_INVALID_ARG	- Invalid cmd specified or a completion was reported when the operation is not in progress.

Notes

During initial registration of a LU or LPORT, unless the user has explicitly offlined the object, the framework automatically calls stmf_ctl to online it.

7 Support Functions

This section describes several support functions provided by STMF to simplify the development of a LU provider or a LPORT provider.

7.1 stmf_scsilib_send_status

Support routine to handle SCSI status phase, calculate residuals and send any check condition.

Format

```
void stmf_scsilib_send_status(scsi_task_t *task,
                              uint8_t st, uint32_t saa);
```

Parameters

task

scsi_task_t for which status is to be sent.

st

8 bit SCSI status.

saa

A combination of 'sense key', ASC and ASCQ. Bits 16-23 is the sense key, Bits 8-15 is the ASC and bits 0-7 is ASCQ. The routine forms the appropriate sense data based on these 3 values, iff the scsi status is 'STATUS_CHECK' (0x02).

Notes

This routine eventually calls stmf_send_scsi_status(). So various transfer lengths should be properly updated before calling this routine. See stmf_send_scsi_status for more details.

7.2 stmf_scsilib_default_handling

Shifts the ownership of a task from a LU to stmf and lets stmf handles the task.

Format

```
void stmf_scsilib_default_handling(scsi_task_t *task,
                                   stmf_data_buf_t *dbuf);
```

Parameters

task

scsi_task_t for which the LU wants to transfer ownership.

dbuf

A pointer to an initial dbuf passed to the LU. This can be NULL.

Notes

If a LU receives a scsi_task_t which it does not understand then instead of responding to the initiator with a check condition, it can call this routine to let stmf decode and respond to this task. Also LUs should not decode commands like REPORT LUN and instead pass those commands to stmf using this routine.

7.3 stmf_scsilib_uniq_lu_id

During LU registration, stmf requires that the lu_id be set to a unique identifier of type GUID.

This support routine can be used by a LU provider to generate a GUID.

Format

```
stmf_status_t stmf_scsilib_uniq_lu_id(uint32_t company_id,
                                      scsi_devid_desc_t *lu_id);
```

Fields

company_id

24 bit NAA company ID. This can be set to COMPANY_ID_NONE if the LU does not care about company identifiers. It defaults to COMPANY_ID_SUN in that case.

lu_id

An empty lu_id which will be filled by this function. The caller must set the ident_length to 0x10 and have 16 bytes allocated for the ident field.

Return Values

STMF_SUCCESS	- lu_id has been created successfully.
STMF_INVALID_ARG	- ident_length was not set to 0x10.

7.4 stmf_scsilib_handle_inquiry_page83

This routine responds to inquiry Page 83 request based on the lu_id field of the stmf_lu_t

Format

```
void stmf_scsilib_handle_inquiry_page83(
     scsi_task_t *task, stmf_data_buf_t *dbuf,
     uint8_t byte0);
```

Parameters

task

Pointer to the scsi_task_t structure representing the inquiry command.

dbuf

The initial dbuf passed in by the LPORT. Can also be NULL.

byte0

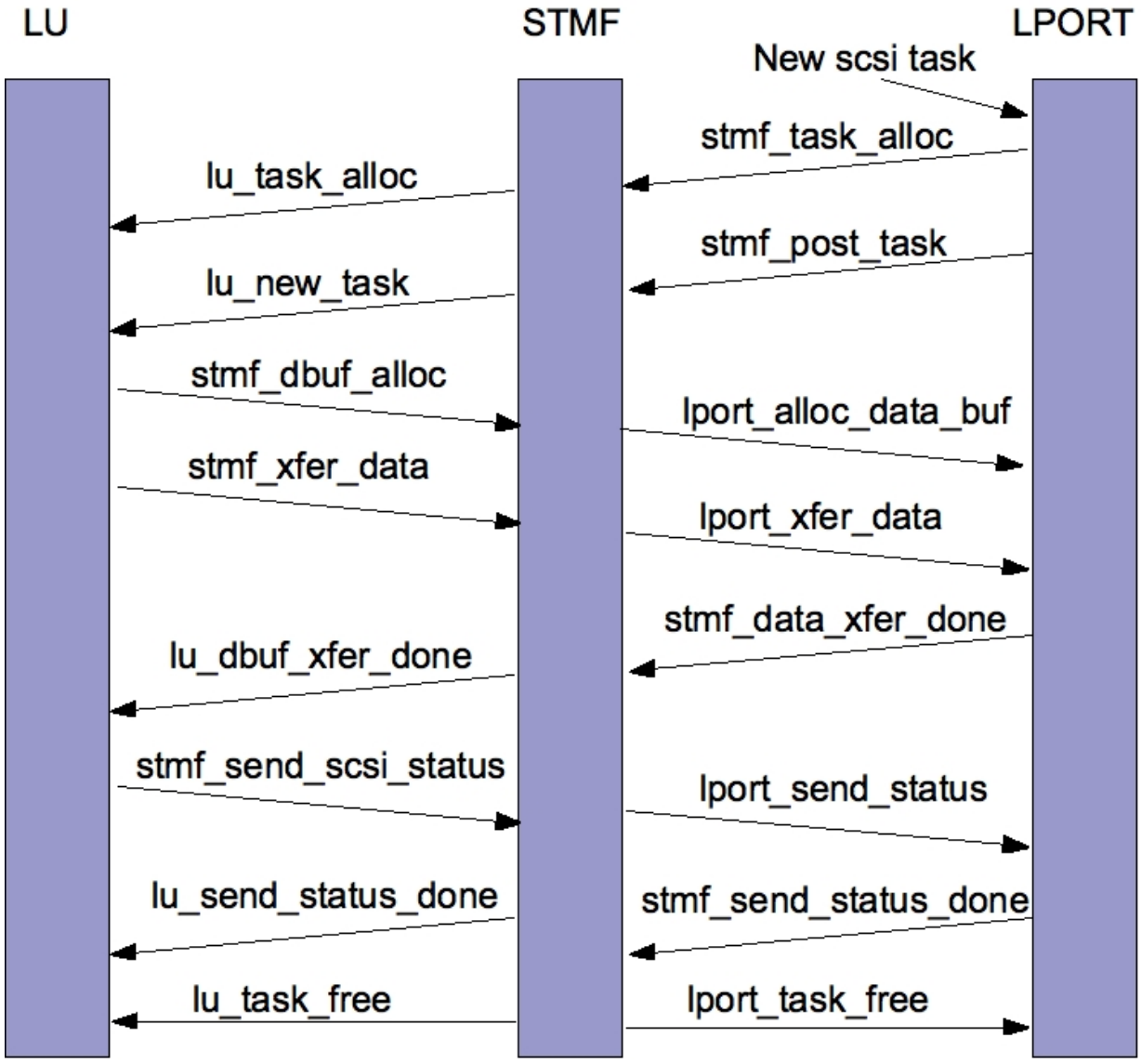
Byte #0 if the inquiry response. This contains the device type identifier.

Notes

If the LU only wants to send one descriptor out for the inquiry page 83 command and that descriptor is based on the lu_id registered with the framework in the stmf_lu_t structure, then the LU can use this support routine to do the same.

8 Flow Diagrams

8.1 SCSI IO Flow



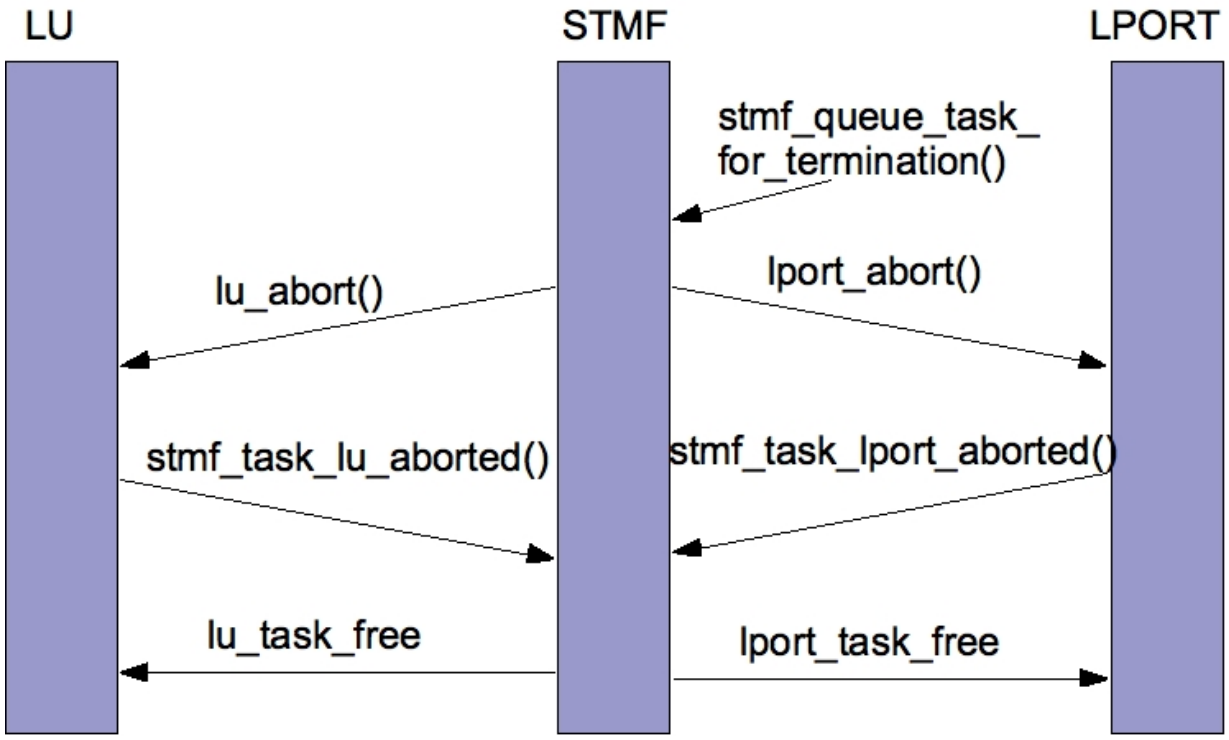
8.1.1 Notes on SCSI IO Flow

STMF maintains a cache of scsi_task_t on a per LU basis. If the task is pulled from the cache, stmf will not call lu_task_alloc(). Similarly STMF maintains a cahe of data bufs on a per lport basis and it may not call lport_alloc_data_buf() when the LU requests a buffer.

A LU has the option of ending a SCSI flow at stmf_xfer_data level (by combining the data xfer and good SCSI status) or at the stmf_send_scsi_status() level by setting the ioflags parameter. See descriptions of these interfaces for detail.

Both the LPORT and LU have the option of terminating an IO abnormally by calling stmf_queue_task_for_termination() before they indicate that they are done with the IO by setting the ioflags. See the abnormal IO termination flow for details.

8.2 Abnormal IO Termination Flow



8.2.1 Notes on abnormal IO termination

Both LU and LPORT can call stmf_queue_task_for_termination anytime before they have indicated to the framework that they are done with the task using ioflags. After a task has been submitted for termination, any attempt by both LU and LPORT to normally complete this task will be ignored by the framework and an explicit acknowledgment of the abort request will be needed.

The framework will not call the lu_task_free or lport_task_free until either the task has been aborted or completed by both LU and LPORT.

The lu_abort and lport_abort can complete synchronously by returning STMF_ABORT_SUCCESS fron the *_abort() entry point in which case a stmf_task_*_aborted() call will not be needed.