



Chelsio iSCSI Target Exported Kernel API Guide

CHELSIO COMMUNICATIONS, INC. Company Confidential
Do Not Reproduce.
This copy belongs to

Copyright © 2005 - 2015 Chelsio Communications Inc. All rights reserved.

This document contains proprietary information of Chelsio Communications. No part of the document may be reproduced. Reverse engineering of the hardware or software is prohibited and is protected by patent law.

This material or any portion of it may not be copied in any form or by any means, stored in a retrieval system, adopted or transmitted in any form or by any means (electronic, mechanical, photographic, graphic, optic or otherwise), or translated in any language or computer language without the prior written permission of Chelsio Communications.

Chelsio Communications makes no representation or warranties with respect to the contents herein and shall not be responsible for any loss or damage caused to the user by the direct or indirect use of this information.

All products or company names or trademarks mentioned herein are used for identification purposes only, and may be trademarks or registered trademarks of their respective owners.

Table of Contents

INTRODUCTION.....	5
1.1 AUDIENCE.....	5
1.2 ABOUT THIS DOCUMENT.....	5
1.3 OTHER RELATED DOCUMENTS.....	5
1.4 DOCUMENT REVISION HISTORY.....	5
ISCSI INTRODUCTION.....	7
2.1 SCSI.....	7
2.2 iSCSI.....	7
ISCSI ACCELERATION WITH CHELSIO'S TERMINATOR ASIC.....	12
3.1 TCP OFFLOAD ENGINE.....	12
3.2 CHELSIO ULP iSCSI ACCELERATION.....	12
CHELSIO ISCSI TARGET MODULE.....	14
4.1 ISCSI SOFTWARE DESIGN.....	15
4.2 ISCSI PROCESSING CONTEXT.....	17
4.2.1 <i>Interface functions</i>	17
REGISTER WITH ISCSI TARGET MODULE.....	18
5.1 ISCSI TARGET CLASS.....	18
5.1.1 <i>Target Class Identification</i>	19
5.1.2 <i>Initiator Login Initialization</i>	19
5.1.3 <i>Login Phase Check</i>	21
5.1.4 <i>Retrieving CHAP Information</i>	22
5.1.5 <i>iSCSI Target Session Management</i>	22
5.1.6 <i>Target Discovery</i>	23
5.1.7 <i>Target Redirection</i>	24
5.2 ISCSI TARGET LUN CLASS.....	24
5.2.1 <i>Target LUN Class Identification</i>	26
5.2.2 <i>Target LUN Class Property</i>	26
5.2.2.1 LUN_CLASS_SCSI_PASS_THRU_BIT.....	26
5.2.2.2 LUN_CLASS_MULTI_PHASE_DATA_BIT.....	26
5.2.2.3 LUN_CLASS_HAS_CMD_QUEUE_BIT.....	26
5.2.3 <i>Target LUN Configuration Functions</i>	27
5.2.4 <i>Handling of iSCSI SCSI Command</i>	27
ISCSI TARGET CONFIGURATION.....	30

6.1	TARGET CONFIGURATION VIA KERNEL APIs.....	30
6.1.1	Adding an iSCSI Target Node.....	30
6.1.2	Removing an iSCSI Target Node.....	31
6.1.3	Modify an iSCSI Target Node's Settings.....	31
6.1.3.1	CHAP Configuration.....	32
	ISCSI TARGET PROCESSING.....	34
7.1	ISCSI SESSION.....	34
7.2	ISCSI DISCOVERY SESSION.....	34
7.3	ISCSI NORMAL SESSION.....	34
7.3.1	The Login Phase.....	35
7.3.1.1	First Login Request.....	35
7.3.1.2	Login Stage Transition.....	37
7.3.1.3	CHAP Processing.....	37
7.3.2	Session Addition.....	38
7.3.3	Session Removal.....	38
7.3.4	Session Abort.....	39
7.3.5	iSCSI SCSI Command Processing.....	39
7.3.5.1	iSCSI SCSI Command.....	39
7.3.5.2	iSCSI SCSI Command Read Operation.....	46
7.3.5.3	iSCSI SCSI Command Write Operation.....	50
7.3.5.4	Abort a SCSI Command.....	54
7.3.5.5	Debugging SCSI Command problem.....	55
7.3.6	iSCSI Task Management Functions.....	55
	TARGET INFORMATION AND STATISTICS.....	58
8.1	TARGET INFORMATION.....	60
8.2	SESSION INFORMATION.....	60
8.3	CONNECTION INFORMATION.....	60
8.4	PORTAL PERFORMANCE DATA.....	61
	CHELSIO ISCSI TARGET API FILES.....	62
1	INTRODUCTION.....	51.1
	AUDIENCE.....	51.2
	ABOUT THIS DOCUMENT.....	51.3
	OTHER RELATED DOCUMENTS.....	51.4
	DOCUMENT REVISION HISTORY.....	52
	ISCSI INTRODUCTION.....	72.1
	ISCSI73.....	72.2
	ISCSI ACCELERATION WITH CHELSIO'S TERMINATOR ASIC.....	103.1
	TCP OFFLOAD ENGINE.....	103.2

CHELSIO ULP ISCSI ACCELERATION.....	104
CHELSIO ISCSI TARGET MODULE.....	124.1ISCSI
SOFTWARE DESIGN.....	134.2ISCSI
PROCESSING CONTEXT.....	144.2.1
INTERFACE FUNCTIONS.....	145
REGISTER WITH ISCSI TARGET MODULE.....	155.1ISCSI
TARGET CLASS.....	155.1.1
TARGET CLASS IDENTIFICATION.....	165.1.2
INITIATOR LOGIN INITIALIZATION.....	165.1.3
LOGIN PHASE CHECK.....	185.1.4
RETRIEVING CHAP INFORMATION.....	195.1.5ISCSI
TARGET SESSION MANAGEMENT.....	195.1.6
TARGET DISCOVERY.....	205.1.7
TARGET REDIRECTION.....	215.2ISCSI
TARGET LUN CLASS.....	215.2.1
TARGET LUN CLASS IDENTIFICATION.....	235.2.2
TARGET LUN CLASS PROPERTY.....	235.2.2.1
LUN_CLASS_SCSI_PASS_THRU_BIT.....	235.2.2.2
LUN_CLASS_MULTI_PHASE_DATA_BIT.....	235.2.2.3
LUN_CLASS_HAS_CMD_QUEUE_BIT.....	235.2.3
TARGET LUN CONFIGURATION FUNCTIONS.....	245.2.4
HANDLING OF ISCSI SCSI COMMAND.....	246ISCSI
TARGET CONFIGURATION.....	276.1
TARGET CONFIGURATION VIA KERNEL APIS.....	276.1.1
ADDING AN ISCSI TARGET NODE.....	276.1.2
REMOVING AN ISCSI TARGET NODE.....	286.1.3
MODIFY AN ISCSI TARGET NODE'S SETTINGS.....	286.1.3.1CHAP
CONFIGURATION.....	297ISCSI
TARGET PROCESSING.....	317.1ISCSI
SESSION.....	317.2ISCSI
DISCOVERY SESSION.....	317.3ISCSI
NORMAL SESSION.....	327.3.1THE
LOGIN PHASE.....	327.3.1.1FIRST
LOGIN REQUEST.....	327.3.1.2
LOGIN STAGE TRANSITION.....	347.3.1.3CHAP
PROCESSING.....	347.3.2
SESSION ADDITION.....	357.3.3
SESSION REMOVAL.....	357.3.4
SESSION ABORT.....	367.3.5ISCSI
SCSI COMMAND PROCESSING.....	367.3.5.1ISCSI
SCSI COMMAND.....	367.3.5.2ISCSI
SCSI COMMAND READ OPERATION.....	437.3.5.3ISCSI
SCSI COMMAND WRITE OPERATION.....	467.3.5.4
ABORT A SCSI COMMAND.....	487.3.5.5
DEBUGGING SCSI COMMAND PROBLEM.....	497.3.6ISCSI
TASK MANAGEMENT FUNCTIONS.....	498

TARGET INFORMATION AND STATISTICS.....	528.1
TARGET INFORMATION.....	548.2
SESSION INFORMATION.....	548.3
CONNECTION INFORMATION.....	548.4
PORTAL PERFORMANCE DATA.....	559
CHELSIO ISCSI TARGET API FILES.....	56APPENDIX A.
CHELSIO ISCSI SOFTWARE FEATURES.....	57APPENDIX B.ISCSI
TEXT KEY SETTINGS.....	58APPENDIX C.
CHELSIO TARGET CONFIGURATION KEYS.....	59APPENDIX D.LOG
AND DEBUG MESSAGES.....	60APPENDIX E.ISCSI
ACRONYMS.....	627
7.....	60

1 Introduction

This document describes the Chelsio iSCSI Target APIs to be used with 3rd-party Storage Driver. The Chelsio iSCSI Target APIs enables the Storage Driver to communicate to the iSCSI Target Module.

- Section 2 introduces the iSCSI system.
- Section 3 explains the Terminator ASIC's ULP Acceleration and its APIs.
- Section 4 overviews Chelsio iSCSI Target Module Software.
- Section 5 defines how the Storage Driver could register with the Target by creating new iSCSI Target Class and iSCSI Target LUN class.
- Section 6 specifies how to add, remove and modify an iSCSI Target Node.
- Section 7 describes the exported APIs in the iSCSI Target processing flow.
- Section 8 describes the exported API for iSCSI Target information and Statistics
- Section 9 shows the source file layout for the data structures and the APIs described in this document.

1.1 Audience

The audience for the Chelsio iSCSI Target API Guide is primarily engineers who want to utilize the Chelsio iSCSI Target Module to support any specialized storage technology as the backend storage for iSCSI Targets.

1.2 About This Document

This manual contains the details of the data structures and kernel APIs exported by the Chelsio iSCSI Target Module.

1.3 Other Related Documents

For additional information about the Chelsio iSCSI Software and the Chelsio Terminator ASIC, refer to the following related documents listed below:

- *Internet Small Computer Systems Interface (iSCSI)*, IETF RFC-3720, April 2004.
- *PPP Challenge Handshake Authentication Protocol (CHAP)*, IETF RFC-1994, August 1996.
- *iSCSI Software Installation and User Guide*, Chelsio communications.

1.4 Document Revision History

Revision	Comment	Date
0.1	Initial version.	07/16/2008
0.5	Updated read/write walkthrough and state diagram from multi-phase	06/02/2009

	data buffer handling	
0.6	Changed fp_chap_info_get function from blocking to non-blocking.	09/17/2009
0.7	Add support of multiple outstanding buffers for multi-phase data buffer handling. Removed requirement that only one outstanding buffer per phase.	09/29/2009
0.8	Add support for buffer overflow and underflow	06/23/2010
0.9	Add more clarification and examples in the buffer setup section as per customer request	07/22/2010
0.9.5	Updated tcp_endpoint structure with new iscsi_tcp_endpoints	02/01/2011
1.0	Update fp_first_login_check_done API	09/09/2011
1.1	Added a return value for fp_session_added(), and pass the value as part of the iscsi_scsi_command	08/22/2013
1.2	Added the private session handle to fp_tmf_execute(). Build #706	10/17/2013
1.3	Remove reference to t3	03/10/2015
1.4	Added details on target process context and API functions.	04/21/2015
1.5	prefix data structures and functions with ch(elsio)_.	06/12/2015

2 iSCSI Introduction

iSCSI (Internet Small Computer System Interface, also commonly referred to as Internet SCSI or “SCSI over IP”) is a storage networking standard that enables the transport of block-level I/O data over an IP network.

2.1 SCSI

The Small Computer Systems Interface (SCSI) is a popular family of protocols for communicating with I/O devices, especially storage devices.

There are two types of devices in the SCSI protocol; the SCSI *Initiators (clients)* start the communications and the *Targets (servers)* responds. The initiators are devices that request commands be executed. Targets are devices that carry out the commands.

A Command Descriptor Block (CDB) is used to carry a command from an application client to a device server. SCSI command execution results in an optional data phase and a status phase. In the data phase, data travels either from the initiator to the target as in a WRITE command or from the target to the initiator as in a READ command. In the status phase, the target returns the final status of the operation. The status response terminates an SCSI command or task.

2.2 iSCSI

iSCSI replaces SCSI’s direct-attached cabling architecture with a network fabric. It works by encapsulating SCSI commands into packets and transporting them via TCP connections. This allows for the deployment of storage networks over a commodity IP network

The iSCSI protocol follows the SCSI’s client-server model. On opposing ends of the network are the iSCSI initiator client and the iSCSI target server.

The main responsibilities of the initiator are to establish a connection to an iSCSI target and to forward the SCSI Commands and transferring data to the target.

The iSCSI target’s primary functions are to broker the requests issued by the initiator to the physical storage and to forward the storage’s response back to the initiator. Essentially, the iSCSI target acts as the bridge between the network and the storage devices such as disks, or RAID arrays.

The figure below depicts an iSCSI network between multiple initiators and targets interconnected through a TCP/IP network.

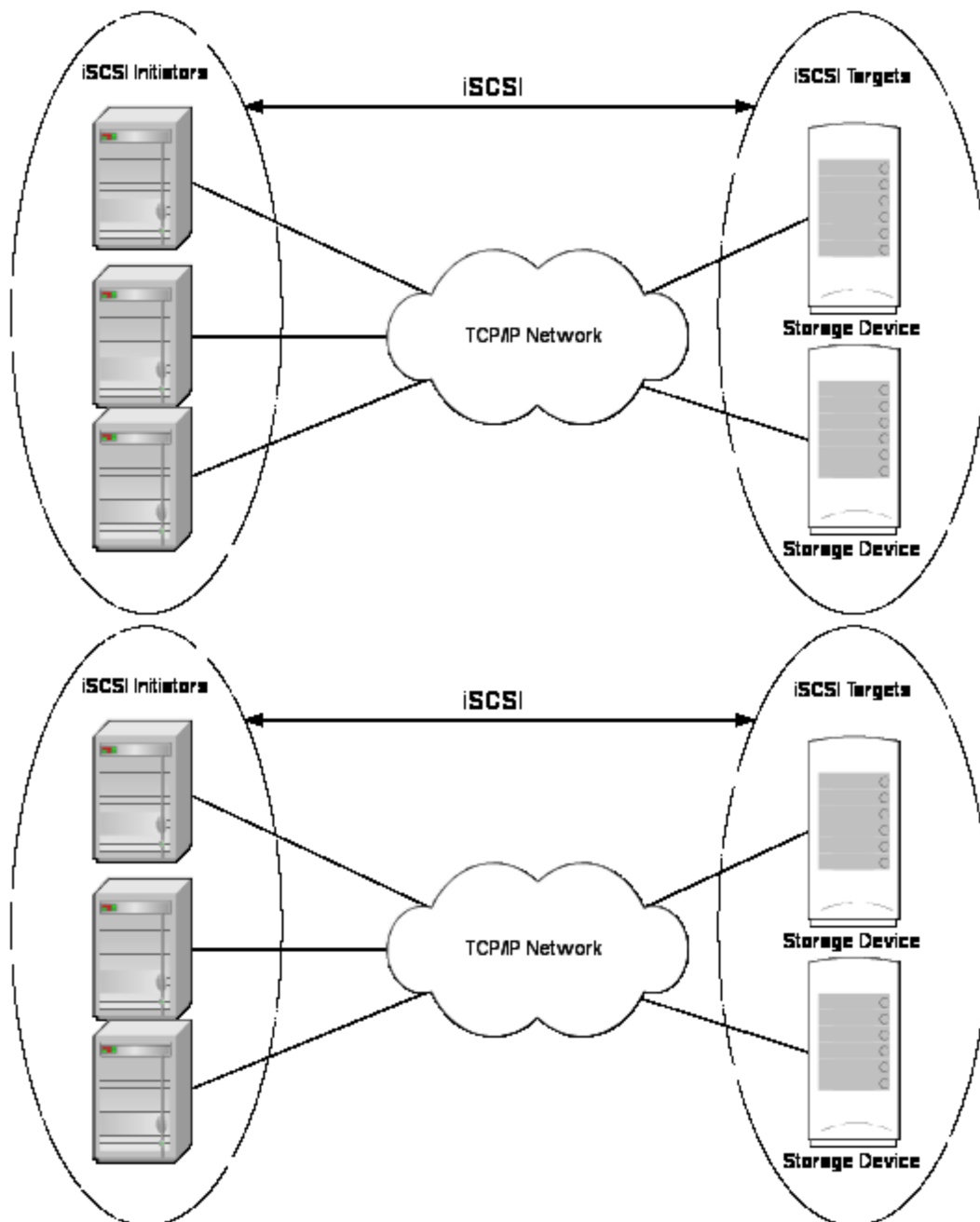


Figure 2-1. iSCSI Network

The messages exchanged between the Initiator and the target are called iSCSI Protocol Data Units (PDUs). They are used to carry SCSI and iSCSI commands from the Initiator to the Target, to transfer data in either direction, and to return responses from the Target to the Initiator.

Because iSCSI storage is presented to the OS as though it is attached locally, rather than as a network share, iSCSI's compatibility with existing software applications is practically guaranteed. It offers great flexibility as users can use the OS's native file

system on the remote storage devices over any IP network infrastructure just like any other local block-level device.

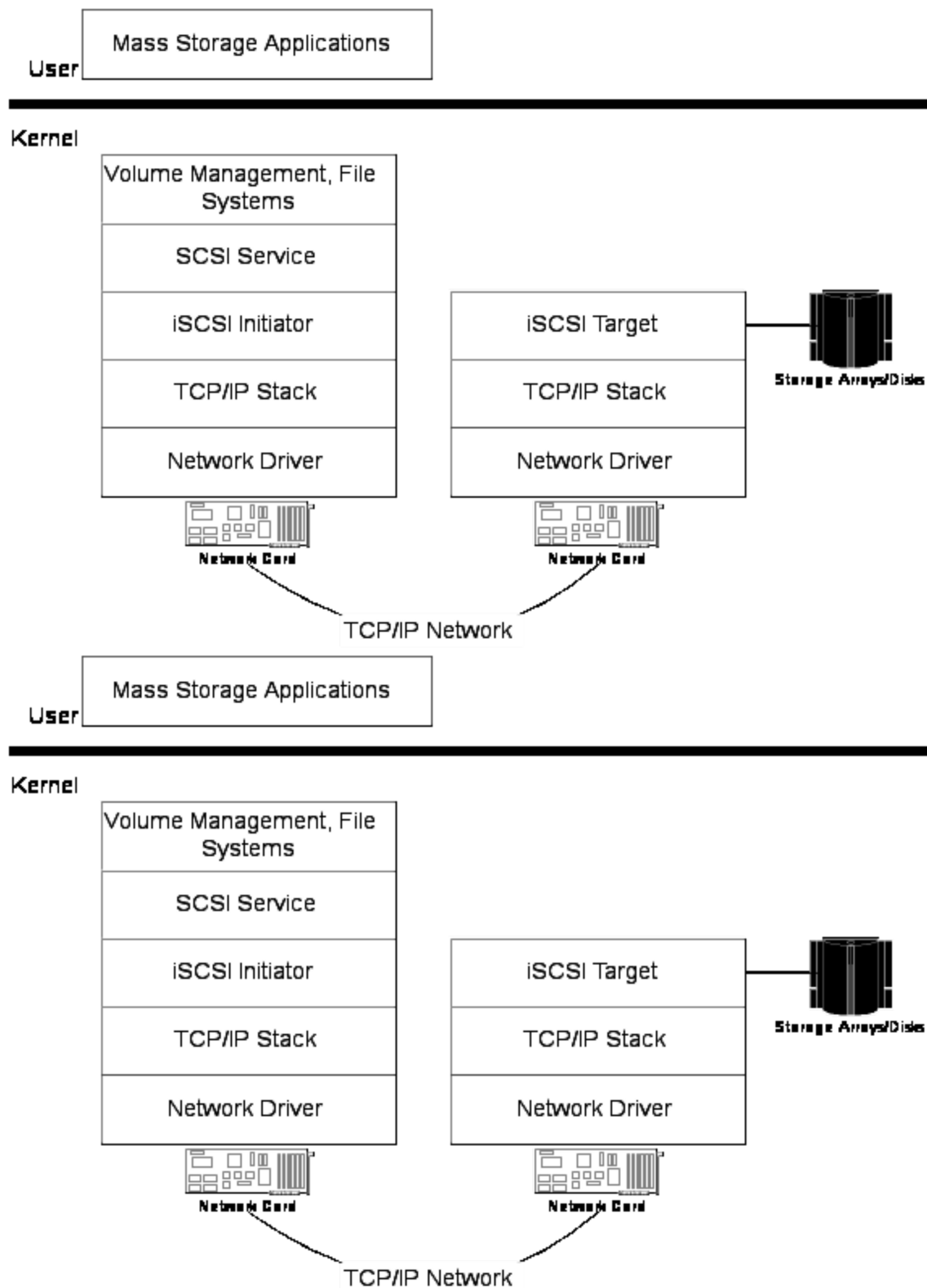


Figure 2-1. iSCSI System Overview

The simplified illustration above shows the software and hardware elements involved in an iSCSI system:

- The iSCSI Target exports the locally attached storage devices to iSCSI initiators. It receives and processes the SCSI requests from iSCSI initiators and delivers the responses back to the initiators.
- The iSCSI Initiator presents the remote storage devices exported by an iSCSI target to the local SCSI subsystem as if they are locally-attached SCSI devices.
- The iSCSI Initiator and Target communicates on top of TCP/IP network.

3 iSCSI Acceleration With Chelsio's Terminator ASIC

Chelsio's Terminator ASIC delivers full TCP offload and provides acceleration to iSCSI's CPU-intensive operations (such as PDU recovery, CRC digest computation and verification, Direct Data Placement into host memory buffers) across the PCI bus.

3.1 TCP Offload Engine

The Terminator's full TCP/IP offload engine handles the connection establishment and teardown, as well as all the exceptions including retransmission and re-ordering in the hardware.

Comparing with the TCP stacks resides in the host the Terminator's offload engine provides much higher performance due to the large reduction in packet and interrupt rate, and zero copy capability, all without requiring any application changes.

The Terminator TCP offload engine also offers a unique zero-copy "direct data placement" feature. It allows dramatic reductions not only in CPU load, but also in memory subsystem usage, which is oftentimes the actual bottleneck in today's systems.

In addition, hardware based retransmission and re-ordering virtually eliminate the impact of packet loss on performance, transforming the normally lossy Ethernet network into a high performance reliable fabric.

3.2 Chelsio ULP iSCSI Acceleration

In addition to offloading TCP processing from the host processor, the Terminator ASIC also supports iSCSI Upper Layer Protocol (ULP) acceleration and iSCSI Direct Data Placement (DDP) where the hardware handles the expensive byte touching operations, such as CRC computation and verification.

- **iSCSI PDU Digest Generation and Verification**

On transmitting, Terminator computes and inserts the Header Digest and the Data Digest into the iSCSI PDUs.

On receiving, the Terminator computes and verifies the Header Digest and the Data Digest of the iSCSI PDUs.

- **Direct Data Placement (DDP)**

Terminator can directly place the iSCSI Data-In or Data-Out PDU's payload into pre-posted final-destination host-memory buffers based on the Initiator Task Tag (ITT) or Target Task Tag (TTT) information contained in the PDU header. The "Page Pod" concept is used to manage the host memory buffers and the tags.

- **PDU Transmit and Recovery**

On transmitting, Terminator accepts the complete PDU (header + data) from the host driver, computes and inserts the digests, decomposes the PDU into multiple TCP segments if necessary, and transmits all the TCP segments onto the wire.

On receiving, Terminator recovers the iSCSI PDU by reassembling TCP segments, separating the header and data, calculating and verifying the digests, then forwards the header to the host. The payload data, if possible, will be directly placed into a pre-posted host DDP buffer. Otherwise, the payload data will be sent to the host too.

NOTE:

- The Terminator ASIC is unaware of iSCSI login, iSCSI sessions and other finer details of the iSCSI protocol. However, the Terminator is aware of TCP connections that carry iSCSI traffic and it performs the above functions for any TCP connection labeled with the iSCSI attribute.

4 Chelsio iSCSI Target Module

Chelsio's iSCSI Target Module implements the iSCSI target-mode protocol stack. It comes with the full support of Chelsio's Terminator ASIC, taking advantage of both the full TCP offload support and the ULP iSCSI acceleration.

The Chelsio's iSCSI Software can also operate on top of host's native TCP/IP stack over any third-party NIC.

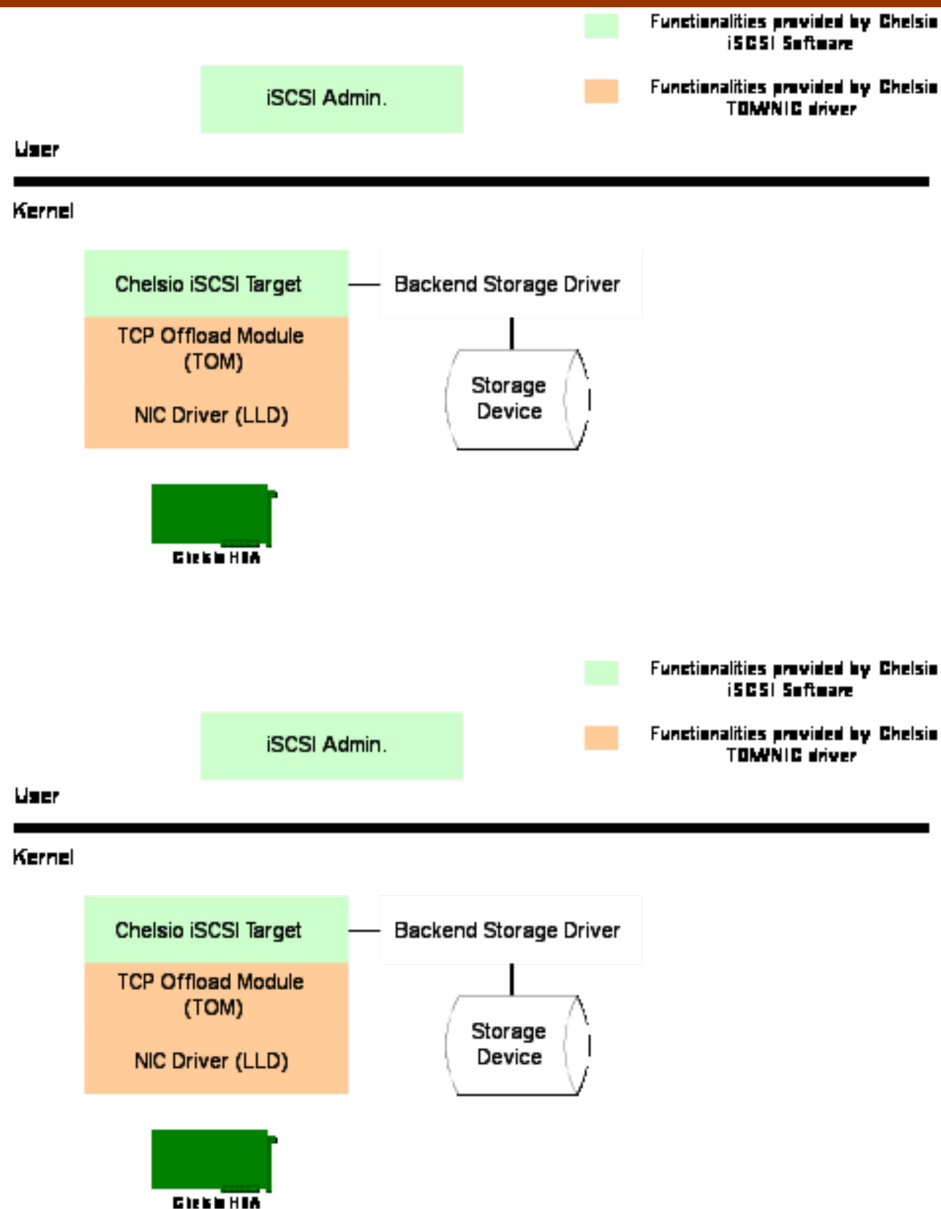


Figure 4-1. Chelsio iSCSI Target Module Overview

The figure above shows a simplified view of the hardware and software components involved:

- The Chelsio protocol-offload Host Bus Adapters (HBAs) are installed in the target systems. These boards use the Terminator ASICs for offloading TCP/IP and accelerating iSCSI.

- The Chelsio TCP offload module (TOM) is a peer to the hosts' native TCP/IP stack. Used by the Chelsio iSCSI Target Module to offload TCP/IP processing, it enables the Host to bypass the native TCP/IP stack, and send and receive TCP data as a byte stream.
- The Chelsio Low-Level Driver (LLD) is the device driver for Chelsio's protocol engine.

NOTE:

1. The Chelsio's TCP offload module (TOM) and Low Level Driver (LLD) are outside of the scope of this document. For more details on these modules please refer to the related TOE driver documents.

4.1 iSCSI Software Design

The Chelsio iSCSI Target Module is designed with the following goals in mind:

- **Performance**
Highly optimized data path to utilize the full potential of Chelsio Terminator ASIC's offloading capabilities (TCP and iSCSI); specialized functions may be developed for better performance.
- **Inter-operability**
To the local host system, Chelsio iSCSI Target Module behaves in the exact same way as traditional direct-attached storage systems. No change is required for the operating system or high-level applications.
- **Storage Management**
The iSCSI Target Module provides target device virtualization. It comes with support for block devices (SCSI, IDE, Serial-ATA disk drives, etc) to be used easily. And it also provides abilities to utilize disk management features like Logical Volume Manager (LVM) and software RAID for flexible storage management and redundancy.

In addition, the iSCSI Target Module exports a set of kernel API to enable communications with any Storage Driver thus expanding the supports of additional target device types.
- **Maintainability**
As various operating systems are continue to evolve, the Chelsio iSCSI software tries to keep the design simple by using as much high-level interfaces as possible. The usage of well-tested, high-level standard OS/kernel interfaces has clear advantages in terms of stability, implementation and maintenance costs, and portability.

4.2 iSCSI Processing Context

The Target Module utilizes the kernel thread for the iSCSI protocol processing:

- 1 main thread

The main thread handles the iSCSI connections in the Login Phase.

- N worker threads (where N = # of logical CPUs in the host system)

The worker thread handles the iSCSI connections in the Full Feature Phase (FFP).

Once an iSCSI connection completes the login phase, the connection is handed from the main thread to one of the worker threads for continuing processing.

4.2.1 Interface functions

For the functions that the Storage Driver provides they will be invoked by the Target Module in one of the kernel thread context. Because these functions are executed in the Target Module kernel thread context, they are expected to be either as

- Blocking

The function is expected to complete its functionality entirely during the call and no callback mechanism into the Target Module is provided.

Most of the control plane functions fall into this category.

- Non-blocking

The function should do minimum processing and return the control to the Target Module as soon as possible. It is expected the bulk of work should be done by the Storage Driver outside of the Target Module kernel thread context. A callback mechanism into the Target Module is provided.

All data plane functions and some of the control plane functions fall into this category.

For the functions that are exported by the Target Module (i.e., called by the Storage Driver), similar to the non-blocking functions, they will perform minimum house-keeping and then wake up the related Target kernel thread to continue the iSCSI processing. Because the target kernel thread could become active before the function returns, there is a small window that the Target thread would call into one of the Storage Driver provided functions at the same time.

5 Register with iSCSI Target Module

To be able to communicate with the Target Module, the Storage Driver should first create and register a new class of iSCSI target and target LUN.

5.1 iSCSI Target Class

To add or remove a new class of iSCSI target, the Storage Driver should use the following exported APIs:

- `int chiscsi_target_class_register (chiscsi_target_class *target_class);`
- `int chiscsi_target_class_deregister (char *class_name);`

The structure *chiscsi_target_class* provides a series of hooks in various points in the iSCSI target protocol processing stack.

```
typedef struct chiscsi_target_class chiscsi_target_class;

struct chiscsi_target_class {
    ...
    char                *class_name;

    void (*fp_first_login_check) (unsigned long hndl,
                                  char          *initiator_name,
                                  char          *target_name,
                                  chiscsi_tcp_endpoint *endpoint);

    void (*fp_login_stage_check) (unsigned long hndl,
                                  unsigned char login_stage,
                                  char          *initiator_name,
                                  char          *target_name,
                                  chiscsi_tcp_endpoint *endpoint);

    void (*fp_chap_info_get) (char          *initiator_name,
                              char          *target_name,
                              chap_info    *chap);

    unsigned long (*fp_session_added) (unsigned long sess_hndl,
                                       unsigned char isid[6],
                                       char          *initiator_name,
                                       char          *target_name);

    void (*fp_session_removed) (unsigned long sess_hndl,
                                char          *initiator_name,
                                char          *target_name);

    int (*fp_discovery_target_accessible)(
        unsigned long hndl,
        char          *initiator_name,
        char          *target_name,
```



```

                                chiscsi_tcp_endpoint *endpoint);
int (*fp_select_redirection_portal)(char *target_name,
                                char *initiator_name,
                                chiscsi_tcp_endpoint *endpoint);
};

```

5.1.1 Target Class Identification

Each iSCSI target class is identified with a unique character string “*class_name*”. It must not be NULL.

5.1.2 Initiator Login Initialization

```

void (*fp_first_login_check) (    unsigned long hndl,
                                char *initiator_name,
                                char *target_name,
                                chiscsi_tcp_endpoint *endpoint);

```

where

- *hndl* is a value uniquely identifies the iSCSI connection
- *initiator_name* points to the initiator name string
- *target_name* points to the target name string
- *endpoint* contains the tcp 4-tuple.

The structure *chiscsi_tcp_endpoints* represents the tcp 4-tuples, and is defined as:

```

typedef struct chiscsi_tcp_endpoints    chiscsi_tcp_endpoints;
struct chiscsi_tcp_endpoints {
    unsigned int    f_ipv6:1;
    unsigned int    f_filler:31;
    struct tcp_endpoint iaddr;    /* initiator tcp address */
    struct tcp_endpoint taddr;    /* target tcp address */
    unsigned int    port_id;    /* for Chelsio HBA only */
};

```

The structure *tcp_endpoints* represents the tcp endpoint, and is defined as:

```

#define ISCSI_IPADDR_LEN    16
struct tcp_endpoint {
    unsigned char    ip[ISCSI_IPADDR_LEN];
    unsigned int    port;
};

```

If the Storage Driver wants to be informed of an Initiator login attempt for a normal session, it could set the *fp_first_login_check()*. This hook is the ideal place for actions such as

- preparing the target for a possible new iSCSI session and/or
- checking if the target resource is available and/or

- performing iSCSI Initiator access control

When the iSCSI protocol stack receives the first Login Request PDU for a normal session from a newly accepted TCP connection, it invokes *fp_first_login_check()*.

The function *fp_first_login_check()* is non-blocking. The Target Module will halt the processing of the particular connection until the Storage Driver notifies the Target Module via the exported API *chiscsi_target_first_login_check_done()*.

```
void chiscsi_target_first_login_check_done(
    unsigned long   hndl,
    unsigned char   login_status_class,
    unsigned char   login_status_detail,
    unsigned int    max_cmd);
```

where

- *hndl* is the value uniquely identified the iSCSI connection, which is passed in via *fp_first_login_check()*.
- *login_status_class* and *login_status_detail* are used to reject a login request.
- *max_cmd* is the maximum number of outstanding SCSI commands this target can handle for the new session. If it is zero then the iSCSI target protocol stack will default to maximum of 128 outstanding SCSI commands per session.

login_status_class and *login_status_detail* should both be set to zero if the Storage Driver completes its processing successfully and wants the login phase to proceed.

In order to use Redirection, storage driver should set the *login_status_class* and *login_status_detail*, both to zero. Chelsio Target driver will treat Redirection status class as Success.

In the case of error, *login_status_class* should be set to one of the following values:

- #define ISCSI_LOGIN_STATUS_CLASS_INITIATOR_ERROR 2
- #define ISCSI_LOGIN_STATUS_CLASS_TARGET_ERROR 3

If *login_status_class* = *ISCSI_LOGIN_STATUS_CLASS_INITIATOR_ERROR* then the possible values for *login_status_detail* are:

- #define ISCSI_LOGIN_STATUS_DETAIL_INIT_ERROR 0
- #define ISCSI_LOGIN_STATUS_DETAIL_NO_PERMS 2
- #define ISCSI_LOGIN_STATUS_DETAIL_TARGET_NOT_FOUND 3
- #define ISCSI_LOGIN_STATUS_DETAIL_TARGET_REMOVED 4
- #define ISCSI_LOGIN_STATUS_DETAIL_INVALID_REQUEST 11

If *login_status_class* = *ISCSI_LOGIN_STATUS_CLASS_TARGET_ERROR* then *login_status_detail* should be set to one of the following values:

- #define ISCSI_LOGIN_STATUS_DETAIL_TARG_ERROR 0

- #define ISCSI_LOGIN_STATUS_DETAIL_SERVICE_UNAVAIL 1
- #define ISCSI_LOGIN_STATUS_DETAIL_NO_RESOURCES 2

5.1.3 Login Phase Check

The iSCSI login phase could proceed in two stages:

1. #define ISCSI_LOGIN_STAGE_SECURITY 0
2. #define ISCSI_LOGIN_STAGE_OPERATIONAL 1

In the login security stage the Initiator and the Target negotiate the authentication method (defined by iSCSI key “*AuthMethod*”) and perform the authentication if needed.

The login operational stage consists of text string negotiation of operating parameters using key=value pairs of login parameter exchanges.

fp_login_stage_check() is invoked when the login security stage and the login operational stage are first entered for the leading connection of a normal session. This hook is the ideal place to perform initiator access control.

```
void (*fp_login_stage_check) (unsigned long hndl,
                             unsigned char login_stage,
                             char *initiator_name,
                             char *target_name,
                             chiscsi_tcp_endpoint *endpoint);
```

where

- *hndl* is a value uniquely identifies the iSCSI connection
- *login_stage* is either *ISCSI_LOGIN_STAGE_SECURITY* or *ISCSI_LOGIN_STAGE_OPERATIONAL*.
- *initiator_name* points to the initiator name string
- *target_name* points to the target name string
- *endpoint* contains the tcp 4-tuple.

fp_login_stage_check() is non-blocking. The Target Module will halt the processing of the connection until the Storage Driver informs the target the checking has been completed via the exported API *chiscsi_target_login_stage_check_done()*.

```
void chiscsi_target_login_stage_check_done(
    unsigned long hndl,
    unsigned char login_status_class,
    unsigned char login_status_detail);
```

where

- *hndl* is the value uniquely identified the iSCSI connection, which is passed in via *fp_login_stage_check()*.
- *login_status_class* and *login_status_detail* are used to reject a login request.

5.1.4 Retrieving CHAP Information

If the Storage Driver manages its own CHAP database, it should set up *fp_chap_info_get()*. This function is used by the Target Module to retrieve CHAP names and secrets from the Storage Driver during the login phase.

```
void (*fp_chap_info_get) (      char      *initiator_name,
                             char      *target_name,
                             chap_info *chap);
```

where

- *initiator_name* points to the initiator name string
- *target_name* points to the target name string
- *chap* contains the CHAP name and secret.

The *chap_info* structure is used to pass CHAP-related information from the Storage Driver to the Target Module.

```
typedef struct chap_info      chap_info;

#define CHAP_FLAG_LOCAL_NAME_VALID      0x1
#define CHAP_FLAG_LOCAL_SECRET_VALID   0x2
#define CHAP_FLAG_REMOTE_NAME_VALID    0x4
#define CHAP_FLAG_REMOTE_SECRET_VALID  0x8
#define CHAP_FLAG_MUTUAL_REQUIRED      0x10

struct chap_info {
    unsigned char      flag;

    unsigned char      local_secret_length;
    unsigned char      remote_secret_length;
    unsigned char      filler;

    unsigned char      local_secret[16];
    unsigned char      local_name[256];

    unsigned char      remote_secret[16];
    unsigned char      remote_name[256];
};
```

5.1.5 iSCSI Target Session Management

If the Storage Driver wants to be informed of the session creation and termination it should set up *fp_session_added()* and *fp_session_removed()*.

The target module calls *fp_session_added()* whenever a session goes into service (i.e., its leading connection transits to the Full Feature Phase).

The function *fp_session_removed()* is called by the Target Module whenever a normal session has been terminated.

```

unsigned long (*fp_session_added) (unsigned long sess_hdl,
                                   unsigned char isid[6],
                                   char          *initiator_name,
                                   char          *target_name);

void (*fp_session_removed) (      unsigned long sess_hdl,
                                   char          *initiator_name,
                                   char          *target_name);

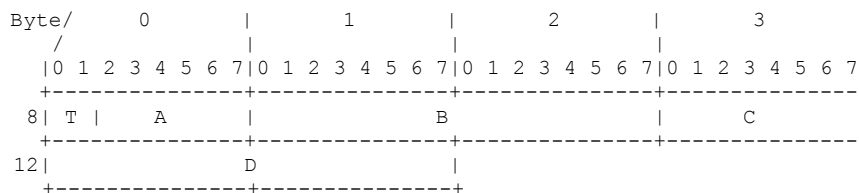
```

where

- *hdl* is the target session handle.
- *isid* is the initiator session id.
- *initiator_name* points to the initiator name string
- *target_name* points to the target name string

The ISID is a unique identifier that an Initiator assigns to its end point of the session. When combined with the iSCSI Initiator Name, it uniquely identifies its SCSI Initiator Port.

According to RFC3720 the ISID is a six-byte value:



For *fp_session_added()*, the Storage Driver can choose to pass back a unsigned value (or a casted pointer) to be associated with the target session handle *hdl*.

5.1.6 Target Discovery

fp_discovery_target_accessible() should be set if the Storage Driver wants to control what targets an initiator should be able to see.

```

int (*fp_discovery_target_accessible)(
                                   unsigned long hndl,
                                   char          *initiator_name,
                                   char          *target_name,
                                   iscsi_tcp_endpoint *endpoint);

```

where

- *hdl* is a value uniquely identifies the iSCSI connection
- *initiator_name* points to the initiator name string
- *target_name* points to the target name string
- *endpoint* contains the tcp 4-tuple.

fp_discovery_target_accessible() is a blocking call. The Storage Driver should return zero (0) if it does not want the target named "*target_name*" to be seen by the initiator

named “*initiator_name*” or returns one (1) otherwise. It should return a value less than zero (<0) in the case of error.

5.1.7 Target Redirection

An iSCSI Target can redirect an initiator to use a different IP address and port (often called a portal) instead of the current one. The redirected-to target portal can either be on the same machine, or on another system.

Chelsio configuration key *ShadowMode* is used to distinguish whether the redirected target portals are on a different system or not. If *ShadowMode=Yes*, then the redirected-to target portals are on another system.

To configure target redirection, specify the redirected-to target portals, and add their portal group tag to local listening portal. In the following example, any login requests received on *10.193.184.81:3260* will be redirected to *10.193.184.85:3261*:

```
PortalGroup=2@10.193.184.85:3261
PortalGroup=1@10.193.184.81:3260,[2]
```

fp_select_redirection_portal() should be set if the Storage Driver wants to control which redirected-to portal the initiator should be connected to.

```
int (*fp_select_redirection_portal)(  char          *target_name,
                                     char          *initiator_name,
                                     chiscsi_tcp_endpoint *endpoint);
```

where

- *target_name* points to the target name string
- *initiator_name* points to the initiator name string
- *endpoint* contains the tcp 4-tuple.

fp_select_redirection_portal() is a blocking call.

The Storage Driver should fill the field *endpoint->taddr* as the redirected-to portal and return zero (0). It should return a value less than zero (<0) in the case of error.

5.2 iSCSI Target LUN Class

An *chiscsi_target_lun_class* defines a LUN type and its behavior for a particular *chiscsi_target_class*.

To relate a LUN class with an existing iSCSI target class uses the following exported API:

- `int chiscsi_target_lun_class_register(`

`chiscsi_target_lun_class`
`*lun_class,`

`)`

```

        char                *target_class_name);
    • int chiscsi_target_lun_class_deregister(
        char    *lun_class_name,
        char    *target_class_name);

```

A *chiscsi_target_class* must contain at least one *chiscsi_target_lun_class*.

An iSCSI target node can accommodate multiple LUNs. Similarly, a *chiscsi_target_class* can have multiple *chiscsi_target_lun_class* registered to it.

```

enum lun_class_property_bits {
    LUN_CLASS_SCSI_PASS_THRU_BIT,
    LUN_CLASS_MULTI_PHASE_DATA_BIT,
    LUN_CLASS_HAS_CMD_QUEUE_BIT, /* LU maintains its own queue, this
                                   should be the default mode, so it
                                   won't exe. in the iscsi target stack
                                   context */
};

```

```

typedef struct chiscsi_target_lun_class chiscsi_target_lun_class;

```

```

struct chiscsi_target_lun_class {
    ...
    char                *class_name;
    unsigned int        property;

    int (*fp_config_parse_options) (unsigned long  hndl,
        char                *buf,
        int                buflen,
        char                *ebuf);

    int (*fp_attach) (unsigned long  hndl,
        char                *ebuf,
        int                ebuflen);

    void (*fp_detach) (unsigned long  hndl);

    int (*fp_reattach) (unsigned long  hndl,
        unsigned long  hndl_new,
        char                *ebuf,
        int                ebuflen);

    int (*fp_flush) (unsigned long  hndl);

    int (*fp_scsi_cmd_cdb_rcved) (chiscsi_scsi_command *scmd);

    int (*fp_scsi_cmd_data_xfer_status)(chiscsi_scsi_command *scmd);

    int (*fp_scsi_cmd_abort) (chiscsi_scsi_command *scmd);

    void (*fp_scsi_cmd_abort_status) (unsigned int  sc_lun,
        unsigned int  sc_cmdsn,
        unsigned int  sc_xfer_sgcnt,

```

```

        unsigned char *sc_xfer_sreq_buf,
        void          *sc_sdev_hndl);

    int (*fp_tmf_execute)

        (unsigned long sess_tclass,
         unsigned long hndl,
         unsigned char immediate_cmd,
         unsigned char tmf_func,
         unsigned int  lun,
         chiscsi_scsi_command *scmd);
};

```

5.2.1 Target LUN Class Identification

Each iSCSI target LUN class is identified with a unique character string “*class_name*”. It must not be NULL.

5.2.2 Target LUN Class Property

5.2.2.1 LUN_CLASS_SCSI_PASS_THRU_BIT

If the Storage Driver wants to handle all the SCSI command (encapsulated in *SCSI Command PDU*) received from the Initiator itself, it should set the *LUN_CLASS_SCSI_PASS_THRU_BIT* in the property (i.e., property = 1 << LUN_CLASS_SCSI_PASS_THRU_BIT).

5.2.2.2 LUN_CLASS_MULTI_PHASE_DATA_BIT

It is preferred that the Storage Driver could supply all the data buffers for a SCSI command in one shot -- i.e., for a read command all the read data is returned together and for a write command all the write data buffers can be allocated and returned together.

If the Storage Driver wants to allocate and return the data buffers for the SCSI command in chunks (i.e., the data buffers are allocated and returned in multiple steps), it should set the *LUN_CLASS_MULTI_PHASE_DATA_BIT* in the property (i.e., property = 1 << LUN_CLASS_MULTI_PHASE_DATA_BIT).

5.2.2.3 LUN_CLASS_HAS_CMD_QUEUE_BIT

It is recommended that the Storage Driver set *LUN_CLASS_HAS_CMD_QUEUE_BIT* in the property (i.e., property = 1 << LUN_CLASS_HAS_CMD_QUEUE_BIT), which means

the Storage Driver maintains its own queue of scsi tasks/commands and its own thread to execute these scsi commands (i.e., not in the Target Module execution context).

5.2.3 Target LUN Configuration Functions

There are ways to configure an iSCSI target node, its LUNs and other iSCSI specific parameters:

1. via a set of exported kernel APIs. These APIs are covered in detailed in the next chapter.
2. via the Chelsio-defined configuration file (default to */etc/chiscsi/chiscsi.conf*).

To be able to use the second method, the Chelsio-defined configuration file, to manage the iSCSI target nodes, the Storage Driver must set the following function hooks properly:

- *fp_config_parse_option()* is invoked when parsing the “*TargetDevice*” key-value line in the configuration file.
- *fp_attach()* attaches to or opens a LUN.
- *fp_rettach()* will be called when a target node is reloaded.
- *fp_detach()* detaches or closes a LUN.
- *fp_flush()* flushes the data in the cache. It is called before a LUN is taken offline (i.e. being detached).

The above hooks do not need to be set if using the first method, via exported kernel APIs.

5.2.4 Handling of iSCSI SCSI Command

After an iSCSI session completes the Login Phase successfully, it moves into the Full Feature Phase (FFP). Once in the Full Feature Phase, an iSCSI initiator can then start to perform I/Os.

The Storage Driver should always set the following function pointers:

```
int (*fp_scsi_cmd_cdb_rcvd)      (chiscsi_scsi_command *scmd);

int (*fp_scsi_cmd_data_xfer_status) (chiscsi_scsi_command *scmd,
                                     unsigned char *sc_xfer_sreq_buf,
                                     unsigned int sc_xfer_sgcnt,
                                     unsigned int sc_xfer_offset,
                                     unsigned int sc_xfer_buflen);

int (*fp_scsi_cmd_abort)         (chiscsi_scsi_command *scmd);

void (*fp_scsi_cmd_abort_status) (unsigned int   sc_lun,
```

```

unsigned int    sc_cmdsn,
unsigned int    sc_itt,
unsigned int    sc_xfer_sgcnt,
unsigned char   *sc_xfer_sreq_buf,
void            *sc_sdev_hdl);

```

```

int (*fp_tmf_execute)
(
    unsigned long sess_tclass,
    unsigned long hndl,
    unsigned char immediate_cmd,
    unsigned char tmf_func,
    unsigned int lun,
    chiscsi_scsi_command *scmd);

```

fp_scsi_cmd_cdb_rcved() is called by the Target Module when the SCSI command CDB is received from the initiator.

fp_scsi_cmd_data_xfer_status() is used by the Target Module to pass data transfer status to the Storage Driver.

fp_scsi_cmd_abort() is used by the Target Module to abort a scsi command that has been forwarded to the Storage Driver.

When the Storage Driver initiates the request to abort a scsi command, the Target Module uses *fp_scsi_cmd_abort_status()* to inform the Storage Driver of the abort is done.

fp_tmf_execute() is called by the Target Module for the *Task Management Function Request PDUs* received.

If any private data is kept by the Storage driver for an *chiscsi_scsi_command*, the following function pointer can be set optionally:

```

void (*fp_scsi_cmd_cleanup)    (chiscsi_scsi_command *scmd);

```

fp_scsi_cmd_cleanup() would be called by the Target Module right before the *chiscsi_scsi_command* structure is released. This function can be used to clean up any private data kept by the Storage driver for the particular *chiscsi_scsi_command*.

All of the above functions are non-blocking.

In order to inform the Target Module the requested operation has been completed, the Target Module exported the following callback function APIs:

```

int chiscsi_scsi_cmd_buffer_ready (chiscsi_scsi_command *scmd,
    unsigned char *sc_xfer_sreq_buf,
    unsigned int sc_xfer_sgcnt,
    unsigned int sc_xfer_offset,

```

```
unsigned int sc_xfer_buflen);
```

```
int chiscsi_scsi_cmd_execution_status(chiscsi_scsi_command *scmd,  
                                     unsigned char *sc_xfer_sreq_buf,  
                                     unsigned int sc_xfer_sgcnt,  
                                     unsigned int sc_xfer_offset,  
                                     unsigned int sc_xfer_buflen);
```

```
int chiscsi_tmf_execution_done      (unsigned long hndl,  
                                     unsigned char tmf_response,  
                                     chiscsi_scsi_command *scmd);
```

The Storage Driver should use the following exported API to request the Target Module to abort a SCSI command:

```
int chiscsi_scsi_cmd_abort          (chiscsi_scsi_command *scmd);
```

```
int chiscsi_scsi_cmd_abort_status   (chiscsi_scsi_command *scmd);
```

All of the SCSI command related functions mentioned above will be explained in detail in Chapter 7.

6 iSCSI Target Configuration

Chelsio Target Module supports two ways of configuring an iSCSI target node:

- via the configuration file. This method uses the `ioctl` interface to pass the configuration information from user space to kernel space.
- via exported kernel APIs. This method is preferred if the configuration management is done from the kernel space.

The following sections described the exported kernel functions by the Target Module.

6.1 Target Configuration via Kernel APIs

The following sections explain the exported target configuration APIs in detail.

6.1.1 Adding an iSCSI Target Node

The Storage Driver should use `chiscsi_target_add()` to create a new iSCSI target node.

```
int chiscsi_target_add(
                        void    *sdev_priv,
                        char    *target_name,
                        char    *target_class_name,
                        char    *config_buffer,
                        int      buflen);
```

where

- `sdev_priv` points to any target node private data maintained by the Storage Driver.
- `target_name` points to the iSCSI target node name string. The string should be a globally unique name.
- `target_class_name` is the name of the iSCSI target class this target node belongs to (i.e. `iscsi_target_class.class_name`).
- `config_buffer` should contain all of the configuration parameters in the form of `<key>=<value>`, each key-value pair should be separated by a `NULL` character, similar to the `iscsi` key-value pairs contained in the login PDUs.
- `buflen` is the length of the `config_buffer`, including the ending `NULL` character.

Any key that is not presented in the `config_buffer` will take the default value if applicable.

The function `chiscsi_target_add()` is blocking. It returns zero if the target node has been added and started successfully or less than zero (`< 0`) in the case of error.

The “*PortalGroup*” includes the portal group tag and a list of associated target IP address(es) and port number(s) that service the login request. *PortalGroup* is one of the Chelsio configuration keys. Its format is described in the Appendix.

Additionally, the Storage driver can also specify the following parameters in the *config_buffer*:

- The iSCSI parameter settings.
- The Initiator Access Control List

6.1.2 Removing an iSCSI Target Node

The function *chiscsi_target_remove()* is provided to the Storage Driver to stop and remove an existing iSCSI target node.

```
int chiscsi_target_remove(          void *sdev_priv,  
                                char *target_name);
```

where

- *sdev_priv* points to the target node private data maintained by the Storage Driver.
- *target_name* points to the iSCSI target node name string.

The function *chiscsi_target_remove()* is blocking. It returns zero if the target node is stopped and removed successfully or less than zero (< 0) in the case of error.

When a target node is removed, all the sessions logged into this target will be terminated.

6.1.3 Modify an iSCSI Target Node's Settings

The function *chiscsi_target_reconfig()* could be used by Storage Driver to change the settings of an existing target node.

```
int chiscsi_target_reconfig(      void *sdev_priv,  
                                char *target_name,  
                                char *target_class_name,  
                                char *config_buffer,  
                                int  buflen);
```

where

- *sdev_priv* points to any target node private data maintained by the Storage Driver.
- *target_name* points to the iSCSI target node name string. *target_class_name* is the name of the iSCSI target class this target node belongs to (i.e. *chiscsi_target_class.class_name*).

- *config_buffer* should contain all of the configuration parameters in the form of *<key>=<value>*. Each key-value pair should be separated by a *NULL* character, similar to the iSCSI key-value pairs contained in the login PDUs.
- *buflen* is the length of the *config_buffer*, including the ending *NULL* character.

Any key that is not presented in the *config_buffer* will take the default value if applicable.

The function is blocking, the Target Module will parse the parameter buffer (pointed by *config_buffer*). If any text key format error is detected the Target Module will return a value less than zero (< 0). The functions will return 0 if no error is detected and the target is reconfigured successfully.

To make an iSCSI target node usable, at a minimum, it should have at least one target portal group with a minimum of one listening portal.

6.1.3.1 CHAP Configuration

CHAP is a protocol that is used to authenticate the peer of a connection and uses the notion of a challenge and response, (i.e., the peer is challenged to prove its identity).

The Target Module supports Challenge-Handshake Authentication Protocol (CHAP) for normal sessions.

The iSCSI key *AuthMethod* controls if an initiator needs to be authenticated or not. It defaults to “None,CHAP”.

To force CHAP authentication on all the connection to a particular target node, the target node should have its “*AuthMethod*” set to “CHAP” only (i.e., *AuthMethod=CHAP*).

Two authentication options are available if CHAP is chosen as the authentication method:

1. One-way CHAP method (also called uni-directional CHAP), and
2. Mutual CHAP method (also called bi-directional CHAP).

With one-way CHAP the target uses CHAP to authenticate the initiator. The initiator does not authenticate the target.

With mutual CHAP, the target uses CHAP to authenticate the initiator. And the initiator also uses CHAP to authenticate the target.

The Chelsio configuration key *Auth_CHAP_Policy* controls which CHAP authentication (*Oneway* or *Mutual*) needs to be performed if CHAP authentication is

needed on a connection. It defaults to oneway CHAP (i.e., *Auth_CHAP_Policy=Oneway*).

To force Mutual CHAP to be performed on all the connections to a target node, “*Auth_CHAP_Policy*” should be set to “*Mutual*” (i.e., *Auth_CHAP_Policy=Mutual*).

7 iSCSI Target Processing

This chapter describes the target interface and the exported APIs usage in the iSCSI target processing flows.

7.1 iSCSI Session

The iSCSI Protocol defines two types of sessions:

- **Normal Session:** a session in which SCSI commands, data and responses can be transferred between an iSCSI initiator and an iSCSI target.
- **Discovery Session:** a session only opened for target discovery.

To exchange SCSI commands, data, and responses, an iSCSI session must exist between the Initiator and the Target.

An iSCSI session identifies all of the connections between an initiator port and a target port. It is defined as a group of one or more TCP connections over which commands and data for a particular SCSI session are transferred.

A session may consist of one or more TCP/IP connections. Across all connections within a session, an initiator sees one and the same target. When multiple commands are striped across multiple TCP connections, the iSCSI session delivers the commands in the order in which they are received.

7.2 iSCSI Discovery Session

The Discovery session is used by the Initiators to find out available iSCSI targets at a given IP address.

After the initiator completes the login successfully, it will issue a *Text Request PDU* with the iSCSI key “*SendTarget*”.

If the function `fp_discovery_target_accessible()` (of `chiscsi_target_class`) is defined, the Target Module will first invoke the function to query the Storage Driver if the target should be included in the discovery response.

After receiving the available target list in the *Text Response PDU* from the Target, the Initiator would terminate the session.

7.3 iSCSI Normal Session

The iSCSI Normal Session is required if an Initiator wants to perform SCSI I/Os from/to an iSCSI target.

7.3.1 The Login Phase

When the Initiator is ready to connect to a target, it first establishes a TCP connection to the Target's listening portal (defined by the Chelsio Configuration key "*PortalGroup*") and then it begins the iSCSI login phase.

iSCSI login is a mechanism used to authenticate the involved parties, negotiates the session's parameters, and marks the connection as belonging to an iSCSI session. The login phase must be completed on each TCP connection before it can be used to transport SCSI commands. The login process may establish a new session, or add the connection to an existing session.

If the Target wants to reject the login, the iSCSI Target Module issues a negative Login response to the Initiator, with the Status Class/Detail (standard iSCSI status code defined in the iSCSI RFC 3720) as determined by the Target System. In this immediate reject case, the iSCSI Target will terminate the TCP connection and clean up any resources associated with this connection.

7.3.1.1 First Login Request

Byte/	0	1	2	3
/				
0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	
+	+	+	+	+
0 . I 0x03	T C . . C S G N S G	Version-max	Version-min	
+	+	+	+	+
4 TotalAHSLength	DataSegmentLength			
+	+	+	+	+
8 ISID				
+	+	+	+	+
12		TSIH		
+	+	+	+	+
16 Initiator Task Tag				
+	+	+	+	+
20 CID		Reserved		
+	+	+	+	+
24 CmdSN				
+	+	+	+	+
28 ExpStatSN or Reserved				
+	+	+	+	+
32 Reserved				
+	+	+	+	+
36 Reserved				
+	+	+	+	+
40 / Reserved				/
+	+	+	+	+
48 / DataSegment - Login Parameters in Text request Format				/
+	+	+	+	+

Figure 7-1. Login Request PDU Format

When the iSCSI Target Module receives the first Login Request PDU without C (continue) bit set from a newly accepted TCP connection, it first verifies

- the request is to establish a new normal session
- the requested target node exists
- the *chiscsi_target_class* of the requested target node has *fp_first_login_check()* set

Then the Target Module invokes *fp_first_login_check()*.

Because *fp_first_login_check()* is assumed to be non-blocking, at this point, the Target Module will halt the login processing on this connection until the Storage Driver calls the exported API *chiscsi_target_first_login_check_done()*.

If the Storage Driver calls *chiscsi_target_first_login_check_done()* with a non-zero *login_status_class*, the Target Module will reject the Initiator's login request and terminates the underlying TCP connection.

In this case the *Login Response PDU* that is constructed and sent to the Initiator would have

- “Status-Class” = *login_status_class* and
- “Status-Detail” = *login_status_detail*

Byte/ /	0	1	2	3
	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
	+	+	+	+
0	. . 0x23	T C . . CSG NSG	Version-max	Version-active
	+	+	+	+
4	TotalAHSLength	DataSegmentLength		
	+	+	+	+
8	ISID			
	+		+	+
12		TSIH		
	+	+	+	+
16	Initiator Task Tag			
	+	+	+	+
20	Reserved			
	+	+	+	+
24	StatSN			
	+	+	+	+
28	ExpCmdSN			
	+	+	+	+
32	MaxCmdSN			
	+	+	+	+
36	Status-Class	Status-Detail	Reserved	
	+	+	+	+
40	/ Reserved			/
	+/			/
	+	+	+	+
48	/ DataSegment - Login Parameters in Text request Format			/
	+/			/
	/+	+	+	+

Figure 7-1. Login Responses PDU Format

If both *login_status_class* and *login_status_detail* are zeroes, indicating the Storage Driver is ready to proceed, the Target Module will continue the login processing of the connection.

The *max_cmd* passed in from the Storage Driver via *chiscsi_target_first_login_check_done()* will be used as the maximum iSCSI command window for the session, if it is zero, the Target Module will default to maximum of 128 command per session.

7.3.1.2 Login Stage Transition

The iSCSI Login Phase could proceed in two stages indicated by the “CSG” field of the *Login Request PDU*:

- *ISCSI_LOGIN_STAGE_SECURITY*
- *ISCSI_LOGIN_STAGE_OPERATIONAL*

If the *chiscsi_target_class* of the requested target node has *fp_login_stage_check()* set, the Target Module would also invoke *fp_login_stage_check()* upon receiving the first *Login Request PDU* without C-bit (continue) set, similar to the case for *fp_first_login_check()*.

If the login phase starts from *ISCSI_LOGIN_STAGE_SECURITY* *fp_login_stage_check()* could be called again when the target agrees to transit to the next stage *ISCSI_LOGIN_STAGE_OPERATIONAL*.

Similar to *fp_first_login_check()*, *fp_login_stage_check()* is non-blocking. The Target Module will halt the login processing on this connection until the Storage Driver calls the exported API *chiscsi_target_login_stage_check_done()*.

The target will reject the initiator’s login request if the *login_status_class* from *chiscsi_target_login_stage_check_done()* is non-zero.

7.3.1.3 CHAP Processing

The Initiator and the Target may decide to authenticate each other in the security stage of the login phase.

In this stage, the target would invokes *fp_chap_info_get()* when it receives CHAP_N and CHAP_R from the initiator.

The target Module will set the *chap_info* structure as follows:

- the flag will have *CHAP_FLAG_REMOTE_NAME_VALID* set,

- the value of `CHAP_N` received will be copied into *remote_name*, the string will be terminated by a NULL character.

The Storage Driver should

- Set *CHAP_FLAG_REMOTE_SECRET_VALID* bit in the flag
- fill in the *remote_secret* and *remote_secret_length*

If the mutual CHAP information is available, it should also

- Set *CHAP_FLAG_LOCAL_NAME_VALID* and *CHAP_FLAG_LOCAL_SECRET_VALID* bits in the flag
- fill in the *remote_name* and *remote_name_length*
- fill in the *remote_secret* and *remote_secret_length*
- If the Mutual CHAP is required, set *CHAP_FLAG_MUTUAL_CHAP_REQUIRED*

fp_chap_info_get() is non-blocking. In case of error, the Target Module will reject the login and terminate the connection.

7.3.2 Session Addition

A session is created or associated with a connection when the connection moves into the login phase.

Once the leading connection of a newly created session completes the login phase successfully and moves into the full feature phase (FFP), the Target Module will call *fp_session_added()* to inform the Storage Driver of the new active session.

fp_session_added() is a blocking call.

7.3.3 Session Removal

In the normal session termination scenario, when the iSCSI Initiator decides to end the iSCSI session, it sends a *Logout Request PDU* to the target. After the target responds with a *Logout Response PDU*, the associated session will be closed and all the resources related to this session will be released.

A session could also be terminated abnormally. The Target may decide that it is unable to continue providing service to a specific Initiator or all Initiators.

An abnormal session termination can be caused by reasons such as

- Administrative action to take the target offline.
- Initiator access control violation.
- Local TCP state machine encountered an unrecoverable error such as a timeout, and the connection is the only connection of that session.

- Received PDU has header digest error or iSCSI protocol error on a connection, and the connection is the only connection of that session.

In both cases the Target Module will call *fp_session_removed()* to inform the Storage Driver of the session is being terminated.

fp_session_removed() is a blocking call.

7.3.4 Session Abort

The Storage Driver could also force the target to terminate an established session via the exported API *chiscsi_target_session_abort()*.

The handle of the session abort request is similar to abnormal session termination in the previous section.

The Storage Driver will still be called via *fp_session_removed()* when the session termination is completed

7.3.5 iSCSI SCSI Command Processing

The Target Module supports SCSI uni-directional read and write commands.

7.3.5.1 iSCSI SCSI Command

There is an *chiscsi_scsi_command* structure for every *SCSI command PDU* received from the initiator.

The fields listed below are public and can be accessed in the Storage Driver provided functions. They will contain meaningful values during the lifetime of the scsi command. Most of them are read-only except when marked as writeable in the following sections.

The Storage Driver could use “*sc_sdev_hndl*” to keep any per SCSI command information/data structures for its own private use. This pointer will not be accessed in the Target Module.

```
typedef struct chiscsi_scsi_command          chiscsi_scsi_command;

struct chiscsi_scsi_command {

    ...

    /* value returned by fp_session_added */
}
```

```

unsigned long          sc_tclass_sess_priv;

chiscsi_tcp_endpoint  *sc_ieps;
void                  *sc_sessl;    /* the iscsi session handle */
void                  *sc_conn;     /* the iscsi connection handle */

unsigned int          sc_flag;

#define SC_FLAG_READ                0x1
#define SC_FLAG_WRITE               0x2
#define SC_FLAG_SENSE                0x4
#define SC_FLAG_TMF_ABORT            0x8
#define SC_FLAG_SESS_ABORT           0x10
#define SC_FLAG_CMD_ABORT            0x20
#define SC_FLAG_XFER_OVERFLOW        0x40
#define SC_FLAG_XFER_UNDERFLOW       0x80
#define SC_FLAG_XFER_BI_OVERFLOW     0x100
#define SC_FLAG_XFER_BI_UNDERFLOW    0x200
#define SC_FLAG_IMMEDIATE_CMD        0x400

unsigned char          sc_attribute; /* ATTR field of SCSI Command PDU */
unsigned char          sc_cmdblen;   /* length of the cdb, >= 16 */
unsigned char          *sc_cmd;      /* CDB from SCSI Command PDU */

unsigned int           sc_cmdsn;      /* iscsi cmdSN */
unsigned int           sc_itt;        /* iscsi initiator task tag */
unsigned int           sc_xfer_len;   /* transfer length specified in
                                       SCSI Command PDU */

unsigned int           sc_lun;        /* lun from SCSI Command PDU*/

/* for constructing SCSI_RESPONSE response, filled by the storage */
unsigned char          sc_status;
unsigned char          sc_response;
unsigned char          sc_sense_key;
unsigned char          sc_sense_asc;
unsigned char          sc_sense_ascq;

unsigned char          sc_sense_buf[SCSI_SENSE_BUFFERSIZE];
unsigned int           sc_xfer_residualcount;

void                  *sc_sdev_hndl; /* by set the Storage Driver for its own
                                       private data if there is any */
...
};

```

When the iSCSI target module receives an SCSI Command PDU, it sets up an *chiscsi_scsi_command* structure according to the received *SCSI Command PDU*.

The format of the *SCSI Command PDU* is as follows:

Byte/	0	1	2	3
	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7

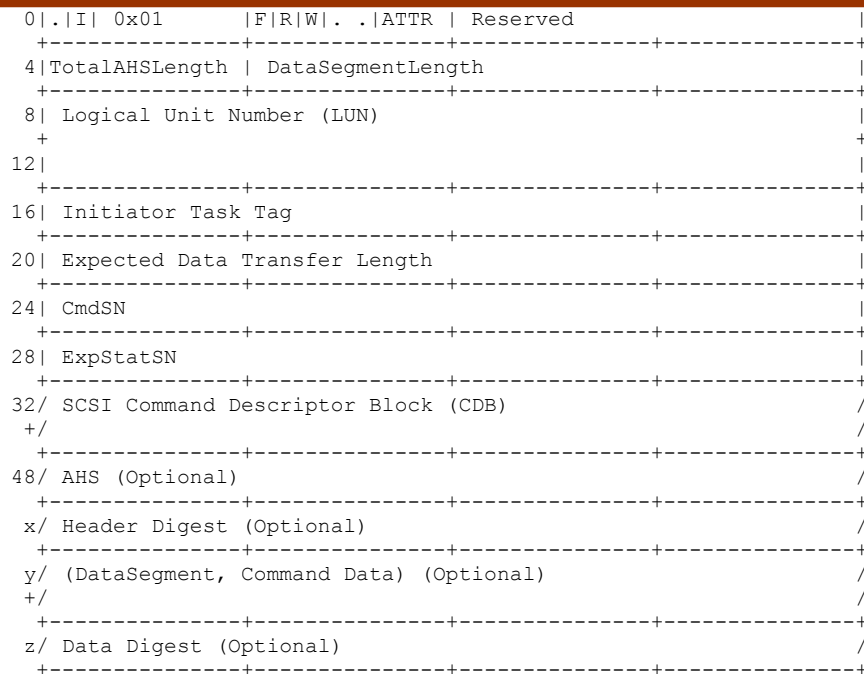


Figure 7-1. SCSI Command PDU Format

Bi-directional SCSI command (i.e. both R-bit and W-bit are set) will be rejected.

The *sc_ieps* field of *chiscsi_scsi_command* contains the tcp 4-tuple.

The *chiscsi_scsi_command* will be marked as a write command (*SC_FLAG_WRITE*) if the *SCSI Command PDU* has its w-bit set. Otherwise, it will be marked as a read command (*SC_FLAG_READ*).

The *chiscsi_scsi_command* is then set up according to the *SCSI Command PDU*:

- *sc_attribute* = PDU's "ATTR" field
- *sc_xfer_len* = PDU's "expected data transfer length" field
- *sc_lun* = PDU's "Logical Unit Number (LUN)" field
- *sc_cmd* = PDU's "SCSI Command Descriptor Block (CDB)" field
- *sc_cmdlen* = 16
- *sc_itt* = PDU's "Initiator Task Tag" field
- *sc_cmdsn* = PDU's "CmdSN" field

The possible values of *sc_attribute* are:

- #define ISCSI_TASK_ATTRIBUTES_UNTAGGED 0
- #define ISCSI_TASK_ATTRIBUTES_SIMPLE 1
- #define ISCSI_TASK_ATTRIBUTES_ORDERED 2
- #define ISCSI_TASK_ATTRIBUTES_HEAD_OF_QUEUE 3
- #define ISCSI_TASK_ATTRIBUTES_ACA 4

Once a SCSI command CDB (i.e., *SCSI Command PDU*) is received, the Target Module hands the *chiscsi_scsi_command* to the Storage Driver by calling *fp_scsi_cmd_cdb_rcved()*.

If the command is a read command the Storage Driver should queue the command up for execution. Once the command is executed, it notifies the Target Module via *chiscsi_scsi_cmd_execution_status()*. The Target Module will then send the data out and return the data transfer status to the Storage Driver via *fp_scsi_cmd_data_xfer_status()*. See section 7.3.5.2 below.

If the Lun has *LUN_CLASS_MULTI_PHASE_DATA_BIT* is set, the Storage Driver should pass the next chunk of read data to the Target Module via *chiscsi_scsi_cmd_execution_status()* once it is available. And the Target Module calls *fp_scsi_cmd_data_xfer_status()* to indicating the chunk of read data has been transmitted.

If the command is a write command the Storage Driver should allocate the data buffers needed. After the Target Module get notified that the write data buffers are available by *chiscsi_scsi_cmd_buffer_ready()* it will solicit remaining write data from the initiator by sending out Ready-To-Transfer PDUs. Once all the write data are received, the command and data are forwarded to the Storage Driver via *fp_scsi_cmd_data_xfer_status()*. See section below.

In the case of *LUN_CLASS_MULTI_PHASE_DATA_BIT* is set for the LUN, the Storage Driver should pass the next chunk of write data to the Target Module via *chiscsi_scsi_cmd_buffer_ready()* once it is available. And the Target Module calls *fp_scsi_cmd_data_xfer_status()* to indicate the chunk of write data has been received. Please note the buffers returned from the Target Module via *fp_scsi_cmd_data_xfer_status()* could be merged from one or multiple of the buffers given by the Storage Driver.

Data Buffer Setup

As mentioned above, the read or write data buffer are allocated by the Storage Driver and pointed by *sc_xfer_sreq_buf*. The *sc_xfer_sgcnt* is the total number of *chiscsi_sgvec* structures.

An *chiscsi_sgvec* structure describes the location and size of a buffer. An array of *chiscsi_sgvec* structures can be used to represent blocks of data buffers which are logically related but may not be contiguous in the memory.

```
typedef struct chiscsi_sgvec      chiscsi_sgvec;

struct chiscsi_sgvec {
    unsigned int    sg_flag;
#define ISCSI_SG_SBUF_DMABLE 0x1
```



```

void          *sg_page;      /* host page if any */
unsigned char *sg_addr;      /* start of the buffer */
unsigned long sg_dma_addr;   /* physical address */
unsigned int  sg_offset;     /* used w/ sg_page: offset into the page */
unsigned int  sg_length;     /* length of the buffer */
....
};

```

All of the fields listed above should be set up properly by the Storage Driver provided functions.

The Storage Driver should allocate *sc_xfer_sgcnt* of *chiscsi_sgvec* structures and initialize each *chiscsi_sgvec* properly. These structures together with the data buffers allocated should be persistent until the data transfer from or to the buffers are complete.

Following are the options for setting up the buffers:

- If the data buffers are host page based
 - Set *sg_page*, *sg_addr*, *sg_offset*, *sg_length*.
 - The Target Module will use Linux Kernel API *pci_map_page()* to obtain the physical address of the page if needed.
 - *sg_offset* field is relevant only when *sg_page* is set.
- If the Storage Driver manages its own memory (i.e., the memory is outside of kernel space or cannot be read or write by the Target Module), it should set *sg_dma_addr*.
 - Set *sg_dma_addr*, *sg_length* and the dma flag (i.e., *sg_flag* = *ISCSI_SG_SBUF_DMABLE*).
 - It is recommended that *sg_addr* is set if Target Module can read and write to the buffer address..
 - If *sg_addr* will not be set (i.e., *sg_addr* = NULL) then the Storage driver must turn off the unsolicited data and immediate data of the corresponding target node:

ImmediateData=No
 InitialR2T=Yes
- Following are the possible combinations for setting buffer addresses:
 - *sg_length* must be set
 - *sg_page*=NULL, *sg_addr* not set, no dmable flag: Not allowed,
 - *sg_page* set, *sg_offset* set, no dmable flag: Allowed..
 - *sg_page*=NULL, *sg_dma_addr* set, dmable flag set: Allowed
 - *sg_page* set, *sg_dma_addr* set, dmable flag set: Allowed, *sg_dma_addr* takes precedence over *sg_page*.

Following is as example for write command requesting 16K data, with dma-able memory. In this case *sc_xfer_sgcnt*=4, *sc_xfer_offset*=0 and *sc_xfer_buflen* 16K. One of the sglist entry would look like this.

- *sg_dma_addr*=<physical_addr>
- *sg_length*=4096
- *sg_addr*= <virtual_addr>
- *sg_page*=NULL
- *sg_offset*=0
- *sg_flag* |= *ISCSI_SBUG_SG_DMABLE*

Multiphase data buffers

If *LUN_CLASS_MULTI_PHASE_DATA_BIT* is set, the Storage Driver should set *sc_xfer_offset* and *sc_xfer_buflen*, in *chiscsi_scsi_cmd_buffer_ready()* and *chiscsi_scsi_cmd_execution_status()* properly:

- *sc_xfer_offset* is the offset into the total data length (i.e., *sc_xfer_len*) indicated by the command.
- *sc_xfer_buflen* is the total length of the data buffers pointed by *sc_xfer_sreq_buf*.

For example, with *LUN_CLASS_MULTI_PHASE_DATA_BIT* set, for a SCSI read command requesting 256KB of data (*sc_xfer_len*=256K), the Storage Driver returns the read data in two phases: 64KB then followed by 192KB. For the first phase of 64KB of data, *sc_xfer_offset* = 0 and *sc_xfer_buflen* = 64KB. And *sc_xfer_offset* = 64KB and *sc_xfer_buflen* = 192KB for the second phase of 192KB of data.

Requirement for DDP

For write command, the Terminator can perform Direct Data Placement (DDP) for direct placement of iSCSI PDU payload data into pre-posted final-destination host-memory buffers.

In order for this mechanism to work, the following guideline must be observed.

For total *n* number of data buffers spanning over multiple phases

- The memory must be DMA-able and/or page based
- All data buffers from 2 ~ (*n* - 1) should be of the same length and length should be either 4K, 8K, 16K, or 64K (i.e. *sg_offset* = 0 and *sg_length* are of the same value).
- The first (1) buffer can be partially used (i.e., *sg_offset* >= 0 and *sg_length* <= the length of buffers from 2 ~ (*n* - 1)).
- The last (*sc_xfer_sgcnt*) buffer can be partially used from the beginning (i.e., *sg_offset* = 0 and *sg_length* <= the length of buffers from 2 ~ (*n* - 1)).

It is preferable to not setup DDP for small IOs (< 2K), hence *sg_addr* should be provided in these cases.

Following is an example of multiphase, DDP buffer setup for a 64KB write command with 8K data buffer length(*sg_length*), with partially used first and last page.

Phase 1: *chiscsi_scsi_cmd_buffer_ready()*, offset=0 buflen 20K, sgcnt 3: Three entries in sglist, with *sg_offset*=4K,0,0 and *sg_length*=4K,8K,8K.

Phase 2: *chiscsi_scsi_cmd_buffer_ready()*, offset 20K buflen 32K, sgcnt 4: Four entries in sglist with *sg_offset*=0,0,0,0 and *sg_length*=8K,8K,8K,8K

Phase 3: *chiscsi_scsi_cmd_buffer_ready()*, offset=52K buflen 12K sgcnt 2: Two entries in sglist with *sg_offset*=0,0 and *sg_length*=8K, 4K

Storage Driver may pass phase 2 buffers to iSCSI target stack before phase 1 buffer has been returned back by the Target Module.

Both *fp_scsi_cmd_cdb_rcved()* and *fp_scsi_cmd_data_xfer_status()* are non-blocking calls, the target will halt the processing of the command until the Storage Driver notifies the Target Module.

If the Storage Driver could not execute the SCSI command or the execution results in error, it should set the following fields properly in the *chiscsi_scsi_command*:

- *sc_status*
- *sc_response*
- *sc_sense_key*
- *sc_sense_asc*
- *sc_sense_ascq*
- *sc_sense_buf*
- *sc_xfer_residualcount*

The Target Module will return the execution status to the Initiator based on the fields mentioned above.

Buffer Overflow and Underflow

The iSCSI Target module provides two flags to handle buffer overflow and buffer underflow, namely *SC_FLAG_XFER_OVERFLOW* and *SC_FLAG_XFER_UNDERFLOW*. The phase in which the flags are set will be assumed to be the last phase which means that Storage Driver should not send any additional buffers after this.

For buffer overflow, the Storage Driver needs to:

1. Set the *SC_FLAG_XFER_OVERFLOW* flag and
2. Set the residual count in *sc->sc_xfer_residualcount*

The Residual Count indicates the number of bytes that were not transferred because the initiator's Expected Data Transfer Length was not sufficient. iSCSI Target Module will transfer only Expected Data Transfer Length of bytes but send *fp_scsi_cmd_data_xfer_status()* for all of the buffers that were provided.

For buffer underflow, The Storage Driver needs to:

1. Set the flag `SC_FLAG_XFER_UNDERFLOW`
2. Set residual count in `sc->sc_xfer_residualcount`
3. Set new `sc_xfer_len` = `bufferlen`

The Residual Count indicates the number of bytes that were not transferred out of the number of bytes that were expected to be transferred.

Additionally the Storage driver should set the appropriate `sc->sc_response` and `sc->sc_status` in `chiscsi_scsi_command` structure.

7.3.5.2 iSCSI SCSI Command Read Operation

For SCSI read request, the Target Module will transfer the data from the memory allocated by the Storage Driver to the Initiator.

The diagram below describes the sequence of actions of a SCSI read operation:

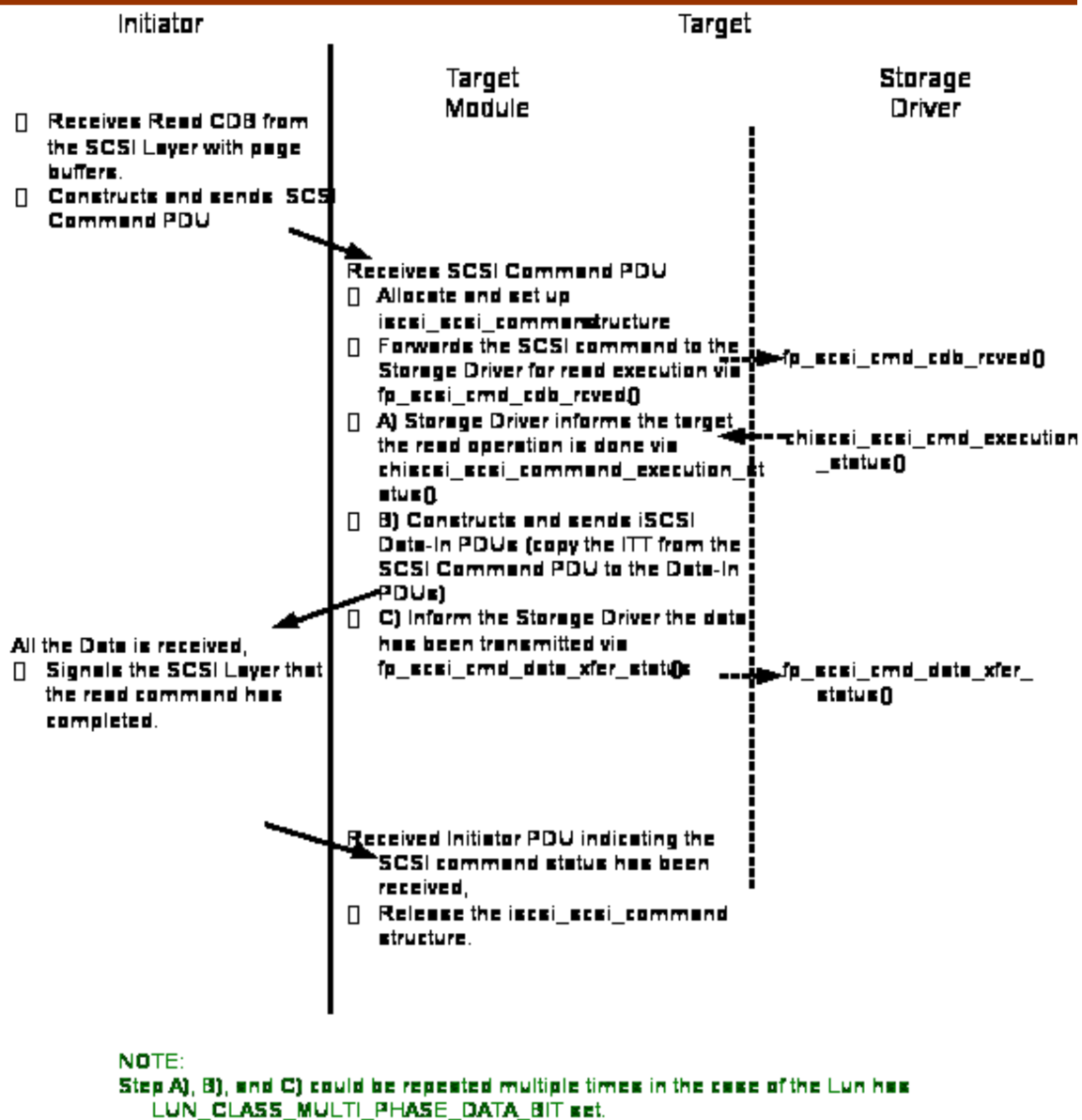
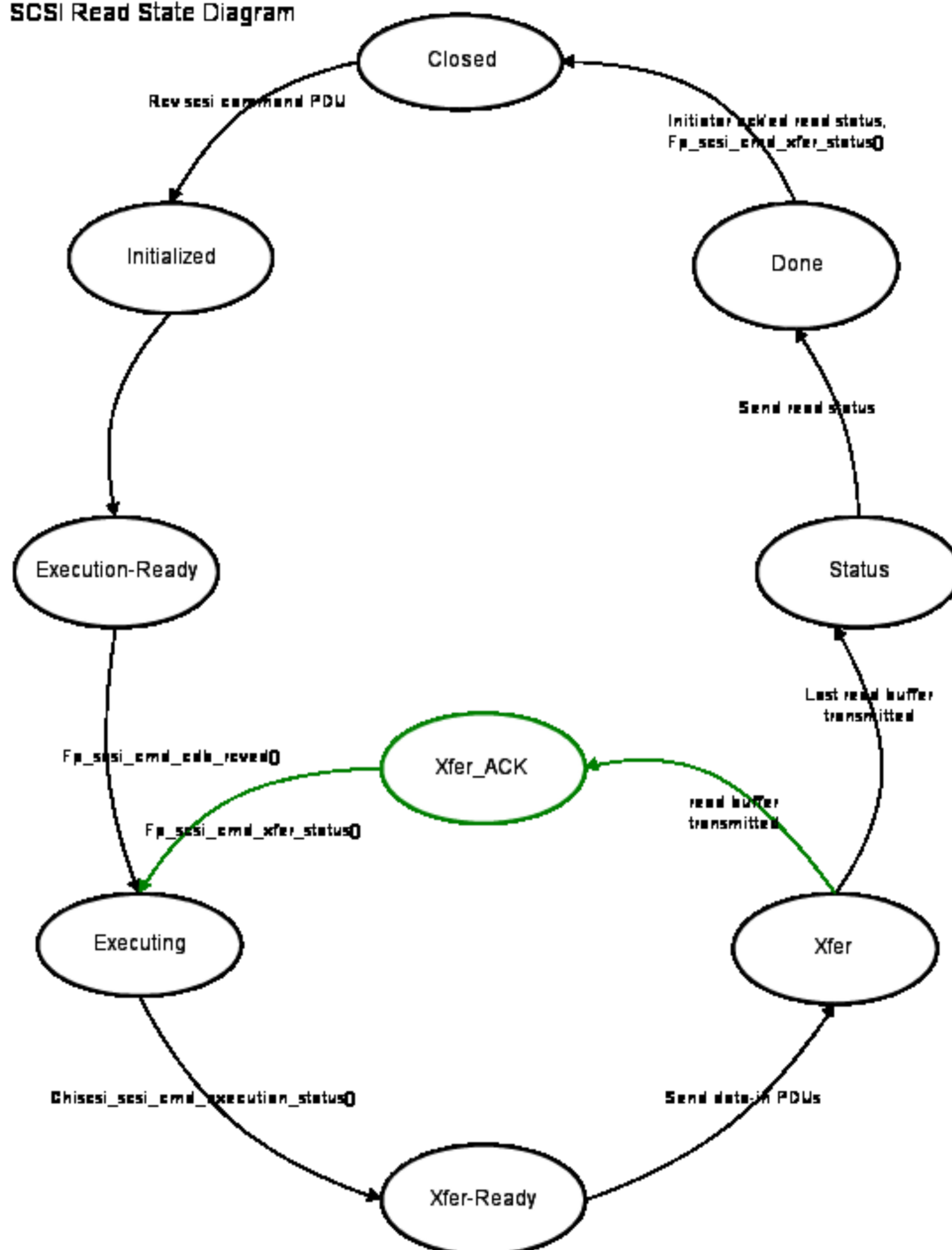
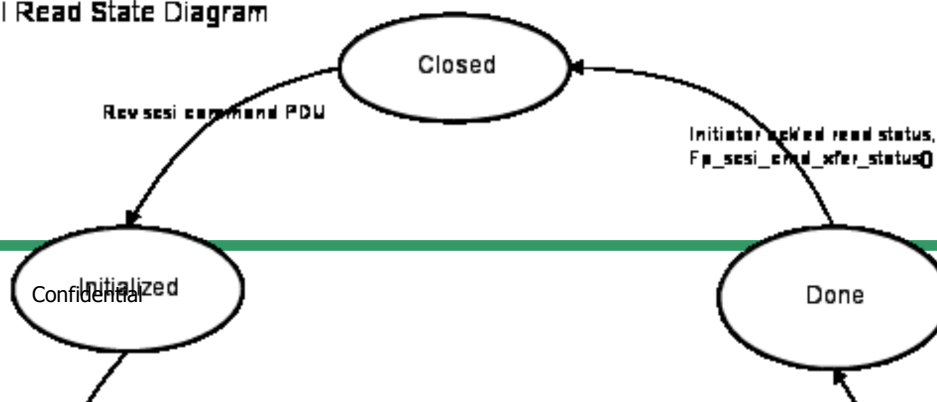


Figure 7-1. iSCSI SCSI Read Walkthrough

SCSI Read State Diagram



SCSI Read State Diagram



SCSI Read State Diagram

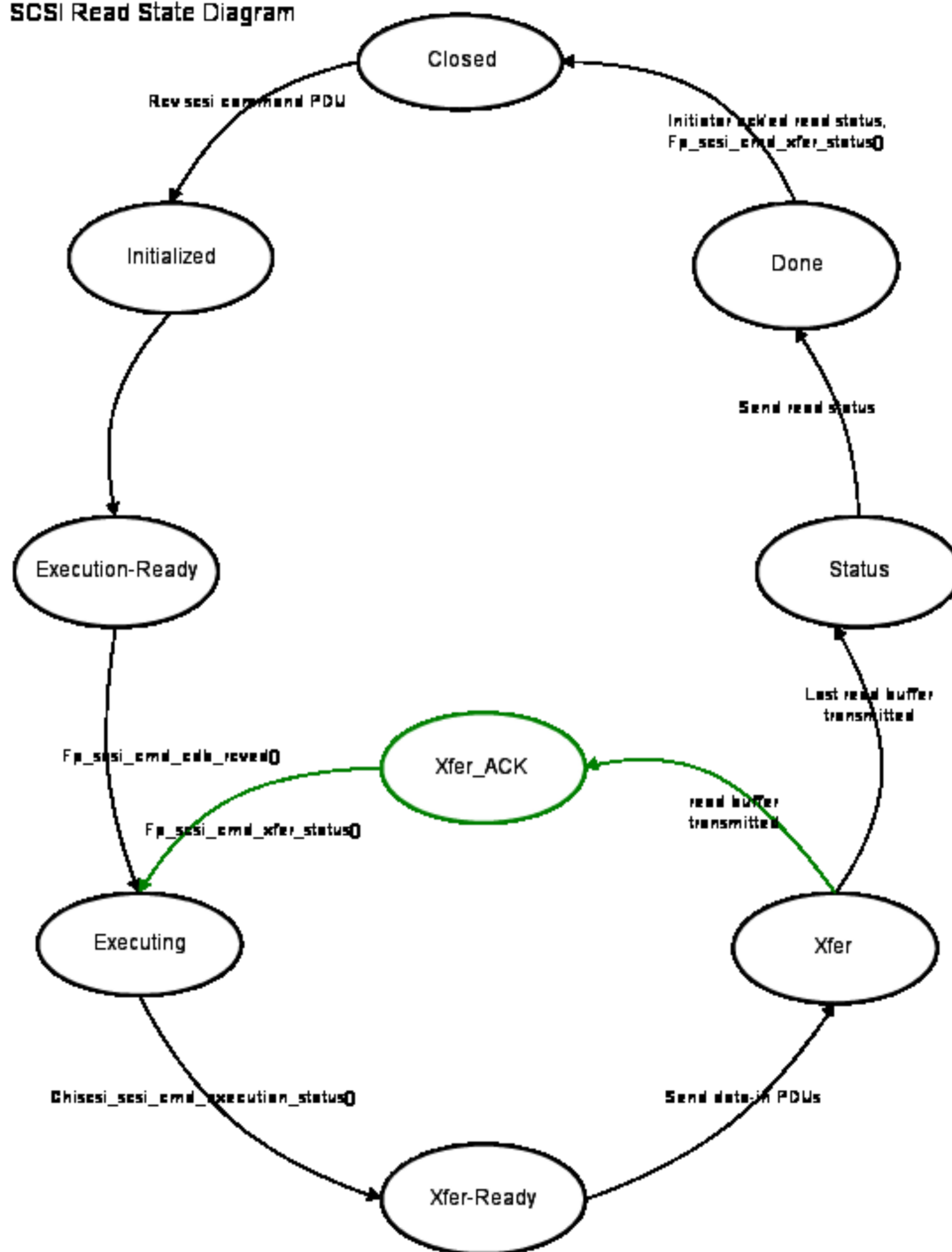


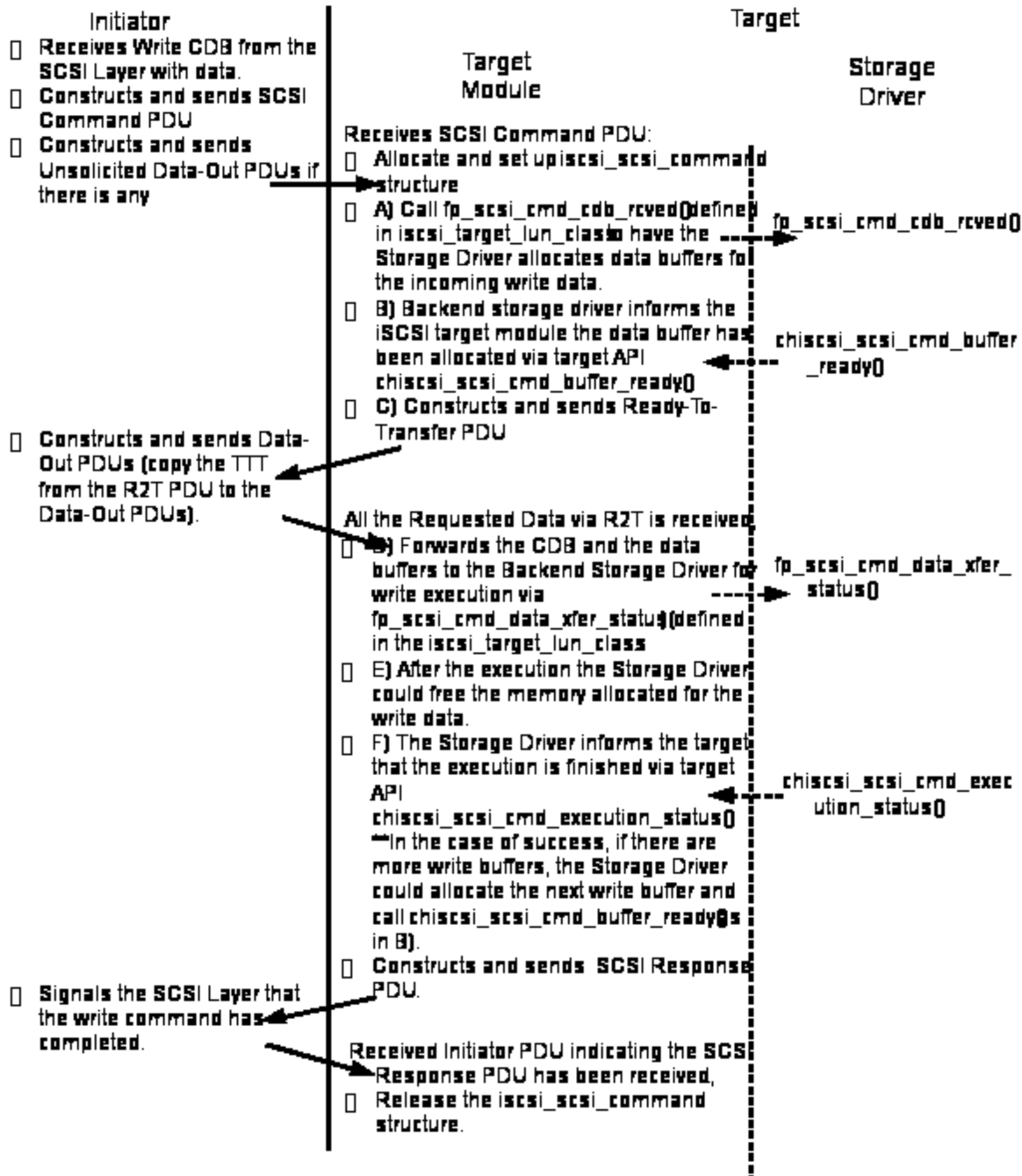
Figure 7-1. iSCSI SCSI Read State Diagram

7.3.5.3 iSCSI SCSI Command Write Operation

For SCSI write request, the Target Module will first call *fp_scsi_cmd_cdb_rcved()* to have the Storage Driver allocate the write data buffers. It will copy or DMA the data received from the initiator to the memory allocated by the Storage Driver. The SCSI command and data then is handed back to the Storage Driver for write execution via *fp_scsi_cmd_data_xfer_status()*.

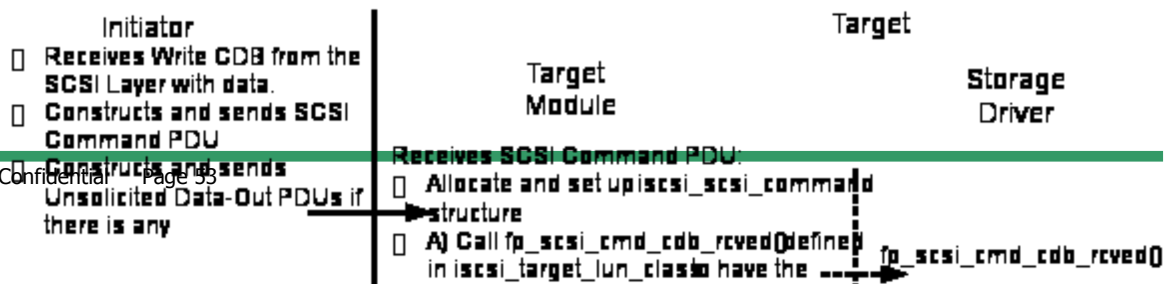
The diagram below describes the action sequence of a SCSI write operation:

SCSI Write Walk-Through

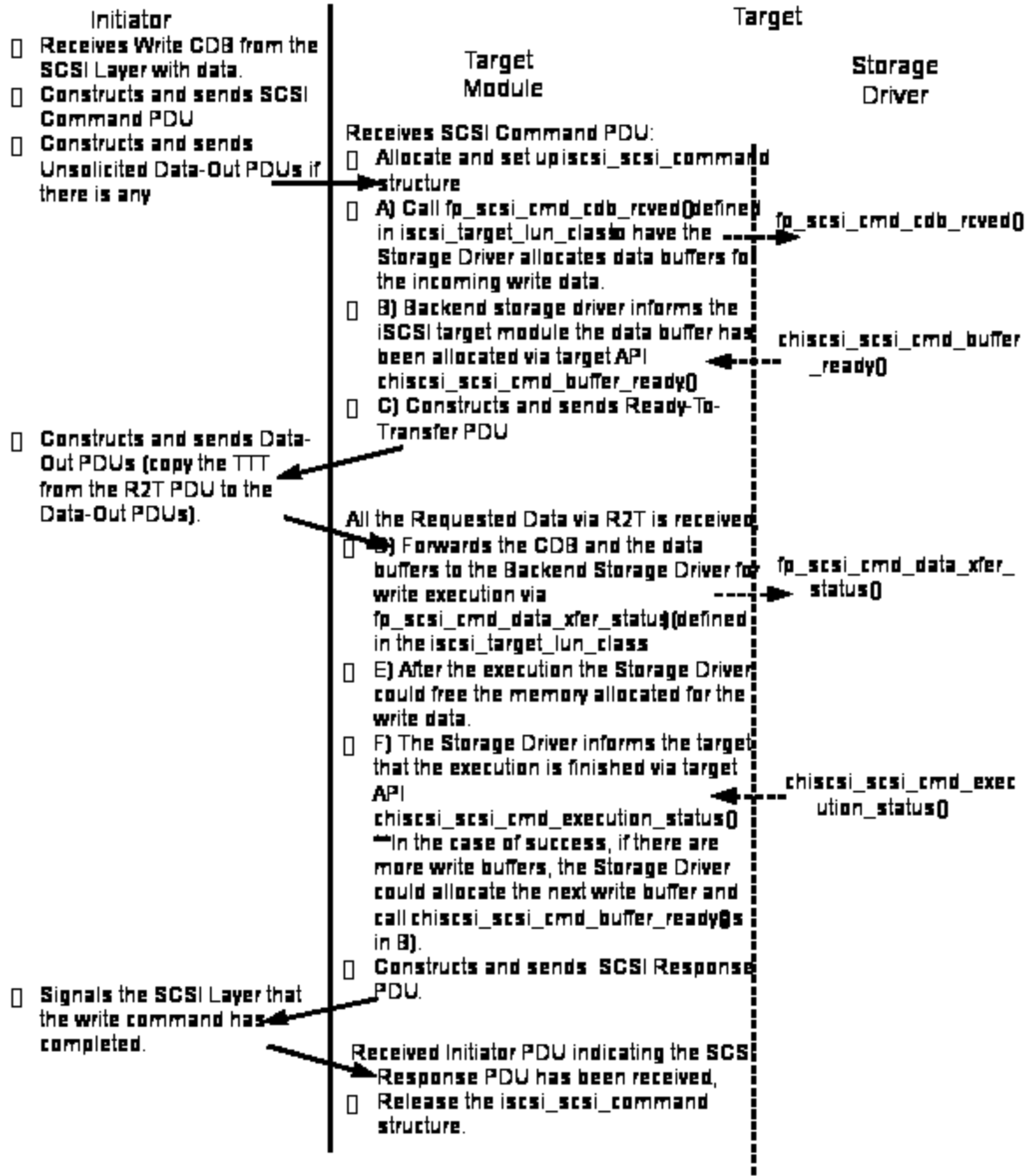


NOTE: In the case of the Lun has `LUN_CLASS_MULTI_PHASE_DATA_BIT` set, Step A), B), C), D), E) and F) could be repeated multiple times (the long path), or Step B), C), D), and E) could be repeated multiple times (the short path).

SCSI Write Walk-Through



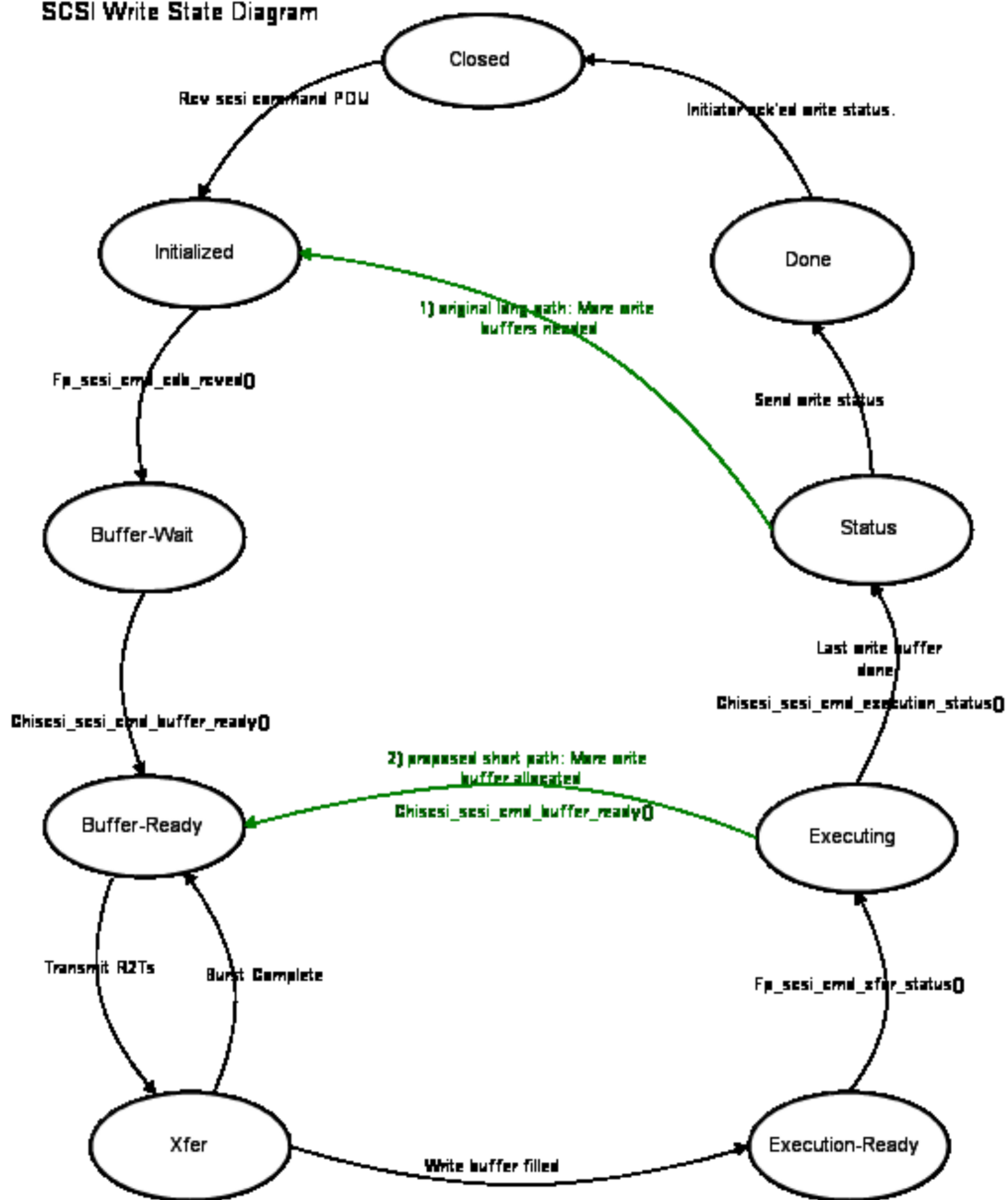
SCSI Write Walk-Through



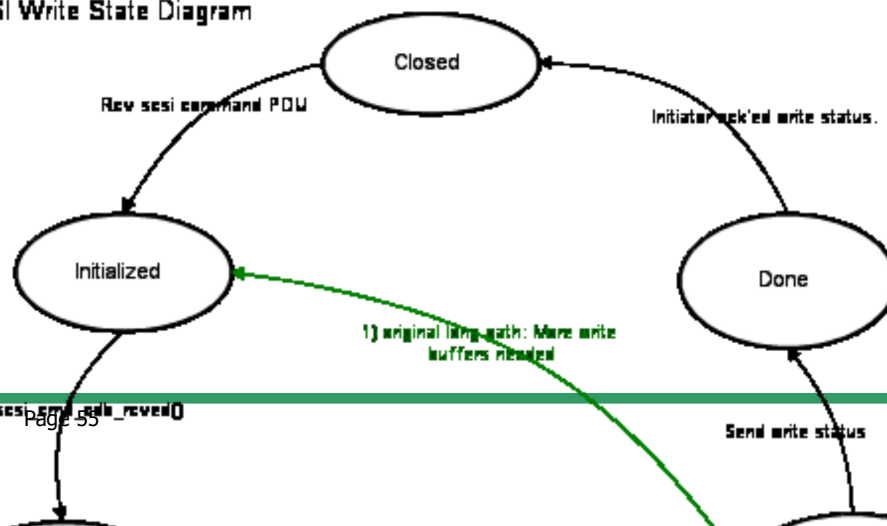
NOTE: In the case of the Lun has `LUN_CLASS_MULTI_PHASE_DATA_BIT` set, Step A), B), C), D), E) and F) could be repeated multiple times (the long path), or Step B), C), D), and E) could be repeated multiple times (the short path).

Figure 7-1. iSCSI SCSI Write Walkthrough

SCSI Write State Diagram



SCSI Write State Diagram



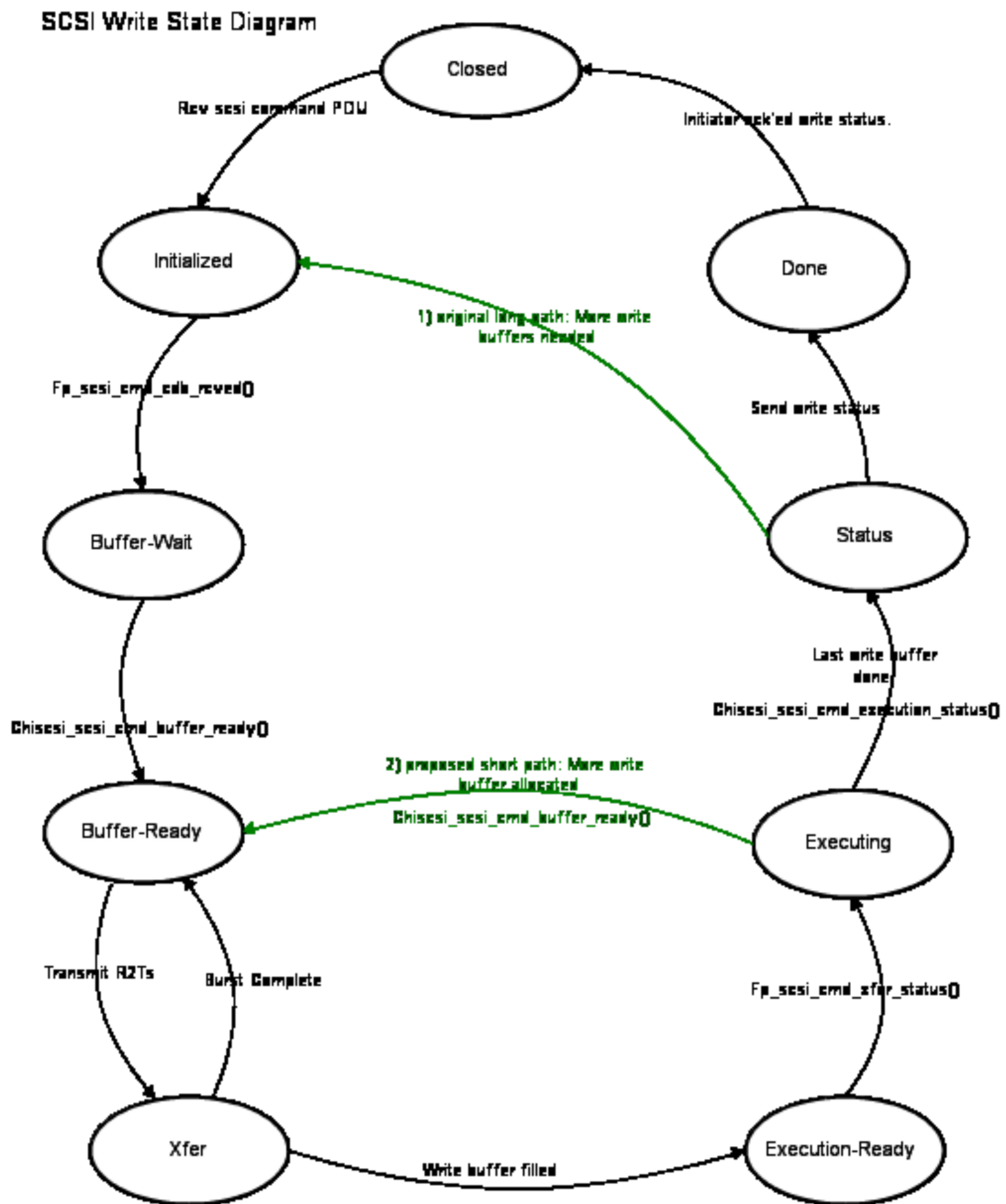


Figure 7-1. iSCSI SCSI Write State Diagram

7.3.5.4 Abort a SCSI Command

When a target session has to be terminated due to either Initiator Requests or iSCSI protocol errors, the Target Module will issue *fp_scsi_cmd_abort()* for every outstanding SCSI command that has been forwarded to the Storage Driver (either via *fp_scsi_cmd_cdb_rcvd()* or *fp_scsi_cmd_data_xfer_status()*).

Each of the *chiscsi_scsi_command* will have *SC_FLAG_SESS_ABORT* set.

After cleaning up the command resources the Storage Driver should notify the Target Module by invoking *chiscsi_scsi_cmd_abort_status()*. The Storage Driver should set up the *chiscsi_scsi_command* similar to the case where the execution results in error.

fp_scsi_cmd_abort() is non-blocking.

If the Storage Driver wants to abort a SCSI command that it has already executed or allocated memory for, it should call the Target Module's exported API function *chiscsi_scsi_cmd_abort()*. The storage driver should set up the *chiscsi_scsi_command* similar to the case where the execution results in error.

chiscsi_scsi_cmd_abort() is non-blocking.

After the Target Module aborts the command it will inform the Storage Driver via *fp_scsi_cmd_abort_status()*.

7.3.5.5 Debugging SCSI Command problem

The Target Module provides a function to dump out an *chiscsi_scsi_command* information:

```
void          chiscsi_iscsi_command_dump (chiscsi_scsi_command *scmd)
```

chiscsi_iscsi_command_dump() will print out the *chiscsi_scsi_command* structure information into the syslog.

7.3.6 iSCSI Task Management Functions

When the Target Module receives a *Task Management Function Request PDU*, it may invoke *fp_tmf_execute()* defined in *iscsi_target_lun_class*.

The possible Task Management Functions (TMF) include

- #define ISCSI_TMF_FUNCTION_ABORT_TASK 1
- #define ISCSI_TMF_FUNCTION_ABORT_TASK_SET 2
- #define ISCSI_TMF_FUNCTION_CLEAR_ACA 3
- #define ISCSI_TMF_FUNCTION_CLEAR_TASK_SET 4
- #define ISCSI_TMF_FUNCTION_LOGICAL_UNIT_RESET 5
- #define ISCSI_TMF_FUNCTION_TARGET_WARM_RESET 6
- #define ISCSI_TMF_FUNCTION_TARGET_COLD_RESET 7
- #define ISCSI_TMF_FUNCTION_TASK_REASSIGN 8

The target will handle the TMF locally under the following conditions (i.e., the Storage Driver will not be involved):

- Function is *ISCSI_TMF_FUNCTION_TASK_REASSIGN*, or
- Function is *ISCSI_TMF_FUNCTION_CLEAR_ACA*, or
- Function is *ISCSI_TMF_FUNCTION_ABORT_TASK* and
 - a. the target could not find the requested SCSI command, or
 - b. the SCSI command has already completed execution (i.e. *chiscsi_scsi_cmd_execution_status()* has already been called for this command), or
 - c. the SCSI command has not been sent to the Storage Driver for execution.

In the rest of the scenarios the target will forward the TMF request to the Storage Driver via *fp_tmf_execute()*.

```
int (*fp_tmf_execute)          (unsigned long sess_tclass,
                                unsigned long  hndl,
                                unsigned char  immediate_cmd,
                                unsigned char  tmf_func,
                                unsigned int   lun,
                                chiscsi_scsi_command *scmd);
```

where

- *sess_tclass* is the private session hndl returned by the Storage Driver from *fp_session_added()*.
- *lun* is valid when the function is *ISCSI_TMF_FUNCTION_ABORT_TASK*, or *ISCSI_TMF_FUNCTION_CLEAR_TASK_SET*, or *ISCSI_TMF_FUNCTION_ABORT_TASK_SET*, or *ISCSI_TMF_FUNCTION_LOGIN_UNIT_RESET*.
- *Scmd* will point to a valid *chiscsi_scsi_command* structure when the function is *ISCSI_TMF_FUNCTION_ABORT_TASK*.

In the case where the scsi command is in the data transfer stage (i.e., *chiscsi_scsi_cmd_buffer_ready()* has been called for the command by the Storage Driver), the Target Module will

- set *chiscsi_scsi_command.sc_flag = SC_FLAG_TMF_ABORT* , and
- invoke *fp_scsi_cmd_abort()*.

For each scsi command being aborted, after cleaning up the command resources the Storage Driver should notify the Target Module by invoke *chiscsi_scsi_cmd_execution_status()*. The Storage Driver should setup the *chiscsi_scsi_command* similar to the cases where the execution results in error.

fp_tmf_execute() is non-blocking, the Target Module will not respond to the initiator with *Task Management Function Response PDU* until the Storage Driver notify the Target Module of the TMF response code via *chiscsi_tmf_execution_done()*.

The possible Task Management Function (TMF) response code include

- #define ISCSI_RESPONSE_TMF_COMPLETE 0
- #define ISCSI_RESPONSE_TMF_INVALID_TASK 1
- #define ISCSI_RESPONSE_TMF_INVALID_LUN 2
- #define ISCSI_RESPONSE_TMF_TASK_STILL_ALLEGIANT 3
- #define ISCSI_RESPONSE_TMF_NO_TASK_FAILOVER 4
- #define ISCSI_RESPONSE_TMF_NOT_SUPPORTED 5
- #define ISCSI_RESPONSE_TMF_AUTH_FAILED 6
- #define ISCSI_RESPONSE_TMF_FUNCTION_REJECTED 0xff

8 Target Information and Statistics

To allow Storage Driver to retrieve target settings and statistical data, the following structures are defined:

```
struct chiscsi_perf_info {
    unsigned long read_bytes;
    unsigned long write_bytes;
    unsigned long read_cmd_cnt;
    unsigned long write_cmd_cnt;
};

struct iscsi_session_settings {
    unsigned char initial_r2t:1;
    unsigned char immediate_data:1;
    unsigned char erl:2;
    unsigned char data_pdu_in_order:1;
    unsigned char data_sequence_in_order:1;
    unsigned char filler:2;

    unsigned int max_conns;
    unsigned int max_r2t;
    unsigned int first_burst;
    unsigned int max_burst;
    unsigned int time2wait;
    unsigned int time2retain;
};

struct iscsi_conn_settings {
    unsigned char header_digest[2];
    unsigned char data_digest[2];

    unsigned int portal_group_tag;
    unsigned int max_rcv_data_segment;
    unsigned int max_xmit_data_segment;
};

struct iscsi_chap_settings {
    unsigned char chap_en:1;
    unsigned char chap_required:1;
    unsigned char mutual_chap_forced:1;
    unsigned char filler:5;

    unsigned int challenge_length;
};

struct iscsi_target_config_settings {
    unsigned char acl_en:1;
    unsigned char isns_register:1;
    unsigned char shadow_mode:1;
    unsigned char filler:5;

    unsigned int sess_max_cmds;
```



```
};

struct chiscsi_target_info {
    char name[256];
    char alias[256];

    struct iscsi_session_settings sess_keys;
    struct iscsi_conn_settings conn_keys;
    struct iscsi_chap_settings chap;
    struct iscsi_target_config_settings config_keys;

    unsigned char auth_order;

    /* private to Chelsio Stack */
    unsigned long hndl;
};

struct chiscsi_session_info {
    char peer_name[256];
    char peer_alias[256];

    struct iscsi_session_settings sess_keys;

    unsigned char type;
    unsigned char isid[6];
    unsigned char conn_cnt;
    unsigned short tsih;

    unsigned int cmdsn;
    unsigned int maxcmdsn;
    unsigned int expcmdsn;

    struct chiscsi_perf_info perf;
    /* private to Chelsio Stack */
    unsigned long hndl;
};

struct chiscsi_connection_info {
    struct chiscsi_tcp_endpoint tcp_endpoints;

    struct iscsi_conn_settings conn_keys;

    unsigned char offloaded:1;
    unsigned char filler:7;

    unsigned int cid;
    unsigned int statsn;
    unsigned int expstatsn;

    /* private to Chelsio Stack */
    unsigned long hndl;
};
```

8.1 Target Information

The Storage driver could get detail information about the target using the exported API *chiscsi_target_get_information()*.

```
int          chiscsi_get_target_info      (char *tname,
                                          struct chiscsi_target_info *target_info);
```

Where,

- *tname* – is the target_name provided by the storage driver.
- *target_info* – is the output structure which will return the details information on the target specified by target_info structure.

8.2 Session Information

The Storage driver could get information about established session parameters using the exported API *chiscsi_target_get_session()*. The Storage Driver can issue this API call anytime it wants to collect the detail information about the established sessions.

```
int          chiscsi_get_session_info    (char *tname,
                                          char *iname,
                                          int sess_info_max,
                                          struct chiscsi_session_info *sess_info);
```

Where,

- *tname* – is the target_name provided by the storage driver.
- *iname* – is the initiator name provided by the storage driver.
- *sess_info_max* – the number of *chiscsi_session_info* structures pointed by *sess_info*.
- *sess_info* – is the pointer to *chiscsi_session_info* structure. This structure will be returned as output once the data has been filled in by chiscsi target stack.

When there are less than *sess_info_max* active sessions between a given initiator and a target, *chiscsi_target_get_session()* will return the number of sessions filled in. Otherwise, it will return the total numbers of active sessions. The *sess_info* would contains the information of the first *sess_info_max* sessions.

8.3 Connection Information

The Storage Driver can retrieve connection information by calling exported API *chiscsi_get_connection_info()*.

```
int          chiscsi_get_connection_info (unsigned long sess_hdl,
                                          int conn_idx,
                                          struct chiscsi_connection_info *conninfo)
```

The connection information API should always be called after session API so that `sess_hdl` and `conn_cnt` parameter can be used to retrieve connection details.

8.4 Portal Performance Data

The Storage Driver can retrieve portal performance data by calling exported API `chiscsi_get_perf_info()`.

```
int chiscsi_get_perf_info      (int ipv6,
                               unsigned int *ip_addr,
                               unsigned int port,
                               struct chiscsi_perf_info *pdata)
```

Where,

- *ipv6* – whether **ip_addr* is ipv6 address or not.
- *address* – ip address for the target portal for which the performance statistics are to be collected.
- *port* – port number of the target portal. If port number is not provided Chelsio target stack will use default (3260) port number.
- *pdata* – pointer to the structure `chiscsi_perf_data` which will be filled in by Chelsio iSCSI stack

A new structure to store performance statistics is created for storage driver to pass pointer to when calling the exported API.

```
struct chiscsi_perf_info {
    unsigned long          read_bytes;
    unsigned long          write_bytes;
    unsigned long          read_cmd_cnt;
    unsigned long          write_cmd_cnt;
};
```

Where,

- *read_bytes* – number of bytes read by the target portal.
- *write_bytes* – number of bytes written by the target portal.
- *read_cmd_cnt* – number of read command executed by the target portal.
- *write_cmd_cnt* – number of write commands executed by the target portal.

9 Chelsio iSCSI Target API Files

All the data structure definitions and iSCSI Target API function declarations described in the previous chapters are under `includes/common/`.

Name	Description
<code>iscsi_target_class.h</code>	<i>iscsi_target_class</i> and <i>iscsi_target_lun_class</i> data structures and related APIs.
<code>chiscsi_sgvec.h</code>	scatter gather vector representation <i>chiscsi_sgvec</i> .
<code>chiscsi_scsi_command.h</code>	Describes an iSCSI SCSI Command request and associated data <i>chiscsi_scsi_command</i> .
<code>iscsi_chap.h</code>	<i>chap_info</i> representation
<code>iscsi_info.h</code>	Target, session, connection settings and performance counters

Table 9-1. Target API Header Files

Appendix A. Chelsio iSCSI Software Features

This table below summarizes the major capabilities supported by Chelsio iSCSI Software:

Feature	Y/N	Comment
Multi-Path I/O	Y	
Discovery	Y	
“iqn.” Naming	Y	
“eui.” Naming	Y	
Login	Y	
Text Key List Negotiations	Y	
Text Key Simple Value Negotiations	Y	
Security Negotiation	Y	
Authentication Method	Y	
Connection Reinstatement	N	
Session Reinstatement	Y	
Session Continuation and Failure	N	
Logout	Y	
CHAP	Y	One way and Mutual
Access Control List	Y	IP address and/or Initiator Name based
Target LUN Masking	Y	
Error Recovery level	Y	Level 0 only
Reject PDU usage	Y	
Connection Timeout	Y	
Immediate Data	Y	
Unsolicited SCSI Data Out	Y	
Outstanding R2T	Y	Support multiple outstanding R2Ts
Task Mgmt Function	Y	
Asynchronous Message	Y	
Text Request/Response	Y	
Login Request/Response	Y	
Logout Request/Response	Y	
SNACK request	N	
NOP-Out / NOP-In	Y	
iSNS Client	Y	

Table I. iSCSI Software Features

Appendix B. iSCSI Text Key Settings

iSCSI Text Entity Parameters pass iSCSI protocol control information to the Chelsio iSCSI module. This information is unique for each target node.

The parameters follow the IETF iSCSI standard RFC 3720 in both definition and syntax. The descriptions below are mostly from this RFC.

Please refer to the Chelsio iSCSI User Guide for detailed description of supported values.

Appendix C. Chelsio Target Configuration Keys

Chelsio Target Configuration Entity Parameters pass control information to the Chelsio iSCSI module. The parameters are specific to Chelsio's implementation of the iSCSI Target and are unique to each target.

The parameters consist of information that can be put into the following categories:

1. Challenge Handshake Authentication Protocol (CHAP).
2. Target specific settings.

All of the following parameters can have multiple instances in one target entity block (i.e., they can be declared multiple times for one particular target).

- Portal Group
- Storage Device
- Access Control List (ACL)

Please refer to the User Guide for detailed description of supported values.

Appendix D. Log and Debug messages.

Chelsio iSCSI Target driver will perform sanity check on the buffer settings and will print following error messages to dmesg.

Buffer Setup Errors

Error Message	Reason
<i>"chiscsi_scsi_cmd_buffer_ready: itt 0x5a, data buf NULL, 0+4096/8192"</i>	Buffer pointer passed in chiscsi_scsi_cmd_buffer_ready is NULL
<i>"chiscsi_scsi_cmd_buffer_ready: itt 0x5a, sg 0, exp. sgl, 0+4096/8192"</i>	sgcnt passed in chiscsi_scsi_cmd_buffer_ready is zero
<i>"chiscsi_scsi_cmd_execution_status: itt 0x5a, buf overflow, 4096!=0+8192."</i>	Received more buffer (0+8192) than ExpectedDataTransferLen (4096) while SC_FLAG_XFER_OVERFLOW flag was not set
<i>"chiscsi_scsi_cmd_execution_status: itt 0x5a, buf underflow, 16384!=0+8192"</i>	Received more buffer(0+8192) than ExpectedDataTransferLen (16384) while SC_FLAG_XFER_UNDERFLOW flag 'and' MULTIPHASE bit not set
<i>"chiscsi_scsi_cmd_buffer_ready: itt 0x5a, buf off 4096+4096, exp 0+4096"</i>	Buffers are out of order. offset + buflen (4096+4096) sent by storage driver is different than what Chelsio iSCSI stack expects (0+4096)

Other Error Messages

Error Message	Reason
<i>chiscsi: sc 0xffff880231c88000, itt 0x69000000, held by storage driver</i>	iSCSI stack is waiting for callback and cannot proceed until reply comes from controller. TMF is issued from initiator in this case
<i>chiscsi:No DDP: sgcnt 2 sgl 0xffff88022d79b0e0, sg_flag 0x0, sg_addr NULL, Cannot copy data!</i>	If sg_addr is not set then command will fail.
<i>chiscsi:2/3, len 3072 xferoff 8192 NOT full pages</i>	One of the middle phase has an incomplete page, which means buffer not suitable for DDP. In this example, in one of the middle phases (indicated with the xferoff of 8192) 2 nd sglst entry out of 3, has sg_length less than page size

Sending debug messages to Chelsio

To trace the flow of the complete `chisci_scsi_command` in Chelsio iSCSI stack, turn on the debug messages by executing following command:

```
iscsictl -G iscsi_verbose_level=0xf,0xc82000
```

Execute a read/write command from initiator after this and send the log to Chelsio for debugging purpose.

Appendix E. iSCSI Acronyms

ACL	Access Control List
CHAP	Challenge-Handshake Authentication Protocol
CPL	Chelsio Protocol Language, the Terminator's programming interface
CRC	Cyclic Redundancy Check
DDP	Direct Data Placement
ERL	Error Recovery Level (0, 1, 2)
EUI	Extended Unique Identifier
FFP	Full Feature Phase
HBA	Host Bus Adaptor
H/W	Hardware
iSCSI	Internet Small Computer System Interface
iSNS	Internet Storage Name Service
ISID	Initiator Session ID
LU	Logical Unit
LVM	Logical Volume Management
MPIO	Multi-Path I/O
MCS	Multiple Connections per Session
NIC	Network Interface Card
N/A	Not applicable
NOP	No Operation
PDU	Protocol Data Unit
R2T	Request To Transmit
SLP	Service Location Protocol
SNACK	Selected Negative Acknowledgement
SNMP	Simple Network Management Protocol
SSID	Session ID
S/W	Software
TBD	To Be Determined
TCP	Transmission Control Protocol
TCB	TCP Control Block
Terminator	Chelsio's ASIC for protocol offload
TOE	TCP/IP Offload Engine
ULP	Upper Layer Protocol