SCST technical description

Vladislav Bolkhovitin

Version 3.0.0 for SCST 3.0.0 and later

Contents

1 Introduction

SCST is a SCSI target mid-level subsystem for Linux. It provides unified consistent interface between SCSI target drivers, backend device handlers and Linux kernel as well as simplifies target drivers development as much as possible.

It has the following features:

- Very low overhead and fine-grained locks, which allow to reach maximum possible performance and scalability that close to theoretical limit.
- Complete SMP support.
- Performs all required pre- and post- processing of incoming requests and all necessary error recovery functionality.
- Emulates necessary functionality of SCSI host adapters, because from a remote initiator's point of view SCST acts as a SCSI host with its own devices. Some of the emulated functions are the following:
  - Generation of necessary UNIT ATTENTIONs, their storage and delivery to all connected remote initiators (sessions).
  - RESERVE/RELEASE functionality, including Persistent Reservations.
  - All types of RESETs and other task management functions.
  - REPORT LUNS command as well as SCSI address space management in order to have consistent address space on all remote initiators, since local SCSI devices could not know about each other to report via REPORT LUNS command. Additionally, SCST responds with error on all commands to non-existing devices and provides access control, so different remote initiators could see different set of devices.
  - Other necessary functionality (task attributes, etc.) as specified in SAM-2, SPC-2, SAM-3, SPC-3 and other SCSI standards.
- Verifies all incoming requests to ensure commands execution reliability and security.
- Device handlers architecture provides extra flexibility by allowing to make additional requests processing, which is completely independent from target drivers, for example, data caching or device dependent exceptional conditions treatment.

2 Terms and Definitions

SCSI initiator device

A SCSI device that originates service and task management requests to be processed by a SCSI target device and receives device service and task management responses from SCSI target devices.

SCSI target device

A SCSI device that receives device service and task management requests for processing and sends device service and task management responses to SCSI initiator devices or drivers.

SCST session

SCST session is the object that describes relationship between a remote initiator and SCST via a target driver. All the commands from the remote initiator is passed to SCST in the session. For example, for connection oriented protocols, like iSCSI, SCST session could be mapped to TCP connection (as well as iSCSI session). SCST session is equivalent of SCSI I_T nexus object.

Local SCSI initiator

A SCSI initiator that is located on the same host as SCST subsystem. Examples are sg and st drivers.

Remote SCSI initiator

A SCSI initiator that is located on the remote host for SCST subsystem and makes client connections to SCST via SCST target drivers.

SCSI target driver

A Linux hardware or logical driver that acts as a SCSI target for remote SCSI initiators, i.e. accepts remote connections, passes incoming SCSI requests to SCST and sends SCSI responses from SCST back to their originators.

Device (backend) handler driver

Also known as "device type specific driver" or "dev handler", SCST driver, which helps SCST to analyze incoming requests and determine parameters, specific to various types of devices as well as perform some processing. See below for more details.

3 SCST Core Architecture

SCST accepts commands and passes them to SCSI mid-level at the same way as SCSI high-level drivers (sg, sd, st) do. Figure 1 shows interaction between SCST, its drivers and Linux SCSI subsystem.

4 Target drivers

4.1 struct scst_tgt_template

To work with SCST a target driver must register its template in SCST by calling **scst_register_target_template()**. The template lets SCST know the target driver's entry points. It is defined as the following:

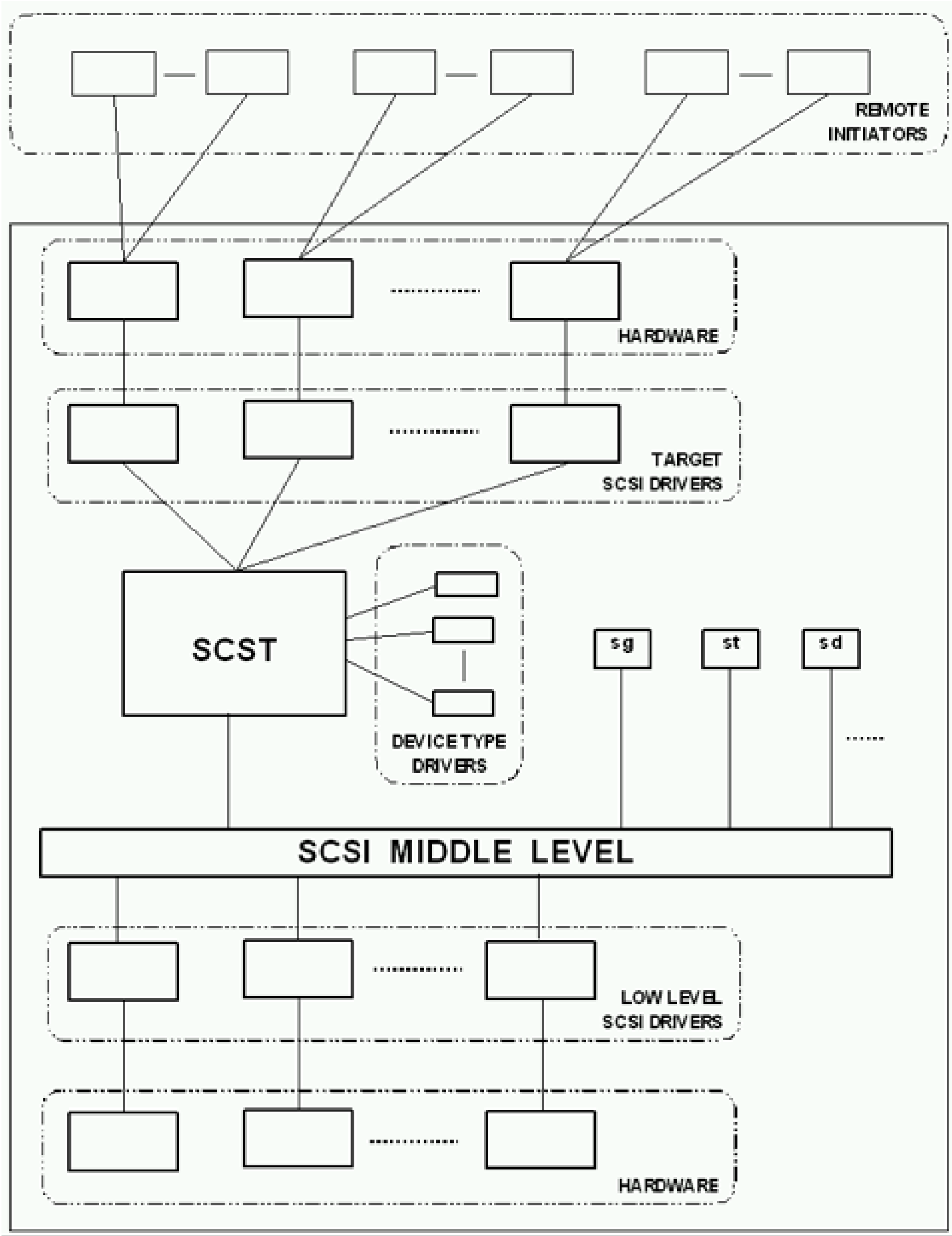

Figure 1: Interaction between SCST, its drivers and Linux SCSI subsystem.

```
struct scst_tgt_template
{
        int sg_tablesize;
        const char name[SCST_MAX_NAME];

        unsigned unchecked_isa_dma:1;
        unsigned use_clustering:1;
        unsigned no_clustering:1;

        unsigned xmit_response_atomic:1;
        unsigned rdy_to_xfer_atomic:1;

        unsigned no_proc_entry:1;

        int max_hw_pending_time;

        int threads_num;

        int (*detect) (struct scst_tgt_template *tgt_template);
        int (*release)(struct scst_tgt *tgt);

        int (*xmit_response)(struct scst_cmd *cmd);
        int (* rdy_to_xfer)(struct scst_cmd *cmd);

        void (*on_hw_pending_cmd_timeout) (struct scst_cmd *cmd);

        void (*on_free_cmd) (struct scst_cmd *cmd);

        int (*alloc_data_buf) (struct scst_cmd *cmd);

        void (*preprocessing_done) (struct scst_cmd *cmd);

        int (*pre_exec) (struct scst_cmd *cmd);

        void (*task_mgmt_affected_cmds_done) (struct scst_mgmt_cmd *mgmt_cmd);
        void (*task_mgmt_fn_done)(struct scst_mgmt_cmd *mgmt_cmd);

        int (*report_aen) (struct scst_aen *aen);

        int (*read_proc) (struct seq_file *seq, struct scst_tgt *tgt);
        int (*write_proc) (char *buffer, char **start, off_t offset,
                int length, int *eof, struct scst_tgt *tgt);

        int (*get_initiator_port_transport_id) (struct scst_session *sess,
                uint8_t **transport_id);
}
```

Where:

- **sg_tablesize** - allows checking whether scatter/gather can be used or not and, if yes, sets the maximum supported count of scatter/gather entries
- **name** - the name of the template. Must be unique to identify the template. Must be defined.
- **unchecked_isa_dma** - true, if this target adapter uses unchecked DMA onto an ISA bus.
- **use_clustering** - true, if this target adapter wants to use clustering (i.e. smaller number of merged segments).
- **no_clustering** - true, if this target adapter doesn't support SG-vector clustering
- **xmit_response_atomic**, **rdy_to_xfer_atomic** - true, if the corresponding function supports execution in the atomic (non-sleeping) context.
- **no_proc_entry** - true, if this template doesn't need the entry in /proc
- **max_hw_pending_time** - The maximum time in seconds cmd can stay inside the target hardware, i.e. after rdy_to_xfer() and xmit_response(), before on_hw_pending_cmd_timeout() will be called, if defined. In the current implementation a cmd will be aborted in time t max_hw_pending_time <= t < 2*max_hw_pending_time.
- **threads_num** - number of additional threads to the pool of dedicated threads. Used if xmit_response() or rdy_to_xfer() is blocking. It is the target driver's duty to ensure that not more, than that number of threads, are blocked in those functions at any time.
- **int (*detect) (struct scst_tgt_template *tgt_template)** - this function is intended to detect the target adapters that are present in the system. Each found adapter should be registered by calling *scst_register_target()*. The function should return a value >= 0 to signify the number of detected target adapters. A negative value should be returned whenever there is an error. Must be defined.
- **int (*release)(struct scst_tgt *tgt)** - this function is intended to free up resources allocated to the device. The function should return 0 to indicate successful release or a negative value if there are some issues with the release. In the current version of SCST the return value is ignored. Must be defined.
- **int (*xmit_response)(struct scst_cmd *cmd)** - this function is equivalent to the SCSI queuecommand(). The target should transmit the response data and the status in the struct scst_cmd. See below for details. Must be defined.
- **int (*rdy_to_xfer)(struct scst_cmd *cmd)** - this function informs the driver that data buffer corresponding to the said command have now been allocated and it is OK to receive data for this command. This function is necessary because a SCSI target does not have any control over the commands it receives. Most lower-level protocols have the corresponding function which informs the initiator that buffers have been allocated e.g., XFER_RDY in Fibre Channel. After the data actually received, the low-level driver should call *scst_rx_data()* in order to continue processing this command. Returns one of the *SCST_TGT_RES_** constants, described below. Pay attention to "atomic" attribute of the command, which can be get via scst_cmd_atomic(). It is true if the function called in the atomic (non-sleeping) context. Must be defined.

- **void (*on_hw_pending_cmd_timeout) (struct scst_cmd *cmd)** - Called if cmd stays inside the target hardware, i.e. after rdy_to_xfer() and xmit_response(), more than max_hw_pending_time time. The target driver supposed to cleanup this command and resume cmd's processing.

- **void (*on_free_cmd)(struct scst_cmd *cmd)** - this function called to notify the driver that the command is about to be freed. Necessary, because for aborted commands xmit_response() could not be called. Could be used on IRQ context. Must be defined.

- **int (*alloc_data_buf) (struct scst_cmd *cmd)** - this function allows target driver to handle data buffer allocations on its own. Target driver doesn't have to always allocate buffer in this function, but if it decided to do it, it must check that scst_cmd_get_data_buff_alloced() returns 0, otherwise to avoid double buffer allocation and memory leaks alloc_data_buf() shall fail. Returns 0 in case of success or < 0 (preferrably -ENOMEM) in case of error, or > 0 if the regular SCST allocation should be done. In case of returning successfully, scst_cmd->tgt_data_buf_alloced will be set by SCST. It is possible that both target driver and dev handler request own memory allocation. If allocation in atomic context, i.e. scst_cmd_atomic() is true, and < 0 is returned, this function will be recalled in thread context. Note that the driver will have to handle itself all relevant details such as scatterlist setup, highmem, freeing the allocated memory, etc.

- **void (*preprocessing_done) (struct scst_cmd *cmd)** - this function informs the driver that data buffer corresponding to the said command have now been allocated and other preprocessing tasks have been done. A target driver could need to do some actions at this stage. After the target driver done the needed actions, it shall call *scst_restart_cmd()* in order to continue processing this command. In case of preliminary commands completion, this function will also be called before xmit_response(). Called only for commands queued using scst_cmd_init_stage1_done() instead of scst_cmd_init_done(). Returns void, the result is expected to be returned using scst_restart_cmd(). This command is expected to be NON-BLOCKING. If it is blocking, consider to set threads_num to some none 0 number. Pay attention to "atomic" attribute of the cmd, which can be get by scst_cmd_atomic(). It is true if the function called in the atomic (non-sleeping) context.

- **int (*pre_exec) (struct scst_cmd *cmd)** - this function informs the driver that the said command is about to be executed. Returns one of the *SCST_PREPROCESS_** constants. This command is expected to be NON-BLOCKING. If it is blocking, consider to set threads_num to some none 0 number.

- **void (*task_mgmt_affected_cmds_done) (struct scst_mgmt_cmd *mgmt_cmd)** - this function informs the driver that all affected by the corresponding task management function commands have beed completed. No return value expected. This function is expected to be NON-BLOCKING. Called without any locks held from a thread context.

- **void (*task_mgmt_fn_done)(struct scst_mgmt_cmd *mgmt_cmd)** - this function informs the driver that a received task management function has been completed. Completion status could be get via *scst_mgmt_cmd_get_status()*. No return value expected. Must be defined, if the target supports task management functionality.

- **int (*report_aen) (struct scst_aen *aen)** - this function is used for Asynchronous Event Notifications. Returns one of the *SCST_AEN_RES_** constants. After AEN is sent, target driver must call *scst_aen_done()* and, optionally, *scst_set_aen_delivery_status()*. This function is expected to be NON-BLOCKING, but can sleep. This function must be prepared to handle AENs between calls for the corresponding session of scst_unregister_session() and unreg_done_fn() callback called or before scst_unregister_session() returned, if its called in the blocking mode. AENs for such sessions should be ignored. Must be defined, if low-level protocol supports AENs.

- **int (*read_proc) (struct seq_file *seq, struct scst_tgt *tgt), int (*write_proc) (char *buffer, char **start, off_t offset, int length, int *eof, struct scst_tgt *tgt)** - those functions can be used to export the driver's statistics and other infos to the world outside the kernel as well as to get some management commands from it. If the driver needs to create additional files in its /proc subdirectory, it can use *scst_proc_get_tgt_root()* function to get the root proc_dir_entry.

- **int (*get_initiator_port_transport_id) (struct scst_session *sess, uint8_t **transport_id)** - this function returns in tr_id the corresponding to sess initiator port TransporID in the form as it's used by PR commands, see "Transport Identifiers" in SPC. Space for the initiator port TransporID must be allocated via kmalloc(). Caller supposed to kfree() it, when it isn't needed anymore. If sess is NULL, this function must return TransportID PROTOCOL IDENTIFIER of this transport. Returns 0 on success or negative error code otherwise. Should be defined, because it's required for Persistent Reservations.

Functions **xmit_response()**, **rdy_to_xfer()** are expected to be non-blocking, i.e. return immediately and don't wait for actual data transfer to finish. Blocking in such command could negatively impact on overall system performance. If blocking is necessary, it is worth to consider creating dedicated thread(s) in target driver, to which the commands would be passed and which would perform blocking operations instead of SCST. If the function allowed to sleep or not is defined by "atomic" attribute of the cmd that can be get via *scst_cmd_atomic()*, which is true, if sleeping is not allowed. In this case, if the function requires sleeping, it can return *SCST_TGT_RES_NEED_THREAD_CTX* in order to be recalled in the thread context, where sleeping is allowed.

Functions **task_mgmt_fn_done()** and **report_aen()** are recommended to be non-blocking as well. Blocking there will stop all management processing for all target drivers in the system (there is only one management thread in the system).

Functions **xmit_response()** and **rdy_to_xfer()** can return the following error codes:

- **SCST_TGT_RES_SUCCESS** - success.
- **SCST_TGT_RES_QUEUE_FULL** - internal device queue is full, retry again later.
- **SCST_TGT_RES_NEED_THREAD_CTX** - it is impossible to complete requested task in atomic context. The command should be restarted in the thread context as described above.
- **SCST_TGT_RES_FATAL_ERROR** - fatal error, i.e. it is unable to perform requested operation. If returned by **xmit_response()** the command will be destroyed, if by **rdy_to_xfer()**, **xmit_response()** will be called with **HARDWARE ERROR** sense data.

4.1.1 More about xmit_response()

As already written above, function xmit_response() should transmit the response data and the status from the cmd parameter.

Sense data, if any, is contained in the buffer, returned by *scst_cmd_get_sense_buffer()*, with length, returned by *scst_cmd_get_sense_buffer_len()*. SCST always works in autosense mode. If a low-level SCSI driver/device doesn't support autosense mode, SCST will issue REQUEST SENSE command, if necessary. Thus, if CHECK CONDITION established, target driver will always see sense in the sense buffer and isn't required to request the sense manually.

After the response is completely sent, the target should call *scst_tgt_cmd_done()* function in order to allow SCST to free the command.

Function xmit_response() returns one of the *SCST_TGT_RES_** constants, described above. Pay attention to "atomic" attribute of the cmd, which can be get via *scst_cmd_atomic()*: it is true if the function called in the atomic (non-sleeping) context.

To detect aborted commands xmit_response() must in the beginning check return status of function **scst_cmd_aborted_on_xmit()**. If it's true, xmit_response() must call **scst_set_delivery_status(cmd, SCST_CMD_DELIVERY_ABORTED)** and terminate further processing by calling **scst_tgt_cmd_done(cmd, SCST_CONTEXT_SAME)**.

4.2 Target driver registration functions

4.2.1 scst_register_target_template()

Function **scst_register_target_template()** is defined as the following:

```
int scst_register_target_template(
        struct scst_tgt_template *vtt)
```

Where:

- **vtt** - pointer to the target driver template

Returns 0 on success or appropriate error code otherwise.

4.2.2 scst_register_target()

Function **scst_register_target()** is defined as the following:

```
struct scst_tgt *scst_register_target(
        struct scst_tgt_template *vtt)
```

Where:

- **vtt** - pointer to the target driver template

Returns target structure based on template vtt or NULL in case of error.

4.3 Target driver unregistration functions

In order to unregister itself target driver should at first call **scst_unregister_target()** for all its adapters and then call **scst_unregister_target_template()** for its template.

4.3.1 scst_unregister_target()

Function **scst_unregister_target()** is defined as the following:

```
void scst_unregister_target(
	struct scst_tgt *tgt)
```

Where:

- **tgt** - pointer to the target driver structure

4.3.2 scst_unregister_target_template()

Function **scst_unregister_target_template()** is defined as the following:

```
void scst_unregister_target_template(
	struct scst_tgt_template *vtt)
```

Where:

- **vtt** - pointer to the target driver template

5 Device specific drivers (backend device handlers)

Device specific drivers are add-ons for SCST, which help SCST to analyze incoming requests and determine parameters, specific to various types of devices as well as actually execute specified SCSI commands. Device handlers are intended for the following:

- To get data transfer length and direction directly from CDB and current device's configuration exactly as an end-target SCSI device does. This serves two purposes:
  - Improves security and reliability by not trusting the data supplied by remote initiator via SCSI low-level protocol.
  - Some low-level SCSI protocols don't provide data transfer length and direction, so that information can be get only directly from CDB and current device's configuration. For example, for tape devices to get data transfer size it might be necessary to know block size setting.
- Execute commands
- To process some exceptional conditions, like ILI on tape devices.
- To initialize incoming commands with some device-specific parameters, like timeout value.
- To allow some additional device-specific commands pre-, post- processing or alternative execution, like copying data from system cache, and do that completely independently from target drivers.

Device handlers considered to be part of SCST, so they could directly access any fields in SCST's structures as well as use the corresponding functions.

Without appropriate device handler SCST hides devices of this type from remote initiators and returns **HARDWARE ERROR** sense data to any requests to them.

5.1 Structure scst_dev_type

Structure **scst_dev_type** is defined as the following:

```
struct scst_dev_type
{
        char name[];
        int type;

        unsigned parse_atomic:1;
        unsigned alloc_data_buf_atomic:1;
        unsigned dev_done_atomic:1;

        unsigned no_proc:1;

        unsigned exec_sync:1;

        unsigned pr_cmds_notifications:1;

        int threads_num;
        enum scst_dev_type_threads_pool_type threads_pool_type;

        int (*attach) (struct scst_device *dev);
        void (*detach) (struct scst_device *dev);

        int (*attach_tgt) (struct scst_tgt_device *tgt_dev);
        void (*detach_tgt) (struct scst_tgt_device *tgt_dev);

        int (*parse) (struct scst_cmd *cmd);
        int (*alloc_data_buf) (struct scst_cmd *cmd);
        int (*exec) (struct scst_cmd *cmd);
        int (*dev_done) (struct scst_cmd *cmd);
        int (*on_free_cmd) (struct scst_cmd *cmd);

        int (*task_mgmt_fn) (struct scst_mgmt_cmd *mgmt_cmd,
                struct scst_tgt_dev *tgt_dev);

        int (*read_proc) (struct seq_file *seq, struct scst_dev_type *dev_type);
        int (*write_proc) (char *buffer, char **start, off_t offset,
                int length, int *eof, struct scst_dev_type *dev_type);
}
```

Where:

- **name** - the name of the device handler. Must be defined and unique.
- **type** - SCSI type of the supported device. Must be defined.

- **parse_atomic**, **alloc_data_buf_atomic**, **dev_done_atomic** - true, if the corresponding callback supports execution in the atomic (non-sleeping) context.
- **no_proc** - true, if no /proc files should be automatically created by SCST for this dev handler
- **exec_sync** - should be true, if exec() is synchronous. This is a hint to SCST core to optimize commands order management.
- **pr_cmds_notifications** - should be set if the device wants to receive notification of Persistent Reservation commands (PR OUT only) Note: The notifications will not be sent if the command failed.
- **threads_num** - sets number of threads in this handler's devices' threads pools. If 0 - no threads will be created, if <0 - creation of the threads pools is prohibited. Also pay attention to *threads_pool_type* below.
- **threads_pool_type** - threads pool type. Valid only if threads_num > 0. Possible values:
  - **SCST_THREADS_POOL_PER_INITIATOR** - each initiator will have dedicated threads pool
  - **SCST_THREADS_POOL_SHARED** - all connected initiators will use shared threads pool
- **int (*attach) (struct scst_device *dev)** - called when new device is being attached to the device handler
- **void (*detach) (struct scst_device *dev)** - called when new device is being detached from the device handler
- **int (*attach_tgt) (struct scst_tgt_device *tgt_dev)** - called when new tgt_dev (session) is being attached to the device handler
- **void (*detach_tgt) (struct scst_tgt_device *tgt_dev)** - called when tgt_dev (session) is being detached from the device handler
- **int (*parse) (struct scst_cmd *cmd, const struct scst_info_cdb *cdb_info)** - called to parse CDB from the cmd and initialize *cmd->bufflen* and *cmd->data_direction* (both - REQUIRED). Returns the command's *next state* or *SCST_CMD_STATE_DEFAULT*, if the next default state should be used, or *SCST_CMD_STATE_NEED_THREAD_CTX* if the function called in atomic context, but requires sleeping, or *SCST_CMD_STATE_STOP* if the command should not be further processed for now. In the SCST_CMD_STATE_NEED_THREAD_CTX case the function will be recalled in the thread context, where sleeping is allowed. Pay attention to "atomic" attribute of the cmd, which can be get by scst_cmd_atomic(). It is true if the function called in the atomic (non-sleeping) context. Must be defined.
- **int (*alloc_data_buf) (struct scst_cmd *cmd)** - this function allows dev handler to handle data buffer allocations on its own. Returns the command's *next state* or *SCST_CMD_STATE_DEFAULT*, if the next default state should be used, or *SCST_CMD_STATE_NEED_THREAD_CTX* if the function called in atomic context, but requires sleeping, or *SCST_CMD_STATE_STOP* if the command should not be further processed for now. In the SCST_CMD_STATE_NEED_THREAD_CTX case the function will be recalled in the thread context, where sleeping is allowed. Pay attention to "atomic" attribute of the cmd, which can be get by scst_cmd_atomic(). It is true if the function called in the atomic (non-sleeping) context.

- **int (*exec) (struct scst_cmd *cmd)** - called to execute CDB. Useful, for instance, to implement data caching. The result of CDB execution is reported via *cmd->scst_cmd_done()* callback.

  Returns:

  - **SCST_EXEC_COMPLETED** - the cmd is done, go to other ones
  - **SCST_EXEC_NOT_COMPLETED** - the cmd should be sent to SCSI mid-level.

  If this function provides sync execution, you should set exec_sync flag and consider to setup dedicated threads by setting *threads_num* > 0.

  Optional, if not set, the commands will be sent directly to SCSI device.

  If this function is implemented, scst_check_local_events() shall be called inside it just before the actual command's execution.

- **int (*dev_done) (struct scst_cmd *cmd)** - called to notify device handler about the result of the command's execution and perform some post processing. If *parse()* function is called, dev_done() is *guaranteed* to be called as well. The command's fields *tgt_resp_flags* and *resp_data_len* should be set by this function, but SCST offers good defaults. Pay attention to "atomic" attribute of the command, which can be get via scst_cmd_atomic(). It is true if the function called in the atomic (non-sleeping) context. Returns the command's *next state* or *SCST_CMD_STATE_DEFAULT*, if the next default state should be used, or *SCST_CMD_STATE_NEED_THREAD_CTX* if the function called in atomic context, but requires sleeping. In the last case, the function will be recalled in the thread context, where sleeping is allowed.

- **void (*on_free_cmd) (struct scst_cmd *cmd)** - called to notify device handler that the command is about to be freed. Could be called on IRQ context.

- **int (*task_mgmt_fn) (struct scst_mgmt_cmd *mgmt_cmd, struct scst_tgt_dev *tgt_dev)** - called to execute a task management command. Returns:

  - **SCST_MGMT_STATUS_SUCCESS** - the command is done with success, no further actions required
  - **SCST_MGMT_STATUS_*** - the command is failed, no further actions required
  - **SCST_DEV_TM_NOT_COMPLETED** - regular standard actions for the command should be done

  NOTE: for **SCST_ABORT_TASK** it is called under spinlock!

- **int (*read_proc) (struct seq_file *seq, struct scst_tgt *tgt), int (*write_proc) (char *buffer, char **start, off_t offset, int length, int *eof, struct scst_tgt *tgt)** - those functions can be used to export the driver's statistics and other infos to the world outside the kernel as well as to get some management commands from it. If the driver needs to create additional files in its /proc subdirectory, it can use *scst_proc_get_dev_type_root()* function to get the root proc_dir_entry.

5.2 Device specific drivers registration

5.2.1 scst_register_dev_driver()

To work with SCST a device specific driver must register itself in SCST by calling **scst_register_dev_driver()**. It is defined as the following:

```
int scst_register_dev_driver(
        struct scst_dev_type *dev_type)
```

Where:

- **dev_type** - device specific driver's description structure

The function returns 0 on success or appropriate error code otherwise.

5.2.2 scst_register_virtual_device()

To create a virtual device a device handler must register it in SCST by calling **scst_register_virtual_device()**. It is defined as the following:

```
int scst_register_virtual_device(
        struct scst_dev_type *dev_handler,
        const char *dev_name)
```

Where:

- **dev_handler** - device specific driver's description structure
- **dev_name** - the new device name, NULL-terminated string. Must be unique among all virtual devices in the system.

The function returns ID assigned to the device on success, or negative value otherwise.

All local real SCSI devices will be registered and unregistered by the SCST core automatically, so pass-through dev handlers don't have to worry about it.

5.3 Device specific drivers unregistration

5.3.1 scst_unregister_virtual_device()

Virtual devices unregistered by calling **scst_unregister_virtual_device()**. It is defined as the following:

```
void scst_unregister_virtual_device(
        int id)
```

Where:

- **id** - the device's ID, returned by the registration function.

5.3.2 scst_unregister_dev_driver()

Device specific driver is unregistered by calling **scst_unregister_dev_driver()**. It is defined as the following:

```
void scst_unregister_dev_driver(
        struct scst_dev_type *dev_type)
```

Where:

- **dev_type** - device specific driver's description structure

6 SCST sessions

6.1 SCST sessions registration

When target driver determines that it needs to create new SCST session (for example, by receiving new TCP connection), it should call **scst_register_session()**, that is defined as the following:

```
struct scst_session *scst_register_session(
        struct scst_tgt *tgt,
        int atomic,
        const char *initiator_name,
        void *tgt_priv,
        void *result_fn_data,
        void (*result_fn) (
                struct scst_session *sess,
                void *data,
                int result))
```

Where:

- **tgt** - target
- **atomic** - true, if the function called in the atomic context
- **initiator_name** - remote initiator's name, any NULL-terminated string, e.g. iSCSI name, which used as the key to found appropriate access control group. Could be NULL, then "default" group is used. The groups are set up via /proc interface.
- **tgt_priv** - pointer to target driver's private data
- **result_fn_data** - data that will be used as the second parameter for **bf**result_fn/()/ function
- **result_fn** - pointer to the function that will be asynchronously called when session initialization finishes. Can be NULL. Parameters:
  - **sess** - session

- **data** - target driver supplied to scst_register_session() data
- **result** - session initialization result, 0 on success or appropriate error code otherwise

A session creation and initialization is a complex task, which requires sleeping state, so it can't be fully done in interrupt context. Therefore the "bottom half" of it, if scst_register_session() is called from atomic context, will be done in SCST thread context. In this case scst_register_session() will return not completely initialized session, but the target driver can supply commands to this session via scst_rx_cmd(). Those commands processing will be delayed inside SCST until the session initialization is finished, then their processing will be restarted. The target driver will be notified about finish of the session initialization by function *result_fn()*. On success the target driver could do nothing, but if the initialization fails, the target driver must ensure that no more new commands being sent or will be sent to SCST after result_fn() returns. All already sent to SCST commands for failed session will be returned in *xmit_response()* with BUSY status. In case of failure the driver shall call *scst_unregister_session()* inside result_fn(), it will NOT be called automatically.

Thus, scst_register_session() can be safely called from IRQ context.

6.2 SCST sessions unregistration

SCST session unregistration basically is the same, except that instead of atomic parameter there is **wait** one.

```
void scst_unregister_session(
        struct scst_session *sess,
        int wait,
        void (*unreg_done_fn)(
                struct scst_session *sess))
```

Where:

- **sess** - session to be unregistered
- **wait** - if true, instructs to wait until all commands, which currently being executed in the session, finished. Otherwise, target driver should be prepared to receive *xmit_response()* for the session after scst_unregister_session() returns.
- **unreg_done_fn** - pointer to the function that will be asynchronously called when the last session's command finishes and the session is about to be completely freed. Can be NULL. Parameter:
  - **sess** - session

All outstanding commands will be finished regularly. After scst_unregister_session() returned no new commands must be sent to SCST via scst_rx_cmd(). Also, the caller must ensure that no scst_rx_cmd() or scst_rx_mgmt_fn_*() is called in parallel with scst_unregister_session().

Function scst_unregister_session()/ can be called before result_fn() of scst_register_session() called, i.e. during the session registration/initialization.

7 Commands processing and interaction between SCST core and its drivers

Consider simplified commands processing example. It assumes that target driver doesn't need own memory allocation, i.e. not defined alloc_data_buf() callback. Example of such target driver is qla2x00t.

The commands processing by SCST started when target driver calls **scst_rx_cmd()**. This function returns SCST's command. Then the target driver finishes the command's initialization, for example, storing necessary target driver specific data there, and calls **scst_cmd_init_done()** telling SCST that it can start the command processing. Then SCST translates the command's LUN to local device, determines the command's data direction and required data buffer size by calling appropriate device handler's **parse()** callback function. Then:

- If the command required no data transfer, it will be passed to SCSI mid-level directly or via device handler's **exec()** callback.
- If the command is a *READ* command (data to the remote/local initiator), necessary space will be allocated and then the command will be passed to SCSI mid-level directly or via device handler's **exec()** callback.
- If the command is a *WRITE* command (data from the remote/local initiator), necessary space will be allocated, then the target's **rdy_to_xfer()** callback will be called, telling the target that the space is ready and it can start data transferring. When all the data are read from the target, it will call **scst_rx_data()**, and the command will be passed to SCSI mid-level directly or via device handler's **exec()** callback.

When the command is finished by SCSI mid-level, device handler's **dev_done()** callback is called to notify it about the command's completion. Then in order to send its response the target's **xmit_response()** callback is called. When the response, including data, if any, is transmitted, the target will call **scst_tgt_cmd_done()** to tell SCST that it can free the command and its data buffer.

Then during the command's deallocation device handler's and the target's **on_free_cmd()** callback will be called in this order, if set.

This sequence is illustrated on Figure 2. To simplify the picture, sign "..." means SCST's waiting state for the corresponding command to complete. During this state SCST and its drivers continue processing of other commands, if there are any. One way arrow, for example to xmit_response(), means that after this function returns, nothing valuable for the current command will be done and SCST goes sleeping or to the next command processing until the corresponding event happens.

7.1 The commands processing functions

7.1.1 scst_rx_cmd()

Function **scst_rx_cmd()** creates and sends new command to SCST. Returns the command on success or NULL otherwise. It is defined as the following:

```
struct scst_cmd *scst_rx_cmd(
        struct scst_session *sess,
```

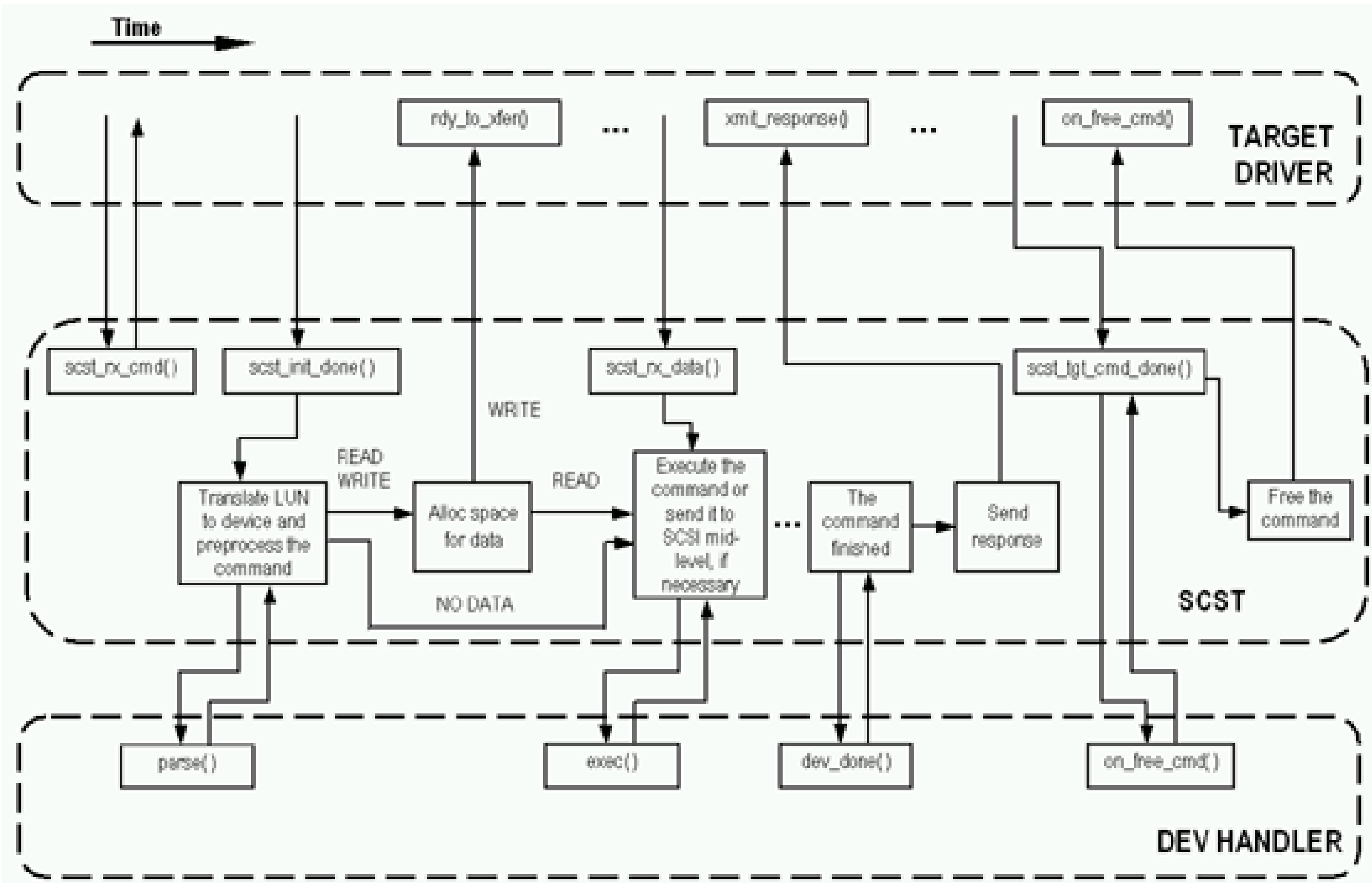

Figure 2: The commands processing flow

```
        const uint8_t *lun,
        int lun_len,
        const uint8_t *cdb,
        int cdb_len,
        int atomic)
```

Where:

- **sess** - SCST's session
- **lun** - pointer to device's LUN as specified by SAM in without any byte order translation. Extended addressing method is not supported.
- **lun_len** - LUN's length
- **cdb** - SCSI CDB
- **cdb_len** - CDB's length. Can be up to 64KB long.
- **atomic** - if true, the command will be allocated with GFP_ATOMIC flag, otherwise GFP_KERNEL will be used

7.1.2 scst_cmd_init_done()

Function **scst_cmd_init_done()** notifies SCST that the driver finished its part of the command initialization, and the command is ready for execution. It is defined as the following:

```
void scst_cmd_init_done(
        struct scst_cmd *cmd,
        enum scst_exec_context pref_context)
```

Where:

- **cmd** - the command
- **pref_context** - preferred command execution context. See *SCST_CONTEXT_** constants below for details.

7.1.3 scst_rx_data()

Function **scst_rx_data()** notifies SCST that the driver received all the necessary data and the command is ready for further processing. It is defined as the following:

```
void scst_rx_data(
        struct scst_cmd *cmd,
        int status,
        enum scst_exec_context pref_context)
```

Where:

- **cmd** - the command
- **status** - completion status, see below.
- **pref_context** - preferred command execution context. See *SCST_CONTEXT_** constants below for details.

Parameter **status** can have one of the following values:

- **SCST_RX_STATUS_SUCCESS** - success
- **SCST_RX_STATUS_ERROR** - data receiving finished with error, so SCST should set the sense and finish the command by calling xmit_response()
- **SCST_RX_STATUS_ERROR_SENSE_SET** - data receiving finished with error and the sense is set, so SCST should finish the command by calling xmit_response()
- **SCST_RX_STATUS_ERROR_FATAL** - data receiving finished with fatal error, so SCST should finish the command, but don't call xmit_response(). In this case the driver must free all associated with the command data before calling scst_rx_data().

7.1.4 scst_tgt_cmd_done()

Function **scst_tgt_cmd_done()** notifies SCST that the driver has sent the data and/or response. It must not been called if there are an error and xmit_response() returned something other, than SCST_TGT_RES_SUCCESS. It is defined as the following:

```
void scst_tgt_cmd_done(
        struct scst_cmd *cmd,
        enum scst_exec_context pref_context)
```

Where:

- **cmd** - the command
- **pref_context** - preferred command execution context. See *SCST_CONTEXT_** constants below for details.

7.2 The commands processing context

Execution context often is a major problem in the kernel drivers development, because many contexts, like IRQ context, greatly limit available functionality, therefore require additional complex code in order to pass processing to more simple context. SCST does its best to undertake most of the context handling.

On the initialization time SCST creates for internal command processing as many threads as there are processors in the system or specified by user via **scst_threads** module parameter. Similarly, as many tasklets created as there are processors in the system.

Each command can be processed in one of four contexts:

1. Directly, i.e. in the caller's context, without limitations
2. Directly atomically, i.e. with sleeping forbidden
3. In the SCST's internal threads
4. In the SCST's per processor tasklets

The target driver sets this context as pref_context parameter for SCST functions. Additionally, target's template's *xmit_response_atomic* and *rdy_to_xfer_atomic* flags have direct influence on the context. If one of them is false, the corresponding function will never be called in the atomic context and, if necessary, the command will be rescheduled to one of the SCST's threads.

SCST in some circumstances can change preferred context to less restrictive one, for example, for large data buffer allocation, if there is not enough GFP_ATOMIC memory.

7.2.1 Preferred context constants

There are the following preferred context constants:

- **SCST_CONTEXT_DIRECT** - sets direct command processing (i.e. regular function calls in the current context) sleeping is allowed, no context restrictions. Supposed to be used when calling from thread context where no locks are held and the driver's architecture allows sleeping without performance degradation or anything like that.
- **SCST_CONTEXT_DIRECT_ATOMIC** - sets direct command processing (i.e. regular function calls in the current context), sleeping is not allowed. Supposed to be used when calling on thread context where there are locks held, when calling on softirq context or the driver's architecture does not allow sleeping without performance degradation or anything like that.
- **SCST_CONTEXT_TASKLET** - tasklet or thread context required for the command processing. Supposed to be used when calling from IRQ context.
- **SCST_CONTEXT_THREAD** - thread context required for the command processing. Supposed to be used if the driver's architecture does not allow using any of above.
- **SCST_CONTEXT_SAME** - context is the same as it was in previous call of the corresponding callback. For example, if dev handler's exec() does sync. data reading this value should be used for scst_cmd_done(). The same is true if scst_tgt_cmd_done() called directly from target driver's xmit_response(). Not allowed in scst_cmd_init_done() and scst_cmd_init_stage1_done().

7.3 SCST commands' processing states

There are the following processing states, which a SCST command passes through during execution and which could be returned by device handler's **parse()** and **dev_done()** (but not all states are allowed to be returned):

- **SCST_CMD_STATE_INIT_WAIT** - the command is created, but *scst_cmd_init_done()* not called

- **SCST_CMD_STATE_INIT** - LUN translation (i.e. *cmd->tgt_dev* assignment) state
- **SCST_CMD_STATE_PARSE** - device handler's *parse()* is going to be called
- **SCST_CMD_STATE_PREPARE_SPACE** - allocation of the command's data buffer
- **SCST_CMD_STATE_PREPROCESSING_DONE_CALLED** - waiting for scst_restart_cmd()
- **SCST_CMD_STATE_RDY_TO_XFER** - target driver's *rdy_to_xfer()* is going to be called
- **SCST_CMD_STATE_DATA_WAIT** - waiting for data from the initiator (until *scst_rx_data()* called)
- **SCST_CMD_STATE_TGT_PRE_EXEC** - target driver's *pre_exec()* is going to be called
- **SCST_CMD_STATE_SEND_FOR_EXEC** - the command is going to be sent for execution
- **SCST_CMD_STATE_EXECUTING** - waiting for the command's execution finish
- **SCST_CMD_STATE_LOCAL_EXEC** - the command is being checked if it should be executed locally
- **SCST_CMD_STATE_REAL_EXEC** - the command is ready for execution
- **SCST_CMD_STATE_REAL_EXECUTING** - waiting for CDB's execution finish
- **SCST_CMD_STATE_PRE_DEV_DONE** - internal post-exec checks
- **SCST_CMD_STATE_MODE_SELECT_CHECKS** - internal MODE SELECT pages related checks
- **SCST_CMD_STATE_DEV_DONE** - device handler's *dev_done()* is going to be called
- **SCST_CMD_STATE_PRE_XMIT_RESP** - checks before target driver's *xmit_response()* is called
- **SCST_CMD_STATE_XMIT_RESP** - target driver's *xmit_response()* is going to be called
- **SCST_CMD_STATE_XMIT_WAIT** - waiting for data/response's transmission finish (until *scst_tgt_cmd_done()* called)
- **SCST_CMD_STATE_FINISHED** - the command finished and going to be freed

8 Task management functions

There are the following task management functions supported:

- **SCST_ABORT_TASK** - this is *ABORT_TASK* SAM task management function. Aborts the specified task (command).
- **SCST_ABORT_TASK_SET** - this is *ABORT_TASK_SET* SAM task management function. Aborts all tasks (commands) in the specified session.

- **SCST_CLEAR_ACA** - this is **CLEAR_ACA** SAM task management function. Currently does nothing.
- **SCST_CLEAR_TASK_SET** - this is **CLEAR_TASK_SET** SAM task management function. Clears task set of commands on the specified device or session.
- **SCST_LUN_RESET** - this is **LUN_RESET** SAM task management function. Resets specified device.
- **SCST_TARGET_RESET** - this is **TARGET_RESET** SAM task management function. Resets all devices visible in this session.
- **SCST_NEXUS_LOSS_SESS** - SCST extension. Notifies about I_T nexus loss event in the corresponding session. Aborts all tasks there, resets the reservation, if any, and sets up the I_T Nexus loss UA.
- **SCST_ABORT_ALL_TASKS_SESS** - SCST extension. Aborts all tasks in the corresponding session.
- **SCST_NEXUS_LOSS** - SCST extension. Notifies about I_T nexus loss event. Aborts all tasks in all sessions of the tgt, resets the reservations, if any, and sets up the I_T Nexus loss UA.
- **SCST_ABORT_ALL_TASKS** - SCST extension. Aborts all tasks in all sessions of the tgt.

All task management functions return completion status via *task_mgmt_fn_done()* when the affected SCSI commands (tasks) are actually aborted, i.e. guaranteed never be executed any time later.

8.1 scst_rx_mgmt_fn_tag()

Function **scst_rx_mgmt_fn_tag()** tells SCST to perform the specified task management function, based on the command's tag. Can be used only for *SCST_ABORT_TASK*.

It is defined as the following:

```
int scst_rx_mgmt_fn_tag(
        struct scst_session *sess,
        int fn,
        uint32_t tag,
        int atomic,
        void *tgt_priv)
```

Where:

- **sess** - the session, on which the command should be performed.
- **fn** - task management function, one of the constants above.
- **tag** - the command's tag.
- **atomic** - true, if the function called in the atomic context.

- **tgt_priv** - pointer to the target driver specific data, can be retrieved in task_mgmt_fn_done() via *scst_mgmt_cmd_get_status()* function.

Returns 0 if the command was successfully created and scheduled for execution, error code otherwise. On success, the completion status of the command will be reported asynchronously via task_mgmt_fn_done() driver's callback.

8.2 scst_rx_mgmt_fn_lun()

Function **scst_rx_mgmt_fn_lun()** tells SCST to perform the specified task management function, based on the LUN. Currently it can be used for any function, except *SCST_ABORT_TASK*.

It is defined as the following:

```
int scst_rx_mgmt_fn_lun(
        struct scst_session *sess,
        int fn,
        const uint8_t *lun,
        int lun_len,
        int atomic,
        void *tgt_priv);
```

Where:

- **sess** - the session, on which the command should be performed.
- **fn** - task management function, one of the constants above.
- **lun** - LUN, the format is the same as for **scst_rx_cmd()**.
- **lun_len** - LUN's length.
- **atomic** - true, if the function called in the atomic context.
- **tgt_priv** - pointer to the target driver specific data, can be retrieved in task_mgmt_fn_done() via *scst_mgmt_cmd_get_status()* function.

Returns 0 if the command was successfully created and scheduled for execution, error code otherwise. On success, the completion status of the command will be reported asynchronously via task_mgmt_fn_done() driver's callback.

Possible status constants which can be returned by **scst_mgmt_cmd_get_status()**:

- **SCST_MGMT_STATUS_SUCCESS** - success
- **SCST_MGMT_STATUS_TASK_NOT_EXIST** - requested task does not exist
- **SCST_MGMT_STATUS_LUN_NOT_EXIST** - requested LUN does not exist
- **SCST_MGMT_STATUS_FN_NOT_SUPPORTED** - requested TM function does not exist.
- **SCST_MGMT_STATUS_REJECTED** - TM function rejected.
- **SCST_MGMT_STATUS_FAILED** - TM function failed.

9 SGV cache

SCST SGV cache is a memory management subsystem in SCST. One can call it a "memory pool", but Linux kernel already have a mempool interface, which serves different purposes. SGV cache provides to SCST core, target drivers and backend dev handlers facilities to allocate, build and cache SG vectors for data buffers. The main advantage of it is the caching facility, when it doesn't free to the system each vector, which is not used anymore, but keeps it for a while (possibly indefinitely) to let it be reused by the next consecutive command. This allows to:

- Reduce commands processing latencies and, hence, improve performance;
- Make commands processing latencies predictable, which is essential for RT applications.

The freed SG vectors are kept by the SGV cache either for some (possibly indefinite) time, or, optionally, until the system needs more memory and asks to free some using the set_shrinker() interface. Also the SGV cache allows to:

- Cluster pages together. "Cluster" means merging adjacent pages in a single SG entry. It allows to have less SG entries in the resulting SG vector, hence improve performance handling it as well as allow to work with bigger buffers on hardware with limited SG capabilities.
- Set custom page allocator functions. For instance, scst_user device handler uses this facility to eliminate unneeded mapping/unmapping of user space pages and avoid unneeded IOCTL calls for buffers allocations. In fileio_tgt application, which uses a regular malloc() function to allocate data buffers, this facility allows 30% less CPU load and considerable performance increase.
- Prevent each initiator or all initiators altogether to allocate too much memory and DoS the target. Consider 10 initiators, which can have access to 10 devices each. Any of them can queue up to 64 commands, each can transfer up to 1MB of data. So, all of them in a peak can allocate up to 10*10*64 = 6.5GB of memory for data buffers. This amount must be limited somehow and the SGV cache performs this function.

9.1 Implementation

From implementation POV the SGV cache is a simple extension of the kmem cache. It can work in 2 modes:

1. With fixed size buffers.
2. With a set of power 2 size buffers. In this mode each SGV cache (struct sgv_pool) has SGV_POOL_ELEMENTS (11 currently) of kmem caches. Each of those kmem caches keeps SGV cache objects (struct sgv_pool_obj) corresponding to SG vectors with size of order X pages. For instance, request to allocate 4 pages will be served from kmem cache[2], since the order of the of number of requested pages is 2. If later request to allocate 11KB comes, the same SG vector with 4 pages will be reused (see below). This mode is in average allows less memory overhead comparing with the fixed size buffers mode.

Consider how the SGV cache works in the set of buffers mode. When a request to allocate new SG vector comes, sgv_pool_alloc() via sgv_get_obj() checks if there is already a cached vector with that order. If yes, then that vector will be reused and its length, if necessary, will be modified to match the requested size. In the above example request for 11KB buffer, 4 pages vector will be reused and modified using trans_tbl to contain 3 pages and the last entry will be modified to contain the requested length - 2*PAGE_SIZE. If there is no cached object, then a new sgv_pool_obj will be allocated from the corresponding kmem cache, chosen by the order of number of requested pages. Then that vector will be filled by pages and returned.

In the fixed size buffers mode the SGV cache works similarly, except that it always allocate buffer with the predefined fixed size. I.e. even for 4K request the whole buffer with predefined size, say, 1MB, will be used.

In both modes, if size of a request exceeds the maximum allowed for caching buffer size, the requested buffer will be allocated, but not cached.

Freed cached sgv_pool_obj objects are actually freed to the system either by the purge work, which is scheduled once in 60 seconds, or in sgv_shrink() called by system, when it's asking for memory.

9.2 Interface

9.2.1 sgv_pool *sgv_pool_create()

```
struct sgv_pool *sgv_pool_create(
        const char *name,
        enum sgv_clustering_types clustered, int single_alloc_pages,
        bool shared, int purge_interval)
```

This function creates and initializes an SGV cache. It has the following arguments:

- **name** - the name of the SGV cache
- **clustered** - sets type of the pages clustering. The type can be:
  - **sgv_no_clustering** - no clustering performed.
  - **sgv_tail_clustering** - a page will only be merged with the latest previously allocated page, so the order of pages in the SG will be preserved
  - **sgv_full_clustering** - free merging of pages at any place in the SG is allowed. This mode usually provides the best merging rate.
- **single_alloc_pages** - if 0, then the SGV cache will work in the set of power 2 size buffers mode. If >0, then the SGV cache will work in the fixed size buffers mode. In this case single_alloc_pages sets the size of each buffer in pages.
- **shared** - sets if the SGV cache can be shared between devices or not. The cache sharing allowed only between devices created inside the same address space. If an SGV cache is shared, each subsequent call of sgv_pool_create() with the same cache name will not create a new cache, but instead return a reference to it.
- **purge_interval** - sets the cache purging interval. I.e. an SG buffer will be freed if it's unused for time t purge_interval <= t < 2*purge_interval. If purge_interval is 0, then the default interval will be used (60 seconds). If purge_interval <0, then the automatic purging will be disabled. Shrinking by the system's demand will also be disabled.

Returns the resulting SGV cache or NULL in case of any error.

9.2.2 void sgv_pool_del()

```
void sgv_pool_del(
        struct sgv_pool *pool)
```

This function deletes the corresponding SGV cache. If the cache is shared, it will decrease its reference counter. If the reference counter reaches 0, the cache will be destroyed.

9.2.3 void sgv_pool_flush()

```
void sgv_pool_flush(
        struct sgv_pool *pool)
```

This function flushes, i.e. frees, all the cached entries in the SGV cache.

9.2.4 void sgv_pool_set_allocator()

```
void sgv_pool_set_allocator(
        struct sgv_pool *pool,
        struct page *(*alloc_pages_fn)(struct scatterlist *sg, gfp_t gfp, void *priv),
        void (*free_pages_fn)(struct scatterlist *sg, int sg_count, void *priv));
```

This function allows to set for the SGV cache a custom pages allocator. For instance, scst_user uses such function to supply to the cache mapped from user space pages.

alloc_pages_fn() has the following parameters:

- **sg** - SG entry, to which the allocated page should be added.
- **gfp** - the allocation GFP flags
- **priv** - pointer to a private data supplied to sgv_pool_alloc()

This function should return the allocated page or NULL, if no page was allocated.

free_pages_fn() has the following parameters:

- **sg** - SG vector to free
- **sg_count** - number of SG entries in the sg
- **priv** - pointer to a private data supplied to the corresponding sgv_pool_alloc()

9.2.5 struct scatterlist *sgv_pool_alloc()

```
struct scatterlist *sgv_pool_alloc(
        struct sgv_pool *pool,
        unsigned int size,
        gfp_t gfp_mask,
```

```
        int flags,
        int *count,
        struct sgv_pool_obj **sgv,
        struct scst_mem_lim *mem_lim,
        void *priv)
```

This function allocates an SG vector from the SGV cache. It has the following parameters:

- **pool** - the cache to alloc from
- **size** - size of the resulting SG vector in bytes
- **gfp_mask** - the allocation mask
- **flags** - the allocation flags. The following flags are possible and can be set using OR operation:
    1. **SGV_POOL_ALLOC_NO_CACHED** - the SG vector must not be cached.
    2. **SGV_POOL_NO_ALLOC_ON_CACHE_MISS** - don't do an allocation on a cache miss.
    3. **SGV_POOL_RETURN_OBJ_ON_ALLOC_FAIL** - return an empty SGV object, i.e. without the SG vector, if the allocation can't be completed. For instance, because SGV_POOL_NO_ALLOC_ON_CACHE_MISS flag set.
- **count** - the resulting count of SG entries in the resulting SG vector.
- **sgv** - the resulting SGV object. It should be used to free the resulting SG vector.
- **mem_lim** - memory limits, see below.
- **priv** - pointer to private for this allocation data. This pointer will be supplied to alloc_pages_fn() and free_pages_fn() and can be retrieved by sgv_get_priv().

This function returns pointer to the resulting SG vector or NULL in case of any error.

9.2.6 void sgv_pool_free()

```
void sgv_pool_free(
        struct sgv_pool_obj *sgv,
        struct scst_mem_lim *mem_lim)
```

This function frees previously allocated SG vector, referenced by SGV cache object sgv.

9.2.7 void *sgv_get_priv(struct sgv_pool_obj *sgv)

```
void *sgv_get_priv(
        struct sgv_pool_obj *sgv)
```

This function allows to get the allocation private data for this SGV cache object sgv. The private data are set by sgv_pool_alloc().

9.2.8 void scst_init_mem_lim()

```
void scst_init_mem_lim(
        struct scst_mem_lim *mem_lim)
```

This function initializes memory limits structure mem_lim according to the current system configuration. This structure should be latter used to track and limit allocated by one or more SGV caches memory.

9.3 Runtime information and statistics.

SGV cache runtime information and statistics is available in */proc/scsi_tgt/sgv*.

10 Target driver qla2x00t

Target driver qla2x00t allows to use QLogic 2xxx based adapters in the target (server) mode.

It consists from two parts:

- **qla2xxx** - patched initiator driver from Linux kernel, which is, among other things, intended to perform all the initialization and shutdown tasks.
- **qla2x00tgt** - target mode add-on for the changed qla2xxx

The initiator driver qla2xxx was changed to:

- To provide support for the target mode add-on via a set of exported callbacks
- To provide extra info and management interface in the driver's sysfs interface (attributes target_mode_enabled, ports_database, etc.)
- To fix some problems uncovered during target mode development and usage.

The changes are relatively small (few thousands lines big patch) and local.

The changed qla2xxx is still capable to work as initiator only. Mode, when a host acts as initiator and target simultaneously, is supported as well.

Since firmware interface for 24xx+ chips is fundamentally different from earlier versions, qla2x00t generally contains 2 separate drivers sharing some common processing.

10.1 Driver initialization

On initialization, qla2x00tgt registers its SCST template tgt2x_template in the SCST core. Then during template registration SCST core calls detect() callback which is function q2t_target_detect().

In this function qla2x00tgt registers its callbacks in qla2xxx by calling qla2xxx_tgt_register_driver(). Qla2xxx_tgt_register_driver() stores pointer to the being registered callbacks in variable qla_target.

Then q2t_target_detect() calls qla2xxx_add_targets(), which calls for each known local FC port (HBA instance) qla_target.tgt_host_action() callback with ADD_TARGET action. Then q2t_host_action() calls q2t_add_target() which registers SCST target for this FC port.

If later a new FC port is hot added, qla2x00_probe_one() will also call for all new local ports qla_target.tgt_host_action() with ADD_TARGET action.

10.2 Driver unload

When a local FC port is being removed, the Linux kernel calls qla2x00_remove_one(), which then qla_target.tgt_host_action() with REMOVE_TARGET action.

Then q2t_host_action() calls q2t_remove_target(), which unregisters the corresponding SCST target in SCST. During unregistration SCST core calls release() callback of tgt2x_template, which is q2t_target_release().

Then q2t_target_release() calls q2t_target_stop(). Then q2t_target_stop() marks this target as stopped by setting flag tgt_stop. When this flag is set, all incoming from initiators commands are refused.

Then q2t_target_stop() schedules deletion of all sessions of the target.

Then q2t_target_stop() waits until all outstanding commands finished and sessions deleted.

Then q2t_target_stop(), if necessary, calls qla2x00_disable_tgt_mode() to disables target mode, which disables target mode of the corresponding HBA and resets it. Then qla2x00_disable_tgt_mode() waits until reset finished.

Then q2t_target_stop() returns and then q2t_target_release() frees the target.

If module qla2x00tgt is being unloaded, q2t_exit() at first takes q2t_unreg_rwsem on writing. Taking it is necessary to make sure that q2t_host_action() will not be active during qla2x00tgt unload.

Then q2t_exit() calls scst_unregister_target_template() for tgt2x_template, which then in a loop will unregister all QLA SCST targets from SCST as described above.

10.3 Enabling target mode

When command to enable target mode received, qla_target.tgt_host_action() with action ENABLE_TARGET_MODE called. Then q2t_host_action() goes over all discovered remote of the being enabled target and adds SCST sessions for all them.

Then it calls qla2x00_enable_tgt_mode(), which enables target mode of the corresponding HBA and resets it. Then qla2x00_enable_tgt_mode() waits until reset finished.

During reset firmware initialization functions detect that target mode is enables and initialize the firmware accordingly.

10.4 Disabling target mode

When command to disable target mode received, qla_target.tgt_host_action() with action DISABLE_TARGET_MODE called. Then q2t_host_action() calls q2t_target_stop(), which processes as describe above.

10.5 SCST sessions management

As required by SCSI and FC standards, each remote initiator FC port has the corresponding SCST session.

Since qla2xxx is not intended to strictly maintain database of remote initiator FC ports as it is needed for target mode, qla2x00t uses mixed approach for SCST sessions management, when both qla2xxx and QLogic firmware generate events and information about currently active remote FC ports.

Remote FC ports management also has to handle changing FC and loop IDs after fabric events, so it needs to constantly monitor FC and loop IDs of the registered FC ports. This is implemented by checks in q2t_create_sess() that being registered FC port already has SCST session and q2t_check_fcport_exist() in q2t_del_sess_work_fn(). See below for more info.

Interaction with qla2xxx is implemented using tgt_fc_port_added() and tgt_fc_port_deleted() qla_target's callbacks.

Callback tgt_fc_port_added() called by qla2xxx when the target driver detects new remote FC port. Assigned to it q2t_fc_port_added() checks if an SCST session already exists for this remote FC port and, if not, creates it.

Callback tgt_fc_port_deleted() called by qla2xxx when it deletes a remote FC port from its database. Assigned to it q2t_fc_port_deleted() checks if an SCST session already exists for this remote FC port and, if yes, schedules it for deletion.

Driver qla2x00tgt has 2 types of SCST sessions: local and not local. Sessions created by q2t_fc_port_added() are not local. Local sessions created if qla2x00tgt receives a command from remote initiator for which there is no know remote FC port and, hence, SCST session. Local sessions are created in tgt->sess_work (q2t_sess_work_fn()) by calling q2t_make_local_sess(). All received from remote initiators commands for local sessions are delayed until the sessions are created.

To minimize affecting initiators by FC fabric events, qla2x00tgt doesn't immediately delete SCST sessions scheduled for deletion, but instead delay them for some time. If during this time a command from an unknown remote initiator received, q2t_make_local_sess()/q2t_create_sess() at first check if a session for this initiator already exists and, if yes, undelete then reuse it after updating its s_id and loop_id to new values.

If a session not reused during the delete delay time, then q2t_del_sess_work_fn() asks the firmware internal database if it knows the corresponding remote FC port. If yes, then this session is undeleted and its s_id and loop_id updated to new values. If no, the session is deleted.

10.6 Handling stuck commands

Driver qla2x00tgt defines in tgt2x_template callback on_hw_pending_cmd_timeout for handling stuck commands in q2t_on_hw_pending_cmd_timeout() function, with max_hw_pending_time timeout set Q2T_MAX_HW_PENDING_TIME (60 seconds). If the firmware doesn't return reply for one or more IOCBs for the corresponding SCST command, SCST core calls this callback.

In this callback all the stuck commands are forcibly finished.

A Debugging and troubleshooting

SCST core and its drivers provide excessive debugging and logging facilities suitable to catch and analyze problems of virtually any level of complexity.

Depending from amount debugging and logging facilities available, there are 3 types of builds:

- **release** - has basic amount of logging, suitable for basic tracing. Extra checking is disabled in this mode. This is the default mode.
- **debug** - has full amount of logging and extrachecks enabled. Has slower and much bigger binary code, but suitable for advanced tracing and debugging. Also in this mode more logging is enabled by default.
- **perf** - has all logging and extrachecks disables. Intended to performance measuremens, including measurements of overhead introduced by the logging and extrachecks facilities.

Switch between build modes is done by calling "make x2y", where "x" - current build mode and "y" - desired build mode. For instance, to switch from release to debug mode you should run "make release2debug".

A.1 Logging levels management

Logging levels management is done using "trace_level" file located in the driver's proc interface subdirectory. Each SCST driver has it, except in the perf build mode. For instance, for SCST core it's located in /proc/scsi_tgt/. For qla2x00t it's located in /proc/scsi_tgt/qla2x00tgt/.

Reading from it you can find currently enabled logging levels.

You can change them by writing in this file, like:

echo "add scsi" >/proc/scsi_tgt/trace_level

The following commands are available:

- **add trace_level** - adds (enables) the corresponding trace level
- **del trace_level** - deletes (disables) the corresponding trace level
- **set mask** - sets all trace levels at ones using a mask, e.g. 0x1538
- **all** - enables all trace levels
- **none** - disables all trace levels
- **default** - sets all trace levels in the default value
- **dump_prs dev_name** - dumps Persistent Reservations states for device "dev_name"

The following trace levels are common for all drivers:

- **function** - enables printing the corresponding function names for each logged messages
- **line** - enables printing the corresponding numbers of line of code for each logged message
- **pid** - enables printing PIDs of the corresponding processes or threads for each logged message

- **scsi** - enables logging of processed SCSI commands and their processing results
- **mgmt** - enables logging of processed Task Management functions
- **minor** - enables logging of minor events, line unknown SCSI commands or difference between buffer lengths encoded in CDBs and expected transfer values
- **out_of_mem** - enables logging of out of memory events
- **entryexit** - enables logging of functions entry and exit. Not available in the release build.
- **mem** - enables logging of memory allocation and freeing. Not available in the release build.
- **debug** - enables various debug logging messages. Not available in the release build.
- **buff** - enables logging of various buffers contain. Not available in the release build.
- **sg** - enables logging of SG vectors manipulations. Not available in the release build.
- **mgmt_dbg** - enables debug logging of Task Management functions processing. Not available in the release build.
- **special** - enables logging of "special" events. Intended to temporary enable logging of some debug messages without enabling the whole "debug" level. Not available in the release build.

The following trace levels are additionally available for SCST core:

- **scsi_serializing** - enables logging of SCSI commands task attributes processings (SIMPLE, ORDERED, etc.). Not available in the release build.
- **retry** - enables logging of retries of rdy_to_xfer() and xmit_response() target drivers callbacks. Not available in the release build.
- **recv_bot**, **send_bot**, **recv_top**, **send_top** - enables logging of commands buffers on various processing stages. Not available in the release build.

A.2 Preparing a debug kernel

SCST logging can produce huge amount of logging, which default kernel configuration can't cope with, so it needs some extra adjustments.

For that you should change in lib/Kconfig.debug or init/Kconfig depending from your kernel version LOG_BUF_SHIFT from "12 21" to "12 25".

Then you should in your .config set CONFIG_LOG_BUF_SHIFT to 25.

Also, Linux kernel has a lot of helpful debug facilities, like lockdep, which allows to catch various deadlocks, or memory allocation debugging. It is recommended to enable them during SCST debugging.

The following options are recommended to be enabled (available depending from your kernel version): CONFIG_SLUB_DEBUG, CONFIG_PRINTK_TIME, CONFIG_MAGIC_SYSRQ, CONFIG_DEBUG_FS, CONFIG_DEBUG_KERNEL, CONFIG_DEBUG_SHIRQ, CONFIG_DETECT_SOFTLOCKUP, CONFIG_DETECT_HUNG_TASK, CONFIG_SLUB_DEBUG_ON, CONFIG_SLUB_STATS, CONFIG_DEBUG_PREEMPT, CONFIG_DEBUG_RT_MUTEXES, CONFIG_DEBUG_PI_LIST, CONFIG_DEBUG_SPINLOCK, CONFIG_DEBUG_MUTEXES, CONFIG_DEBUG_LOCK_ALLOC,

CONFIG_PROVE_LOCKING, CONFIG_LOCKDEP, CONFIG_LOCK_STAT, CONFIG_DEBUG_SPINLOCK_SLEEP, CONFIG_STACKTRACE, CONFIG_DEBUG_BUGVERBOSE, CONFIG_DEBUG_VM, CONFIG_DEBUG_VIRTUAL, CONFIG_DEBUG_WRITECOUNT, CONFIG_DEBUG_MEMORY_INIT, CONFIG_DEBUG_LIST, CONFIG_DEBUG_SG, CONFIG_DEBUG_NOTIFIERS, CONFIG_FRAME_POINTER, CONFIG_FAULT_INJECTION, CONFIG_FAILSLAB, CONFIG_FAIL_PAGE_ALLOC, CONFIG_FAIL_MAKE_REQUEST, CONFIG_FAIL_IO_TIMEOUT, CONFIG_FAULT_INJECTION_DEBUG_FS, CONFIG_FAULT_INJECTION_STACKTRACE_FILTER.

A.3 Preparing logging subsystem

It is recommended that you system logger daemon on the target configured:

- To store kernel logs in separate files on the fastest disk you have. It will be better if this disk is dedicated for logging or, at least, doesn't contain your LUNs data.
- To write the kernel logs to the disk in asynchronous manner, i.e. without calling fsync() after each written message. Usually, you can achieve it, if you add a '-' sign before the corresponding file path in your syslog daemon conf file, like:

  kern.* -/var/log/kern.log

A.4 Decoding OOPS messages

You can decode an OOPS message to the corresponding line in C file using gdb "l" command. For example, an OOPS message has a line:

```
[<ffffffff88646174>] :iscsi_scst:iscsi_extracheck_is_rd_thread+0x94/0xb0
```

You can decode it by:

```
$ gdb iscsi-scst.ko
(gdb) l *iscsi_scst:iscsi_extracheck_is_rd_thread+0x94
```

For that the corresponding module (iscsi-scst.ko) should be build with debug info. But modules not always have debug info built-in. To workaround it you can add "-g" flag in the corresponding Makefile (without changing anything else!) or enable in .config using "make menuconfig" building kernel with debug info. Then rebuild only the .o file you need.

For instance, to decode OOPS in mm/filemap.c in the kernel you need enable in .config building kernel with debug info and then run:

```
$ make mm/filemap.o
...
$ gdb mm/filemap.o
```