StorageTek®

4400 Automated Cartridge System

UNIX® Storage Server

Programmer's Guide

PN 9036

PRELIMINARY

Information contained in this publication is subject to change. In the event of changes, the publication will be revised. Comments concerning the contents of this manual should be directed to Technical Publications at the following address:

Storage Technology Corporation,
Technical Publications, 2270 South 88th Street, Louisville, CO 80028-2286.

DOCUMENTATION SURVEY

1. Rate the following documentation factors:

	Excellent	Good	Fair	Poor
Table of Contents	[ ]	[ ]	[ ]	[ ]
Written material	[ ]	[ ]	[ ]	[ ]
Organization	[ ]	[ ]	[ ]	[ ]
Illustrations	[ ]	[ ]	[ ]	[ ]
Index	[ ]	[ ]	[ ]	[ ]

Suggestions for improvement: _____

_____

_____

_____

_____

2. Does this manual contain the necessary information and procedures for using this product?

If No, please explain: _____

_____

_____

_____

_____

3. How important is this manual as an aid in performing your job?

[ ] Very Important [ ] Important [ ] Somewhat Important [ ] Not Important

4. How often do you use this manual?

[ ] Daily [ ] Weekly [ ] Monthly [ ] As needed [ ] Never

5. When have you consulted this manual? You may select more than one response.

[ ] For a product overview
[ ] For command syntax reference
[ ] For installation planning
[ ] For maintenance procedures
[ ] For installation instructions
[ ] To answer a specific question
[ ] For operating instructions
[ ] During a training course

Other (please explain): _____

_____

_____

6. Do you know where your manual is right now? Yes [ ] No [ ]

If No, please explain: _____

_____

_____

7. What is your position/title? _____

TABLE OF CONTENTS

Section **Page**

(INTENTIONALLY LEFT BLANK)

LIST OF ILLUSTRATIONS

(INTENTIONALLY LEFT BLANK)

LIST OF TABLES

(INTENTIONALLY LEFT BLANK)

PREFACE

PURPOSE

This manual provides the following information:

- Functional description of the StorageTek 4400 Automated Cartridge System and the UNIX Storage Server
- Functional descriptions of the Storage Server software components and the required Server System Interface (SSI)
- Format and contents of data structures used by the Storage Server, an SSI, and a client application

AUDIENCE

The *UNIX Storage Server Programmer's Guide* is written for the designer and programmer of a Server System Interface (SSI) and a client application. The reader must have a working knowledge of the following:

- UNIX Operating System
- "C" programming language

USING THIS MANUAL

This manual is organized as follows:

- *Chapter 1: ACS Overview.* Describes the 4400 ACS hardware components and the UNIX Storage Server software components.
- *Chapter 2: ACSLM Processes.* Describes the primary functions of the ACS Library Manager (ACSLM) software component.
- *Chapter 3: ACSLM Common Data Structures.* Describes the format and content of ACSLM request and response messages. Defines common status codes and common variables.
- *Chapter 4: ACSLM Command Structures.* Describes the specific format and content of each ACSLM command.
- *Chapter 5: CSI Processes.* Describes the primary functions of the Client System Interface (CSI).

- *Chapter 6: SSI Requirements.* Describes specific functional requirements of the Storage Server Interface (SSI) and instructions for programming.
- *Chapter 7: CSI Data Structures.* Describes the format and content of data structures used by the CSI and the SSI. Identifies common environment variables and external software libraries.
- *Appendix A: Event Log Messages.* Identifies and describes messages written to the Event Log by all Storage Server software components.
- *Appendix B: ACSLM Header Files.* Includes listings of system-wide header files required for ACSLM processing.
- *Appendix C: XDR Translation Functions.* Includes listings of the StorageTek-supplied XDR translation functions.
- *Appendix D: CSI and SSI Required Files.* Includes listings of files used by the CSI and the SSI.

CONVENTIONS

The following conventions are used throughout this manual for library and UNIX commands.

command	Literal user entries are shown in **Courier bold** type.
variable_entry	Variable entries (text that may vary each time it is entered) are shown in ***bold italic*** type. Do not enter the actual characters shown.
message	System messages are shown in Courier plain type.
variable_msg	Variable messages (text that may vary each time the message is displayed) are shown in *italic* type.
...	The text immediately preceding the ellipsis may be entered or displayed multiple times. Do not enter the ellipsis itself.
[*optional*]	Text presented between square brackets is optional. Do not enter the brackets themselves. Text that is *not* in brackets is always required.
text1 | *text2*	The vertical bar represents "or". Only one of the text strings separated by a vertical bar can be entered or displayed. Do not enter the vertical bar itself.

All library commands and parameters are shown in lowercase letters. User entries can be any combination of lowercase and uppercase letters, however.

CHAPTER 1:
ACS LIBRARY OVERVIEW

INTRODUCTION

The StorageTek® UNIX® Storage Server / 4400 Automated Cartridge System (ACS) is a fully automated, cartridge-based, 18-track storage and retrieval system. It provides automated tape cartridge library services to a network of heterogeneous client systems. The client systems may range from workstations to supercomputers. They may be located in the same data center or spread across multiple locations.

The basic hardware component of the system is a Library Storage Module (LSM), a 12-sided structure containing the following:

- Storage cells for approximately 6000 tape cartridges.
- A robot that retrieves and moves the cartridges.
- Apertures in the walls of the structure, through which cartridges can be passed to load and unload cartridge drives outside the LSM.

Figure 1-1 shows an LSM with associated electronic modules and attached cartridge drives.

The system is controlled by the Storage Server software residing on a server system. The Storage Server receives mount and dismount messages from client systems or library operators and translates them into robot movement commands.

Figure 1-2 illustrates how an ACS is divided in terms of function.

ACS LIBRARY FUNCTIONS

The ACS library performs automated mounts and dismounts of tape cartridges in response to requests received from client applications or library users. The ACS library controls only the movement and locations of tape cartridges, not the data recorded on them. The library identifies cartridges by their external bar-coded labels; it does not verify the external labels against magnetically-recorded internal labels.

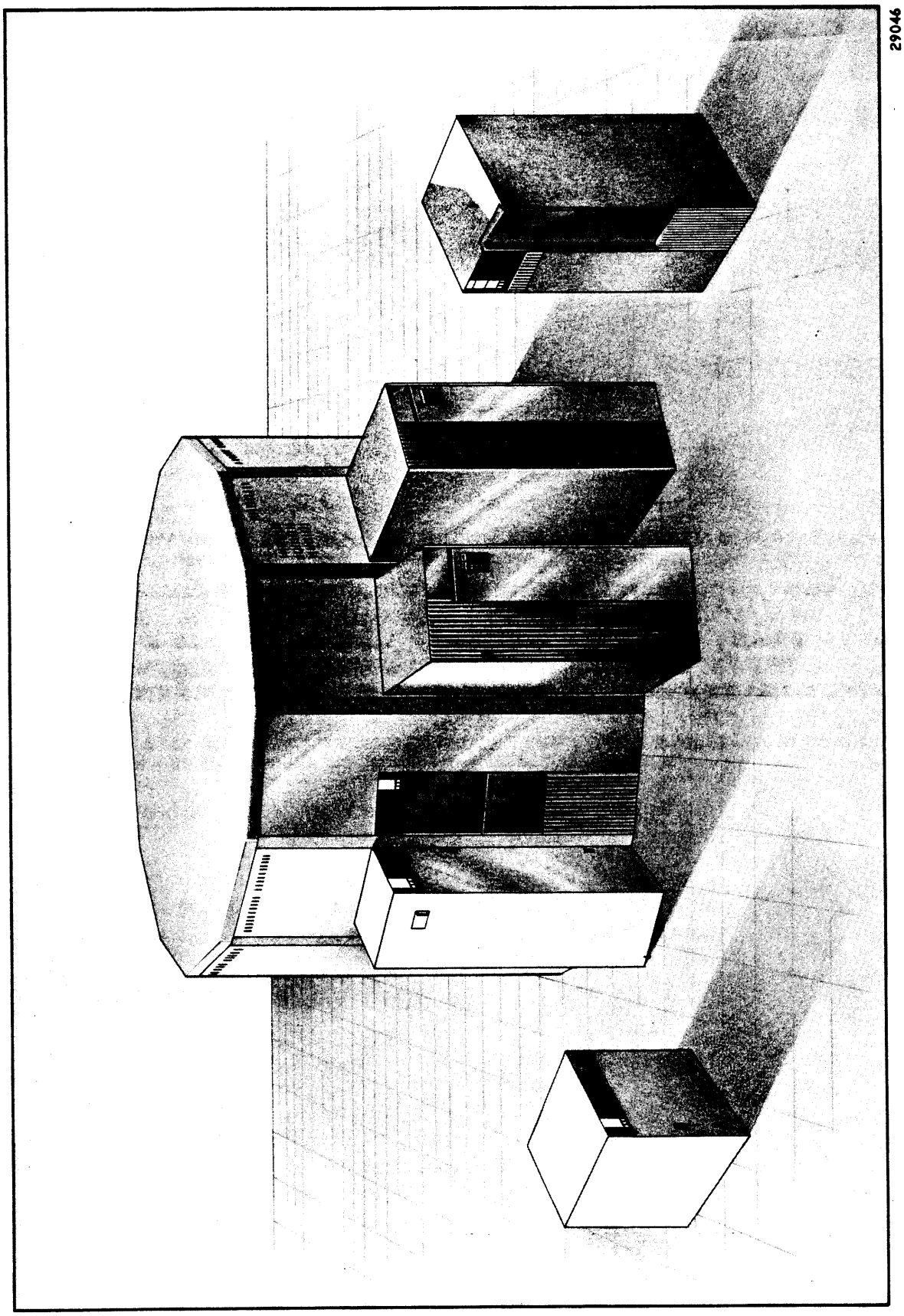

Figure 1-1. 4400 Automated Cartridge System

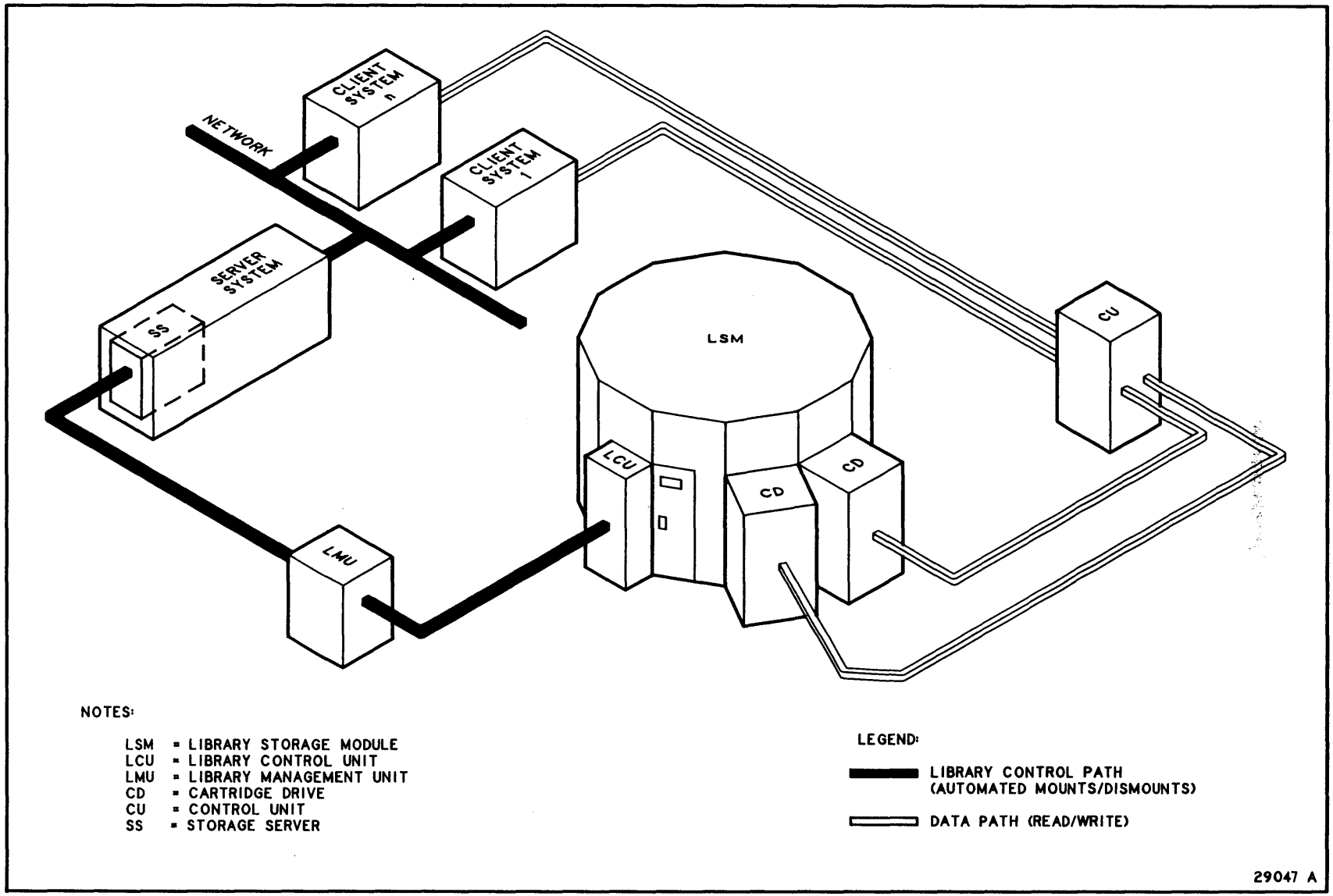

Figure 1-2. 4400 ACS Library Control/Data Paths

Client applications determine when automated tape handling is required. They allocate specific library tape drives and resolve any resource deadlocks. The Storage Server software controlling the ACS library rejects any requests that specify cartridges or tape drives outside of the library.

The client applications control the library tape drives, including transfer of data to and from a cartridge, detecting and recovering from tape data transfer errors, and determining write protection for a cartridge. Storage Server control of tape drives is restricted to forcing rewinds and unloads.

BENEFITS

The 4400 ACS provides the following benefits:

- *Storage capacity is available in increments of approximately 6000 cartridges.* The maximum capacity of a library is approximately 24 million cartridges.
- *Faster tape access time than manual systems.* Cartridge select and mount time averages 11 seconds for a cartridge in the same LSM as the cartridge drive.
- *More dependable tape operations.* Automated cartridge handling reduces the potential of human error. Newer technology is less prone to the mechanical alignment problems of older automated tape libraries.
- *Less expensive tape operations.* Automated cartridge handling reduces the need for manual labor to handle tapes. Reduced floor space, power, and air conditioning requirements generate additional cost savings.
- *Broader access to ACS library services.* The Storage Server permits systems with appropriate data paths and software to store, mount, dismount and retrieve tape cartridges automatically.
- *Darkened data center.* The Storage Server can be located in a remote, "darkened," data center with entry by personnel required only for maintenance and entry and ejection of cartridges.

ACS LIBRARY HARDWARE COMPONENTS

A 4400 ACS consists of the following hardware components:

- Library Storage Module (LSM)
- Library Control Unit (LCU)
- Library Management Unit (LMU)
- 4480 Cartridge Subsystem
- Server system

Library Storage Module (LSM)

The LSM (Figure 1-3) consists of the tape cartridge storage area and an internal robot for moving the cartridges. Each LSM provides storage cells for approximately 6000 cartridges and connections for up to 16 cartridge transports. Up to 16 LSMs can be interconnected through Pass-Thru Ports (PTPs) in adjacent LSM walls.

The robot can retrieve any cartridge in the LSM and deliver it to another cell, a transport, or a Pass-Thru Port (PTP). The robot has an optical system that identifies the correct cartridge by its external bar-code label and an electro-mechanical system that picks up the cartridge and delivers it to the correct location. Cartridges can be passed from one LSM to another through the Pass-Thru Ports.

Each LSM has doors in the outer and inner walls allowing access to the interior. The access door in the outer wall contains a cartridge access port (CAP) which is used to enter cartridges into and eject them from the LSM without opening the door. The CAP holds 21 cartridges at a time.

Cartridges are stored in cells located on both the outer and inner LSM walls. The outside wall contains twelve panels, while the inner wall contains eight panels. Figure 1-4 illustrates two typical panels with cartridge storage cells. Each panel is divided into columns and rows for cartridge storage. Cartridges are placed into a library location defined by an ACS identifier, an LSM number, a panel number, a row number, and a column number.

Pass-Thru Ports are also shown in Figure 1-3. Each PTP occupies half of two rows at the base of a special wall panel. A PTP is installed by a Customer Services Engineer in a master/slave relationship. In addition, circuitry to control the PTP is connected to the master. As viewed from inside the LSM, the master side of the PTP is on the right, while the slave side is on the left. In Figure 1-3, the LSM on the left is the master side of the PTP, while the LSM on the right is the slave. On the PTP panel, the half-rows adjacent to the PTP are used for storing cartridges. Each LSM can have up to four PTPs.

Library Control Unit (LCU)

The LCU is a microprocessor that controls the robot's movements. There is one LCU per LSM. It is attached to the panel immediately to the left of the LSM access door.

The LCU translates gross movement requests received from the LMU into the discrete servo commands required to control the robot.

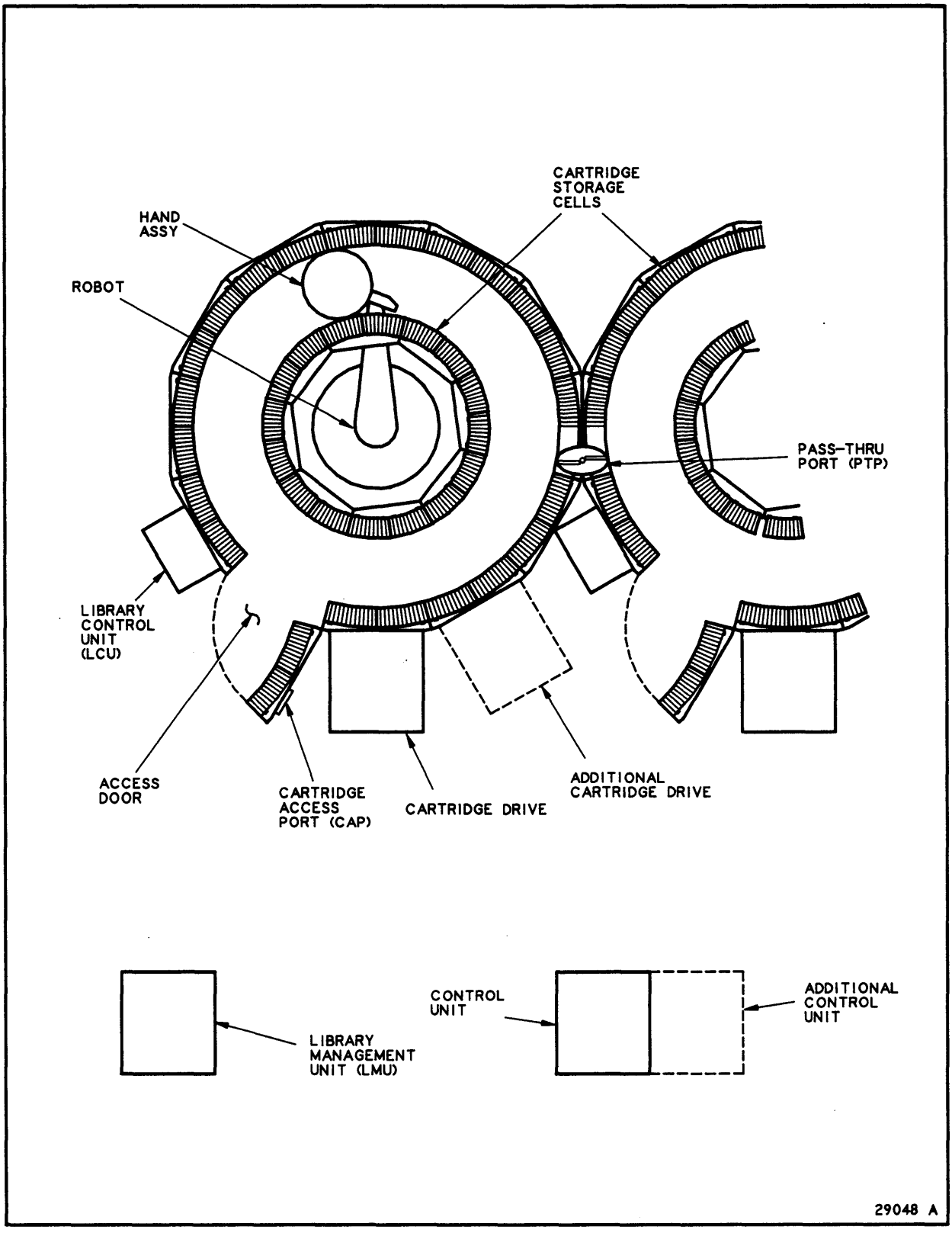

Figure 1-3. 4400 ACS Hardware

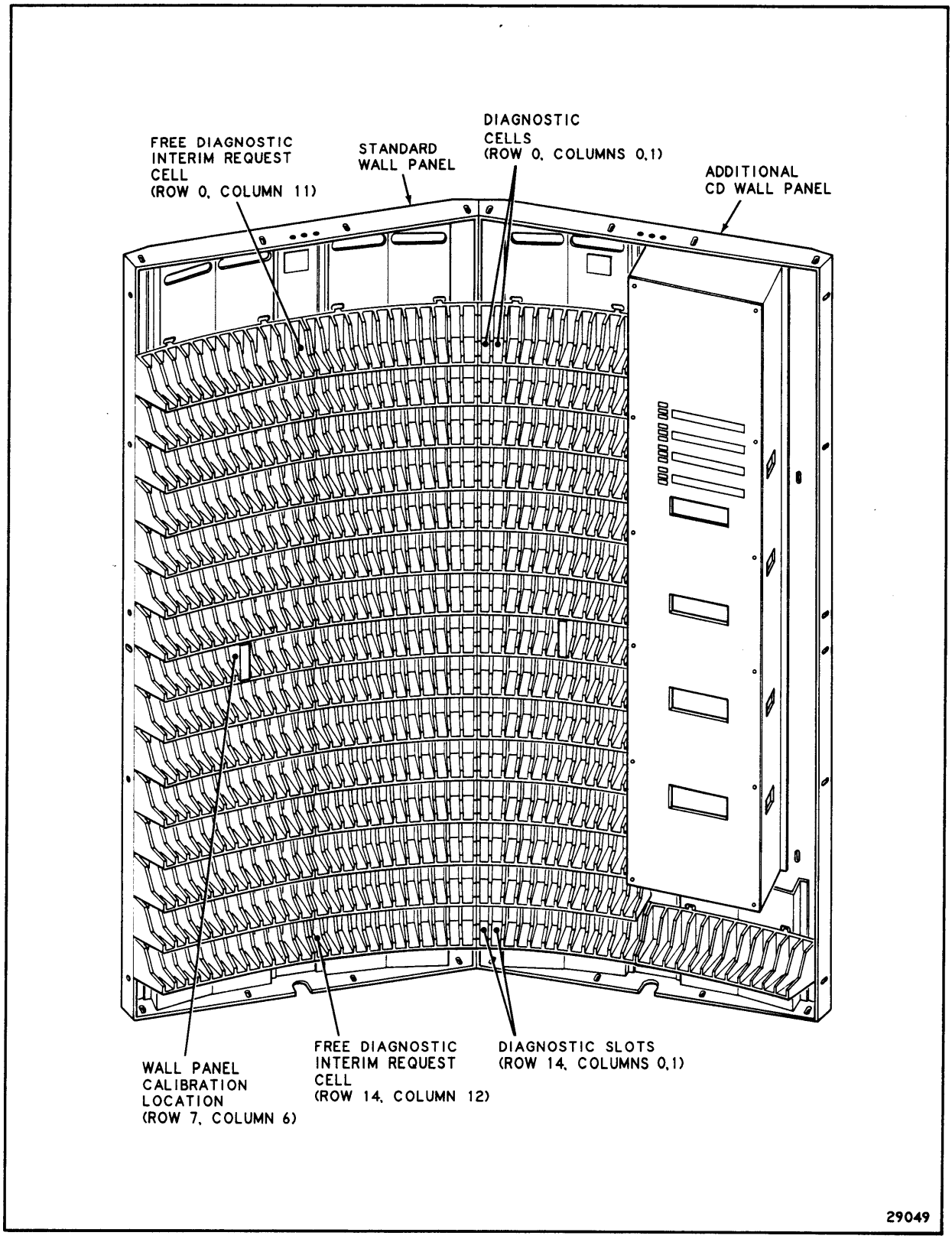

Figure 1-4. LSM Cartridge Storage Cells

Library Management Unit (LMU)

The LMU manages all the LSMs in an ACS. There is one LMU per ACS. The LMU receives cartridge movement requests from the Storage Server, translates them into robot movement instructions, and relays these instructions to the correct LCU. The LMU also passes ending status from the LCU back to the Storage Server. The LMU allocates LSM resources (robot, CAP, ports, etc.) to optimize and coordinate cartridge movement within and among LSMs.

The LMU communicates with the LSMs through a Local Area Network (LAN). It communicates with the Storage Server through an RS423 interface.

4480 Cartridge Subsystem

The StorageTek 4480 Cartridge Subsystem consists of a control unit (CU) and at least one cartridge drive (CD) containing either two or four transports. There can be up to four cartridge drive units attached to each LSM. See the *4480 Cartridge Subsystem Operator's Manual* for details on the 4480 cartridge drives and CUs.

4480 Control Unit

The 4480 CU is the controller/interface between the client systems and up to eight transports. The CUs are connected to client systems either directly via I/O channels or indirectly via a data network. Each CU is controlled by dual microprocessors and contains a data buffer which is used to maximize transfer rates at the channel interface.

The CU interprets and distributes commands to the appropriate transport, provides data formatting (including error correction and detection) for the subsystem, and reports CU and transport status.

A CU coupler feature is available. This feature, along with two CUs and a maximum of sixteen transports (four cartridge drives), enables a configuration in which each CU is capable of directly addressing any of the sixteen transports, since each transport is attached by cables to both CUs.

4480 Cartridge Drive

The cartridge drive units contain two or four transports each, along with the supporting pneumatic equipment and power supplies. Library drives differ from manual drives only in that they are attached to an LSM and are controlled by the Storage Server rather than by an operator. All transports in a library drive can be operated concurrently.

Each transport performs the following functions:

- Read/Write functions (using a standard 18-track cartridge)
- Automatic threading and positioning of the tape
- Status reporting to the CU

The drive mechanism moves tape across the head at a precise speed through servo-driven motors. Rewind speed is also controlled by the servo electronics and drive motors.

Table 1-1. 4480 Performance Specifications

Tape Speed:	70 IPS (2 meters/sec)	Read/Write
	158 IPS (4 meters/sec)	Rewind & Search
Tape Density:	37,871 bytes/inch	
	1,491 bytes/millimeter	

Server System

The server system hardware is the residence for the Storage Server software. The server system consists of a UNIX-based processor, a network adaptor, one 1/4" cartridge tape drive, one hard disk, and a terminal. The network adaptor acts as a buffered communications controller to move messages between the server and the client systems. Examples of network adaptors include Ethernet™ controllers and HYPERchannel® processor adaptors.

The server system is the interface between any number of heterogeneous client systems and one library. No other system can be connected to the library. The server system is connected directly to each LMU through an RS423 connection. At least two connections between the server system and LMU are recommended for redundancy.

Data path connections between the server system and library drives are not supported.

ACS LIBRARY SOFTWARE COMPONENTS

Storage Server and Client Software Interaction

The client software resides on any number of distributed, heterogeneous client systems. This software manages tape cartridge contents, generates requests for cartridges, and transfers data to and from cartridges. The client software is *not* part of the Storage Server

product. It must meet certain requirements, however, in order to be able to communicate with the Storage Server.

The Storage Server software resides on the server system. It manages the storage and movement of tape cartridges and the use of library resources. It translates requests for tape cartridges, received from the client software, into cartridge movement requests for the LMU.

Figure 1-5 illustrates the Storage Server and client system software components and their interfaces. These components are described in detail in the paragraphs that follow.

A client application generates cartridge movement requests which are translated by the client Storage Server Interface (SSI) into a format that can be interpreted by the Storage Server. The requests are then passed from the client system to the server system via the network interfaces.

The Client System Interface receives the requests from the network interface, reformats them, and passes them to the ACS Library Manager. The ACSLM validates the requests, then translates and routes them to the LMU. If either the CSI or ACSLM encounters any errors, they are sent to the Event Logger. After the request is completed, a response is returned, through the same channels, to the client application.

Storage Server Software

The Storage Server software executes within a UNIX System environment that complies with the System V Interface Definition (SVID). One exception to SVID compliance is the use of BSD sockets as the interprocess communications mechanism.

The Storage Server consists of the following major components:

- ACS Library Manager (ACSLM)
- ACS System Administrator (ACSSA)
- Client System Interface (CSI)
- ACS Event Logger (ACSEL)
- Network Interface (NI)
- Storage Server data base

All of these components reside on one server system; distribution of these components across multiple server systems is not supported.

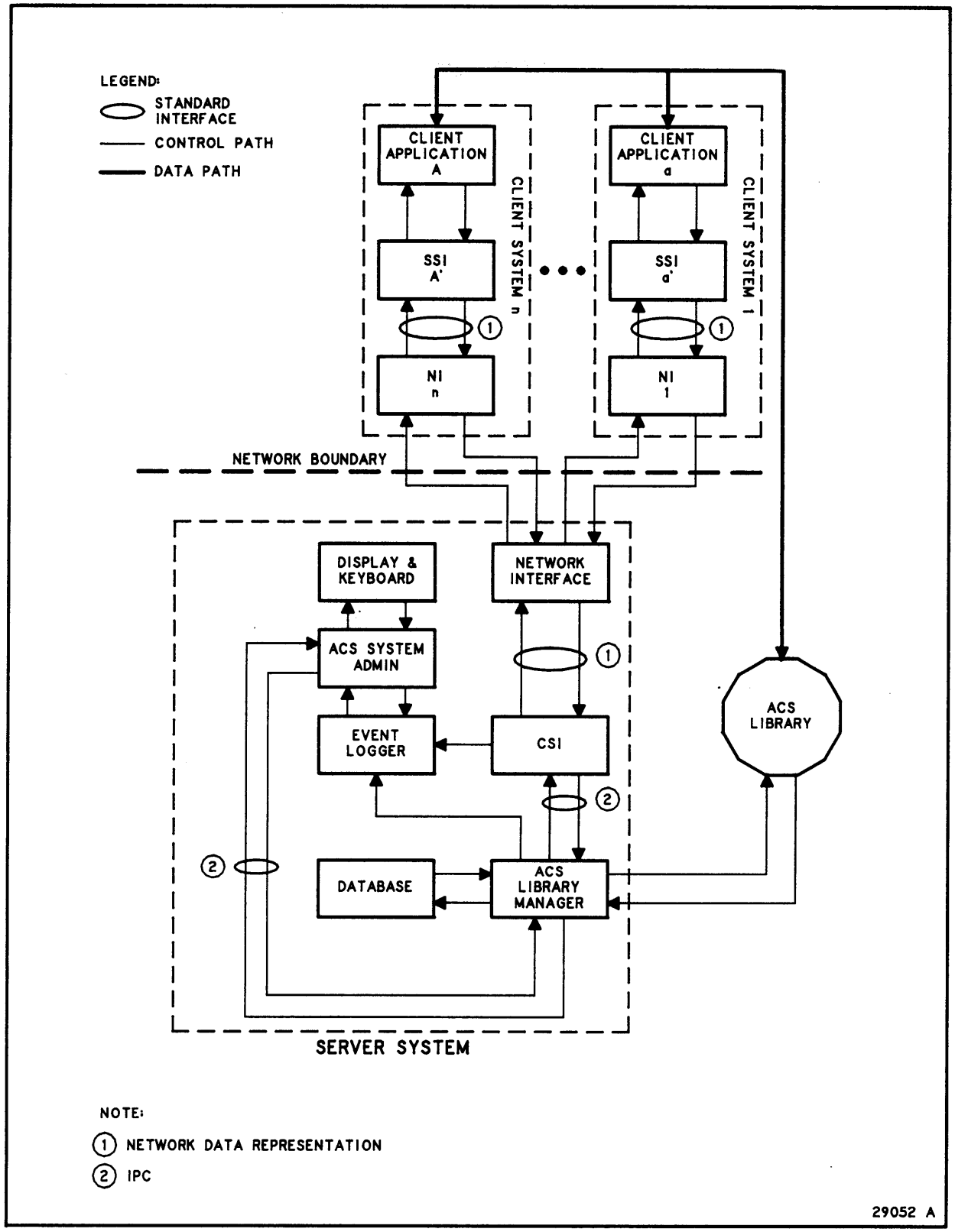

Figure 1-5. Storage Server / Client Software Components

ACS Library Manager (ACSLM)

The ACSLM processes library requests originating from client applications (through the CSI) or library users (through the ACSSA). It validates these requests and routes valid ones to the LMU. When responses are returned from the LMU, the ACSLM routes them to the appropriate requestor (either the ACSSA or the CSI).

If the ACSLM encounters errors, it routes event messages to the Event Logger. Additionally, the ACSLM routes unsolicited messages to the ACSSA when it is notified of a significant event occurring in the library.

The ACSLM also maintains the configuration and cartridge location data base. The data base supports checkpointing and journaling to facilitate recovery from errors. See the *UNIX Storage Server System Administrator's Guide* for details.

The ACSLM performs the following functions to recover from errors with little or no operator intervention:

- Detects, notifies, and recovers from library failures. These include the loss of an LMU, LSM component, etc.
- Works with the LMU to recover an LSM after it has failed. In particular, this includes determining that there are volumes in-transit, discovering their external labels, and disposing of them properly.
- Detects, isolates, reports, and recovers from communication line failures (for example, loss of terminal port).
- Attempts to reestablish data paths if a communications failure occurs with a CSI or the ACSSA.
- Detects, isolates, reports, and recovers from software errors (for example, program interrupts, operator cancellations, process terminations).

ACS System Administrator (ACSSA)

The ACSSA provides a screen interface that enables library operators and users to monitor and control Storage Server operations. The screen interface is referred to as the Command Processor.

The Command Processor receives requests from a user and performs basic syntax validations on the input. If it detects errors in a request, the Command Processor displays error messages and prompts for the correct entry. If a request has no errors, the ACSSA passes it to the ACSLM for further processing.

The ACSSA also receives and processes responses from the ACSLM; resulting error messages are displayed by the Command Processor.

When the ACSLM returns more than one response for a request, the ACSSA displays each one as it is received.

The ACSSA supports multiple Command Processors. That is, several users can be entering requests at one time through separate terminals or Command Processor windows.

See the *UNIX Storage Server System Administrator's Guide* for the specific commands supported by the Command Processor.

Client System Interface (CSI)

The CSI serves as the interface between the ACSLM and the Storage Server Interfaces (SSIs). The CSI presents a network- and host-independent control path message format to client applications. It receives requests from an SSI and translates them into a format that can be interpreted by the ACSLM. It also translates ACSLM responses and routes them to the appropriate SSI. The CSI communicates with the SSIs through the network interfaces.

The CSI attempts to reestablish communication paths if a communication failure occurs between the ACSLM and CSI or between the CSI and NI. The CSI can control the flow of messages when network or processor congestion occurs. It also routes error messages to the Event Logger.

ACS Event Logger (ACSEL)

The ACSEL records messages describing library errors and software errors not normally tracked by the operating system. This data can be used for later tracking and analysis.

The ACSLM and CSI independently notify the ACSEL of abnormal events. The ACSEL writes records of these events to a centralized file known as the Event Log. See the *Event Logging* section in *Chapter 2* for details.

Network Interface (NI)

The NI implements a customer-specified network communications protocol. The NI resident on the server system interacts with the NIs on the client systems to maintain connections, control the flow of requests and responses, and perform error recovery as necessary.

Client Software

The client system software components described below are supplied by the customer and are *not* part of the Storage Server product. Their descriptions are provided only to clarify the differences between Storage Server and client application functions.

- Network Interface (NI)
- Storage Server Interface (SSI)
- Client applications

Network Interface (NI)

The NIs on the client systems function in the same manner as the NI on the server system. They implement a customer-specified network communications protocol to allow for the transfer of messages between the server and client systems.

Server System Interface (SSI)

Each SSI serves as the interface between the CSI and the client applications residing on that client system. Any number of client applications can issue Storage Server requests. The SSI processes these requests in the order that it receives them. It translates the requests into a format that can be interpreted by the CSI, and sends them to the CSI through the network interfaces.

The SSI also receives response messages from the CSI, translates them, and sends them to the appropriate client applications.

Client Applications

Any number of client applications can manage volumes contained in the ACS library. A Tape Library Management System (TLMS) is one example of a client application that would interact with the library. Consistency between multiple applications is maintained by the applications themselves, not by the Storage Server.

The client applications manage cartridge contents, whereas the Storage Server manages cartridge locations. The only information provided to client applications by the Storage Server are lists of volumes entered, ejected, or currently residing in the library.

Client applications gain access to tape cartridges by interacting with the Storage Server through the *control path.* They read and write data on tape cartridges by interacting directly with a cartridge drive through the *data path.* Data path interactions do not affect Storage Server operations. For example, an application issuing an unload request to a cartridge drive does not cause the Storage Server to move the unloaded

cartridge to a library storage cell; the application must issue a separate request across the control path to move the cartridge.

STORAGE SERVER ARCHITECTURE

Adaptability

The Storage Server software can run within any environment compliant with UNIX System V Interface Definition (SVID), as defined for UNIX System V Release 2. The environment must also allow for the BSD implementation of sockets. The software has the following characteristics which allow it to be run on a variety of hardware platforms.

- *Adaptability*. The software is written using replaceable modules with well-defined interfaces isolating system dependent features.
- *Consistent user interface*. Command arguments and options that are common to more than one command have the same syntax and meaning.
- *Modularity*. The architecture allows for future variability in hardware and software, including different kinds of connections between the server system and the LMU, different data base management systems, different network protocols, different interprocess communication methods, and different library authorization methods.
- *Portability*. The Storage Server software allows porting to other server system environments, both hardware and software. This requirement is accomplished with the following design characteristics:
  - The software is modular and based on an object-oriented design. The software incorporates object-oriented techniques including data encapsulation.
  - All software modules comply with the System V Interface Definition, as defined for UNIX System V Release 2.
  - All operating system calls are isolated in replaceable software modules.
  - All software modules are written in the "C" source language.

OPEN SYSTEMS INTERCONNECTION (OSI) MODEL

Overview

The International Standards Organization (ISO) Open Systems Interconnection (OSI) model describes the functions required to reliably transmit data between two applications on different host systems. This model uses a seven-layer architecture, as follows:

1. Physical layer
2. Data link layer
3. Network layer
4. Transport layer
5. Session layer
6. Presentation layer
7. Application layer

The Storage Server software components can be applied to the OSI model. Figure 1-6 illustrates how they are applied, and the following sections provide supporting discussions.

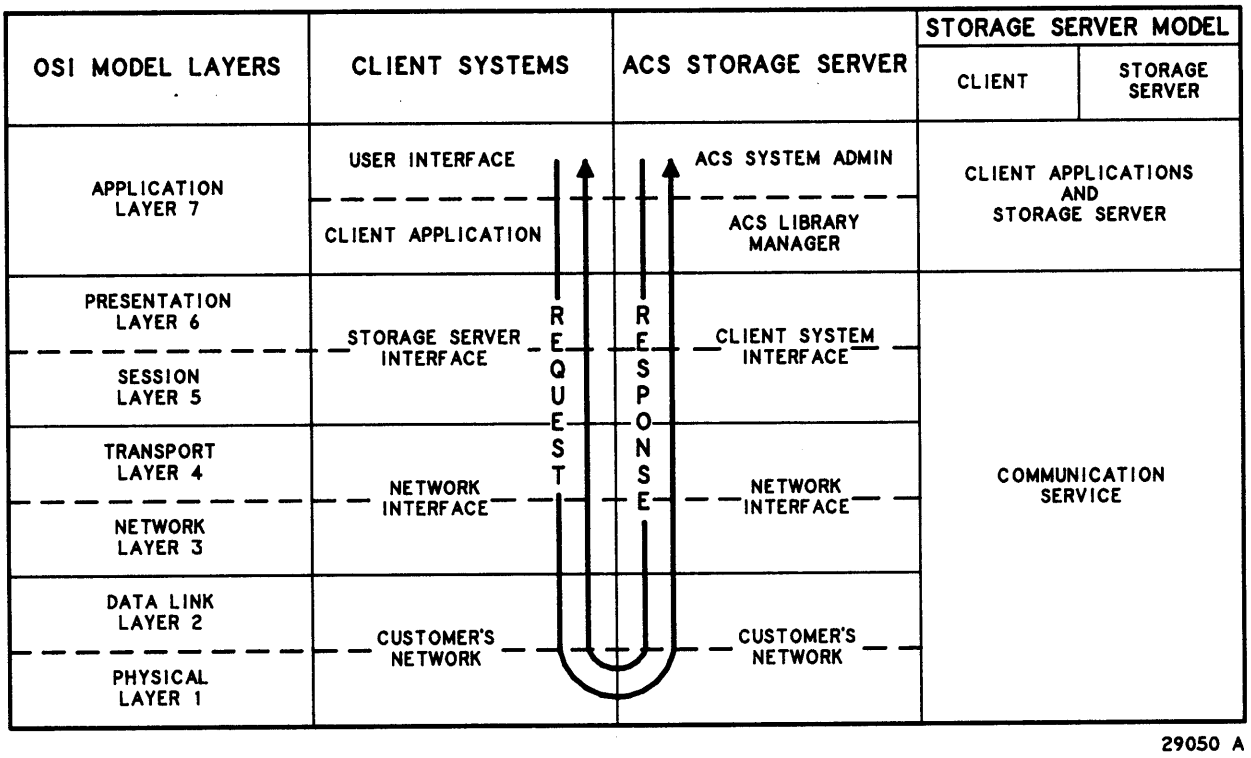

Figure 1-6. Storage Server Component Relationship to OSI Model

Layers 1 and 2 – Physical and Data Link Layers

Layers 1 and 2 provide for the physical transfer of messages between network nodes. The customer's network provide these layers between the client systems and the server system.

Layers 3 and 4 – Network and Transport Layers

Layers 3 and 4 provide for the transfer of data between network nodes and perform any necessary error recovery and flow control. The Network Interfaces (NIs) on the client systems and the server system fulfill these layers. Each NI implements the same customer-specified network protocol, therefore they cooperate to establish, maintain, and terminate connections between client systems and the Storage Server network nodes.

Examples of Layer 3 – Network Layer protocols are:

- Internet Protocol (IP)
- ISO Connectionless Network Service (CLNS)

Examples of Layer 4 – Transport Layer protocols are:

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)
- ISO Connectionless Transport

Layers 5 and 6 – Session and Presentation Layers

Layers 5 provides for the control of application interactions, and Layer 6 provides for the exchange of messages between those applications. The CSI and SSIs fulfill these layers.

At Layer 5, the CSI and SSIs cooperate to control the exchange of client application requests and ACSLM responses. They interact with the NIs to receive requests from and send responses to one another, thereby insulating the client and Storage Server applications from the NIs. CSI and SSIs ensure that the ACSLM and client applications receive complete messages from the lower layers.

At Layer 6, each SSI and the CSI implement the same network- and host-independent message format for exchanging requests and responses across the network. They translate control path messages from the network message format to the internal message format and data representation of their respective host systems, and vice versa. The requests and responses defined by the ACSLM programmatic interface dictate the contents of control path messages. The Layer 6 protocol dictates the format and encoding of those messages.

Different SSIs are used for different client operating systems and network protocols. Different CSIs are used for different network protocols.

An example of Layer 5 – Session Layer protocols is:

- Sun Microsystems™ Remote Procedure Call (RPC)

Examples of Layer 6 – Presentation Layer protocols are:

- Sun Microsystems External Data Representation (XDR)
- ISO Abstract Syntax Notation One (ASN.1)

A matched pair of session and presentation layers define a particular CSI. Changing either layer defines a new CSI.

Layer 7 – Application Layer

Layer 7 provides application functionality to system users. The ACSLM, ACSSA, and client system applications fulfill this layer. Client system applications interact with the ACSLM, through the lower OSI layers, to provide users with access to ACSLM functions.

CHAPTER 2:
ACSLM PROCESSES

OVERVIEW

This chapter describes the primary functions of the ACSLM, which are as follows:

- Storage Server initiation
- Storage Server termination
- Library request processing
- Routing of unsolicited messages
- Event logging
- Storage Server recovery

STORAGE SERVER INITIATION

Overview

Storage Server software initiation involves starting the ACSLM, ACSSA, and CSI components on the server system.

The Storage Server initiation command file is automatically installed as part of the Storage Server installation. The command file is called

acsss_home`/rc.acsss`

where *acsss_home* is the directory in which the Storage Server software was installed, usually `/usr/ACSSS`.

The command file can be invoked in two ways:

- *Manually* by invoking the command file at a server system terminal.
- *Automatically at system IPL* by referencing the initiation command file in the system startup file.

Storage server initiation can only be invoked by the Storage Server user ID, `acsss`, or the server system superuser.

Normally, the ACSLM automatically enters the STATE_RUN state after initiation is complete. To automatically put the ACSLM in the STATE_IDLE state after initiation, the IDLE qualifier can be included in the invocation command. If initiation is invoked manually, the IDLE qualifier must be typed as part of the request:

```
/usr/ACSSS/rc.acsss IDLE
```

If initiation is invoked as part of system IPL, the *acsss_home*/rc.acsss file must be modified to include the IDLE qualifier.

Initiation Process

During initiation, the ACSLM performs the following functions:

- Ensures that only one copy of the Storage Server software is running at any given time.
- Establishes the Storage Server infrastructure. The CSI establishes communication paths with the NI and the ACSLM, and the ACSSA establishes communication paths with the ACSLM. Any failures in this process will halt initiation.
- Confirms the integrity of the data base. This includes examining the table structures and contents for consistency. It also includes checking for correct data base file access permissions and the existence and state of journal files. Any structural or access problems with the data base will halt initiation.
- Verifies the library configuration recorded in the LMU against that recorded in the data base. Inconsistencies will halt initiation.
- Performs Storage Server recovery procedures to bring the library to an operable state. This includes attempting to recover from CPU or operating system failures. See the *Storage Server Recovery* section in this chapter for details.
- Attempts to put all library components online. Failure of a library component to go online is noted in the Event Log and initiation continues. If the data base indicates that a library component is in the offline or diagnostic state, no attempt is made to put that component online, or change its state.

Error messages will be generated for any LMU requests outstanding at the time of initiation.

LIBRARY REQUEST PROCESSING

Overview

The ACSLM receives requests from the following sources:

- A system user through the ACSSA Command Processor
- A client application through the CSI

The ACSLM returns responses to the appropriate request originator. Following are the requests that the ACSLM processes:

```
audit
cancel
dismount
eject
enter
idle
mount
query
start
vary
```

See *Chapter 4: ACSLM Command Structures* for the structure and contents of ACSLM requests and responses.

ACSLM Processing States

The ACSLM can be in one of four states, as described below:

- `STATE_RUN`. The normal operating state. The ACSLM processes all library requests received from a CSI or the ACSSA.
- `STATE_IDLE`. The ACSLM rejects all requests involving library operations. Only the following requests are processed: `cancel`, `idle`, `query`, `start`, and `vary`.
- `STATE_IDLE_PENDING`. A transition state that occurs when the Storage Server is taken from run to idle. All new requests involving library operations are rejected, but current and pending requests are processed to completion. Only the following new requests are processed: `cancel`, `idle`, `query`, `start`, and `vary`.
- `STATE_RECOVERY`. A transition state that occurs when the Storage Server is taken from idle to run. The only request that will be processed is `query server`. All other requests are rejected while recovery processing takes place.

Table 2-1 identifies which requests the ACSLM processes when it is in each of the four states.

Table 2-1. Library Commands and ACSLM States

Request	RECOVERY	RUN	IDLE	IDLE_PENDING
audit		X		
cancel		X	X	X
dismount		X		
enter		X		
eject		X		
idle		X	X	X
mount		X		
query	X	X	X	X
start		X	X	X
vary		X	X	X

ACSLM State Transitions

The `start` and `idle` requests move the ACSLM between these states. These transitions occur as follows:

- The `start` request causes the ACSLM to go into STATE_RECOVERY while it performs recovery procedures on the library (see the *Storage Server Recovery* section in this chapter for details). When all recovery procedures have been completed successfully, the ACSLM moves into STATE_RUN.
- An unqualified `idle` request (that is, without the `force` option) causes the ACSLM to go into STATE_IDLE_PENDING initially. The ACSLM processes all current and pending requests to completion before entering STATE_IDLE.
- An `idle` request with the `force` option puts the ACSLM in STATE_IDLE immediately, causing any current or pending requests to be aborted.

Interactions With Other Storage Server Components

Requests are passed from either the ACSSA or a CSI to the ACSLM. The ACSLM returns responses to the appropriate request originator, either the ACSSA or a CSI. The following subsections summarize how requests and responses are passed between these components. See *Chapter 3* and *Chapter 4* for details on request and response formats.

Client Application — ACSLM Interactions

Overview

Client system users can issue library requests through client applications which may be system commands, procedures, utilities, or interactive scripts. The applications generate ACSLM requests which must pass across the network and through several software layers (that is, SSI, NIs, and CSI) before being processed by the ACSLM.

For each request, the ACSLM generates one or more responses which must pass through the same software layers before being received by the client application originating the request. These software layers generally are transparent to the ACSLM and the client applications.

Response Types

There are three response types generated by the ACSLM:

- *Acknowledge*. An acknowledge response is returned for each syntactically correct request. It indicates that the request has been received.
- *Intermediate*. An intermediate response is returned if interim information must be returned to the requestor, or if all necessary response information cannot fit in a single response message.
- *Final*. A final response is returned for each request. It indicates the success or failure of the request.

Response Coordination

Any number of client applications can exist in a network, and each application asynchronously sends multiple requests to the ACSLM. All requests are queued and processed by the ACSLM in First In First Out (FIFO) order. The ACSLM attempts to maximize library resource usage by processing as many requests as possible. As a result, the ACSLM may intermix responses to different requests from a single client application.

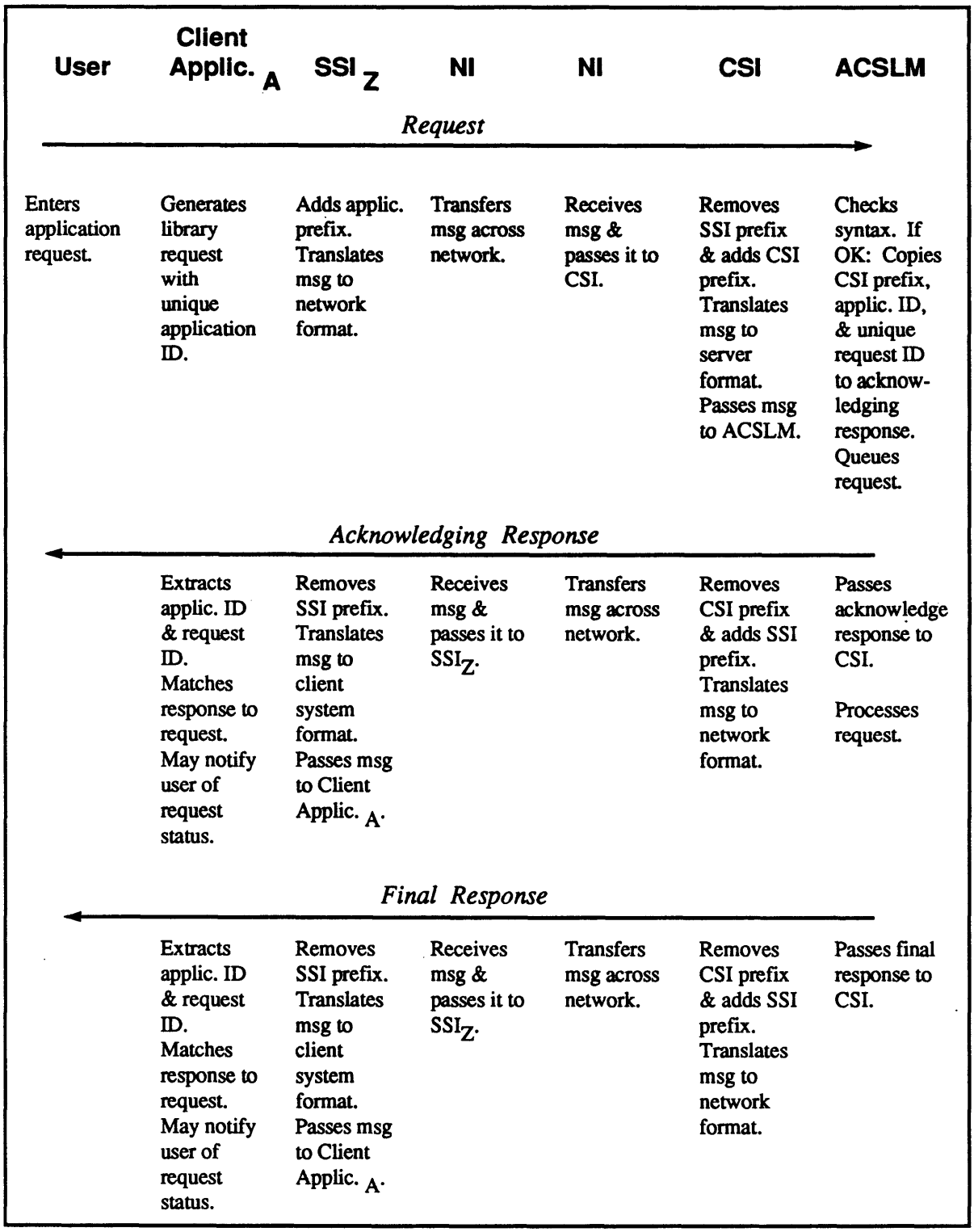

Figure 2-1. Client Application — ACSLM Message Flow

For example, if an application issues, in order, an `audit`, a `mount`, and a `query` request, the acknowledging responses will be received in the same order, but the intermediate and final responses will probably be intermixed. The final responses may be returned in a different order due to the processing time required for each request. Only the order of responses within request is guaranteed.

Interaction Process

Figure 2-1 summarizes how a request is sent from a user to the ACSLM through a client application. It also shows how acknowledging and final responses are returned to the client application. The arrows in the figure indicate the message flow direction.

Figure 2-2 through Figure 2-4 show the possible message flows between a client application and the ACSLM. Figure 2-2 shows how only a final response is returned from the ACSLM if it finds a syntax error in the request. ⊗ indicates that message processing occurs at the indicated layer.

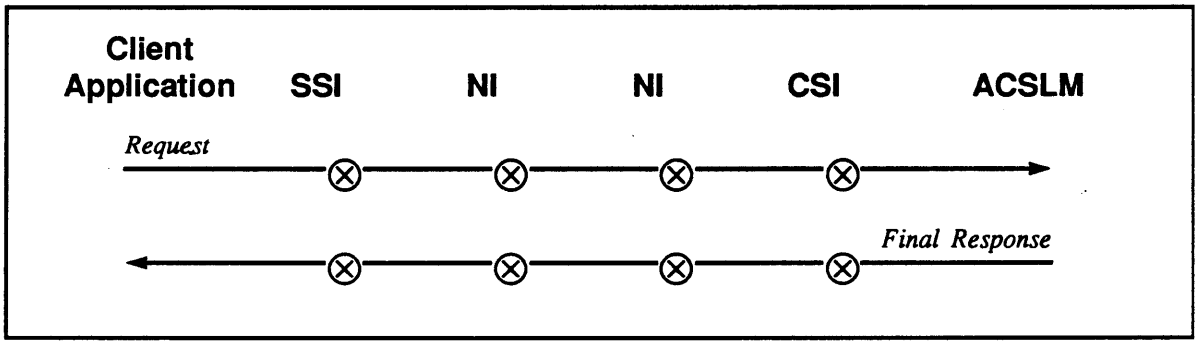

Figure 2-2. Request With a Syntax Error

An acknowledging response is always returned for syntactically correct requests. Figure 2-3 shows such a request with no intermediate responses. ⊗ indicates that message processing occurs at the indicated layer.

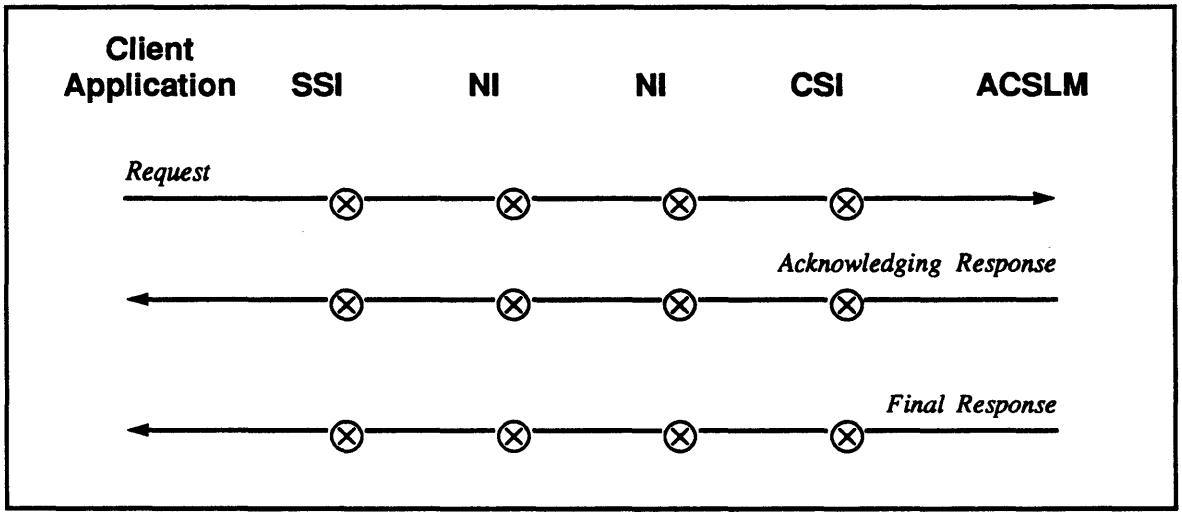

Figure 2-3. Request With No Intermediate Response

Figure 2-4 shows a syntactically correct request with one or more intermediate responses. ⊗ indicates that message processing occurs at the indicated layer.

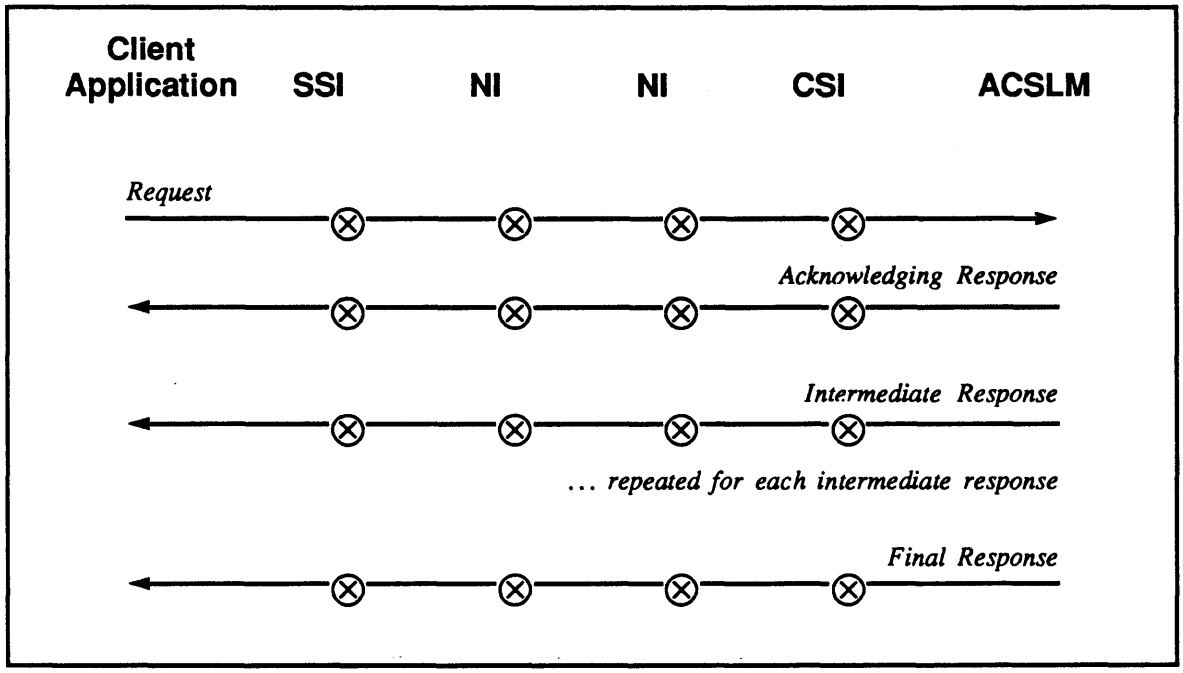

Figure 2-4. Request With Intermediate Responses

ACSSA — ACSLM Interactions

Overview

Server system users can access ACSLM services through the ACSSA Command Processor. The ACSSA translates commands from each user to equivalent ACSLM requests and multiplexes the requests to the ACSLM. The ACSSA ensures that the responses to each request are returned to the correct user.

Users must log into the special user identifier, `acssa`, to access the Command Processor. Any number of users can be using Command Processors at one time. The Command Processor ensures that each user issues only one command at a time; this prevents responses to several requests from being intermixed on the same terminal display.

The interaction between the ACSSA and ACSLM is the same as that between a client application and the ACSLM, but without the intervening software layers. The ACSSA appears as a CSI to the ACSLM since the ACSSA uses the same ACSLM programmatic interface that the CSIs use.

The ACSSA generally has no more, or less, privilege than a client application. The one exception is that ACSSA users are allowed to perform library operations on components in the diagnostic state. This feature allows maintenance personnel to exercise components without interference from client system users.

Interaction Process

Figure 2-5 summarizes how a request is sent from a user to the ACSLM through the ACSSA. It also shows how acknowledging and final responses are returned to the ACSSA. The arrows in the figure indicate the message flow direction.

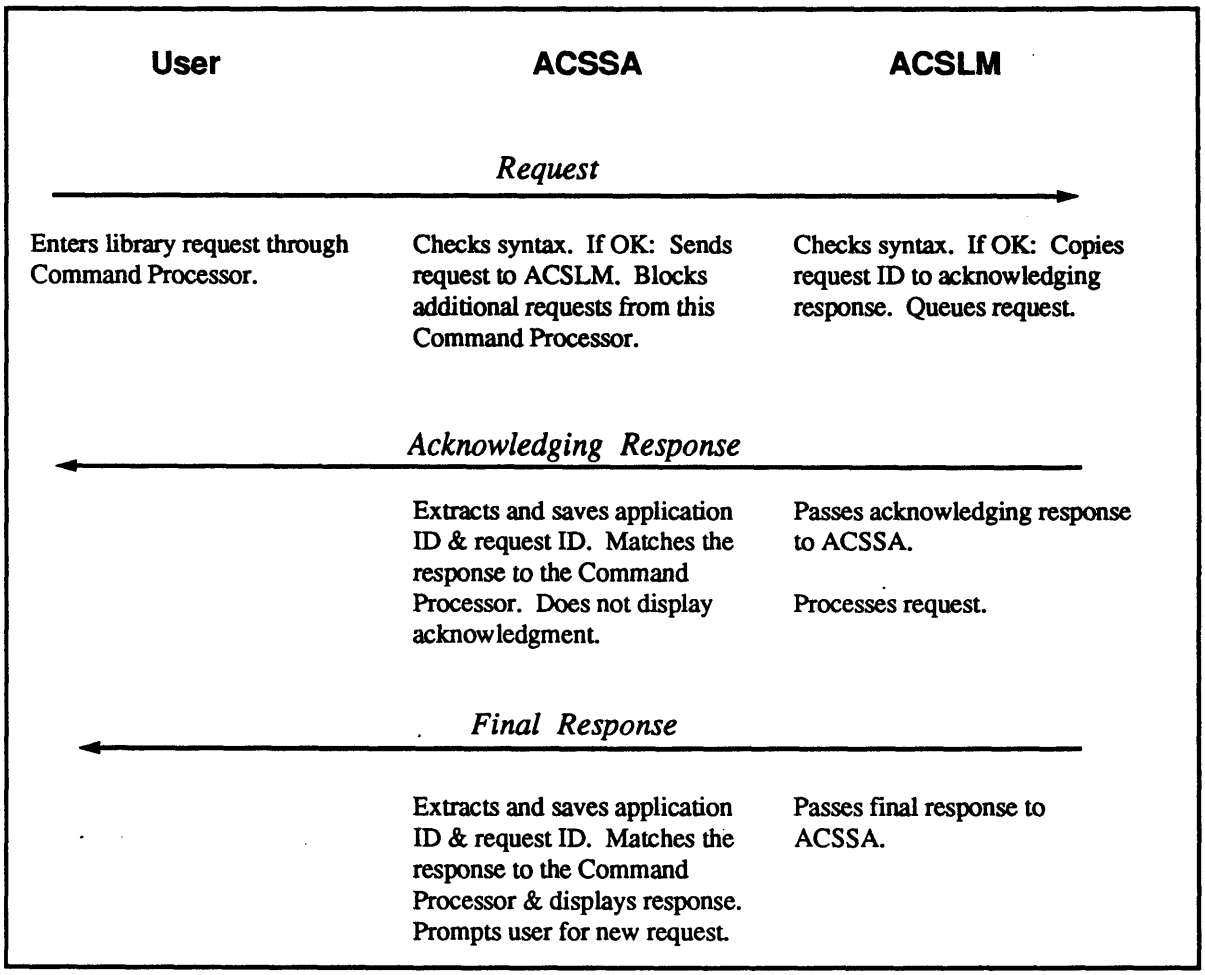

Figure 2-5. ACSSA — ACSLM Message Flow

Figure 2-6 through Figure 2-8 show the possible message flows between the ACSSA and ACSLM. Figure 2-6 shows how only a final response is returned from the ACSLM if it finds a syntax error in the request.

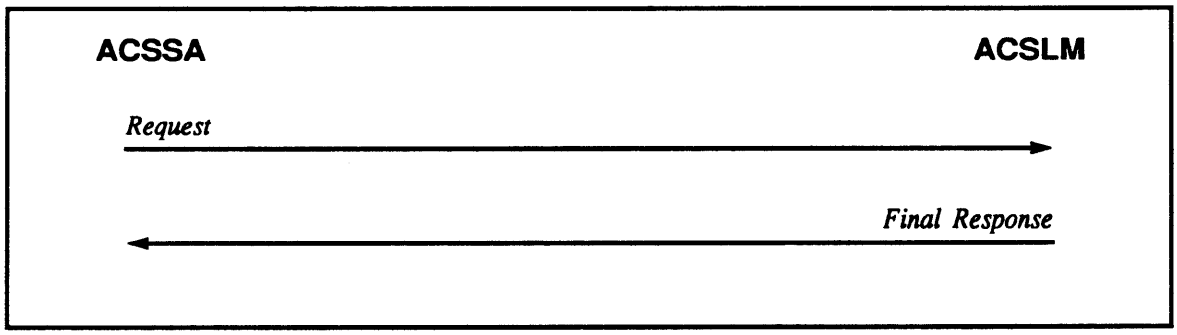

Figure 2-6. Request Containing a Syntax Error

An acknowledging response is always returned for syntactically correct requests. Figure 2-7 shows such a request with no intermediate responses.

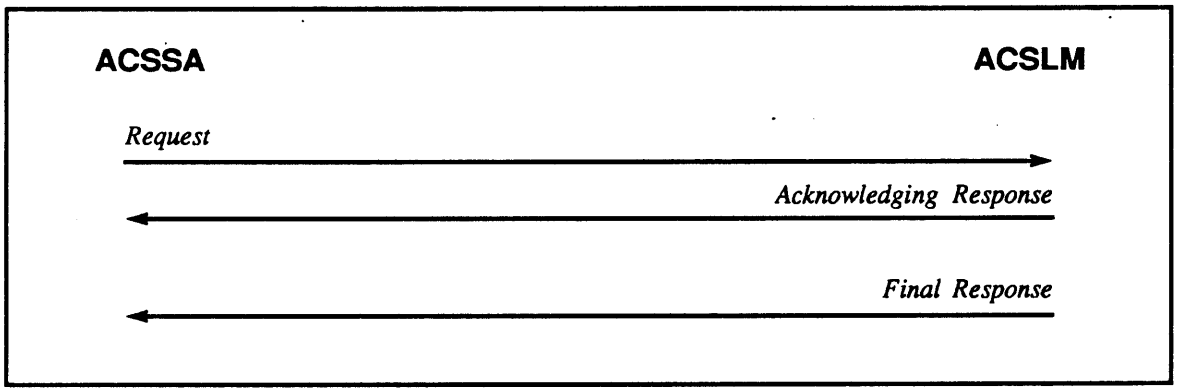

Figure 2-7. Request With No Intermediate Response

Figure 2-8 shows a syntactically correct request with one or more intermediate responses.

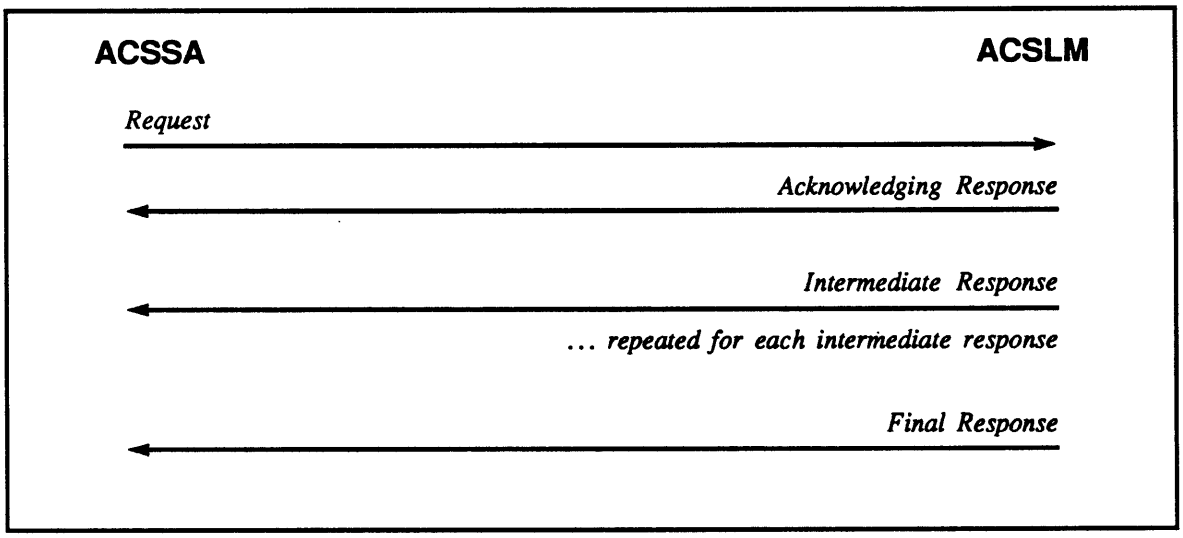

Figure 2-8. Request With Intermediate Responses

Programming Considerations

The following are special considerations for designing client applications that will submit requests to the ACSLM.

Handling Outstanding Requests

A client application must be able to handle situations where it sends a request to the ACSLM but never receives a response. These errors may be the result of communications or software failures. The method of handling the outstanding request depends on whether an acknowledge response was received, as outlined below.

- *If the application never received an acknowledge response*: The application should issue a `query server` request to determine whether the Storage Server is still functioning. If the Storage Server is functioning, the application should reissue the request.
- *If the application received an acknowledge, but not a final response*: The application should issue a `query request` request using the request ID received in the acknowledge response. If the request is still pending, the application should wait for some predefined period of time before timing out. If the request is *not* current or pending, the application should issue a `query` on some component of the request (such as a drive or volume) to determine whether the request has been completed successfully. If the request has *not* been completed, the application should reissue it.

See *Chapter 4: ACSLM Command Structures* for the format and contents of the `query` request.

Library Drive Addresses

Library drive locations are unique within an ACS. The Storage Server uses a physical identifier to locate library drives. If client systems elect to use logical drive identifiers, it is the responsibility of the client systems to map the logical drive identifier to the physical drive identifier prior to issuing a Storage Server request.

STORAGE SERVER TERMINATION

Overview

Storage Server software termination involves terminating the ACSLM, ACSSA and CSI components on the server system. The Storage Server is terminated through a command file which must be invoked manually from the server system console.

The Storage Server termination command file is automatically installed as part of the Storage Server installation. The command file is called

acsss_home`/kill.acsss`

where *acsss_home* is the directory in which the Storage Server software was installed, usually `/usr/ACSSS`.

Termination Process

Storage Server termination performs the following functions:

- All current and pending library requests are aborted. New requests are ignored.
- All data base files and the Event Log file are closed.
- All Storage Server processes are destroyed.

The ACSLM should be in the idle state when the Storage Server is terminated, otherwise data base inconsistencies and unrecoverable in-transit cartridges may result.

It is also recommended that all LSMs be in the online state when the Storage Server is terminated; this will enable LSM and drive configurations and reserved cell contents to be verified completely when the Storage Server is reinitiated. See the *Storage Server Recovery* section in this chapter for details.

STORAGE SERVER RECOVERY

Overview

Storage Server recovery procedures take place automatically under the following circumstances:

- The Storage Server is initiated. See the *Storage Server Initiation* section in this chapter for details.
- A major Storage Server failure occurs.

Recovery processing does *not* need to be initiated by the System Administrator.

During Storage Server recovery, the ACSLM performs the following processes for each ACS in the library:

- Verifies that all online ports can communicate with the ACS.
- Verifies that the library configuration recorded in the data base matches that recorded in the LMU.
- If possible, varies each ACS and its LSMs online, and marks them online in the data base.
- Directs the LSM robot to scan the physical contents of each of the following locations, and updates the data base to match:
  - Reserved storage cells
  - Cartridge drives
  - Last known location of each cartridge selected for use

Once these processes are completed successfully, request processing can resume.

Storage Server Recovery Process

The following are the steps the ACSLM goes through in performing Storage Server recovery. All data base changes that occur as a result of this procedure are logged in the Event Log. If the recovery fails, additional error messages detailing the reasons for the failure will also be found in the Event Log. See *Appendix A: Event Log* for the Event Log entries that may be made during recovery.

Note: The ACSLM will not be able to verify configuration or contents of LSMs that were in the offline or diagnostic state at the time the Storage Server failed or was terminated. This is because an offline LSM is unable to provide configuration data and the LSM robot is unable to scan storage cells and tape drives for their contents. The ACSLM will perform as much of the recovery procedure as possible and will note in the Event Log that the LSM is offline.

1. Issues the following unsolicited message to the Display Area of the Command Processor:

   ```
   Server system recovery started
   ```

2. Updates all ACS records in the data base as follows:
   - ACSs in the recovery state are changed to online.
   - ACSs in the diagnostic or offline-pending states are changed to offline.

3. Attempts to communicate with each ACS, using each port that the data base indicates is online. The ACSLM must find at least one port that can successfully communicate with the library in order for recovery processing to continue.
4. Verifies that the LSM and drive configurations in the Storage Server data base match those defined in the LMU. Discrepancies are noted in the Event Log.
5. Varies online all LSMs attached to an online ACS, if possible. Cartridge recovery is performed as part of this step.
6. Directs the LSM robot to scan the contents of all cell locations marked "reserved" in the data base. These are locations that tape cartridges were being moved either to or from at the time the system failure occurred. The ACSLM updates the data base to reflect the actual physical contents of these cells, as determined by the robot.
7. Updates the data base to reflect the true status of all library tape drives (that is, available, in use, offline).
8. Directs the LSM robot to scan the contents of all library drives that the data base indicates are in use. Updates the data base to reflect the true physical contents.
9. Directs the LSM robot to scan the contents of the last known location of each cartridge selected for use at the time of the system failure. Updates the data base with the true contents of these cells. If a cartridge is not found in its last known location it is deleted from the data base.
10. Displays either of the following unsolicited messages in the Display Area of the Command Processor, based on whether the recovery process was successful or not.

    ```
    Server system recovery complete
    ```
    – or –
    ```
    Server system recovery failed
    ```

UNSOLICITED MESSAGES

The ACSLM sends an unsolicited message to the ACSSA whenever an event requiring operator or System Administrator action occurs. The ACSSA, in turn, displays the message in the Display Area of the Command Processor screen and sends the message to the Event Logger. The Event Log entry may show additional detail concerning the event. See *Appendix A: Event Log* for the specific entries that may be written to the Event Log.

Unsolicited messages are "asynchronous," meaning that their timing is not necessarily related to the processing of a particular request. Most unsolicited messages indicate an error, although some (particularly those related to CAP processing) serve to notify the library operator when a particular routine action can be taken.

The status codes for all unsolicited messages are listed below in alphabetical order.

- `STATUS_ACSLM_IDLE` if the ACSLM has been placed in the idle state and is therefore unavailable for requests using library resources. See *Library Request Processing* in this chapter for details on ACSLM states.
- `STATUS_ACTIVITY_START` when the ACSLM has been placed in the run state.
- `STATUS_CARTRIDGES_IN_CAP` if cartridges are detected in the CAP and need to be removed by the operator.
- `STATUS_CLEAN_DRIVE` if a drive needs to be cleaned.
- `STATUS_CONFIGURATION_ERROR` if the library configuration specified in the Storage Server data base is not the same as that defined in the LMU by a Customer Services Engineer, or if a component appears in the data base but fails to respond to LMU commands.
- `STATUS_DATABASE_ERROR` if the ACSLM is unable to access the data base.
- `STATUS_DEGRADED_MODE` if the library hardware is operable, but with degraded performance.
- `STATUS_DIAGNOSTIC` if the specified device has been varied to the diagnostic state and is therefore available for requests submitted through the Command Processor only. See the `vary` command description in *Chapter 4* for additional details.
- `STATUS_EVENT_LOG_FAILURE` if the Event Logger is unable to open or write to the Event Log file.

- STATUS_EVENT_LOG_FULL if the Event Log has reached the maximum size defined during installation. This unsolicited message will be sent at one minute intervals until the size of the Log is reduced. See the *Event Logging* section in this chapter for details.
- STATUS_IDLE_PENDING if the ACSLM is in an idle-pending state and is therefore unavailable for requests using library resources. See the *Library Request Processing* section in this chapter for details on ACSLM states.
- STATUS_INPUT_CARTRIDGES if a CAP is ready to receive cartridges.
- STATUS_IPC_FAILURE if the ACSLM or CSI cannot communicate with another Storage Server process.
- STATUS_LIBRARY_FAILURE if a library hardware error occurred while the ACSLM was processing a request.
- STATUS_NI_TIMEDOUT if the CSI is unable to establish a connection with the Network Interface. Data may have been lost.
- STATUS_OFFLINE if a device has been varied offline. See the `vary` command description in *Chapter 4* for additional details.
- STATUS_ONLINE if a device has been varied online. See the `vary` command description in *Chapter 4* for additional details.
- STATUS_RECOVERY_COMPLETE when Storage Server recovery has been completed successfully. See the *Storage Server Recovery* section in this chapter for details.
- STATUS_RECOVERY_FAILED if Storage Server recovery has failed. See the *Storage Server Recovery* section in this chapter for details.
- STATUS_RECOVERY_INCOMPLETE if the specified LSM has failed to recover in-transit cartridges during Storage Server recovery. See the *Storage Server Recovery* section in this chapter for details.
- STATUS_RECOVERY_STARTED when Storage Server recovery has been initiated. See the *Storage Server Recovery* section in this chapter for details.
- STATUS_REMOVE_CARTRIDGES if a CAP contains cartridges and is ready for the operator to remove them.
- STATUS_RPC_FAILURE if the CSI has encountered a Remote Procedure Call (RPC) failure. Data may have been lost.

EVENT LOGGING

Description

One system-wide Event Log contains information about library events and errors. All Storage Server software components log events to the Log through the centralized Event Logger.

The information in this Log permits later analysis and tracking of normal library events as well as errors. Logged events include:

- *Library errors.* Both fatal and nonfatal hardware and software errors are logged. Examples include LSM failures, problems with cartridges, data base errors, interprocess and library communications failures, and software failures not normally handled by the operating system.
- *Significant events.* These are normal events that may be of significance in monitoring library operations. For example, events are logged when an `audit` is initiated or terminated, a device changes state, or a CAP is opened or closed.

The Event Log is automatically created when the Storage Server software is installed. The Log exists in the file

acsss_home`/log/acsss_event.log`

where *acsss_home* is the directory in which the Storage Server software was installed, usually `/usr/ACSSS`.

How Events Are Logged

To log an event, a Storage Server component such as the ACSLM, ACSSA, or CSI, sends a message to the centralized Event Logger. The Event Logger accepts the message and updates the Event Log in the following manner.

1. Reformats the message by applying a standard prefix.
2. Opens the Event Log file, or creates it if it does not already exist.
3. Appends the Event Log message to the end of the file.
4. Checks the current file size against the limit parameter specified at installation. If the current size exceeds the specified limit, the Event Logger sends an unsolicited message to the ACSSA to alert the System Administrator.
5. Closes the Event Log file.

Updating the Event Log in this manner keeps the Log entries sequential and allows the System Administrator to truncate or delete the file at any time during system operation.

Event Log Messages

See *Appendix A: Event Log Messages* for significant Event Log messages, as well as a description of how Event Log entries are formatted.

(INTENTIONALLY LEFT BLANK)

CHAPTER 3:
ACSLM COMMON DATA STRUCTURES

OVERVIEW

This chapter describes the general request, response, and data structure formats that a client application must be able to generate for the ACSLM or interpret from the ACSLM.

See *Chapter 2: ACSLM Processes* for a description of how the ACSLM processes requests. See *Chapter 4: ACSLM Command Structures* for the structure and contents of specific ACSLM commands.

REQUESTS

Request Format

Request messages are sent to the ACSLM from either a CSI or the ACSSA. The format of a request is:

```
struct {
     IPC_HEADER       ipc_header;
     MESSAGE_HEADER   message_header;
     •
     •
     message_data
     •
};
```

The IPC_HEADER and MESSAGE_HEADER are the *fixed* portions of the request; this information must appear in every request. The *message_data* is the *variable* portion of the request; its appearance and length is dependent on the specific request.

If the ACSLM finds an error in the fixed portion of a request, it will reject it immediately, without checking the variable portion. If the ACSLM finds an error in the variable-length portion, it will note the error and continue validating the entire variable-length portion; the ACSLM will reject the request, but the response will indicate all elements that are invalid.

ipc_header

The IPC_HEADER is generated by a CSI or the ACSSA and contains information specific to the particular operating environment. See *Chapter 7: CSI Data Structures* for the IPC_HEADER format currently used. The ACSLM does not use the IPC_HEADER, but preserves it and includes it in associated responses.

message_header

The MESSAGE_HEADER is generated by the client application (or ACSSA) and specifies ACSLM request information. The MESSAGE_HEADER has the following format:

```
typedef struct {
     unsigned short   packet_id;
     COMMAND          command;
     unsigned char    message_options;
} MESSAGE_HEADER;
```

packet_id is a unique identifier generated and used by the client application to synchronize on a forthcoming response. The *packet_id* is not used by the ACSLM, but is preserved by the ACSLM and included in its responses.

command designates the action to be performed by the ACSLM. See *Chapter 4: ACSLM Command Structures* for details about these actions.

message_options are request qualifiers generated by the client application. FORCE is the only valid *message_option* for requests. The commands that can use the FORCE *message_option* are:

```
dismount
idle
vary
```

message_data

The *message_data* is generated by the CSI or the ACSSA and varies based on the request. *message_data* may contain a repeated group of information with the form:

```
TYPE             type;
unsigned short   count;
          •
          •
    (information)
          •
```

type defines the nature of the repeated information. Only one *type* of information may appear in a message.

count is the number of times the information is repeated. For some commands, *count* can be set to 0, which indicates that the request to be performed on all objects of the specified *type*. See *Chapter 4: ACSLM Command Structures* for the individual command descriptions.

information is the repeated data. The maximum number of times it can be repeated is MAX_ID. See the *Common Variables* section in this chapter for the definition of MAX_ID.

The different possibilities for repeated information are represented as a "C" `union` declaration. This syntax is used only to denote a choice in this usage. The contents and memory allocation for a particular request or response are based not on the typical allocation rules for "C" `unions`, but on the information *type* and the *count* occurrences of it. See *Chapter 4: ACSLM Command Structures* for the `union` declarations.

RESPONSES

Description

The ACSLM generates at least one response for each request it receives. Following are the three types of responses the ACSLM can generate:

- *Acknowledge*. An acknowledge response is returned only for requests with a valid fixed-length portion. This response indicates that the request has been received and verified. It includes a request ID, generated by the ACSLM, which the user can use to perform queries on the request or to cancel it.
- *Intermediate*. An intermediate response is returned only for valid requests and only if it is required by the particular request. It is required if interim information must be returned to the requestor, or if all necessary information cannot fit in a single response.
- *Final*. A final response is returned for all requests. This response indicates whether the request succeeded or failed. If the request failed, this response indicates the nature of the error. If there are errors in the variable-length portion of the request, all errors will be noted in the final response.

General Response Format

The general format of a response is:

```
struct response {
     IPC_HEADER        ipc_header;
     MESSAGE_HEADER    message_header;
     RESPONSE_STATUS   message_status;
     •
    (parameters)
     •
     •
};
```

ipc_header

The IPC_HEADER is preserved from the request by the ACSLM. It provides information that assists the requesting process in synchronizing requests and responses. See *Chapter 7: CSI Data Structures* for the IPC_HEADER format.

message_header

The MESSAGE_HEADER has the same general format as the MESSAGE_HEADER in the request. The format is as follows:

```
typedef struct {
     unsigned short   packet_id;
     COMMAND          command;
     unsigned char    message_options;
} MESSAGE_HEADER;
```

Note: The *message_options* is the only part of the response MESSAGE_HEADER that differs from the request.

The *packet_id* is preserved from the request by the ACSLM. It is returned in the response in order for the client application to be able to match the request to a response.

The *command* is also preserved from the request by the ACSLM and returned in the response. See *Chapter 4: ACSLM Command Structures* for details about these actions.

message_options can have the following values in the response:

```
ACKNOWLEDGE
FORCE
INTERMEDIATE
```

Table 3-1 lists which response *message_options* are valid for each command response:

Table 3-1. ACSLM Response Message Options

Command	Intermediate	Force	Acknowledge
audit	X	-	X
cancel	X	-	X
dismount	X	X	X
enter	X	-	X
eject	X	-	X
idle	X	X	X
mount	X	-	X
query	X	-	X
start	X	-	X
vary	X	X	X

response_status

The RESPONSE_STATUS defines the completion status of the request. The format is as follows:

```
typedef struct {
     STATUS       status;
     TYPE         type;
     IDENTIFIER   identifier;
} RESPONSE_STATUS;
```

status values vary by request. See *Chapter 4: ACSLM Command Structures* for details.

type indicates the type of *identifier* that the *status* refers to. When no *identifier* is appropriate, the value TYPE_NONE is used and the contents of *identifier* are undefined.

identifier specifies the device or object to which the *status* refers. *identifier* is defined as:

```
typedef union {
     ACS          acs_id;
     CAPID        cap_id;
     CELLID       cell_id;
     DRIVEID      drive_id;
     LSMID        lsm_id;
     PANELID      panel_id;
     PORTID       port_id;
     SUBPANELID   subpanel_id;
     VOLID        vol_id;
     char         socket_name[14];
} IDENTIFIER;
```

See the *Common Variables* section in this chapter for the definitions and formats of each IDENTIFIER.

Parameters

The *parameters* vary depending upon the nature of the response. See *Chapter 4: ACSLM Command Structures* for the possible parameters.

Specific Response Formats

All of the possible response formats and their contents are described below.

Acknowledge Response

The format of an acknowledge response is:

```
struct acknowledge_response {
     IPC_HEADER          ipc_header;
     MESSAGE_HEADER      message_header;
     RESPONSE_STATUS     message_status;
     MESSAGE_ID          message_id;
};
```

Within MESSAGE_HEADER, *message_options* is set to ACKNOWLEDGE.

Within RESPONSE_STATUS, *message_status* is set to STATUS_VALID, *type* is set to TYPE_NONE, and *identifier* is not used.

The *message_id* is generated by the ACSLM and uniquely identifies the request. The requestor can use this value to query the status of the request or to cancel the request.

Intermediate Response

Within MESSAGE_HEADER, *message_options* is set to INTERMEDIATE.

See the individual request descriptions in *Chapter 4* to determine if the ACSLM returns an intermediate response for a request, and for a description of the specific format of the responses.

Final Response – Successful Request

See the individual request descriptions in *Chapter 4* for the format of each final response.

Final Response – Failed Request

When the ACSLM finds a syntax error in the fixed-length portion of a request, the final response has the following format:

```
struct final_response {
     IPC_HEADER          ipc_header;
     MESSAGE_HEADER      message_header;
     RESPONSE_STATUS     message_status;
     TYPE                type;
     unsigned short      count;
};
```

Within RESPONSE_STATUS, *message_status* identifies the syntax error. *type* indicates the type of *identifier* that the *status* refers to; if no *identifier* is appropriate, TYPE_NONE is used and the contents of *identifier* are undefined.

Within MESSAGE_HEADER, no *message_options* qualifiers are set. For example, if an invalid value is specified for the *type* field of a query request, the final response is returned to the requestor with a *status* of STATUS_INVALID_TYPE, *type* of TYPE_NONE, and *identifier* undefined.

TYPE and *count* are included for certain commands only; they are omitted from the response if they were not required in the request. See *Chapter 4: ACSLM Command Structures* for the individual command descriptions.

This form of a final response is also returned for a cancelled audit, eject, enter, or query request that was pending in the ACSLM. All fields through *count* are returned for these commands.

COMMON STATUSES

This section identifies the *status* values that are common to all requests. These statuses can appear in either the fixed portion or the variable portion of a response. The statuses are listed in alphabetical order. (See the *db_defs.h Header File* in *Appendix B* for the "C" enumeration values.)

- Any of the following:

  ```
  STATUS_ACS_NOT_IN_LIBRARY
  STATUS_LSM_NOT_IN_LIBRARY
  STATUS_DRIVE_NOT_IN_LIBRARY
  STATUS_PORT_NOT_IN_LIBRARY
  ```

  if the respective *identifier* values are not found in the data base.

  For example, if LSM 5 was not configured in ACS 1, any IDENTIFIER specifying LSM 5 causes a *status* value of STATUS_LSM_NOT_IN_LIBRARY to be returned, with *type* set to TYPE_LSM, and IDENTIFIER set to an LSMID with an *acs_number* of 1 and an *lsm_number* of 5.

- STATUS_AUDIT_IN_PROGRESS (in the *message_status* or *status)* if another request attempts to access a cell locked by a current audit request.
- STATUS_CONFIGURATION_ERROR, and an unsolicited message is issued, if the ACSLM detects an inconsistency between the data base and the physical library configuration.
- STATUS_COUNT_TOO_LARGE if the *count* is greater than allowed.
- STATUS_COUNT_TOO_SMALL if the *count* is less than allowed.
- STATUS_DATABASE_ERROR if the ACSLM detects a data base consistency error during request processing.
- Either of the following:

  ```
  STATUS_DRIVE_OFFLINE
  STATUS_LSM_OFFLINE
  ```

  if the specified component is in the STATE_OFFLINE_PENDING or STATE_OFFLINE states, or in the STATE_DIAGNOSTIC state and the request originated from a CSI.

  For example, if volume ABC123 is specified in a mount request and the LSM it resides in is STATE_OFFLINE, a STATUS_LSM_OFFLINE *status* value is returned.

- Any of the following:

  ```
  STATUS_INVALID_ACS
  STATUS_INVALID_LSM
  STATUS_INVALID_PANEL
  STATUS_INVALID_DRIVE
  STATUS_INVALID_ROW
  STATUS_INVALID_COLUMN
  STATUS_INVALID_VOLUME
  STATUS_INVALID_PORT
  STATUS_INVALID_MESSAGE
  ```

  if any of the variables in the IDENTIFIER have incorrect syntax.

For example, if the LSM identifier in a *panel_id* exceeds MAX_LSM, a STATUS_INVALID_LSM value is returned in *status*, *type* is set to TYPE_LSM, and IDENTIFIER contains an LSMID.

- STATUS_INVALID_COMMAND if *command* is not COMMAND_AUDIT, COMMAND_CANCEL, COMMAND_DISMOUNT, COMMAND_EJECT, COMMAND_ENTER, COMMAND_IDLE, COMMAND_MOUNT, COMMAND_QUERY, COMMAND_START or COMMAND_VARY.
- STATUS_INVALID_OPTION if a value other than FORCE, is set in *message_options*.
- STATUS_INVALID_TYPE if the request specifies a device or object *type* that is not valid for any command.
- STATUS_IPC_FAILURE if the CSI IPC communication mechanism cannot accept a pending message from the ACSLM after an appropriate number of retries with timeouts.
- STATUS_LIBRARY_BUSY if the ACSLM is unable to communicate with the library after retries and time-outs.
- STATUS_LIBRARY_FAILURE, and an unsolicited message is issued, if request processing fails due to failure of a library component. If a tape cartridge is involved, it is returned to an available location.
- STATUS_LIBRARY_NOT_AVAILABLE if a request other than a query server is received while the ACSLM is in the STATE_RECOVERY state, or if a request requiring library resources is received while the ACSLM is in the STATE_IDLE or STATE_IDLE_PENDING states.
- STATUS_MESSAGE_NOT_FOUND if the *message_id* is not a current or pending request.
- STATUS_MESSAGE_TOO_LARGE if the message size calculated by the ACSLM exceeds the received number of bytes.
- STATUS_MESSAGE_TOO_SMALL if the message size calculated by the ACSLM is less than the received number of bytes.
- STATUS_PROCESS_FAILURE if the ACSLM cannot spawn a process to handle a request, or a spawned process fails.
- STATUS_UNSUPPORTED_OPTION if FORCE is set in *message_options* for a command that does not support it.
- STATUS_UNSUPPORTED_TYPE if the request specifies a device or object *type* that is not supported for a particular command.
- STATUS_VOLUME_NOT_IN_LIBRARY (in *message_status* or *identifier_status*) if the specified volume identifier is not found in the data base.

COMMON VARIABLES

This section describes common variables used by the ACSLM. Minimums and maximums define the range of possible values for a variable.

ACS

Description

ACS uniquely identifies an ACS within a library.

Definition

ACS is defined as:

```
typedef unsigned char ACS;
```

ACS has a range from MIN_ACS to MAX_ACS which are defined as:

```
#define MIN_ACS 0
#define MAX_ACS 255
```

CAPID

Description

CAPID uniquely identifies a CAP within a library. The CAP identifier has the same value as the LSM identifier to which the CAP is attached.

Definition

CAPID is defined as:

```
typedef LSMID CAPID;
```

CAP_SIZE

Description

CAP_SIZE is the number of storage cells in a CAP.

Definition

The CAP storage cells are arranged in three rows of seven columns. Therefore, CAP_SIZE is defined as:

```
#define CAP_SIZE 21
```

CELLID

Description

CELLID uniquely identifies a storage cell within an ACS.

Definition

CELLID contains an ACS identifier, an LSM number, a panel number, a panel row number, and a panel column number:

```
typedef struct {
     PANELID   panel_id;
     ROW       row;
     COL       col;
} CELLID;
```

row is defined as:

```
typedef unsigned char ROW;
```

row has a range from MIN_ROW to MAX_ROW which are defined as:

```
#define MIN_ROW 0
#define MAX_ROW 14
```

col is defined as:

```
typedef unsigned char COL;
```

col has a range from MIN_COLUMN to MAX_COLUMN which are defined as:

```
#define MIN_COL 0
#define MAX_COL 23
```

COMMAND

Description

COMMAND uniquely identifies an operation to the ACSLM.

Values

COMMAND can be any of the following. (See the *defs.h Header File* in *Appendix B* for the "C" enumeration values.)

```
COMMAND_ABORT
COMMAND_AUDIT
COMMAND_CANCEL
COMMAND_DISMOUNT
COMMAND_EJECT
COMMAND_ENTER
COMMAND_IDLE
COMMAND_MOUNT
COMMAND_QUERY
```

```
COMMAND_RECOVERY
COMMAND_START
COMMAND_TERMINATE
COMMAND_UNSOLICITED_EVENT
COMMAND_VARY
```

COMMAND_ABORT, COMMAND_RECOVERY, COMMAND_TERMINATE, and COMMAND_UNSOLICITED_EVENT are for ACSLM internal use only. All other COMMAND values can be generated by a client application.

DRIVEID

Description

DRIVEID uniquely identifies a library drive within the library.

Definition

DRIVEID contains an ACS identifier, an LSM number, a panel number, and a drive number:

```
typedef struct {
    PANELID panel_id;
    DRIVE   drive_number;
} DRIVEID;
```

drive_number is defined as:

```
typedef unsigned char DRIVE;
```

drive_number has a range from MIN_DRIVE to MAX_DRIVE which are defined as:

```
#define MIN_DRIVE 0
#define MAX_DRIVE 3
```

FREECELLS

Description

FREECELLS is the number of unoccupied cells in an ACS or LSM.

Definition

FREECELLS is defined as:

```
typedef unsigned long FREECELLS;
```

FUNCTION

Description

FUNCTION is an action performed by the ACSLM.

Values

FUNCTION can have the following values:

```
audit
cancel
dismount
eject
enter
idle
initiation
mount
query
recovery
start
termination
```

LOCATION

Description

LOCATION describes the class of storage location.

Values

LOCATION can be any of the following. (See the *db_defs.h Header File* in *Appendix B* for the "C" enumeration values.)

```
LOCATION_CELL
LOCATION_DRIVE
```

LSMID

Description

LSMID uniquely identifies an LSM within a library.

Definition

LSMID contains an ACS identifier and an LSM number.

```
typedef struct {
    ACS acs_number;
    LSM lsm_number;
} LSMID;
```

lsm_number is defined as:

```
typedef unsigned char LSM;
```

lsm_number has a range from MIN_LSM to MAX_LSM which are defined as:

```
#define MIN_LSM 0
#define MAX_LSM 15
```

MAX_ACS_DRIVES

Description

MAX_ACS_DRIVES is the maximum number of drives in the library.

Definition

MAX_ACS_DRIVES is defined as:

```
#define MAX_ACS_DRIVES 128
```

MAX_ID

Description

MAX_ID is the maximum number of identifiers that can be specified in a request.

Definition

MAX_ID is defined as:

```
#define MAX_ID  21
```

MAX_MESSAGE_SIZE

Description

MAX_MESSAGE_SIZE is the maximum size of an IPC request or response packet.

Definition

MAX_MESSAGE_SIZE is defined as:

```
#define MAX_MESSAGE_SIZE  4096
```

MAX_PORTS

Description

MAX_PORTS is the maximum number of ports per ACS.

Definition

MAX_PORTS is defined as:

```
#define MAX_PORTS 16
```

MESSAGE_ID

Description

MESSAGE_ID is a unique number generated and used by the ACSLM to identify a specific request from a CSI or the ACSSA.

Definition

MESSAGE_ID is defined as:

```
typedef unsigned short MESSAGE_ID;
```

MESSAGE_ID has a range from MIN_MESSAGE to MAX_MESSAGE which are defined as:

```
#define MIN_MESSAGE 1
#define MAX_MESSAGE 65535
```

PANELID

Description

PANELID uniquely identifies a storage panel within an ACS.

Definition

The panel identifier contains an ACS identifier, an LSM number, and a panel number:

```
typedef struct {
     LSMID lsm_id;
     PANEL panel_number;
} PANELID;
```

panel_number is defined as:

```
typedef unsigned char PANEL;
```

panel_number values can range from MIN_PANEL to MAX_PANEL which are defined as:

```
#define MIN_PANEL 0
#define MAX_PANEL 19
```

PORTID

Description

PORTID uniquely identifies a single communication line between a server system communications port and an LMU host interface. The ACSLM uses the communication line to interact with an ACS.

Definition

PORT_ID contains an ACS identifier and a port number:

```
typedef struct {
     ACS  acs_id;
     PORT port_number;
} PORTID;
```

port_number is defined as:

```
typedef unsigned char PORT;
```

port_number values can range from MIN_PORT to MAX_PORT which are defined as:

```
#define MIN_PORT 0
#define MAX_PORT 15
```

STATE

Description

STATE describes the characteristics that a device or Storage Server component can take.

Values

STATE can be any of the following. (See the *db_defs.h Header File* in *Appendix B* for the "C" enumeration values.)

```
STATE_CANCELLED
STATE_DIAGNOSTIC
STATE_IDLE
STATE_IDLE_PENDING
STATE_OFFLINE
STATE_OFFLINE_PENDING
STATE_ONLINE
STATE_RECOVERY
STATE_RUN
```

STATE_CANCELLED describes a state of a request.

STATE_ONLINE, STATE_OFFLINE, STATE_OFFLINE_PENDING, and STATE_DIAGNOSTIC describe the different states of an ACS, LSM, or library drive.

STATE_ONLINE and STATE_OFFLINE describe the different states of a port.

STATE_RUN, STATE_IDLE, STATE_IDLE_PENDING, and STATE_RECOVERY describe the different states of the ACSLM.

SUBPANELID

Description

SUBPANELID uniquely identifies a subset of cells within a panel. It specifies the upper-left corner (startrow, startcolumn) and lower-right corner (endrow, endcolumn) of a rectangular area wholly contained in a single panel.

Definition

SUBPANELID contains an ACS identifier, an LSM number, a panel number, a starting row number, a starting column number, an ending row number, and an ending column number:

```
typedef struct {
     PANELID   panel_id;
     ROW       begin_row;
     COL       begin_col;
     ROW       end_row;
     COL       end_col;
} SUBPANELID;
```

TYPE

Description

TYPE identifies a class of devices, Storage Server software components, or commands.

Values

Valid TYPE values and their definitions are as follows. (See the *defs.h Header File* in *Appendix B* for the "C" enumeration values.)

TYPE_ACS	ACS
TYPE_AUDIT	audit request process
TYPE_CAP	Cartridge Access Port (CAP)
TYPE_CELL	Cell identifier
TYPE_CP	ACSSA command process
TYPE_CSI	CSI
TYPE_DISMOUNT	dismount request process
TYPE_EJECT	eject request process
TYPE_EL	Event Logger
TYPE_ENTER	enter request process

TYPE_DRIVE	Library tape drive
TYPE_IP	Interprocess communication
TYPE_LH	ACS Library Handler
TYPE_LM	ACS Library Manager (ACSLM)
TYPE_LSM	LSM
TYPE_MOUNT	mount request process
TYPE_NONE	no identifier specified
TYPE_PANEL	LSM panel
TYPE_PORT	ACS communication port
TYPE_QUERY	query request process
TYPE_RECOVERY	Storage Server recovery process
TYPE_REQUEST	Storage Server request
TYPE_SA	ACS System Administrator (ACSSA)
TYPE_SERVER	Storage Server
TYPE_SUBPANEL	LSM subpanel
TYPE_VARY	vary request process
TYPE_VOLUME	Tape cartridge

VOLID

Description

VOLID is the external tape cartridge label which uniquely identifies a tape cartridge within an ACS.

Definition

VOLID is defined as:

```
typedef struct {
    char external_label[EXTERNAL_LABEL_SIZE+1];
} VOLID;
```

EXTERNAL_LABEL_SIZE is defined as:

```
#define EXTERNAL_LABEL_SIZE 6
```

VOLID is a string of characters less than or equal to EXTERNAL_LABEL_SIZE characters long and null terminated. The uppercase characters ("A" through "Z"), digits ("0" through "9"), and blank (" ") are valid. VOLIDs containing embedded blanks are reserved for diagnostic and maintenance cartridges only; leading and trailing blanks are ignored.

CHAPTER 4:
ACSLM COMMAND STRUCTURES

OVERVIEW

This chapter illustrates the data structures for each command processed by the ACSLM. The ACSLM supports the following commands:

```
audit
cancel
dismount
eject
enter
idle
mount
query
start
vary
```

For each command, the following data structures are shown:

- Request structures and values
- Intermediate response structures and values, where applicable
- Final response structures and values
- Final response structures and values for *cancelled* requests, where applicable

See *Chapter 2: ACSLM Processes* for a description of how the ACSLM processes requests. See *Chapter 3: ACSLM Common Data Structures* for data structures that are common to all requests and responses.

AUDIT

Name

`audit` – Performs a physical inventory on the tape cartridges in a library, ACS, LSM, panel, or subpanel.

Description

The `audit` request performs a physical inventory of one or more specified ACSs, LSMs, LSM panels, or LSM subpanels. It can be used to resolve inconsistencies between the data base and the physical contents of the library. These inconsistencies may be the result of a person physically entering the LSM and manually adding, removing, or moving cartridges in the storage cells.

Only one device type at a time can be audited. Within that type, up to `MAX_ID` different devices can be specified in each request.

The LSM robot physically scans each cell in the specified object. The cell contents are compared with the contents recorded in the data base. If there is a difference, or if the robot finds a duplicate or unreadable external label, the physical contents of the cell are *rechecked.* If there is still a discrepancy, the data base is corrected and a record of the change is written to the Event Log. Through this process, the data base is updated to reflect the observed contents in cell storage. It is recommended that the data base be backed up after the completion of an audit.

Any cartridges with duplicate or unreadable external labels are ejected through the specified CAP. The CAP is reserved for the entire audit. The audit process does not begin ejecting cartridges until after it has made all necessary data base updates.

Concurrent audits are allowed as long as they do not overlap one another. They are not recommended, however, when duplicate volumes are suspected.

Requests

Request Format

```
struct audit_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     CAPID                         cap_id;
     TYPE                          type;
     unsigned short                count;
     union {
          ACS                      acs_id;
          LSMID                    lsm_id;
          PANELID                  panel_id;
          SUBPANELID               subpanel_id;
     } identifier;
};
```

Request Values

Within MESSAGE_HEADER, *command* is set to COMMAND_AUDIT.

cap_id is the CAP used for ejection of cartridges.

type is the type of object to audit. Only one *type* can be specified in a single request. *type* is one of the following:

```
TYPE_SERVER
TYPE_ACS
TYPE_LSM
TYPE_PANEL
TYPE_SUBPANEL
```

count is the number of identifiers that follows. For TYPE_SERVER, *count* must equal 1. For all other *types*, *count* must equal 1 to MAX_ID.

Within identifier, *acs_id*, *lsm_id*, *panel_id*, or *subpanel_id* is the unique ID of the object to be audited.

Responses

Intermediate Response Format

Part of audit processing is ejection of tape cartridges. The ACSLM returns an intermediate response when a tape cartridge is added or deleted from the data base. In the case of duplicate or unreadable labels, the tape cartridge is also physically ejected from the library through the CAP specified in the request.

The format of the intermediate response is:

```
struct eject_enter {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
     CAPID                         cap_id;
     unsigned short                count;
     struct {
         VOLID                     vol_id;
         RESPONSE_STATUS           status;
     } volume_status;
};
```

Intermediate Response Values

Within MESSAGE_HEADER:

- *command* is COMMAND_AUDIT.
- *message_options* is set to INTERMEDIATE, indicating a partial response to the audit request.

Within RESPONSE_STATUS, *status* is STATUS_AUDIT_ACTIVITY.

cap_id is the identifier of the CAP, specified in the request, that is used to eject cartridges.

count is the number of tape cartridges added, deleted, or ejected associated with this response. The number ranges from 1 to MAX_ID. See the *Common Variables* section in *Chapter 3* for a description of MAX_ID.

Within volume_status, *vol_id* is the external tape cartridge label.

Within volume_status, *status* is the disposition of each *vol_id*. Following are *status* values specific to this command.

- STATUS_DUPLICATE_LABEL if the robot finds a tape cartridge with a duplicate external label; the cartridge is ejected from the LSM.
- STATUS_VOLUME_ADDED if the robot finds a tape cartridge that is not listed in the data base; the *vol_id* is added to the data base.
- STATUS_VOLUME_NOT_IN_LIBRARY if a tape cartridge listed in the data base is not found in the library; the *vol_id* is deleted from the data base.
- STATUS_UNREADABLE_LABEL if the robot finds a tape cartridge with an unreadable external label; the cartridge is ejected from the library. Within *vol_id* question marks (?) are substituted for the characters that the robot was unable to read.

Final Response Format

```
struct audit_response {
    IPC_HEADER                  ipc_header;
    MESSAGE_HEADER              message_header;
    RESPONSE_STATUS             message_status;
    CAPID                       cap_id;
    TYPE                        type;
    unsigned short              count;
    union {
        struct {
           ACS                  acs_id;
           RESPONSE_STATUS      status;
        } acs_status;
        struct {
           LSMID                lsm_id;
           RESPONSE_STATUS      status;
        } lsm_status;
        struct {
           PANELID              panel_id;
           RESPONSE_STATUS      status;
        } panel_status;
        struct {
           SUBPANELID           subpanel_id;
           RESPONSE_STATUS      status;
        } subpanel_status;
    } identifier_status;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_AUDIT.

Within RESPONSE_STATUS, *status* is the disposition of the entire request. Following are *status* values specific to this command.

- STATUS_AUDIT_FAILED if the request fails during audit processing. See the identifier_status for the nature of the failure.
- STATUS_AUDIT_IN_PROGRESS, for TYPE_SERVER *only*, if another in-process audit is already auditing the library.
- STATUS_CANCELLED if the request is cancelled. See *Final Response Values – Cancelled Request.*
- STATUS_CAP_IN_USE if the specified CAP is being used by an eject, enter, or another audit request.
- STATUS_MULTI_ACS_AUDIT if the request identifier list spans more than one ACS.
- STATUS_NOT_IN_SAME_ACS if the *cap_id* and the identifier do not specify the same ACS.
- STATUS_SUCCESS if the audit completed successfully.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_CONFIGURATION_ERROR
- STATUS_COUNT_TOO_LARGE
- STATUS_COUNT_TOO_SMALL
- STATUS_DATABASE_ERROR
- STATUS_INVALID_ACS
- STATUS_INVALID_LSM
- STATUS_INVALID_OPTION
- STATUS_INVALID_TYPE
- STATUS_LIBRARY_BUSY
- STATUS_LIBRARY_FAILURE
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_LSM_OFFLINE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FAILURE
- STATUS_UNSUPPORTED_OPTION
- STATUS_UNSUPPORTED_TYPE

cap_id is the identifier of the CAP, specified in the request, that is used to eject cartridges.

type is the type of object being audited, as indicated in the request.

count is the number of IDENTIFIERS, as indicated in the request. For TYPE_SERVER, *count* is always 0.

Within identifier_status, *acs_id*, *lsm_id*, *panel_id*, or *subpanel_id* is the unique ID of the object being audited, as indicated in the request.

Within identifier_status, *status* is the disposition of the identifier. Following are *status* values specific to this command.

- STATUS_AUDIT_IN_PROGRESS if another in-process audit is already auditing the LSM specified in identifier. Only one audit may be active on any single LSM.
- STATUS_CANCELLED if the request is cancelled. See *Final Response Values – Cancelled Request.*
- STATUS_DUPLICATE_IDENTIFIER if the IDENTIFIER duplicates or overlaps a previous IDENTIFIER in the list.

- STATUS_INVALID_SUBPANEL if the subpanel is not correctly specified. A subpanel identifier indicates the upper-left and lower-right corners of a panel subsection. The ending row must be > the beginning row, and the ending column must be > the beginning column. This status is returned if these conditions are not met.
- STATUS_VALID if the IDENTIFIER was successfully validated.

Following are common identifier_status *status* values. See the *Common Statuses* section in *Chapter 3* for a description.

- STATUS_INVALID_ACS
- STATUS_INVALID_COLUMN
- STATUS_INVALID_LSM
- STATUS_INVALID_PANEL
- STATUS_INVALID_ROW
- STATUS_LSM_OFFLINE
- STATUS_LSM_NOT_IN_LIBRARY

Final Response Values – Cancelled Request

Within MESSAGE_HEADER, *command* is COMMAND_AUDIT.

Within RESPONSE_STATUS, the *status* is STATUS_CANCELLED.

cap_id is the identifier of the CAP, specified in the request, that is used to eject cartridges.

type is the type of object being audited, as indicated in the request.

count is the number of identifiers that were validated prior to the cancel. For a cancelled *pending* request *count* is 0 and no identifier_status records follow. For a cancelled *current* request, *count* ranges from 1 to MAX_ID. See *Common Variables* section in *Chapter 3* for a definition of MAX_ID.

Within identifier_status, the *status* for each identifier is set to STATUS_VALID.

Notes

- Messages are written to the Event Log whenever an audit starts, terminates, or is cancelled.
- A STATUS_REMOVE_CARTRIDGES unsolicited message is sent to the ACSSA if the CAP becomes full during audit processing. Audit processing is suspended until the CAP is unloaded and resumes when the CAP is closed.

- A STATUS_REMOVE_CARTRIDGES unsolicited message is sent to the ACSSA if a `cancel` request is issued against a current `audit` request while it is ejecting cartridges. An `audit` cannot be cancelled once this message has been issued.

See Also

- The `cancel` command for details on cancelling a pending or current `audit` request.
- *Appendix A* for entries that may be written to the Event Log.

CANCEL

Name

`cancel` – Terminates a current or pending `audit`, `eject`, `enter`, or `query` request.

Description

The `cancel` request terminates current or pending `query`, `eject`, `enter`, or `audit` activity. A cancellation function is provided for these requests because they can take an extended period of time to complete.

The cancelled request may continue to run while it releases allocated resources. Activity is terminated before the next LMU command is issued. Current LMU commands are processed to completion. No attempt is made to undo any activity that was completed *before* the `cancel` request was received by the ACSLM.

The requestor must know the request ID of the request to cancel. The request ID is included in the acknowledging response. It can also be determined by issuing a `query` request, which returns a list of all current and pending requests.

When *pending* requests are cancelled they are removed from the ACSLM's request queue and are not processed.

When *current* requests are cancelled the following activities occur:

- *Cancelling a current* `audit` – The audit halts, and, if cartridges have been moved to the CAP, a message to remove the cartridges is displayed. Cartridges already ejected are not reentered. Cancelling a current audit may result in inconsistencies between the data base and the actual physical contents of the LSM.
- *Cancelling a current* `eject` – The eject is halted, and a message to remove the cartridges is displayed. After the CAP is emptied and closed, a message indicating the number of cartridges acted on is displayed. Cartridges already ejected are not reentered.
- *Cancelling a current* `enter` – The enter is halted. If cartridges are in the CAP, a message to remove the cartridges is displayed. After the CAP is emptied, a message indicating the number of cartridges acted on is displayed. Cartridges already entered into the LSM are not ejected.
- *Cancelling a current* `query` – The processing of status information is aborted.

Requests

Request Format

```
struct cancel_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     MESSAGE_ID                    request_id;
};
```

Request Values

Within MESSAGE_HEADER, *command* is COMMAND_CANCEL.

request_id is the message ID of the request to cancel.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct cancel_response {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
     MESSAGE_ID                    request_id;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_CANCEL.

Within RESPONSE_STATUS, *status* is the disposition of the entire request. Following are *status* values specific to this command.

- STATUS_INVALID_MESSAGE if the value of *request_id* is outside the range of valid values, as defined by MESSAGE_ID, or if the *request_id* specified designates a command that is *not* an audit, eject, enter, or query. See *Common Variables* in *Chapter 3* for a definition of MESSAGE_ID.
- STATUS_MESSAGE_NOT_FOUND if *request_id* is valid but is not a current or pending request.
- STATUS_SUCCESS if the *request_id* was cancelled successfully.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_INVALID_OPTION
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_UNSUPPORTED_OPTION

request_id is the request ID, as indicated in the cancel request.

Final Response Values – Cancelled Request

Not applicable.

Notes

- A query server request cannot be cancelled.
- A cancelled request may continue to run while it releases allocated resources. Refer to the descriptions of the audit, eject, enter, and query requests for additional information on the effects of a cancel request. In all cases, the specified request is cancelled and the RESPONSE_STATUS *status* STATUS_SUCCESS is immediately returned to the request originator.

See Also

- The query, eject, enter, and audit commands for details on their functions.
- *Appendix A* for entries that may be written to the Event Log.

DISMOUNT

Name

`dismount` – Dismounts a tape cartridge from a library drive.

Description

The `dismount` request dismounts a tape cartridge from a library drive.

The *message_option* FORCE is used to automatically dismount the tape cartridge from the specified library drive, even if its *vol_id* does not match that in the drive, and even if the drive is not unloaded.

Upon receipt of a `dismount` request, the LSM robot does the following:

1. Moves to the specified drive and validates the external tape cartridge label.
2. Dismounts the tape cartridge from the drive.
3. Returns the tape cartridge to an available storage cell in the library.

Once the dismount is completed, the data base is updated with the new location of the cartridge.

Unforced Dismount

All of the following conditions must be met for a successful unforced dismount:

- Both the cartridge and the drive must be in the library
- The library drive must be online to ACSLM control
- The cartridge must be in the specified library drive
- The drive must be unloaded

Forced Dismount

When the FORCE *message_option* is used, the system does not verify the tape cartridge label or data base information. It also does not require that the library drive be ready for dismounting. The Storage Server automatically rewinds, unloads, and dismounts whatever tape cartridge is found in the specified library drive.

This option can be used to dismount a cartridge with an unreadable or unknown label, or a cartridge that, for some reason, did not get unloaded by the client application system.

The following conditions must be met for a successful forced dismount:

- The drive must be in the library configuration
- The library drive must be currently online to ACSLM control

Requests

Request Format

```
struct dismount_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     VOLID                         vol_id;
     DRIVEID                       drive_id;
};
```

Request Values

Within MESSAGE_HEADER:

- *command* is COMMAND_DISMOUNT.
- *message_options* can be FORCE. When this is used, the ACSLM does not verify tape cartridge labels or data base information. The ACSLM rewinds, unloads, and dismounts the tape cartridge found in the requested library drive. Therefore, this *message_option* can dismount a cartridge with an unreadable label.

vol_id is the external label of the tape cartridge to be dismounted.

drive_id is the library drive containing the tape cartridge.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct dismount_response {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
     VOLID                         vol_id;
     DRIVEID                       drive_id;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_DISMOUNT.

Within RESPONSE_STATUS, *status* is the disposition of the request. Following are *status* values specific to this command:

- STATUS_ACS_FULL if an available cell location cannot be found in the data base to dismount the cartridge into. The cartridge is left in the tape drive.
- STATUS_AUDIT_IN_PROGRESS, if an in-process audit has locked out access to all available cell records in the data base.
- STATUS_DRIVE_AVAILABLE if the library drive does not contain a tape cartridge.
- STATUS_DRIVE_IN_USE if the dismount fails because the cartridge was not unloaded on the library drive.
- STATUS_MISPLACED_TAPE if the external tape cartridge label of the tape cartridge in the library drive does not match the volume identifier of the tape cartridge in the request. The data base is updated with the volume identifier of the tape cartridge in the library drive.
- STATUS_NOT_IN_SAME_ACS if the tape cartridge and the library drive are not in the same ACS.
- STATUS_SUCCESS if the tape cartridge was dismounted successfully.
- STATUS_UNREADABLE_LABEL if the tape cartridge label is unreadable. If this is an unforced dismount, the request is rejected, and a message is issued to the Event Log. If this is a forced dismount, the cartridge is successfully dismounted. Within *vol_id*, questions marks (?) are substituted for the characters that the robot was unable to read.
- STATUS_VOLUME_NOT_IN_DRIVE if the data base shows that the requested volume identifier is not in the requested drive.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_DATABASE_ERROR
- STATUS_DRIVE_NOT_IN_LIBRARY
- STATUS_DRIVE_OFFLINE
- STATUS_INVALID_ACS
- STATUS_INVALID_DRIVE
- STATUS_INVALID_LSM
- STATUS_INVALID_OPTION
- STATUS_INVALID_VOLUME
- STATUS_LIBRARY_BUSY

- STATUS_LIBRARY_FAILURE
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_LSM_OFFLINE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FAILURE
- STATUS_UNSUPPORTED_OPTION
- STATUS_VOLUME_NOT_IN_LIBRARY

vol_id is the tape cartridge identifier, as indicated in the request.

drive_id is the library tape drive, as indicated in the request.

Final Response Values – Cancelled Request

Not applicable.

Notes

None.

See Also

- The `mount` command for details on mounting a cartridge on a library drive.
- *Appendix A* for entries that may be written to the Event Log.

EJECT

Name

`eject` – Ejects from one to `MAX_ID` tape cartridges from an LSM.

Description

The `eject` request ejects tape cartridges from the library, removing them from library control. Cartridges are ejected through a specified CAP. From one to `MAX_ID` cartridges can be ejected at a time. See *Chapter 3: ACSLM Common Data Structures* for a definition of `MAX_ID`.

For each cartridge to be ejected, the LSM robot does the following:

1. Moves to the cell location indicated in the data base.
2. Reads the external label of the cartridge and verifies that it is the specified cartridge,
3. Moves the cartridge to an available cell in the specified CAP.

When the specified cartridges have been moved to the CAP, the cell locations of the ejected cartridges are deassigned, and the cartridges are removed from the data base. The Command Processor then displays an unsolicited message in the Display Area to remove the cartridges from the CAP.

After the cartridges are removed and the CAP door is closed, the Command Processor displays one message in the Command Area for each cartridge designated for ejection. In each message, the status indicates whether or not the cartridge was actually ejected.

If, for any reason, a specified cartridge cannot be ejected, an error message is displayed in the Command Area, and an entry is made in the Event Log.

Requests

Request Format

```
struct eject_request {
    IPC_HEADER                  ipc_header;
    MESSAGE_HEADER              message_header;
    CAPID                       cap_id;
    unsigned short              count;
    VOLID                       vol_id;
};
```

Request Values

Within MESSAGE_HEADER, *command* is set to COMMAND_EJECT.

cap_id is the CAP used to eject the tape cartridge.

count is the number of *vol_ids* to eject. *count* has a range of 1 to MAX_ID.

Each *vol_id* is the external label of a tape cartridge to be ejected.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct eject_response {
    IPC_HEADER                  ipc_header;
    MESSAGE_HEADER              message_header;
    RESPONSE_STATUS             message_status;
    CAPID                       cap_id;
    unsigned short              count;
    struct {
        VOLID                   vol_id;
        RESPONSE_STATUS         status;
    } volume_status;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_EJECT.

Within RESPONSE_STATUS, *status* indicates the disposition of the entire request. Following are *status* values specific to this command.

- STATUS_CANCELLED if the request is cancelled. See *Final Response Values – Cancelled Request.*

- STATUS_CAP_IN_USE if the request cannot be processed because the CAP is being used by an audit, an enter, or another eject request.
- STATUS_SUCCESS if the fixed portion of the request is correct.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_COUNT_TOO_LARGE
- STATUS_COUNT_TOO_SMALL
- STATUS_DATABASE_ERROR
- STATUS_INVALID_ACS
- STATUS_INVALID_LSM
- STATUS_INVALID_OPTION
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_LSM_BUSY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_LSM_OFFLINE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FALURE
- STATUS_UNSUPPORTED_OPTION

cap_id is the CAP used to eject the tape cartridge.

count is the number of tape cartridges that the ACSLM attempted to eject.

Within volume_status, *vol_id* is the external label of each tape cartridge the ACSLM attempted to eject.

Within volume_status, *status* is the disposition of each *vol_id*. Following are *status* values specific to this command.

- STATUS_AUDIT_IN_PROGRESS, if an in-process audit has locked out access to a cell record in the data base. The cartridges are not ejected. This *status* is issued only after the data base has attempted retries on the cell record.
- STATUS_CAP_FULL if the CAP is full before all *vol_ids* have been processed. Remaining *vol_ids* are not ejected.
- STATUS_MISPLACED_TAPE if a different tape cartridge is in the location specified by the data base. The data base is updated with the external tape cartridge label of the tape cartridge found in the storage location.

- STATUS_NOT_IN_SAME_ACS if the CAP identifier and volume identifier are not in the same ACS.
- STATUS_SUCCESS if the tape cartridge was ejected successfully.
- STATUS_VOLUME_IN_DRIVE if the tape cartridge is in a library drive. The cartridge is not ejected.
- STATUS_VOLUME_IN_USE if the tape cartridge is in use by another request. The cartridge is not ejected.
- STATUS_VOLUME_NOT_IN_LIBRARY if the *vol_id* does not exist in the data base, or if the tape cartridge has already been ejected as part of this request. If no tape cartridge is in the location specified by the data base and the volume is not in transit or in a library drive, the data base entry is removed.

Following are common volume_status *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_INVALID_VOLUME
- STATUS_LIBRARY_FAILURE
- STATUS_LIBRARY_BUSY
- STATUS_LSM_OFFLINE

Final Response Values – Cancelled Request

Within MESSAGE_HEADER, *command* is COMMAND_EJECT.

Within RESPONSE_STATUS, *status* is STATUS_CANCELLED.

cap_id is the CAP used to eject the tape cartridges.

count is the number of tape cartridges to be ejected, as specified in the request.

Within volume_status, *vol_id* is the external label of each tape cartridge the ACSLM attempted to eject.

Within volume_status, *status* is as follows:

- STATUS_CANCELLED if the tape cartridge was *not* processed before the request was cancelled.
- Any valid *status* value if the tape cartridge was processed before the request was cancelled. See *Final Response Values* above for the valid final response *status* values.

Notes

- If all *vol_ids* have been processed and at least one cartridge has been moved to the CAP, or the CAP is full, a `STATUS_REMOVE_CARTRIDGES` unsolicited message is sent to the ACSSA. The final response is not returned until the CAP is closed.

See Also

- The `cancel` command for details on cancelling a current or pending `eject` request.
- The `enter` command for details on entering cartridges into the ACS.
- *Appendix A* for entries that may be written to the Event Log.

ENTER

Name

`enter` – Enters one to `MAX_ID` tape cartridges into an LSM.

Description

The `enter` request enters tape cartridges into the ACS, placing them under library control. The cartridges are entered through a specified CAP. From one to `MAX_ID` cartridges can be entered at a time.

Upon receiving an `enter` request, the CAP is unlocked, and an unsolicited message is displayed, instructing the operator to place the cartridges in the CAP.

The cartridges should be loaded from left to right, top to bottom, starting with the upper left-most CAP cell. The first row should be filled completely before beginning with the next row, etc. The robot stops looking for cartridges in the CAP once it encounters an empty CAP cell, so skipping CAP cells will cause all cartridges after the empty cell not to be entered into the LSM.

After the CAP is closed, the LSM robot does the following for each cartridge in the CAP:

1. Verifies that the external label is readable.
2. Verifies that the label is not a duplicate.
3. Moves the cartridge from the CAP to an unassigned cell in the ACS.
4. Assigns the location to the cartridge and adds the cartridge to the data base.

Once the robot has finished unloading the CAP, the external label of each cartridge found in the CAP is displayed in the Command Area. For each cartridge in the list, the status indicates whether or not the cartridge was actually entered into the LSM.

If any cartridges have unreadable or duplicate labels, the CAP is unlocked, and an unsolicited message is displayed, instructing the operator to remove the cartridges from the CAP; this is done after all cartridges that could be successfully entered are moved into the LSM.

Requests

Request Format

```
struct enter_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     CAPID                         cap_id;
};
```

Request Values

Within MESSAGE_HEADER, *command* is COMMAND_ENTER.

cap_id is the CAP used to enter the tape cartridges.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct enter_response {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
     CAPID                         cap_id;
     unsigned short                count;
     struct {
         VOLID                     vol_id;
         RESPONSE_STATUS           status;
     } volume_status;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_ENTER.

Within RESPONSE_STATUS, *status* indicates the disposition of the entire request. Following are *status* values specific to this command.

- STATUS_CANCELLED if the request is cancelled. See *Final Response Values – Cancelled Request.*
- STATUS_CAP_IN_USE, if the CAP is being used by an `audit`, an `eject`, or another `enter` request.
- STATUS_SUCCESS, if the fixed portion of the request is correct.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_DATABASE_ERROR
- STATUS_INVALID_ACS
- STATUS_INVALID_LSM
- STATUS_INVALID_OPTION
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_LSM_OFFLINE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FAILURE
- STATUS_UNSUPPORTED_OPTION

cap_id is the CAP used to enter the tape cartridges.

count is the number of tape cartridges that the ACSLM attempted to enter into the library. The value is the same as for the request. *count* has a range from 1 to MAX_ID (see the *Common Variables* section in *Chapter 3* for a definition of MAX_ID).

Within volume_status, each *vol_id* is the external label of the tape cartridge that the ACSLM attempted to enter.

Within volume_status, *status* indicates the disposition of each *vol_id*. Following are *status* values specific to this command:

- STATUS_ACS_FULL, if there are no storage cells available in the library for the tape cartridge. The tape cartridge is left in the CAP.
- STATUS_AUDIT_IN_PROGRESS, if an in-process audit has locked out access to a cell record in the data base. The cartridges are left in the CAP. This *status* is issued only after the data base has attempted retries on the cell record.
- STATUS_DUPLICATE_LABEL, if the external label on the tape cartridge is a duplicate of one that already exists in the library. The tape cartridge is left in the CAP and is not entered into the library.
- STATUS_SUCCESS, if the tape cartridge was entered successfully.
- STATUS_UNREADABLE_LABEL, if the external tape cartridge label is unreadable. The tape cartridge is left in the CAP and is not entered into the library. Within *vol_id*, question marks (?) are substituted for the characters the robot was unable to read.

Following are common `volume_status` *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- `STATUS_INVALID_VOLUME`
- `STATUS_LIBRARY_BUSY`
- `STATUS_LIBRARY_FAILURE`
- `STATUS_LSM_OFFLINE`

Final Response Values – Cancelled Request

Within `MESSAGE_HEADER`, *command* is `COMMAND_ENTER`.

Within `RESPONSE_STATUS`, *status* is `STATUS_CANCELLED`.

cap_id is the CAP used to enter the tape cartridges.

count is the number of tape cartridges that the ACSLM acted upon before receiving the `cancel` request. Any cartridges not processed are *not* included in the response.

Within `volume_status`, each *vol_id* is the external label of the tape cartridge that the ACSLM processed.

Within `volume_status`, *status* can be any of the values that are valid for a final response.

Notes

- When the ACSLM begins processing the request, it sends a `STATUS_INPUT_CARTRIDGES` unsolicited message to the ACSSA.
- If the ACSLM does not enter all cartridges in the request, cartridges remain in the CAP. The ACSLM sends a `STATUS_REMOVE_CARTRIDGES` unsolicited message to the ACSSA.
- If a `cancel` request is issued against a current `enter` request, enter processing is halted for that request.
- If any cartridges are left in the CAP after a request is cancelled, the ACSLM issues a `STATUS_REMOVE_CARTRIDGES` unsolicited message to the ACSSA and waits for the operator to remove the cartridges before returning the final response.

See Also

- The `cancel` command for details on cancelling a current or pending `enter` request.
- The `eject` command for details on removing cartridges from the ACS.
- *Appendix A* for entries that may be written to the Event Log.

IDLE

Name

`idle` – Stops ACSLM request processing.

Description

The `idle` request is used to place the Storage Server in a quiescent state prior to maintenance activity. It will remain in that state until it receives a `start` request.

The Storage Server can be in one of the four following states: `STATE_RUN`, `STATE_IDLE`, `STATE_IDLE_PENDING`, and `STATE_RECOVERY`. See the *Library Request Processing* section in *Chapter 2* for details on these states.

Unforced Idle

Upon receipt of an unforced `idle` request, the Storage Server is immediately placed in `STATE_IDLE_PENDING`. While the Storage Server is in this state, new requests involving library operations are rejected, and current and pending requests are processed to completion. The Storage Server is not placed in `STATE_IDLE` until all current and pending requests have been completed.

Forced Idle

An `idle` request with the `FORCE` *message_option* abruptly puts the Storage Server in `STATE_IDLE`. Current and pending requests are aborted, not processed to completion. New requests are rejected.

Requests

Request Format

```
struct idle_request {
    IPC_HEADER                  ipc_header;
    MESSAGE_HEADER              message_header;
};
```

Request Values

Within `MESSAGE_HEADER`, *command* is `COMMAND_IDLE`.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct idle_response {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_IDLE.

Within RESPONSE_STATUS, *status* is the disposition of the request. Following are *status* values specific to this command.

- STATUS_IDLE_PENDING if the ACSLM is processing a previous idle request, and the Storage Server is therefore already in STATE_IDLE_PENDING.
- STATUS_LIBRARY_NOT_AVAILABLE if the Storage Server is in STATE_RECOVERY.
- STATUS_SUCCESS if the Storage Server was successfully put in STATE_IDLE.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_INVALID_OPTION
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_UNSUPPORTED_OPTION

Final Response Values – Cancelled Request

Not applicable.

Notes

- The ACSLM sends a STATUS_IDLE_PENDING unsolicited message to the ACSSA when the Storage Server is put in the STATE_IDLE_PENDING state.
- The ACSLM sends a STATUS_ACSLM_IDLE unsolicited message to the ACSSA when the Storage Server is put in the STATE_IDLE state.

See Also

- The `query` command for details on displaying the current state of the Storage Server.
- The `start` command for details on bringing an idle Storage Server into the `STATE_RUN`.
- *Appendix A* for entries that may be written to the Event Log.

MOUNT

Name

`mount` – Mounts a tape cartridge onto a specified library drive.

Description

The `mount` request mounts a specified tape cartridge on a specified library drive.

Upon receipt of a `mount` request, the LSM robot does the following:

1. Moves to the appropriate cell location and validates the external label of the tape cartridge.
2. Mounts the tape cartridge on the drive.

Once the cartridge is successfully mounted, the data base is updated with the status of the drive and the current location of the cartridge.

If for any reason the `mount` cannot be completed, the tape cartridge is returned to its original location, if possible, or to another available storage cell.

All of the following conditions must be met for a successful `mount`:

- Both the cartridge and the drive must be in the library
- The library drive must be online to ACSLM control and unloaded
- The cartridge must be available

Request

Request Format

```
struct mount_request {
     IPC_HEADER                ipc_header;
     MESSAGE_HEADER            message_header;
     VOLID                     vol_id;
     unsigned short            count;
     DRIVEID                   drive_id;
};
```

Request Values

Within `MESSAGE_HEADER`, *command* is `COMMAND_MOUNT`.

vol_id is the external label of the tape cartridge to be mounted.

count is the number of library drives eligible for mounting tape cartridges. Currently, *count* must equal `1`.

drive_id is a library drive on which the tape cartridge is to be mounted.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct mount_response {
     IPC_HEADER                ipc_header;
     MESSAGE_HEADER            message_header;
     RESPONSE_STATUS           message_status;
     VOLID                     vol_id;
     DRIVEID                   drive_id;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_MOUNT.

Within RESPONSE_STATUS, *status* is the disposition of the request. Following are *status* values specific to this command.

- STATUS_AUDIT_IN_PROGRESS, if an in-process audit has locked out access to the specified cartridge's cell record in the data base. This *status* is issued only after the data base has attempted retries on the cell record.
- STATUS_DRIVE_IN_USE if the requested library drive already contains a tape cartridge.
- STATUS_MISPLACED_TAPE if the external tape cartridge label of the tape cartridge found in the location indicated by the data base does not match the *vol_id* in the request. The data base is corrected and a message is written to the Event Log.
- STATUS_NOT_IN_SAME_ACS if the tape cartridge and the tape drive are not in the same ACS.
- STATUS_SUCCESS if the tape cartridge was successfully mounted on the drive.
- STATUS_UNREADABLE_LABEL if the tape cartridge found in the location indicated by the data base has an unreadable external label. A message is written to the Event Log. Within *vol_id*, question marks (?) are substituted for the characters that the robot was unable to read.
- STATUS_VOLUME_IN_DRIVE if the tape cartridge is already mounted in a library drive.
- STATUS_VOLUME_IN_USE if the tape cartridge is marked in the data base as reserved by another request.

- STATUS_VOLUME_NOT_IN_LIBRARY if the *vol_id* specified in the request is not found in the data base.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_DRIVE_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_COUNT_TOO_LARGE
- STATUS_COUNT_TOO_SMALL
- STATUS_DATABASE_ERROR
- STATUS_DRIVE_OFFLINE
- STATUS_LSM_OFFLINE
- STATUS_INVALID_ACS
- STATUS_INVALID_DRIVE
- STATUS_INVALID_LSM
- STATUS_INVALID_OPTION
- STATUS_INVALID_VOLUME
- STATUS_LIBRARY_BUSY
- STATUS_LIBRARY_FAILURE
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FAILURE
- STATUS_UNSUPPORTED_OPTION

vol_id is the external label of the tape cartridge, as indicated in the request.

drive_id is the library drive used to mount the tape cartridge, as indicated in the request.

Final Response Values – Cancelled Request

Not applicable.

Notes

None.

See Also

- The `dismount` command for details on dismounting a cartridge from a library drive.

- The `query` command for details on displaying the closest library drive to a specified tape cartridge.
- *Appendix A* for entries that may be written to the Event Log.

QUERY

Name

query – Displays information about the Storage Server, an ACS, an LSM, a port, a CAP, a library drive, a tape cartridge, or a request.

Description

The query request returns status information for one or more of the following object types:

- The Storage Server
- ACS
- LSM
- CAP
- Library tape drive
- Tape cartridge
- Port
- Library request
- Tape cartridge mount status

The user can specify more than one object in a single request, as long as they all have the same type. For example, a single request can specify two ACSs, but not an ACS and a library drive. From 1 to MAX_ID objects can be specified in a single request.

Requests

Request Format

```
struct query_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     TYPE                          type;
     unsigned short                count;
     union {
          ACS                      acs_id;
          LSMID                    lsm_id;
          CAPID                    cap_id;
          DRIVEID                  drive_id;
          VOLID                    vol_id;
          MESSAGE_ID               request_id;
          PORTID                   port_id;
     } identifier;
};
```

Request Values

Within MESSAGE_HEADER, *command* is COMMAND_QUERY.

type is one of the following:

```
TYPE_ACS
TYPE_CAP
TYPE_DRIVE
TYPE_LSM
TYPE_MOUNT
TYPE_PORT
TYPE_REQUEST
TYPE_SERVER
TYPE_VOLUME
```

count is the number of items that follow in `identifier`. The maximum number of `identifier` items in a single request is MAX_ID (see the *Common Variables* section in *Chapter 3* for a description of MAX_ID).

Each item in `identifier` is one of the following, based on *type*:

acs_id
lsm_id
cap_id
drive_id
port_id
request_id
vol_id

Only one *type* of `identifier` can be specified in a single request. If *count* = 0, the request is performed on all items in the data base matching the specified *type*. *count* cannot be 0 if *type* is TYPE_MOUNT. *count* must be 1 if *type* is TYPE_SERVER.

Responses

Response Format

```
struct query_response {
    IPC_HEADER                  ipc_header;
    MESSAGE_HEADER              message_header;
    RESPONSE_STATUS             message_status;
    TYPE                        type;
    unsigned short              count;
    union {
        struct {
           STATE                state;
           FREECELLS            freecells;
           unsigned short       requests[MAX_COMMANDS][2];
        } server_status;
        struct {
           ACS                  acs_id;
           STATE                state;
```

```
            FREECELLS           freecells;
            unsigned short      requests[MAX_COMMANDS][2];
            STATUS              status;
          } acs_status;
          struct {
            LSMID               lsm_id;
            STATE               state;
            FREECELLS           freecells;
            unsigned short      requests[MAX_COMMANDS][2];
            STATUS              status;
          } lsm_status;
          struct {
            CAPID               cap_id;
            STATUS              status;
          } cap_status;
          struct {
            DRIVEID             drive_id;
            STATE               state;
            VOLID               vol_id;
            STATUS              status;
          } drive_status;
          struct {
            VOLID               vol_id;
            LOCATION            location_type;
            union {
              DRIVEID           drive_id;
              CELLID            cell_id;
            } location;
            STATUS              status;
          } volume_status;
          struct {
            VOLID               vol_id;
            STATUS              status;
            unsigned short      drive_count;
            DRIVEID             drive_id[MAX_ACS_DRIVES];
          } mount_status;
          struct {
            PORTID              port_id;
            STATE               state;
            STATUS              status;
          } port_status;
          struct {
            MESSAGE_ID          request;
            COMMAND             command;
            STATUS              status;
          } request_status;
        } status_response;
};
```

Intermediate Response Values

If the response is greater than MAX_MESSAGE_SIZE, the ACSLM breaks the response into one or more intermediate responses, containing MAX_ID status_responses, and a final response containing MAX_ID or less status_responses. See the *Common Variables* section in *Chapter 3* for a definition of MAX_ID and MAX_MESSAGE_SIZE.

Each intermediate response has the same format and values as the final response, with the following exception:

- Within MESSAGE_HEADER, *message_options* is set to INTERMEDIATE.

See the *Final Response* sections below for the other values in the intermediate response.

Note: Because of the complexity of this response format, the fixed portion of the response and the individual status_response structures are described separately in the following sections.

Final Response – Fixed Portion

Within MESSAGE_HEADER, *command* is COMMAND_QUERY.

Within RESPONSE_STATUS, *status* is the disposition of the request. Following are *status* values specific to this command.

- STATUS_AUDIT_IN_PROGRESS, if an in-process audit has locked out access to a cell record in the data base. This *status* is issued only after the data base has attempted retries on the cell record.
- STATUS_CANCELLED if the request is cancelled. See *Final Response Values – Cancelled Request.*
- STATUS_LIBRARY_NOT_AVAILABLE if the ACSLM is in the STATE_RECOVERY state and a query request is received specifying a *type* other than TYPE_SERVER.
- STATUS_SUCCESS if the request is executed successfully.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_COUNT_TOO_LARGE
- STATUS_COUNT_TOO_SMALL
- STATUS_DATABASE_ERROR
- STATUS_INVALID_OPTION
- STATUS_INVALID_TYPE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FAILURE
- STATUS_UNSUPPORTED_OPTION

- STATUS_UNSUPPORTED_TYPE

type is one of the following, as indicated in the request:

```
TYPE_ACS
TYPE_CAP
TYPE_DRIVE
TYPE_LSM
TYPE_MOUNT
TYPE_PORT
TYPE_REQUEST
TYPE_SERVER
TYPE_VOLUME
```

count is the number of status_response entries in the response. There is one status_response entry for each identifier item that was processed. The maximum number of status_response entries in a single response is MAX_ID.

Final Response – ACS Status

Within status_response, acs_status format and values are as follows:

```
struct  {
     ACS                          acs_id;
     STATE                        state;
     FREECELLS                    freecells;
     unsigned short               requests[MAX_COMMANDS][2];
     STATUS                       status;
} acs_status;
```

type is TYPE_ACS.

acs_id is the ACS identifier, as indicated in the request.

state is the current state of the ACS and is one of the following:

```
STATE_ONLINE
STATE_OFFLINE
STATE_OFFLINE_PENDING
STATE_DIAGNOSTIC
```

freecells is the total number of unoccupied storage cells in the ACS.

requests is a two-dimensional array of numbers describing the distribution of requests for the ACS. The array has MAX_COMMANDS columns and two rows. MAX_COMMANDS is defined as:

```
#define MAX_COMMANDS 5
```

Each column is a command (audit, mount, dismount, enter, and eject). Each row is the processing status of the request (current, pending).

Both current and pending requests are returned in the *requests* array. These requests are defined as follows:

Current Executing.

Pending Received by the ACSLM, but not yet executed.

status is the disposition of each *acs_id*. Following are *status* values specific to this command.

- STATUS_SUCCESS if status information was successfully retrieved for this ACS.

Following are common acs_status *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_DATABASE_ERROR
- STATUS_INVALID_ACS

Final Response – CAP Status

Within status_response, cap_status format and values are as follows:

```
struct  {
     CAPID                          cap_id;
     STATUS                         status;
} cap_status;
```

type is TYPE_CAP.

cap_id is the CAP identifier, as indicated in the request.

status is the disposition of each *cap_id*. Following are status values specific to the cap_status.

- STATUS_AUDIT_ACTIVITY if the CAP is reserved by an audit.
- STATUS_CAP_AVAILABLE if the CAP is not being used for any activity.
- STATUS_EJECT_ACTIVITY if cartridges are being ejected from the CAP.
- STATUS_ENTER_ACTIVITY if cartridges are being entered into the CAP.

Following are common cap_status *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_DATABASE_ERROR

- STATUS_INVALID_ACS
- STATUS_INVALID_LSM

Final Response – Drive Status

Within status_response, drive_status format and values are as follows:

```
struct  {
     DRIVEID                    drive_id;
     STATE                      state;
     VOLID                      vol_id;
     STATUS                     status;
} drive_status;
```

type is TYPE_DRIVE.

drive_id is the drive identifier, as indicated in the request.

state is the current state of the drive and is one of the following:

```
STATE_ONLINE
STATE_OFFLINE
STATE_DIAGNOSTIC
```

vol_id is the volume identifier of the tape cartridge in the drive. If no volume is in the drive, *vol_id* is null.

status is the disposition of each *drive_id*. Following are status values specific to the drive_status.

- STATUS_DRIVE_AVAILABLE if the drive does not contain a tape cartridge.
- STATUS_DRIVE_IN_USE if the drive contains a tape cartridge or is reserved for a mount.

Following are common drive_status *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_DATABASE_ERROR
- STATUS_DRIVE_NOT_IN_LIBRARY
- STATUS_INVALID_ACS
- STATUS_INVALID_DRIVE
- STATUS_INVALID_LSM

Final Response – LSM Status

Within `status_response`, `lsm_status` format and values are as follows:

```
struct   {
     LSMID                          lsm_id;
     STATE                          state;
     FREECELLS                      freecells;
     unsigned short                 requests[MAX_COMMANDS][2];
     STATUS                         status;
} lsm_status;
```

type is `TYPE_LSM`.

lsm_id is the LSM identifier, as indicated in the request.

state is the current state of the LSM and is one of the following:

```
STATE_ONLINE
STATE_OFFLINE
STATE_OFFLINE_PENDING
STATE_DIAGNOSTIC
```

freecells is the total number of unoccupied storage cells in the LSM.

requests is a two-dimensional array of numbers describing the distribution of requests for the LSM. The array has `MAX_COMMANDS` columns and two rows. `MAX_COMMANDS` is defined as:

```
#define MAX_COMMANDS 5
```

Each column is a command (audit, mount, dismount, enter, and eject). Each row is the processing status of the request (current, pending).

Both current and pending requests are returned in the *requests* array. These requests are defined as follows:

Current	Executing.
Pending	Received by the ACSLM, but not yet executed.

status is the disposition of each *lsm_id*. Following are status values specific to the `lsm_status`.

- `STATUS_AUDIT_ACTIVITY` if the LSM is being audited.
- `STATUS_CAP_AVAILABLE` if the LSM is not being used for any activity.
- `STATUS_EJECT_ACTIVITY` if cartridges are being ejected from the LSM.
- `STATUS_ENTER_ACTIVITY` if cartridges are being entered into the LSM.

Following are common `lsm_status` *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- `STATUS_ACS_NOT_IN_LIBRARY`
- `STATUS_LSM_NOT_IN_LIBRARY`
- `STATUS_DATABASE_ERROR`
- `STATUS_INVALID_ACS`
- `STATUS_INVALID_LSM`

Final Response – Mount Status

Within `status_response`, `mount_status` format and values are as follows:

```
struct {
     VOLID                          vol_id;
     STATUS                         status;
     unsigned short                 drive_count;
     DRIVEID                        drive_id[MAX_ACS_DRIVES];
     } mount_status;
```

The *type* is `TYPE_MOUNT`.

vol_id is a tape cartridge, as indicated in the request.

status is the disposition of each *vol_id*. Following are status values specific to the `mount_status`.

- `STATUS_INVALID_VOLUME` if the volume identifier is invalid.
- `STATUS_VOLUME_HOME` if the cartridge is in a storage cell.
- `STATUS_VOLUME_IN_DRIVE` if the cartridge is in a library drive.
- `STATUS_VOLUME_IN_TRANSIT` if the cartridge is being moved from one location to another or has been selected by another request.
- `STATUS_VOLUME_NOT_IN_LIBRARY` if the volume identifier is not listed in the data base.

Following are common `mount_status` *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- `STATUS_DATABASE_ERROR`

drive_count indicates the number of *drive_ids* to follow.

The *drive_id* list specifies online, available library tape drives, ordered by proximity to the current location of the tape cartridge. Proximity is defined in relation to the number of Pass-Thru Ports (PTPs) between the cartridge and the drive, therefore all drives in an LSM are considered equal in proximity. There can be up to `MAX_ACS_DRIVES` in

the list. See the *Common Variables* section in *Chapter 3* for a definition of MAX_ACS_DRIVES.

Final Response – Port Status

Within status_response, port_status format and values are as follows:

```
struct {
        PORTID                  port_id;
        STATE                   state;
        STATUS                  status;
} port_status;
```

type is TYPE_PORT.

port_id is the port identifier, as indicated in the request.

state is the current state of the port and can be one of the following:

```
STATE_ONLINE
STATE_OFFLINE
```

status is the disposition of each *port_id*. Following are status values specific to the port_status.

- STATUS_SUCCESS if status information was successfully retrieved for this port.

Following are common port_status *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_PORT_NOT_IN_LIBRARY
- STATUS_DATABASE_ERROR
- STATUS_INVALID_ACS
- STATUS_INVALID_PORT

Final Response – Request Status

Within status_response, request_status format and values are as follows:

```
struct {
     MESSAGE_ID                 request_id;
     COMMAND                    command;
     STATUS                     status;
} request_status;
```

type is TYPE_REQUEST.

request is the ACSLM request identifier, as indicated in the request.

command is the command associated with the request.

status is the disposition of each *request_id*. Following are status values specific to the `request_status`.

- `STATUS_CURRENT` if the request is being executed.
- `STATUS_INVALID_MESSAGE` if the format of the *request_id* specified in the `query` request is invalid.
- `STATUS_MESSAGE_NOT_FOUND` if the *request_id* specified in the `query` request is not a current or pending request.
- `STATUS_PENDING` if the request has been received by the ACSLM, but has not been executed.

Final Response – Server Status

Within `status_response`, `server_status` format and values are as follows:

```
struct  {
     STATE                          state;
     FREECELLS                      freecells;
     unsigned short                 requests[MAX_COMMANDS][2];
} server_status;
```

type is `TYPE_SERVER`. *count* is `1`.

freecells is the number of unoccupied storage cells in the library.

state is the current state of the ACSLM and is one of the following:

```
STATE_RECOVERY
STATE_RUN
STATE_IDLE
STATE_IDLE_PENDING
```

requests is a two-dimensional array of numbers describing the distribution of requests for library resources. The array has `MAX_COMMANDS` columns and two rows. `MAX_COMMANDS` is defined as:

```
#define MAX_COMMANDS 5
```

Each column is a command (audit, mount, dismount, enter, and eject). Each row is the processing status of the request (current, pending).

Both current and pending requests are returned in the *requests* array. These requests are defined as follows:

Current	Executing.
Pending	Received by the ACSLM, but not yet executed.

Final Response – Volume Status

Within `status_response`, `volume_status` format and values are as follows:

```
struct  {
     VOLID                          vol_id;
     LOCATION                       location_type;
     union {
       DRIVEID                      drive_id;
       CELLID                       cell_id;
     } location;
     STATUS                         status;
} volume_status;
```

type is `TYPE_VOLUME`.

vol_id is the tape cartridge volume identifier, as indicated in the request.

location_type is the type of location where the cartridge is currently located, according to the data base. It is one of the following:

```
LOCATION_CELL
LOCATION_DRIVE
```

Within `location`, either *drive_id* or *cell_id* has a value, depending on the *location_type*:

- *drive_id* is the library drive where the cartridge is located.
- *cell_id* is the location of the storage cell where the cartridge is located.

status is the disposition of each *vol_id*. Following are status values specific to the `volume_status`.

- `STATUS_VOLUME_HOME` if the cartridge is in a storage cell. `location` is a *cell_id*.
- `STATUS_VOLUME_IN_DRIVE` if the cartridge is in a library drive. `location` is a *drive_id*.
- `STATUS_VOLUME_IN_TRANSIT` if the cartridge is being moved from one location to another or has been selected by another request. In this case, `location` indicates the last location of the cartridge and can be either a *drive_id* or a *cell_id*.
- `STATUS_VOLUME_NOT_IN_LIBRARY` if the cartridge is not listed in the data base.

Following are common `volume_status` *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- `STATUS_INVALID_VOLUME`
- `STATUS_DATABASE_ERROR`

Final Response Values – Cancelled Request

Within `MESSAGE_HEADER`, *command* is `COMMAND_QUERY`.

Within `RESPONSE_STATUS`, *status* is `STATUS_CANCELLED`.

type is the type of `identifier` being queried, as indicated in the request.

count is the number of `identifiers` processed prior to the `cancel` request. `identifiers` not processed are not included in the response.

Within `status_response`, values are assigned according to the type of `identifier` in the request. See the *Final Response* sections for the formats and values.

Notes

- A `query server` request cannot be cancelled.

See Also

- The `cancel` command for details on canceling a current or pending `query` request.
- *Appendix A* for entries that may be written to the Event Log.

START

Name

`start` – Initiates ACSLM request processing.

Description

The `start` request places the ACSLM in `STATE_RUN`, enabling processing of Storage Server requests. The `start` command has no options.

If the `start` is successful, the Storage Server becomes ready to receive requests. If the `start` is unsuccessful, the Storage Server does not become ready to receive requests.

See the *Library Request Processing* section in *Chapter 2* for details on the possible Storage Server states.

Requests

Request Format

```
struct start_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
};
```

Request Values

Within `MESSAGE_HEADER`, *command* is `COMMAND_START`.

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct start_response {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_START.

Within RESPONSE_STATUS, *status* is the disposition of the request. Following are *status* values specific to this command.

- STATUS_SUCCESS if the Storage Server was successfully put in STATE_RUN.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_INVALID_OPTION
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_UNSUPPORTED_OPTION

Final Response Values – Cancelled Request

Not applicable.

Notes

None.

See Also

- The idle command for information on placing the Storage Server in STATE_IDLE.
- The query command for details on displaying the current state of the Storage Server.
- *Appendix A* for entries that may be written to the Event Log.

VARY

Name

`vary` – Changes the state of an ACS, LSM, library drive or port.

Description

The `vary` request changes the state of an ACS, LSM, library drive, or port. Only one device type at a time can be varied. Within that type, from one to `MAX_ID` different devices can be specified in each request.

The *message_option* `FORCE` changes the state of the device abruptly. `FORCE` is valid only when the device is an ACS or an LSM and it device varied offline.

An ACS, LSM, or library drive can be changed to online, offline, or diagnostic. A port can be changed to online or offline.

Device States

A device can be in one of five states, as described below:

- `STATE_ONLINE`. The normal operating state. The device is available for library processing.
- `STATE_OFFLINE`. A state in which the device is logically disabled. Requests involving offline devices are rejected.
- `STATE_OFFLINE_PENDING`. A transition state that occurs when an ACS or LSM is taken from online or diagnostic to offline. All new requests for the device are rejected, but current and pending requests are processed to completion. This state is not valid for cartridge drives or ports.
- `STATE_RECOVERY`. A transition state that occurs when an ACS or LSM is taken from offline to diagnostic or online. A recovery process is performed. New requests are rejected while the device is in this state. This state is not valid for cartridge drives or ports.
- `STATE_DIAGNOSTIC`. A state in which the device is not available to client application requests, but is available to operator requests from the Command Processor. This state allows for diagnostic activity to be performed on the device without interference from client applications. This state is not valid for ports.

Table 4-1 identifies which states are valid for each device.

Table 4-1. Valid Device States

Device	Online	Offline	Offline-Pend.	Recovery	Diag.
ACS	x	x	x	x	x
LSM	x	x	x	x	x
Drive	x	x			x
Port	x	x			

Device State Transitions

The `vary` request moves a device between these states. The transitions occur as follows:

- A `vary online` request on a drive or port immediately places the device in the `STATE_ONLINE` state.
- An `vary online` request on an ACS or LSM places the device in the `STATE_RECOVERY` state while it attempts to recover in-transit cartridges. When this recovery process has been completed successfully, the device is placed in the `STATE_ONLINE` state.
- A `vary offline` request with the `force` *message_option* puts the device in the `STATE_OFFLINE` state immediately, causing any current or pending requests for the device to be aborted.
- An unqualified `vary offline` request (that is, without the `force` *message_option*) is processed according to the type of device:
  - For an ACS or an LSM, the request causes the device to go into the `STATE_OFFLINE_PENDING` state initially. The ACSLM processes all current and pending requests for the device to completion before placing it in the `STATE_OFFLINE` state.
  - For a cartridge drive, the request is rejected if the drive is in use. If the drive is available, it is placed in the `STATE_OFFLINE` state immediately.
  - For a port, the request is rejected if the ACS it is connected to is online and it is the only online port for that ACS. If the ACS is in the `STATE_OFFLINE` state, or if there are other ports in the `STATE_ONLINE` state for that ACS, the port is placed in the `STATE_OFFLINE` state immediately.
- A `vary diagnostic` request places the device in `STATE_DIAGNOSTIC`. The ACSLM processes all current and pending requests for the device to completion. It accepts new requests from the ACSSA, but rejects new requests from a CSI. A port cannot be varied to the `STATE_DIAGNOSTIC` state, but all other devices can.

Requests

Request Format

```
struct vary_request {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     STATE                         state;
     TYPE                          type;
     unsigned short                count;
     union {
         ACS                       acs;
         PORTID                    port_id;
         LSMID                     lsm_id;
         DRIVEID                   drive_id;
     } identifier;
};
```

Request Values

Within MESSAGE_HEADER, *command* is COMMAND_VARY.

Within MESSAGE_HEADER, *message_options* can be FORCE if the *state* is STATE_OFFLINE. This causes request processing for the requested ACS or LSM to be stopped immediately. The specified component's state is marked STATE_OFFLINE in the data base.

state is one of the following:

```
STATE_ONLINE
STATE_OFFLINE
STATE_DIAGNOSTIC
```

state STATE_DIAGNOSTIC cannot be used when *type* is TYPE_PORT.

type is one of the following:

```
TYPE_ACS
TYPE_LSM
TYPE_PORT
TYPE_DRIVE
```

Only one *type* is allowed in a single request.

count is the number of identifier entries that follow. *count* must be 1 to MAX_ID.

Within identifier, the device ID is one of the following, based on *type*:

acs
lsm_id
port_id
drive_id

Responses

Intermediate Response Format

Not applicable.

Final Response Format

```
struct vary_response {
     IPC_HEADER                    ipc_header;
     MESSAGE_HEADER                message_header;
     RESPONSE_STATUS               message_status;
     STATE                         state;
     TYPE                          type;
     unsigned short                count;
     union {
          struct {
             ACS                   acs_id;
             RESPONSE_STATUS       status;
          } acs_status;
          struct {
             PORTID                port_id;
             RESPONSE_STATUS       status;
          } port_status;
          struct {
             LSMID                 lsm_id;
             RESPONSE_STATUS       status;
          } lsm_status;
          struct {
             DRIVEID               drive_id;
             RESPONSE_STATUS       status;
          } drive_status;
     } device_status;
};
```

Final Response Values

Within MESSAGE_HEADER, *command* is COMMAND_VARY.

Within RESPONSE_STATUS, *status* is the disposition of the request. Following are *status* values specific to this command.

- STATUS_SUCCESS if the fixed portion of the request was correct.

Following are common RESPONSE_STATUS *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_CONFIGURATION_ERROR
- STATUS_COUNT_TOO_LARGE
- STATUS_COUNT_TOO_SMALL
- STATUS_DATABASE_ERROR

- STATUS_INVALID_OPTION
- STATUS_INVALID_STATE
- STATUS_INVALID_TYPE
- STATUS_LIBRARY_BUSY
- STATUS_LIBRARY_FAILURE
- STATUS_LIBRARY_NOT_AVAILABLE
- STATUS_MESSAGE_TOO_LARGE
- STATUS_MESSAGE_TOO_SMALL
- STATUS_PROCESS_FAILURE
- STATUS_UNSUPPORTED_OPTION
- STATUS_UNSUPPORTED_STATE
- STATUS_UNSUPPORTED_TYPE

state is one of the following:

```
STATE_ONLINE
STATE_OFFLINE
STATE_DIAGNOSTIC
```

type is one of the following:

```
TYPE_ACS
TYPE_LSM
TYPE_PORT
TYPE_DRIVE
```

count is the number of `device_status` entries that follow. *count* must be 1 to MAX_ID.

Within `device_status`, the device identifier is one of the following, based on *type*:

acs
lsm_id
port_id
drive_id

Within `device_status`, *status* is the disposition of the particular device. Following are *status* values specific to this command:

- STATUS_RECOVERY_INCOMPLETE if recovery of in-transit cartridges is unsuccessful while varying an LSM or ACS online. The LSM is marked STATE_ONLINE in the data base, but the unrecorded in-transit cartridges may restrict use of PTPs or the robot's hands. An unsolicited message is sent to the ACSSA.
- STATUS_STATE_UNCHANGED if the device is already in the requested state.

- STATUS_SUCCESS if the device was successfully varied to the specified state.
- STATUS_VARY_DISALLOWED for any of the following conditions:
  - The request specifies an ACS or LSM that is currently in the STATE_OFFLINE_PENDING or STATE_RECOVERY state. These are transition states which are one-way only and indicate a vary request in progress.
  - The request specifies an ACS, LSM, or library drive be varied to or from STATE_DIAGNOSTIC and the originator is not the ACSSA.
  - The request is to vary an LSM online, but the ACS to which the LSM is attached is offline.
  - If the request is to vary an ACS, but not all LSMs within the ACS can be varied accordingly. The entire request fails, and all LSMs are left in their original state.

Following are common device_status *status* values. See the *Common Statuses* section in *Chapter 3* for their descriptions.

- STATUS_ACS_NOT_IN_LIBRARY
- STATUS_DRIVE_NOT_IN_LIBRARY
- STATUS_LSM_NOT_IN_LIBRARY
- STATUS_PORT_NOT_IN_LIBRARY
- STATUS_INVALID_ACS
- STATUS_INVALID_DRIVE
- STATUS_INVALID_LSM
- STATUS_INVALID_PORT
- STATUS_LIBRARY_BUSY
- STATUS_LIBRARY_FAILURE

Final Response Values – Cancelled Request

Not applicable.

Notes

- A vary request for an ACS is applied to all LSMs belonging to that ACS. If an LSM is not available, an error status value will be returned in the lsm_status *status*.
- When a device changes state, a STATUS_DIAGNOSTIC, STATUS_ONLINE, or STATUS_OFFLINE unsolicited message is sent to the ACSSA.
- If during recovery of in-transit cartridges either no available storage cells can be found or a cartridge label fails to validate, the cartridge is moved to the CAP. A STATUS_CARTRIDGES_IN_CAP unsolicited

message is sent to the ACSSA to notify the operator to empty the CAP. If the CAP is full or cannot accept all in-transit cartridges, recovery of in-transit cartridges is unsuccessful.

See Also

- The `query` command for details or displaying the current state of a library device.
- *Appendix A* for entries that may be written to the Event Log.

(INTENTIONALLY LEFT BLANK)

CHAPTER 5:
CSI PROCESSES

OVERVIEW

This chapter describes the basic functions performed by the CSI. It also identifies function calls and common data structures used by the CSI. See *Appendix C: XDR Translation Functions* for listings of StorageTek-supplied functions. See *Chapter 7: CSI Data Structures* for the formats and contents of the data structures.

Note: Since the functions of the SSI are essentially reciprocal to those of the CSI, an understanding of CSI functionality is required in order to derive SSI requirements. See *Chapter 6: SSI Requirements* for specific details on SSI requirements and programming.

This chapter covers the following topics:

- CSI Functions
- CSI Architecture
- Communications Methodology
- CSI Initiation
- Message Processing
- Error Detection and Recovery
- Termination

CSI FUNCTIONS

The CSI essentially functions as an asynchronous communications switchboard for the ACSLM. It receives input from both the ACSLM and an SSI via the Network Interface (NI). This input consists of Storage Server request and response packets.

Working together with the client system SSIs, the CSI provides a programmatic interface to the Storage Server that allows client applications to request remote tape service by executing library commands in a location-independent manner. The only restrictions on client application architecture are:

- *Communications protocol.* The client application must use a predefined communications protocol to send requests and receive responses.
- *The Storage Server command set.* The client application must format library requests according to the ACSLM command structures. It also must be able to interpret ACSLM responses. See *Chapter 3* and *Chapter 4* for these formats.

Figure 5-1 is a diagram of how library requests and responses are passed between a client application and the Storage Server.

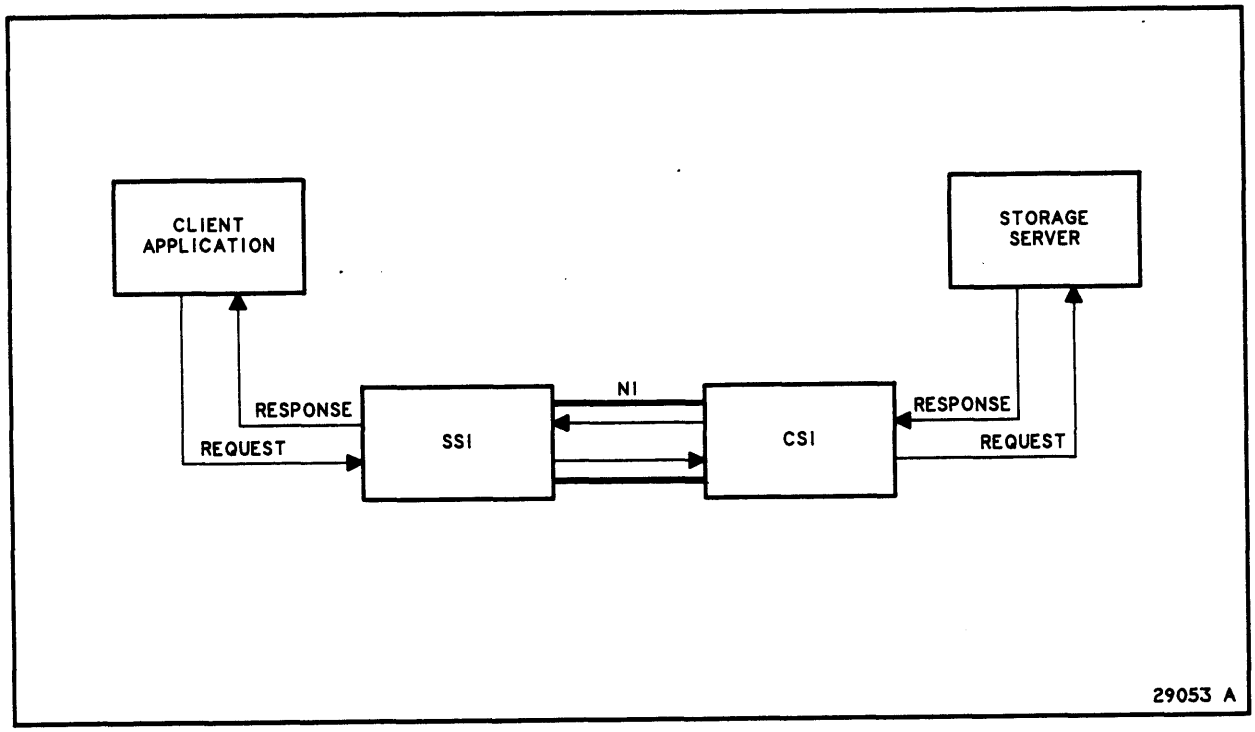

Figure 5-1. Overview of the Client System Interface

CSI ARCHITECTURE

Overview

The CSI communicates with the client system SSIs on a peer-to-peer basis, using a client–server model. The CSI conforms to the OSI layering model for peer-to-peer communications.

Within the client–server model, the CSI is the server, while the SSIs are clients. Following is a summary of request and response processing performed by the CSI and the SSIs.

A client application sends a Storage Server request through its SSI communications layer to a CSI server on the network. The SSI uses XDR library functions to encode, or *serialize*, the request into a host-independent data format, and then performs RPC calls to send the request across the network.

The CSI passively listens on the network for requests. It uses RPC functions to receive requests, uses XDR functions to *deserialize* them from host-independent to host-dependent format, and then passes them to the ACSLM.

After the ACSLM performs the designated function it issues one or more request responses, routing them back to the client through the CSI. The CSI keeps track of the routing pertaining to a particular request. When the CSI receives a response from the ACSLM, it matches the response to a particular client SSI and establishes the routing (address) from internal tables. Then the CSI uses an XDR function to serialize the response, and executes an RPC callback to send the response across the network to the SSI.

Meanwhile, the SSI passively listens for responses to its requests. It uses RPC calls to retrieve the responses from the network, uses XDR functions to deserialize them, and then passes them up to the client application.

OSI Model

See *Chapter 1: ACS Overview* for a summary of the Open Systems Interface (OSI) model.

In the overall Storage Server software communications model, the CSI represents the following two OSI layers:

- Layer 6 – Presentation Layer
- Layer 5 – Session Layer

Figure 5-2 illustrates how the Storage Server and client system components map onto the OSI model.

Almost all CSI programming represents Presentation Layer functionality. Presentation Layer programming focuses on the following:

- Storage of SSI return addresses
- Packet conversion, using XDR translation services
- Network output queueing
- Duplicate packet detection

Session Layer communications programming focuses on making system calls to detect and multiplex module interprocess communications (IPC) and network connections. Session Layer network functionality is accomplished through calls to Sun Microsystems Remote Procedure Call (RPC) functions.

Transport Layer (Layer 4) functionality is accomplished through UDP/IP or TCP/IP, or both running concurrently. This layer of software is transparent to the programmer, as it is supported entirely by RPC and the SunOS implementation of sockets. No TCP or UDP programming is needed.

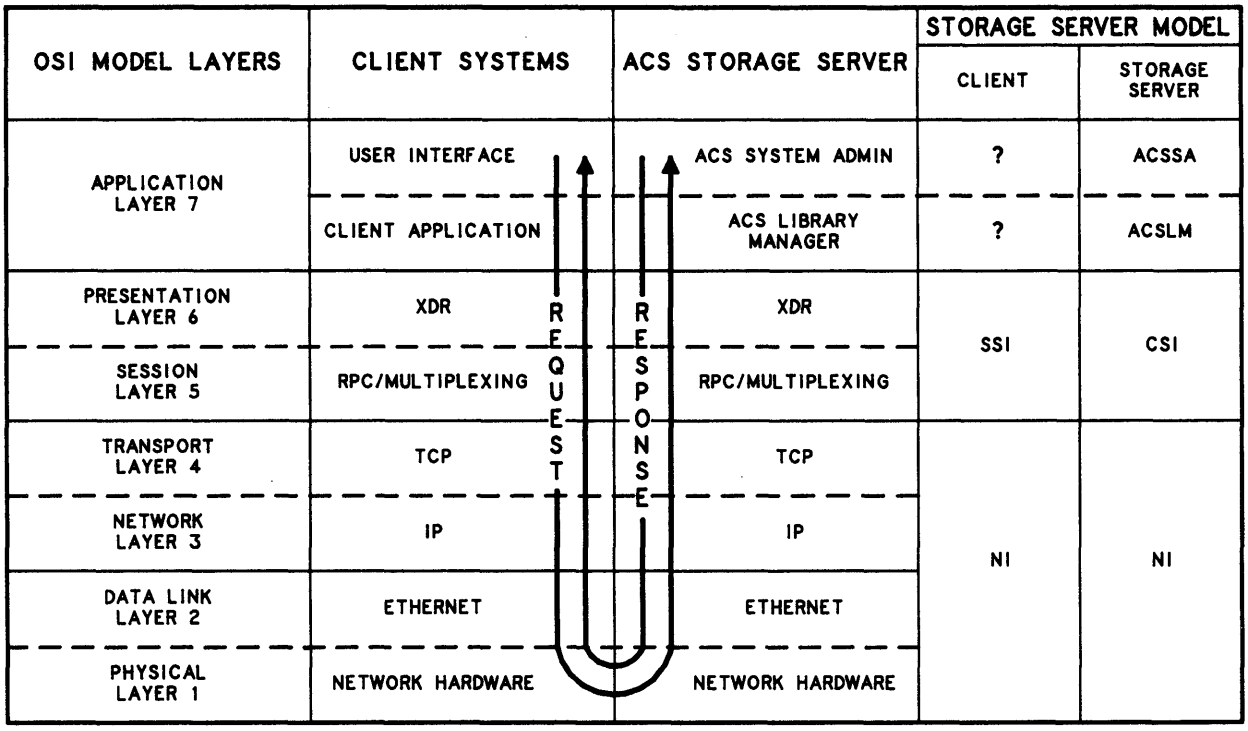

Figure 5-2. Mapping to the OSI Model

COMMUNICATIONS METHODOLOGY

Overview

The standard client–server architecture is based on *synchronous communications*. Essentially, this means that the client issues a request and then blocks until it receives an acknowledge that the requested operation has been performed.

The CSI's communications methodology uses *asynchronous* communications at the Applications Layer and *synchronous* communications at the Session Layer. The CSI receives an arbitrary number of requests from an arbitrary number of SSIs. After the ACSLM processes the requests, it returns one or more responses to the appropriate SSI. As a result, the CSI cannot block indefinitely to complete a network transmission because at any given moment it may have another task to perform, such as servicing new NI RPC connections, SSI Storage Server requests, or ACSLM responses.

There are several RPC methodologies for asynchronous communications. The CSI and SSI use the *callback* model. In this model, the requestor includes in its requests a return address where responses are to be sent. In the case of the Storage Server, the SSI includes its return address in the `CSI_HEADER` portion of the request; the CSI then sends responses to that address. See *Chapter 7: CSI Data Structures* for the format and content of the `CSI_HEADER` structure.

In order to employ the asynchronous model, Storage Server requests employ an application-level protocol made up of groups of discrete, synchronous transactions. Within this protocol, the CSI and the SSI must *immediately* acknowledge all messages, and they must be able to match responses to a particular request. A request is a distinct synchronous transaction, and a response is a separate distinct synchronous transaction. All RPC calls are immediately acknowledged and carry no response data.

The asynchronous nature of the model is accomplished through the high-level RPC protocol employed by the CSI and the SSIs. A Storage Server request from an SSI initiates a series of synchronous transactions which lead to the completion of an entire Storage Server operation. After the SSI initiates the request, it is free to perform other work, if so designed. After the CSI sends a request or response to the appropriate destination, it is free to perform other work. The entire storage server operation is complete when the SSI receives a *final response* from the CSI.

Interprocess Communications

CSI – ACSLM and CSI – ACSSA communications are accomplished through BSD datagram sockets.

The CSI, ACSLM, and ACSSA each create and maintain a single, named input socket used to receive messages from the other processes. This simplifies initialization and error recovery in each process. These sockets are defined at initiation.

Network Communications

Data Stream Representation

The data stream representation supported for the CSI is XDR. The XDR format for data is a host-independent serial byte stream. All data transmitted across the network is serialized to XDR format, transmitted, and then deserialized on the receiving end. See *CSI Message Processing* in this chapter for descriptions of the XDR translation routines.

The only portion of a message going to or from the NI that the CSI interprets is the `CSI_HEADER`. The CSI does, however, *parse* all information in a message as part of the serialization/deserialization process.

Session Connection

Communications services are solely for connection and data transfer. These two operations are encapsulated into a single, *brief* operation; a connection is closed *immediately* after data transfer is complete. This design is necessary because of the following constraints:

- *Network interruptions.* With current technology, network service tends to be interrupted periodically. When this happens the network connections are no longer valid. Limiting connection time reduces the chances of losing a connection.
- *Limited number of file descriptors that can be held at one time.* Maintenance of a TCP/IP connection on UNIX systems requires holding one or more open file descriptors. Limiting the time that connections are held reduces the chances of running out of available file descriptors.

RPC Service Registration

Within the callback model, both a *primary* client–server relationship and a *temporarily inverted* client–server relationship are defined. The primary relationship exists when the SSI requests tape services of the ACSLM via the CSI:

- The SSI functions as *primary client* when it makes RPC calls to transmit requests to the CSI.
- The CSI functions as *primary server* when it receives and processes the requests and passes them to the ACSLM.

This relationship is inverted, however, when responses are transmitted:

- Effectively, the CSI briefly functions as a *client* when it makes an RPC callback to the SSI.
- The SSI has a role as a *secondary server* in order to receive the callback.

Since both the CSI and SSI have server roles, they must each register as RPC servers at initiation. Client SSIs must interface with the RPC layer of the CSI. The descriptions below will clarify some of the processes documented in the *Sun Network Programming Manual.*

RPC Program Numbers and Port Mappings

A port mapping uniquely identifies a logical path that RPC uses in executing a remote function. The port mapping is initialized by making RPC calls to the RPC portmapper program. Each RPC procedure is identified by:

- *A unique program number*, used to interrelate various procedures (function calls) that are remotely executed.
- *One or more version numbers*, assigned to each program number so that the program number does not need to change when the service is changed.
- *A procedure number.*

In order to program the call for a particular remote procedure, the programmer looks up these numbers in a published list, and codes them as is appropriate on the function call interface. The *Sun Network Programming Manual* ("Remote Procedure Call Programming Guide"; "Higher Layers of RPC", "Assigning Program Numbers") defines the following categories of program numbers:

- *Sun-Defined Program Numbers.* Sun-defined program numbers exist in the following range:

```
0x0–0x1fffffff
```

Ultimately, the CSI program number will be derived from within this range by Sun, after StorageTek applies for a program number for the Storage Server product. It is not yet known when the CSI will begin using a Sun-registered program number.

- *User-Defined Program Numbers.* Until Sun-registered program numbers are employed, StorageTek will use user-defined program numbers for the CSI. Permanent CSI service program numbers should exist in the following range:

  ```
  0x20000000-0x3fffffff
  ```

  The program number currently used for the CSI is defined in `csi.h` as `CSI_PROGRAM`. See *Appendix D* for a listing of the `csi.h` header file.

  The SSI callback service could register within this range, but this might limit the number of SSIs that could run on a single host.

- *Transient Program Numbers.* Transient, or temporary, program numbers are primarily intended to exist only for the life of a particular execution of an application, barring a software error that prevents their deletion from the portmapper. These numbers should be dynamically assigned at program startup time, and unmapped with the port-mapper upon program shutdown. The SSI should use this category of program numbers.

  The transient program number is passed in the *program* variable in the `CSI_HANDLE_RPC` structure. See the *Input to the CSI From the SSI* section in *Chapter 7* for the format and content of this structure.

  The range of transient program numbers is:

  ```
  0x40000000-0x5fffffff
  ```

  The `gettransient()` function, which is shown in Figure 6-4, can be used to obtain the mapping.

CSI Registration

The CSI is the primary server; therefore it registers as the primary server at a *permanent*, advertised port/program number. It registers by calling the `svc_register()` RPC function; the parameters and values passed, as defined in the `csi.h` header file, are as follows.

- The transport handle:

  xprt

- The program number of the CSI server:

  ```
  #define CSI_PROGRAM 0x200000fe
  ```

- The version number of the CSI:

  ```
  #define CSI_UDP_VERSION 1
  #define CSI_TCP_VERSION 2
  ```

- A procedural dispatching routine:

```
#define CSI_ACSLM_PROC 1000
```

- A parameter that respecifies this as a TCP/IP or UDP/IP based service:

```
IPPROTO_TCP
IPPROTO_UDP
```

SSI Registration

When the SSI receives responses from the CSI it functions as a *server*, apart from its normal role as *primary client*. This provides the CSI with port, program number, and procedure number mapping used to direct an RPC callback. An integral part of this process is obtaining a transient port/program number. See the Initializing the SSI as a Callback Server topic in the *Programming an SSI* section in *Chapter 6* for details on SSI registration.

CSI INITIATION

Overview

The Storage Server daemon, `rc.acsss`, initiates the CSI; it can also reinitiate the CSI when necessary. CSI initiation performs the following functions:

- Establishes an interprocess communications input endpoint to receive messages from the ACSLM and the ACSSA.
- Establishes itself as an RPC network communications server.

The CSI writes entries to the Event Log when initiation begins and ends. See *Appendix A: Event Log Messages* for these messages.

Environment Variables

The `rc.acsss` startup script sets and exports the CSI environment variables. See the *Environment Variables* section in *Chapter 7* for a list of these variables.

Network Buffer Allocation

Upon startup, the CSI allocates a single network buffer into which the XDR interface functions place data during deserialization. The network buffer is allocated within the `CSI_MSGBUF` global message buffer description structure as an array called `data`. This array is at least `MAX_MESSAGE_SIZE` bytes long. See *Message Translation Structures* in *Chapter 7* for the format and contents of the `CSI_MSGBUF` structure. See *Common Variables* in *Chapter 3* for a definition of `MAX_MESSAGE_SIZE`.

RPC Service Initiation

At initiation, the CSI establishes communications with the NI by calling RPC library functions. Figure 5-3 identifies the functions and the order in which they are called. Normally, the CSI server is established before the SSI attempts to send calls to it.

Note: Figure 5-3 calls the `svctcp_create()` function to establish TCP/IP connections. To establish UDP/IP connections, the `svcudp_create()` function would be called instead.

```
                    /* CSI non-communications Initialization */
socket              /* establish ACSLM IPC or other connections */
pmap_unset()        /* unmap residual port mappings */
svctcp_create()     /* establish CSI server and port connection */
svc_register()      /* register CSI service with portmapper */
csi_process()       /* enter main CSI processing loop */
svc_destroy()       /* de-allocate the service transport resources */
```

Figure 5-3. Initializing the CSI Server

Figure 5-4 is a fragment of pseudocode that provides additional detail to the functions in Figure 5-3. The numerical references in the illustration are explained following the figure.

Note: The pseudocode in Figure 5-4 calls the `svctcp_create()` function to establish TCP/IP connections. To establish UDP/IP connections, the `svcudp_create()` function would be called instead.

```
    #include <rpc/rpc.h>
    #include <stdio.h>
    #include <sys/socket.h>
    #include "csi.h"

    main(argc, argv)
    int          argc;        /* number of command line arguments */
    char         **argv;      /* command line arguments */
    {
        SVCXPRT *xprt;        /* transport service handle */

    /*
     *  init log file, perform other Initialization
     */
    /*
       general, module-related initialization.....

     *  unmap from the portmapper if already mapped
     */
1   pmap_unset(CSI_PROGRAM, CSI_TCP_VERSION);

    /* csi to acslm socket */
    if ((cs_sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
      handle error.....

2   /* csi rpc socket */
    if ((cs_rcvsock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      handle error.....

3   /* reserve a port for the csi server rpc service */
    if (NULL == (xprt = svctcp_create(cs_rcvsock, 0, 0) ))
                          — or —
    if (NULL == (xprt = svctcp_create(RPC_ANYSOCK, 0, 0) ))
4     handle error.....

    /* register the service */
    if (0 == svc_register(xprt, CSI_PROGRAM, CSI_TCP_VERSION,
                                  address_of_dispatch_program, IPPROTO_TCP))
    /*
5    * enter main processing loop
     */
      main processing loop
    cs_process()

    /*
     *  termination, unmap from the portmapper
     */
6   pmap_unset(CSI_PROGRAM, CSI_VERSION);
    }
```

Figure 5-4. CSI RPC Service Initiation Logic

1. `pmap_unset()` is called to deregister the server with the portmapper. Unsetting the port mapping guarantees port mapping cleanup in case the registration from a previous initiation was not removed because the CSI terminated in an unplanned manner. Without this cleanup, the call to `svc_register()` could fail.
2. An NI socket is initialized. This is an Internet domain socket, of family `AF_INET`, and TCP/IP stream sockets of type `SOCK_STREAM`. The `cs_rcvsock` socket is the main RPC service socket for accepting client connections and receiving request packets from, and sending response packets to, the SSI. Either of the following two methods can be used to initialize the socket: 1) the socket can be created by the CSI/SSI, or 2) the `cs_rcvsock` variable can be initialized to `RPC_ANYSOCK`, causing RPC to initialize the socket in `svctcp_create()`. The CSI uses the second method.
3. `svctcp_create()` is called, reserving a port for TCP/IP communications. This function returns the transport handle, *xprt*, which is a structural representation of the service environment. It is passed to `svc_register()` for service registration.

   The parameters in this call should be set as follows:

   - The first parameter is the input socket. It can be set to an already opened socket or to `RPC_ANYSOCK`, causing RPC to initialize it.
   - The second parameter defines the *send* buffer size and should be set to `CSI_DEF_TCPSENDBUF`. Only very unusual applications of the CSI will require this parameter to be changed, since RPC can handle this buffering.
   - The last parameter defines the *receive* buffer size and should be set to `CSI_DEF_TCPRECVBUF`. Only very unusual applications of the CSI will require this parameter to be changed, since RPC can handle this buffering.
4. `svc_register()` is called to register the CSI with the portmapper. This allows incoming RPC calls from the SSI to be multiplexed to the correct socket, program number, version number, and procedure number. See the *Communications Methodology* section in this chapter for details on RPC registration.
5. The CSI enters a main processing loop which listens for client connections and data transfers, and multiplexes the input and output accordingly. See the *CSI Message Processing* section in this chapter for details on these processes.
6. At termination, `pmap_unset()` is called to deregister the CSI from the portmapper.

CSI MESSAGE PROCESSING

Overview

In acting as a communications switchboard between the SSIs and the ACSLM, the CSI performs the following activities:

- *Application-Level Messaging Protocol.* The CSI employs a high-level RPC protocol for message translation and transmission.
- *Interprocess Communications.* The CSI sends requests to and receives responses from the ACSLM.
- *Network Communications.* The CSI and the SSIs receive and send messages to one another via the NI.
- *Message Packet Decoding.* The CSI serializes message packets to be sent over the network and deserializes message packets read from the network.

Application-Level Messaging Protocol

Figure 5-5 through Figure 5-8 are sample RPC protocol sets describing the basic function calls and transactions necessary to initiate a Storage Server operation and carry it to completion. The protocol samples are listed as transaction pairs, two per page. On each page, the first figure represents the initiator of a particular transaction, and the second figure represents the response.

Entering its main processing loop, and blocking on `select()`, the SSI receives a Storage Server request from a client application and makes an RPC call to send it to the CSI.

```
LOOP {
/* merge RPC (svc_fds global) and other file descriptors into mask */
select()          /* block, waiting for application or RPC input */
                  /* determined that input is from application */
read()            /* read application request from application socket */
                  /* initialize CSI_HEADER structure, build request packet */
                        /* set csi_proto to CSI_PROTOCOL_TCP */
                        /* set csi_ctype to CSI_CONNECT_RPCSOCK */
      gethostbyname()   /* get address of this SSI host */
                        /* set csi_handle(program) to transient program# */
                        /* set csi_handle(version) to callback version# */
                        /* set csi_handle(proc) to callback procedure# */
                        /* set csi_handle(raddr) to struct sockaddr_in
                           returned from gethostbyname() */
gethostbyname() /* get the address of the remote CSI host */
bcopy()           /* copy remote host address to a sockaddr_in structure */
clnttcp_create()/* connect to the remote CSI server at CSI_PROGRAM */
clnt_call()       /* issue an RPC call to the CSI server at CSI_ACSLM_PROC */
csi_xdrrequest()/* clnt_call() calls XDR serialization function */
clnt_destroy()    /* close TCP/IP connection to CSI */
} END
```

Figure 5-5. Sending a Request From an Application to the CSI

The CSI server receives the request on one of its RPC sockets and repackages it for transmission to the ACSLM.

```
LOOP {
/* merge RPC (svc_fds global) and other file descriptors into mask */
select()         /* block, waiting for ACSLM or RPC input */
                 /* determined that input is RPC */
svc_getreq()     /* handle rpc input, calls CSI dispatcher CSI_ACSLM_PROC */
      dispatcher()     /* svc_getreq() invokes dispatcher */
            svc_getargs()       /* dispatcher gets data off network */
                  csi_xdrrequest()/* called to () deserialize request */
                        xdr_??()  /* csi_xdrrequest calls XDR primitives */
            svc_sendreply();        /* ACK SSI telling it received request */
            /* strip off CSI_HEADER and store in connection table */
            /* put IPC_HEADER into request packet */
            write()                 /* send request packet to ACSLM */
} END
```

Figure 5-6. Sending a Request From the CSI to the ACSLM

Later, the CSI receives a response from the ACSLM, repackages it and issues an RPC callback to the destination SSI server at the designated remote transient program number and dispatching procedure number.

```
LOOP  {
/* merge RPC (svc_fds global) and ACSLM file descriptors into mask */
select()          /* block, waiting for ACSLM or RPC input */
                  /* determined that input is from ACSLM*/
read()            /* read storage server request from ACSLM socket */
                  /* get csi_header from connect table key=ipc_identifier */
                  /* extract sockaddr_in from client handle in csi_header */
                  /* extract transient program#, version#, procedure#
                  /* strip IPC_HEADER, add CSI_HEADER to response packet */
clnttcp_create()/* create a connection to the remote SSI server */
clnt_call()       /* make RPC call to SSI at transient program#/procedure# */
      csi_xdrresponse() /* clnt_call() calls XDR serialization function */
clnt_destroy()  /* close TCP/IP connection to SSI */
}
```

Figure 5-7. Sending a Response From the ACSLM to the SSI

Finally, the SSI callback server dispatch function receives the response packet input on its RPC input socket and repackages it for transmission to the client application.

```
LOOP {
/* merge RPC (svc_fds global) and other file descriptors into mask */
select()            /* block, waiting for application or RPC input */
                    /* determined that input is RPC */
svc_getreq()        /* handle rpc input, calls SSI RPC callback dispatcher */
      dispatcher()                /* svc_getreq() invokes dispatcher */
            svc_getargs()         /* dispatcher gets data off network */
                  csi_xdrresponse()/* called to deserialize response */
                        xdr_??() /* csi_xdrresponse calls XDR primitives */
            svc_sendreply();      /* ACK CSI telling him received packet */
            /* strip CSI_HEADER from packet */
            /* process response, repackage into application format */
            /* send response to client application */
} END
```

Figure 5-8. Sending a Response From the SSI to the Application

Interprocess Communications

Message Handling

All IPC messages between the CSI and the ACSLM have a common top layer containing a MESSAGE_HEADER and an IPC_HEADER, which together make up a REQUEST_HEADER.

The client application creates the MESSAGE_HEADER, describing the ACSLM command request, for each Storage Server request it generates. See the *Requests* section in *Chapter 3* for details on the format and contents of the MESSAGE_HEADER.

The CSI creates an IPC_HEADER for each request sent to the ACSLM. The contents of the IPC_HEADER structure are specific to the IPC mechanism being used. See *Input to the CSI From the ACSLM* in *Chapter 7* for the current format of this structure.

The IPC_HEADER contains application IPC return address information and a unique identifier that allows responses to be matched to a request. When the CSI receives a response from the ACSLM, it uses the request identifier in the IPC_HEADER to match the response to the request and to filter out any duplicate packets that may have been generated. The CSI strips the IPC_HEADER from the response before passing it on to the SSI.

Message Size

The datagram communication style ensures that messages are sent and received atomically, eliminating the need for processes to deal with partial messages. The size of individual datagrams (requests or responses) is limited to MAXIMUM_MESSAGE_SIZE bytes, which is defined as:

```
#define MAXIMUM_MESSAGE_SIZE   4096
```

Network Communications

Functions Called

The CSI calls the following RPC functions to establish network communications and to send and receive messages:

- `clnttcp_create()`, to create TCP/IP connections.
- `clntudp_create()`, to create UDP/IP connections.
- `clnt_call()`, to serialize and send a message across the network.
- `svc_getreq()` or `svc_getreqset()`, to receive and deserialize a message from an SSI.

- `svc_sendreply()`, to acknowledge receiving an SSI message; *xdr_void* is passed as one of the parameters to indicate that the acknowledge carries no data.

Message Routing

All messages between the SSI and the CSI have a common header, `CSI_HEADER`. The SSI creates a `CSI_HEADER` for each request. The `CSI_HEADER` contains the return address for callback responses to the SSI.

When the CSI receives a request, it strips the `CSI_HEADER` from the packet and stores it in its return address queue. When the CSI receives a response from the ACSLM, it retrieves the `CSI_HEADER` from the queue by using the *ipc_identifier* in the `IPC_HEADER`. It then adds the `CSI_HEADER` to the response and sends the response to the indicated SSI network address.

See the *Input to the CSI From the SSI* section in *Chapter 7* for the format of the `CSI_HEADER`.

Timing Considerations

After the CSI sends a message across the network it blocks on input until it receives an RPC acknowledge that the message has been received. A network transmission timeout will occur if the CSI does not receive the acknowledge within the time period defined by the `CSI_RETRY_TIMEOUT` environment variable.

When a timeout occurs, the CSI saves the unacknowledged message in its network output queue. The CSI maintains one output queue for all messages to be sent to *all* SSIs. The messages are sequenced according to the order in which they were received from the ACSLM by the CSI. The `CSI_XID` in the `CSI_HEADER` is used to indicate the sequence (see *Chapter 7: CSI Data Structures* for the format and contents of these structures). The network output queue frees the CSI to process new input without losing output that has not been acknowledged and must therefore be transmitted again.

Once a message is saved after a timeout, the CSI checks for new input from the ACSLM and proceeds as follows:

- If there is new input, the CSI services it immediately. This ensures that input is not lost.
- If there is *no* new input, the CSI checks its network output queue and transmits any messages intended for other SSIs.

- Once all messages for other SSIs are flushed from the queue *and* after a waiting period greater than `CSI_SELECT_TIMEOUT`, the CSI retries the original message for the first SSI. The CSI will attempt as many retries as are specified by the `CSI_RETRY_TRIES` environment variable.

Despite a timeout by the CSI, the SSI may still receive the original message, although later than expected. The SSI must therefore identify the retry message(s) as duplicates and discard them.

Message Packet Decoding

XDR Translation Functions

The CSI serializes and deserializes message packets by applying XDR primitives to the message structures. The CSI calls high-level routines which control complex translations by calling a series of lower-level routines dedicated to data conversion.

StorageTek supplies two high-level XDR translation interface routines, `csi_xdrrequest()` and `csi_xdrresponse()`, as well as supporting functions, in "C" source code format. They can be ported, by customer option, to the client host and called from CSIs and SSIs. These routines call the lower-level supporting translation routines which, in turn, call XDR primitives.

If the supplied routines are not used by the client SSI, they can be used as a template for creating custom SSI XDR translation routines. Any custom routines must preserve the order and functionality of the serialization and deserialization processes in the supplied routines, however.

See *Appendix C: XDR Translation Functions* for listings of these functions and details on what they do.

csi_xdrrequest() Function

The `csi_xdrrequest()` function handles the serialization and deserialization of Storage Server *requests*, as follows:

- For an SSI sending a request, the routine converts a request packet from host-specific format to XDR stream format for transmission across the NI.
- For a CSI receiving a Storage Server request, the routine converts the input from XDR stream format to a host-specific format request packet.

See the *Input to the CSI From the SSI* section in *Chapter 7* for information on the structure of request packets.

csi_xdrresponse() Function

The `csi_xdrresponse()` function handles the serialization and deserialization of Storage Server *responses* as follows:

- For a CSI sending a Storage Server response packet to an SSI, the routine converts the packet from host-specific format to XDR stream format for transmission across the NI.
- For an SSI receiving a response, it converts the XDR data stream received across the NI to a Storage Server response packet.

See the *Output From the CSI to the SSI* section in *Chapter 7* for information on the format of response packets.

Calling the Supplied Functions

The `csi_xdrresponse()` and `csi_xdrrequest()` routines have identical interfaces. They are usually called from the `svc_getargs()` or `clnt_call()` RPC library routines. On call, they are passed an XDR handle (a client handle or transport handle), as well as a message buffer description structure (`CSI_MSGBUF`). See the *Message Translation Structures* section in *Chapter 7* for the `CSI_MSGBUF` format.

- Following is an example of a call to `csi_xdrresponse()` on `client_call()`:

  `clnt_call(`*client_handle*`,`*procedure_number*`,csi_xdrresponse,`
  msg_buffer_desc`,`*xdr_void*`,NULL);`

  `csi_xdrresponse()` parses the message contents at the address called `data` specified in the *msg_buffer_desc* structure; it also calls its own sub-library of XDR translation routines which, in turn, call XDR translation primitives. As each low-level XDR primitive is called, the resulting serialized data stream is placed into the *client_handle* for subsequent transmission across the NI.

- Following is an example of a call to `csi_xdrrequest()` on `svc_getargs()`:

  `svc_getargs(`*transport_handle*`,csi_xdrrequest,`*msg_buffer_desc*`);`

Duplicate Packet Detection

The XDR functions detect duplicate message packets. See *Error Detection and Recovery* in this chapter for details.

Packet Tracing

The CSI is able to trace message packets and their contents. When packet tracing is enabled, the CSI logs an ASCII dump of packets in the execution trace log as they are processed. The log entry includes a formatted header showing the critical elements of the `CSI_HEADER` and

the *message_options* from the MESSAGE_HEADER. The output indicates the source of the message (either ACSLM or NI).

Figure 5-9 is an example of a packet trace.

```
Packet source: ACSLM
ssi_identifier: 39                  message_options: 0
ssi client addr: 129.80.32.3        ssi client port: 255
Message contents (hex bytes):
0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010: 00 00 00 00
```

Figure 5-9. Sample Packet Trace

Packet tracing is enabled by setting the trace environment variable, TRACE_VALUE, to 00800000 *before* initiating the CSI.

Request Processing Summary

The following series of steps outlines the interaction between an SSI and the CSI to process a Storage Server request. Only the steps performed by the CSI are detailed. See the *Programming an SSI* section in *Chapter 6* for details on the steps performed by the SSI.

SSI: 1. Receives a request from one of its client applications, translates it, and sends it to the CSI.

CSI: 2. Calls either the `svc_getreq()` or `svc_getreqset()` RPC function to receive the message. The following occurs:

2.1 The `svc_getreq()` or `svc_getreqset()` function calls `csi_rpcdisp()`, the dispatching service that was registered during initiation.

2.2 `csi_rpcdisp()` calls `svc_getargs()` to get the data off the network.

2.3 `svc_getargs()` calls a deserialization function to convert the response into host-dependent format. See the Message Packet Decoding topic in this section for a description of the StorageTek-supplied XDR translation functions.

3. Immediately acknowledges receipt of the message by calling the `svc_sendreply()` RPC function, passing no data.

4. Modifies the request header into a format that can be interpreted by the ACSLM:

- Strips off and stores the CSI_HEADER which contains SSI return address information.
- Adds an IPC_HEADER containing Storage Server IPC "return address" information.

See the Interprocess Communications topic in this section.

5. Sends the request packet to the ACSLM.
6. Receives a response from the ACSLM.
7. Modifies the response header into a format that can be interpreted by the SSI:
   - Strips off the `IPC_HEADER`.
   - Retrieves and adds the `CSI_HEADER` which contains the SSI's `CSI_HEADER` return address.
8. Calls either the `clnttcp_create()` or the `clntudp_create()` RPC function to establish a TCP/IP or UDP/IP connection with the SSI at the transient port/program number indicated in the `CSI_HEADER`. See the *Communications Methodology* section in this chapter.
9. Calls the `clnt_call()` RPC function to send the response message.
10. `clnt_call()` calls a serialization function to convert the response into XDR format. See the Message Packet Decoding topic in this section for a description of the StorageTek-supplied XDR translation functions.
11. If this is a *final response*, deletes the SSI's return address from the connect queue.

SSI:

12. Receives the message, translates it, and routes it to the appropriate application.

ERROR DETECTION AND RECOVERY

Overview

The CSI can detect, isolate, report, and recover from various internal software errors. All errors that the CSI encounters are handled in the following manner:

- All errors are logged to the Event Log. See *Appendix A: Event Log Messages* for the error text and their explanations.
- Selected errors are also sent as unsolicited messages to the ACSSA. This messages are displayed in the Display Area of the Command Processor window. See the *Common Statuses* section in *Chapter 3* for the message statuses.

External Errors

The CSI only tracks errors within its domain. This is because the CSI does not interpret the content of messages passed between the client and the Storage Server. All errors, except those sensed internally, are treated as application errors. Any error messages or responses

passed from the Storage Server to the client application are treated as data.

In the case of either a network or application module connection failure, the CSI attempts to recover the connection. It does not inform the SSI of these errors, except as provided for by the RPC mechanism. The RPC mechanism for communicating RPC errors between a CSI and an SSI is employed when the RPC routines prefixed "`svcerr`" are used (see the *Sun Network Programming Manual*). The CSI performs error recovery for its own connections to the ACSLM, the ACSSA, and the NI.

CSI Error Handling

The following sections describe major error detection and recovery processes of the CSI.

IPC Failure

IPC failures occur whenever the CSI is unable to perform one of the following functions:

- Establish inter-module input communications endpoints during initiation
- Read a message from the ACSLM
- Send a message to the ACSLM
- Send a message to the ACSSA

The CSI will attempt retries with timeouts and will issue error messages if communications is still unsuccessful. See the *Environment Variables* section in *Chapter 7* for an explanation of how the retries and timeouts are defined.

CSI Process Failure

A CSI process failure is a general error indicating a serious failure of some internal mechanism. The Storage Server daemon, `rc.acsss`, may attempt to reinitiate the CSI, depending on the nature of the failure. The CSI will then attempt to reestablish itself as a network service, using the same process employed during initiation. See *CSI Initiation* in this chapter for a description of this process.

Operating System Failure

The CSI can indirectly detect some operating system errors. These errors will be reported as IPC failures, process failures, or CSI-specific error messages. Error messages caused by operating system failures list a system error number. See your UNIX documentation for a description of these error numbers.

Network Transmission Failure

Depending on system state, there may be an arbitrary number of instances over an arbitrary length of time in which connections and data transfers will *never* complete. Both the CSI and the SSI must be able to detect these as errors by using timeout and retry algorithms. It is left up to the client application to designate the appropriate handling for connection errors.

Response messages from outstanding ACSLM requests cause the CSI to attempt to establish communications with the client system. If the CSI is unsuccessful, it will attempt retries with timeouts, using a retry algorithm. The algorithm functionality is controlled by setting the `CSI_RETRY_TIMEOUT` and `CSI_RETRY_TRIES` environment variables before starting the CSI. If a connection still cannot be established, the pending messages will be discarded and their intended destination logged.

Network Message Translation Failure

Network message translation failures usually occur when a message of incorrect format or size is detected by the high-level CSI XDR translation routines. The CSI logs all translation failures to the Event Log. Translation failures may occur for a variety of reasons, as outlined below:

- If the message has an incorrect format (that is, the `CSI_REQUEST_HEADER` portion is missing or has an invalid structure), the CSI is unable to deliver it and so discards it.
- The CSI must convert the message representation based on the contents of the message. The contents of key fields (for example, *command*, *type*, *count*, and *message_options*) determine how the remainder of the message is interpreted. When the CSI detects a message from the NI containing an invalid value in a key field, the message is truncated to the field in error and passed to the ACSLM for error processing.
- The CSI receives IPC input from the ACSLM only. The CSI will discard messages with *module_type* set to something other than `TYPE_LM`.
- A network message translation failure may be logged if a failure takes place in the Storage Server. The error is logged because a packet of invalid size for the command type is detected in the CSI. The packet is still transmitted in its (usually) truncated form.

The CSI will transmit a message as long as it can correctly translate the `CSI_REQUEST_HEADER` portion, despite other translation errors. Therefore the receiver of a message should always check the

CSI_MSGBUF structure for either of the following conditions, both of which indicate a message translation failure:

- *packet_status* is set to CSI_PAKSTAT_XLATE_ERROR.
- *translated_size* has an insufficient size for the command type.

Duplicate Network Packets

See the Timing Considerations topic in the *CSI Message Processing* section in this chapter for an explanation of how duplicate packets may be generated. The XDR translation functions will detect duplicate message packets. Only the CSI_REQUEST_HEADER portion of the message will be decoded, and the XDR translation routine will set the *packet_status* in CSI_MSGBUF to CSI_PAKSTAT_DUPLICATE_PACKET. The CSI discards duplicate packets.

CSI Request / Return Address Aging

The CSI's return address queue contains return addresses for requests awaiting final response. When the CSI receives a final response for a request, it deletes the return address from the queue. It is possible, however, for some processing or system error to prevent the CSI from receiving a final response. In order to maintain the queue, the CSI will automatically delete all return addresses that are older than a predefined maximum age. This age is defined in the CSI_CONNECT_AGETIME environment variable which is defined at initiation. See the *Environment Variables* section in *Chapter 7* for additional information about this variable.

CSI TERMINATION

Upon receipt of SIGTERM, the CSI immediately terminates, performing only the cleanup that is necessary to leave its files and connections in a recoverable state.

During termination the CSI may detect entries in its return address queue. This means that some Storage Server requests are still pending on a final response. The CSI will log the return addresses of these responses to the Event Log.

CHAPTER 6:
SSI REQUIREMENTS

OVERVIEW

This chapter describes special functional requirements for SSI design. Since the SSI essentially mirrors the CSI, it is intended that the functional requirements for the SSI be derived from the functionality of the CSI, specifically the CSI's interface requirements.

The following sections explain how to program an SSI. These sections are intended as extensions to the previous sections on CSI design. See *Chapter 5: CSI Processes* for details on CSI functionality. See *Chapter 7: CSI Data Structures* for the format and contents of CSI data structures used by the SSI and the CSI.

Note: Reference is made in this chapter to the StorageTek SSI. This is *not* a StorageTek product, and it is *not* distributed to customers. This SSI is referenced only in order to describe the strategies and techniques that StorageTek has successfully used to create an SSI for in-house use. This information can serve to guide the SSI designer and programmer in creating their own SSI.

DESIGNING AN SSI

Overview

The SSI, residing on a client host, functions as an communications switchboard for client system applications issuing requests for tape services to the Storage Server. Its primary functions are as follows:

- The SSI receives Storage Server requests as input from client applications, serializes them, and sends them across the network by making an RPC call to the CSI.
- The SSI receives Storage Server responses from the CSI via its RPC dispatcher, deserializes them, then provides necessary services to return the responses to the issuing applications.

Required Functionality

Any SSI implementation requires the following functionality:

- Porting and implementation of StorageTek's XDR translation routines
- Initiation of the SSI as a transient network server so that responses can be received
- Initiation of the interface between the SSI and an application requesting network tape services of the Storage Server
- Allocation of a buffer for data transferred to or from the Network Interface (NI)
- Ability to poll for both application and network input
- Ability to receive messages from applications using tape services
- Ability to format application requests into Storage Server request packets
- Ability to send Storage Server request packets on the network via XDR/RPC
- Ability to receive CSI response packets from the network via an RPC dispatcher
- Ability to match a response received from the CSI to a particular request and to route the response to the correct application
- Implementation of network-related error handling for timeouts, transmission errors, and duplicate packets

Architectural Notes

The SSI can be implemented as an interface library linked to a client application or as a separate executable module that communicates with the client application via IPC mechanisms.

The functionality of an SSI can be viewed as a mirror image of the CSI. If this approach is taken, then SSI source code can be derived directly from CSI source code.

Architectural Constraints

The only constraints on SSI architecture are as follows:

- *Communications and data translation protocols.* The SSI must employ the same protocols as the CSI for network connection and data translation:
  - XDR at the Presentation Layer
  - RPC at the Session Layer
  - TCP/IP or UDP/IP at the Transport Layer

- *Application-level protocols*. The SSI must conform to application-level protocols allowing for asynchronous transactions. See *Communications Methodology* in *Chapter 5*.
- *Data structures*. The SSI must include the standard message header, `CSI_REQUEST_HEADER`, and other predefined Storage Server command data structures in all requests sent to the CSI. See the *Input to the CSI From the SSI* section in *Chapter 7* for the format and content of this structure.
- *Convention of maintaining only brief connections*. It is recommended that the SSI encapsulate network connection and data transfer into one, brief operation, in a similar manner as the CSI. See *CSI Message Processing* section in *Chapter 5* for a discussion of the reasons for this.

SSI – CSI Architectural Comparisons

Overview

Depending on the nature of the client application, SSI functionality can be seen as a mirror image of CSI functionality. For example when Storage Server requests are processed, on the client side they are passed *down* from the application layer to the SSI, while on the server side they are passed *up* from the CSI to the application layer.

In order to simplify maintenance of CSIs and SSIs, StorageTek has built both applications using the same body of source code. Conditional compilation statements are be used in areas of the CSI source code where CSI and SSI functionality are inverse. Conditional compilation statements are used in the following areas of the code:

- Sending messages down-layer
- Sending messages up-layer
- RPC initialization

The following discussion compares the differences between the functionality of the StorageTek CSI and StorageTek SSI at points where the functionality in each is a mirror image of the other. Discussion centers on the handling of information as it crosses the various OSI-modeled software layers.

Sending Messages Down-Layer

When sending messages down-layer, the CSI *retrieves* return addresses, whereas the SSI *stores* them.

The CSI sends responses down-layer from the ACSLM to the NI. When it receives the response, the CSI retrieves the RPC return address information from the return address queue in order to be able

to return the response to the correct client. The data structure retrieved is of type `CSI_HEADER`.

The SSI sends requests down-layer from a client application to the NI. When it receives the request, the SSI stores the IPC return address information in the return address queue. The data structure is of type `IPC_HEADER`.

The `csi_lminput` routine can be used by both the CSI and the SSI to receive and process messages down-layer. The difference between the CSI and SSI functions is accomplished with an `#ifdef` statement in the code.

Sending Messages Up-Layer

When sending messages up-layer, the CSI *stores* return addresses, whereas the SSI *retrieves* them.

The CSI sends requests up-layer from the NI to the ACSLM. When it sends requests up-layer, the CSI stores the network return address in the return address queue. The data structure is of type `CSI_HEADER`.

The SSI sends responses up-layer from the NI to the client application. When it sends responses up-layer, the SSI retrieves the IPC address information structure from the return address queue. The data structure is of type `IPC_HEADER`. The SSI uses the key value *ssi_identifier* in the `CSI_HEADER` structure in order to direct the response to the correct client application.

The `csi_rpcdisp.c` routine can be used by both the CSI and the SSI to receive and process messages up-layer. The difference between the CSI and SSI functions is accomplished with an `#ifdef` statement in the code.

CSI and SSI Initialization

During RPC initialization, the CSI registers as a *permanent* server at a predefined permanent program number, whereas the SSI registers as a *transient* server at a transient program number.

The `csi_rpctinit.c` routine can be used by both the CSI and the SSI for RPC TCP initialization, and the `csi_rpcuinit.c` routine can used for RPC UDP initialization. The difference between the initialization sequences of the CSI and SSI is accomplished with an `#ifdef` statement in the code.

PORTING STORAGETEK XDR ROUTINES

StorageTek offers XDR translation functions in "C" source code format, either as a template for, or for direct porting to, client SSI source code. These routines perform XDR serialization and

deserialization of Storage Server packets. These routines are structured with a common high-level interface for the SSI. The interface has the following two parameters:

- A pointer to the network packet buffer supplied by the programmer (of type `CSI_MSGBUF`)
- A pointer to the XDR handle (of type `XDR`) supplied by RPC.

These XDR interfaces are never called directly by the programmer. They are always indirectly called on behalf of the programmer via RPC routines. When porting the StorageTek XDR routines to the customer host system, the programmer must be careful to preserve the intrinsic order and structure of translation.

Serialization of Requests

The SSI serializes message packets when it sends a request to the CSI via the NI. The SSI calls `clnt_call()`, which in turn calls `csi_xdrrequest()`. Figure 6-1 shows the layering of calls.

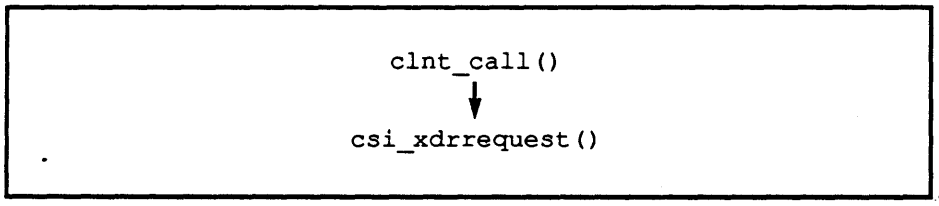

Figure 6-1. Layering of Calls — Request Serialization

Deserialization of Responses

The SSI deserializes message packets when it receives a response from the CSI via the NI. The SSI calls the `svc_getreq()` routine, which calls the SSI's RPC dispatcher function (defined by the SSI programmer). The dispatcher function calls `svc_getargs()`, which in turn calls `csi_xdrresponse()`. Figure 6-2 shows the layering of these calls.

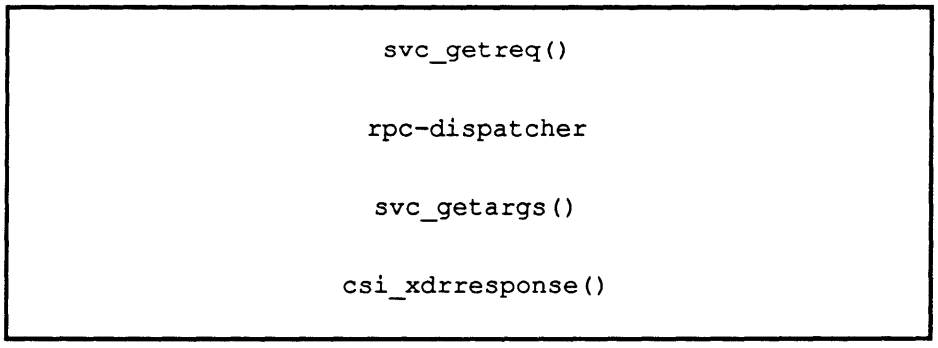

Figure 6-2. Layering of Calls — Response Deserialization

PROGRAMMING AN SSI

Initializing the SSI as a Callback Server

When the SSI receives responses from the CSI, it functions as a *server*, apart from its normal role as *primary client*. This provides the CSI with port, program number, and procedure number mapping to direct an RPC callback.

Because the SSI functions as a transient server it must be assigned a *transient* program number. The transient number is dynamically assigned when the SSI is initiated and should be unmapped with the portmapper when the SSI is terminated in an orderly manner. When the SSI is restarted later, a new program number will be assigned.

Figure 6-3 depicts the process of initializing the SSI as a transient server.

```
                  /* SSI non-communications Initialization */
                  /* establish application IPC or other connections */
pmap_unset()      /* unmap residual port mappings */

gettransient()    /* get transient RPC program# & socket */

    socket()      /* create a TCP/IP callback socket */
    bind()        /* bind socket to get port */
    getsockname() /* retrieve assigned port number */
    loop {
      pmap_set()
    }             /* count up from starting transient
                     program #, when pmap_set succeeds
                     you have a transient program number */
    return (program-number) + socket file descriptor, by reference */
svctcp_create()   /* establish callback service port */
svc_register()    /* register callback service with port mapper */
ssi_process()     /* enter main SSI processing loop */
svc_destroy()     /* de-allocate the service transport resources */
```

Figure 6-3. Initializing the SSI Server

To initialize the SSI as a network service, the following is required:

- *A unique transient program number*, which the CSI can use when issuing an RPC callback to the SSI.
- *A version number*, assigned to each program number so that the program number does not need to change when the NI transport service is changed.
- *A procedure number*.

Obtaining a Unique Program Number

In order to obtain a unique transient program number, which the CSI will use to issue an RPC callback to the SSI (for transmitting Storage Server responses), the SSI programmer must code a function that obtains a transient port mapping. The `gettransient()` function, shown in Figure 6-4, is a sample of such a function.

The `gettransient()` function is derived from the following sources:

- A prototype application
- The *Sun Network Programming Manual,* "Remote Procedure Call Programming Guide," (subsection: "More Examples/Callback Procedures").

The number obtained by the function must be in the range for transient program numbers. See *Communications Methodology* in *Chapter 5* for this range.

Initializing the SSI — Application Interface

There are no constraints on the design of the interface between the SSI and the application that it services. It is defined by the customer.

```
#include <rpc/rpc.h>
#include <stdio.h>
#include <sys/socket.h>
#define START_TRANSIENT 0x40000000   /* start number for transient progs */
#define END_TRANSIENT   0x5ffffffe   /* last transient prog# available */

/*
 *    gettransient()
 *
 *    Description: Returns the next available RPC transient program number
 *                 Assign a socket if the contents of sockp = RPC_ANYSOCK.
 *    Returns:
 *          On Call:
 *                (int)       - Next available transient program #
 *                            - 0 if ERROR
 *          By Reference:
 *                (int *)     - new socket file descriptor
 */
gettransient(proto, vers, sockp)
int  proto;             /* socket protocol */
int  vers;              /* version number */
int *sockp;             /* pointer to socket */
{
    static int         prognum = START_TRANSIENT;
    int                s, len, socktype; /* s=socket fd, len=size of addr */
    struct sockaddr_in addr;             /* internet type socket struct */

    switch (proto) {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
            return(0);
    } /* end of switch */

    if (RPC_ANYSOCK == *sockp) {
        if ((s = socket(AF_INET, socktype, 0)) < 0) {
            return(0);
        }
        *sockp = s;
    }
    else
        s = *sockp;                     /* use the socket passed in */

    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
    bind(s, &addr, len);  /* maybe already bound so don't check for error */
    if (getsockname(s, &addr, &len) < 0)
        return(0);

    /*count up from first program number until find one that is available */
    for (prognum = START_TRANSIENT; prognum < END_TRANSIENT; prognum++) {
        if (pmap_set(prognum, vers, proto, ntohs(addr.sin_port)) > 0)
            return(prognum);
    }
    return(0);                          /* error */
}
```

Figure 6-4. Code for Obtaining a Transient Program Number

Allocating the Network Buffer

The SSI must allocate a buffer for network input which both stores and describes the state of network packets. This buffer is of type `CSI_MSGBUF`. See *Message Translation Structures* in *Chapter 7* for the format and content of this structure. For clarification, the definition is duplicated below.

```
typedef struct {
    int          offset;
    int          size;
    int          maxsize;
    int          translated_size;
    CSI_PAKSTAT  packet_status;
    CSI_Q_MGMT   q_mgmt
    char         data[1];
} CSI_MSGBUF;
```

The network buffer is to be allocated to the size of the `CSI_MSGBUF` data structure (which is the size of the structure from *offset* through *data*), plus the size of *data* for the maximum size of messages that are expected. A "C" source code algorithm for this allocation is:

buffer_pointer=`malloc(name,sizeof(CSI_MSGBUF)+MAX_MESSAGE_SIZE)`

The `char` *data*`[1]` "C" coding construct used in defining `CSI_MSGBUF`, allows the *data* area to be dynamically allocated at the same time as the other portions of the buffer structure. See your UNIX documentation for a description of the `malloc()` system routine.

Polling for Application or NI Input

The SSI must poll for input from both the network and client applications resident on the client host. The method of polling for network and application input varies depending on the client host. Under 4.3BSD UNIX the `select()` system call is employed, for example.

The following are requirements for using `select()`:

- Establish a file descriptor from which the SSI can read application input
- Establish the RPC input file descriptors used in managing RPC network services
- Construct an input file descriptor mask to be passed to `select()`
- Upon return from `select()`, differentiate NI input from application input and make the appropriate function call:
  - Call `svc_getreq()` network service function for network input
  - Call a user function to process application input

Receiving a Request

The SSI receives requests from client applications. There are no constraints on the method the SSI uses to do this; the interface may be either a function call or an IPC mechanism. There is an implied constraint, however, in that the SSI must be able to relate a response to a request so that the response can be sent to the appropriate application.

The StorageTek SSI communicates with an application via a BSD socket IPC mechanism. In order to match a response with the application that issued the original request, it employs a queueing mechanism that stores the IPC return address of the application. In this case, the IPC return address is an application input socket number.

When the application request is first received, its return address is placed on the queue. The location on the queue is described by an integer returned from the queueing function. This integer is assigned to the *ssi_identifier* in the `CSI_HEADER` structure and is preserved by the CSI in its responses. When the SSI receives a Storage Server response, it uses the *ssi_identifier* to retrieve the application's IPC address from the queue, then it sends the response to the application at that IPC address.

For an example of how this mechanism works in the StorageTek SSI, see the CSI source file `csi_lminput.c`.

Formatting a Request Into a Storage Server Packet

Since there are no constraints with respect to the format and structure of messages sent to the SSI by an application, this requirement is to be defined by the customer.

The StorageTek SSI receives input as application-level Storage Server packets. They are distinguished from SSI–CSI Storage Server packets in that they have an `IPC_HEADER` at the topmost layer of the packet instead of a `CSI_HEADER`.

Since all packets sent from the SSI across the network must contain a `CSI_HEADER`, the `IPC_HEADER` is stripped off and replaced by a `CSI_HEADER` containing the RPC return address of the SSI. See *Input to the CSI From the SSI* in *Chapter 7* for a description of how to initialize the `CSI_HEADER`. See the CSI "C" source file `csi_lminput.c` for a description of how the StorageTek SSI strips off the `IPC_HEADER` and replaces it with a `CSI_HEADER`. See the CSI "C" source file `csi_rpccall.c` for a description of how the `CSI_HEADER` is initialized prior to making an RPC call to the CSI.

Sending a Storage Server Packet on the NI

The requirements for an SSI issuing an RPC call to the CSI are as follows:

- The `CSI_HEADER` must be initialized with the RPC return address of the SSI so that the CSI can send responses back to the SSI.
- The network buffer structure state information must be initialized properly before entry into `xdrrequest()`.
- The SSI must obtain the network address and, optionally, the port number of the CSI.
- The SSI must implement an appropriate RPC timeout–retry algorithm.
- The SSI must initiate a connection to the remote CSI via a call to either `clntudp_create()` or `clnttcp_create()`.
- The SSI must call `clnt_call()` which sends the request packet to the CSI's RPC dispatcher.
- The SSI must block while waiting for the CSI to send an acknowledge via a `svc_sendreply()` call.
- The SSI must call `clnt_destroy()` to terminate the connection to the CSI.

Initializing the CSI_HEADER

The SSI must initialize the top layer of a request, which is the `CSI_HEADER`, with its return address so that the CSI can use this address in sending Storage Server responses to the SSI. See *Input to the CSI From the SSI* in *Chapter 7* for a description of how to initialize the RPC return address.

The `CSI_HEADER` must also be stamped with a transaction ID (of type `CSI_XID`) consisting of the following:

- The SSI's return address (redundant for this implementation of the SSI/CSI, but not for future extensions)
- The current process ID
- A packet sequence number

This architecture was chosen so that the CSI can not only identify packet sequence, but also differentiate between SSIs running on different hosts and multiple SSIs running on a single host. See *Input to the CSI From the SSI* in *Chapter 7* for a description of how to initialize the `CSI_XID`.

Initializing the Network Buffer Structure

The SSI must initialize the network buffer structure (of type `CSI_MSGBUF`) before calling the `clnt_call()` RPC routine, since `clnt_call()` calls the XDR translation routine `csi_xdrrequest()`. See *Message Translation Structures* in *Chapter 7* for details on the required state of the network buffer on entry to `csi_xdrrequest()`.

Obtaining the Network Address of the CSI

The SSI must obtain network address of the CSI before the SSI calls `clnt_call()` in order to be able to properly initialize the first parameter on the interface. This parameter is a pointer to an Internet addressing structure (of type `struct sockaddr_in`).

The following are requirements for obtaining the remote address of the CSI:

- Prior knowledge of the name of the CSI host
- An entry for the CSI host in the SSI host's `/etc/hosts` file (or its equivalent)
- A system call, roughly equivalent to the UNIX system call `gethostbyname()`, which returns the address for a remote host.

Note: If the *sin_port* variable (the CSI port number in the Internet address structure used by the SSI on the `clnt_call()` interface) is set to `0`, the portmapper will always be consulted to obtain a target remote CSI port number before packets are sent.

Implementing a Timeout–Retry Algorithm

It is necessary to implement a timeout and retry algorithm, due to variations in the quality of network service and load on both the SSI and Storage Server host. The SSI detects poor or failed network service by exercising a timeout on a particular send attempt. Functionality for a timeout–retry scheme is provided within RPC. Depending on the nature of the client application, however, it may be desirable to implement the timeout–retry algorithm at a higher level, since once invoked, the built-in RPC method is outside of user control.

The RPC method of timeout–retry is specified by declaring a per-try timeout when initiating a connection via the `clntudp_create()` RPC call, and an overall timeout on the subsequent `clnt_call()`. Effectively, the overall number of tries is equal to the overall timeout divided by the per-try-timeout. For example, if the per-try-timeout passed to `clntudp_create()` is specified as `4` seconds, and the overall timeout passed to `clnt_call()` is specified as `20` seconds, then five tries at sending will be attempted within `clnt_call()`. If the `RPC_SUCCESS` return status is not forthcoming and no other RPC error condition exists, then an `RPC_TIMEDOUT` error condition is returned after

the *full* timeout has been attempted. The timeout factors need to be tailored to the individual computer installation.

The `clnt_call()`-embedded RPC retry scheme hypothetically could have various negative consequences on an SSI. One major consequence is that neither application input nor network input can be detected until the call returns. Consequently, packets may be dropped at either the application or the network input endpoints.

In order to enhance input detection and prevent the dropping of packets, it may be desirable to implement the retry algorithm within the SSI program code, rather than within the RPC call.

Following are the requirements for recovering from poor network service:

- Specify a per-try timeout, either within SSI program code or within RPC as a parameter to `clntudp_create()`.
- Specify an overall timeout, either within SSI program code or within RPC as a parameter to `clnt_call()`.
- Provide a mechanism for RPC error detection and recovery in case of RPC failures or RPC timeouts which prevents packets from being dropped.
- Depending on the individual SSI application, provide a mechanism whereby sending network output will not block the detection of either application or network input.

In order to provide a robust system, StorageTek has chosen to favor input over output in both its SSI and its CSI. If input is detected during network transmissions, transmission is temporarily halted, and the packet is placed on a network output queue. The CSI or SSI then receives either application or network input. This prevents the dropping of packets.

After input has been read, if no more input is pending, the StorageTek SSI flushes its network output queue, sending currently queued packets to the appropriate target SSI, and then sends the most recently received packet.

In all cases, it is imperative that the ordering of packets between a specified CSI/ SSI pair be preserved. Packets must be sent in the correct order.

In the StorageTek SSI, environment variables are used to fine-tune the retry algorithm so that the values for timeout and number-of-tries need not be hard-coded. The StorageTek SSI has its retry algorithm implemented outside of RPC. In general, the most successful timeout–retry approach has been to increase the number of retries rather than the duration of the timeout. StorageTek has successfully used a four second timeout combined with five retries in in-house testing.

Initiating a Connection to the CSI

The SSI initiates a connection to the CSI by calling either `clnttcp_create()` or `clntudp_create()`. The parameters in this call are set as follows:

- `struct sockaddr_in` is initialized to the CSI Internet address. To get the CSI address may require a call to `gethostbyname()` or its equivalent. The `/etc/hosts` file must contain the CSI host network address.
- The program number is set to `CSI_PROGRAM`.
- The version number is set to either `CSI_TCPVERSION` or `CSI_UDPVERSION`.
- The socket pointer is set to an open socket of the appropriate Internet family type, or to `RPC_ANYSOCK` which will cause RPC to create the socket.
- If using `clntudp_create()`, the timing parameter is set as discussed in the "Implementing a Timeout–Retry Algorithm" topic in this section.

This call returns a client handle structure which will be used in the subsequent invocation of `clnt_call()`.

Sending a Request to the CSI

The SSI sends a message packet to the CSI RPC dispatcher via a call to `clnt_call()`. The parameters in this call are set as follows:

- The first parameter is set to the client handle returned from `clnttcp_create()` or `clntudp_create()`.
- The procedure number is `CSI_ACSLM_PROC`.
- The XDR serialization procedure is `csi_xdrrequest()`.
- The pointer to the network buffer is set to a structure of type `CSI_MSGBUF`.
- The XDR procedure for data returned is not used, therefore it is set to `xdr_void()`.
- The network buffer for data returned is not used, therefore it is set to null.
- The overall timeout to be used is set as discussed in the Retry and Timeout section.

Invoking XDR Translation

In order to translate the request, the SSI issues a call to `svc_getargs()`. One of the parameters to this function is the XDR translation function, in this case `csi_xdrrequest()`, and the other is the network buffer where the serialized packet and information

describing its size and state will be placed. See *CSI Message Processing* in *Chapter 5* for a description of the operation of `csi_xdrrequest()`. See *Message Translation Structures* in *Chapter 7* for a description of the status information returned in the network buffer.

Reading a Response From the NI

The SSI receives Storage Server responses via the NI. Input from the network is detected when a poll of input file descriptors yields a network file descriptor as being active.

Depending on the particular implementation of the SSI, this input will usually be in the form and structure of an XDR-encoded Storage Server response packet sent by the CSI. Typically, this means that one of the file descriptors in the `svc_fds` global file descriptor mask, defined in the RPC library, has input pending. Following are the requirements for processing this input within the SSI:

- Detect network input as a result of input file descriptor polling.
- Issue a call to `svc_getreq()` which transparently further demultiplexes the RPC input.
- `svc_getreq()` calls the SSI's RPC dispatcher at a designated procedure number (the one originally specified by the SSI in the `CSI_HEADER` portion of the request).
- Call `svc_getargs()` within the RPC dispatcher, to retrieve the packet from the NI.
- `svc_getargs()` calls the XDR translation function passed to it, in this case `csi_xdrresponse()` to deserialize the packet.
- `csi_xdrresponse()` decodes the packet and determines if it is a duplicate of a previous transmission.
- Drop duplicate packets.
- Determine the address of the application destined to receive the packet.

Detection of Network Input

As stated previously, the StorageTek SSI uses the `select()` system call to detect network input. In this case, detection of network input involves comparing the global RPC `svc_fds` mask of network-dedicated file descriptors that are currently being used to the one(s) returned from `select()`. If network input is active, the call to `select()` returns a file descriptor mask with one or more bits set, indicating RPC input file descriptors.

Invoking RPC Handling of Input

Handling of RPC input is invoked by calling the `svc_getreq()` function. This routine has no parameters and initiates handling of input in a somewhat transparent manner, since it calls the SSI's RPC dispatcher on behalf of the SSI. It knows which dispatcher to call because the SSI previously registered its dispatcher during initiation on call to `svc_register()`.

Invoking the RPC Dispatcher

If the switch and case "C" coding construct has been used in construction of the SSI dispatcher, the switch trips on `request-handle-->procedure-number`, and the case entered will be at the value previously defined as the procedure number that the SSI originally defined in the `CSI_HEADER`. See the *Sun Network Programming Manual* for details on the switch and case construct.

Invoking XDR Translation

Within the procedure number specified by the SSI's RPC dispatcher, the SSI issues a call to `svc_getargs()` in order to have the network input translated and placed in the network buffer. One of the parameters to this function is the XDR translation function, in this case, `csi_xdrresponse()`, and the other is the network buffer where the deserialized packet and information describing its size and state will be placed. See *CSI Message Processing* in *Chapter 5* for a description of the operation of `csi_xdrresponse()`. See *Message Translation Structures* in *Chapter 7* for a description of the status information returned in the network buffer.

Detecting Duplicate Packets

The SSI must be able to identify and handle duplicate message packets. Even if the SSI exercises a timeout, the original message may still reach the CSI, although later than expected. If this happens, it is likely that the CSI will send a duplicate response to the request. See the "Timing Considerations" topic in the *CSI Message Processing* section in *Chapter 5* for a more detailed explanation of how duplicate packets occur.

`csi_xdrresponse()` detects duplicate packets using the `CSI_XID` in the `CSI_HEADER`: the address, process ID, and sequence number of the duplicate packet are identical to a previously received packet. Upon return from `csi_xdrresponse()`, the *packet_status* in the `CSI_MSGBUF` structure is set to `CSI_PAKSTAT_DUPLICATE_PACKET` for duplicate packets.

Depending on the implementation of the SSI, it may be desirable to drop duplicate packets. Currently `csi_xdrresponse()` detects a

duplicate packet in the initial phases of the deserialization process, returning immediately upon detecting the condition. Only the `CSI_REQUEST_HEADER` has been deserialized at this point. If duplicate packets are *not* to be dropped, then `csi_xdrresponse()` must be changed to completely deserialize the packet before returning.

Determining the Destination Application Address

There is no constraint with respect to communications between the SSI and the application. However there is a requirement that a response be paired with a particular request in order that the response can be sent to the appropriate application. See the "Receiving a Request" topic in this section for details.

Formatting the Storage Server Response

The Storage Server packet received from the network must be converted to a form that is intelligible by the client application. There is no constraint with respect to communications between the SSI and the application. The nature of this operation is determined by the customer.

Sending the Response to the Application

There is no constraint on communications between the SSI and the application. The form and content of this transaction is determined by the customer.

Request Processing Summary

The following series of steps outlines the interaction between an SSI and the CSI to process a Storage Server request. Only the steps performed by the SSI are detailed. See the *CSI Message Processing* section in *Chapter 5* for details on the steps performed by the CSI.

SSI:
1. Receives a request from one of its client applications.
2. Calls either the `clnttcp_create()` or the `clntudp_create()` RPC function to establish a TCP/IP or UDP/IP connection to the CSI at its published RPC address.
3. Calls the `clnt_call()` RPC function to send the request across the network.
4. `clnt_call()` calls a serialization function to convert the response into XDR format. See the *CSI Message Processing* section in *Chapter 5* for a description of the StorageTek-supplied XDR translation functions.

CSI:
5. Receives the request, translates it, and sends it to the ACSLM. Then, receives and translates responses from the ACSLM, and routes them to the appropriate SSI.

SSI:

6. Calls the `svc_getreq()` or `svc_getreqset()` RPC function to receive the message. The following occurs:
   - 6.1 The `svc_getreq()` or `svc_getreqset()` function calls `csi_rpcdisp()`, the dispatching service that was registered during initiation.
   - 6.2 `csi_rpcdisp()` calls `svc_getargs()` to get the data off the network.
   - 6.3 `svc_getargs()` calls a deserialization function to convert the response into host-dependent format. See the *CSI Message Processing* section in *Chapter 5* for a description of the StorageTek-supplied XDR translation functions.
7. Immediately acknowledges receipt of the message by calling the `svc_sendreply()` RPC function, passing no data.
8. If this is a *final response*, deletes the CSI's return address from the connect queue.
9. Calls an RPC function to close the connection.

CHAPTER 7:
CSI DATA STRUCTURES

OVERVIEW

This chapter describes the common data structures, variables, and external libraries used by the CSI and the SSI. The use of common structures supports both message decoding and extraction of routing information. Common elements consist of the following:

- External interfaces
- Environment variables
- Input to the CSI from the SSI
- Input to the CSI from the ACSLM
- Output from the CSI to the SSI
- Message translation data structures

EXTERNAL INTERFACES

Overview

This section identifies software libraries, functions, and variables that are required for CSI and SSI operation. This software is not supplied by StorageTek.

Software Libraries

The following software libraries, supplied with the Storage Server, are required for the CSI:

- UNIX operating system
- Sun Microsystems Remote Procedure Call (RPC)
- Sun Microsystems External Data Representation (XDR)

Communications Protocol Interfaces

The following communications protocols are required for transmission of data:

- TCP/IP (transparently handled by RPC)
- UDP/IP (transparently handled by RPC)
- Special application-level protocol superimposed on RPC protocol. See the *Communications Methodology* section in *Chapter 5*.

Both the CSI and the SSI use the low-level RPC routines since TCP/IP is not supported by the intermediate-level RPC calls.

Functions and Variables

The following RPC & UNIX functions and global variables are used in the CSI:

```
svc_getargs()
svc_getreq()
clnt_call()
clnt_destroy()
clnt_sperrno()
clnttcp_create()
clntudp_create()
pmap_set()
pmap_unset()
svc_destroy()
svc_freeargs()
svc_getargs()
svc_register()
svc_sendreply()
svctcp_create()
svcudp_create()
svcerr_noproc()
svcerr_decode()
xdr_array()
xdr_bool()
xdr_bytes()
xdr_char()
xdr_double()
xdr_enum()
xdr_float()
xdr_free()
xdr_int()
xdr_long()
xdr_opaque()
xdr_short()
xdr_string()
xdr_u_char()
xdr_u_int()
xdr_u_long()
xdr_u_short()
xdr_vector()
xdr_void()
xdr_wrapstring()
svc_fds (file descriptors)
clnt_stat (data structure)
rpc_createerr
gethostbyname()(UNIX 3N )
gethostname()(UNIX 3N)
```

ENVIRONMENT VARIABLES

The following environment variables are used by the CSI. They are listed below as defined in the `csi.h` header file.

```
#define CSI_TCP_RPCSERVICE   "CSI_TCP_RPCSERVICE"
#define CSI_UDP_RPCSERVICE   "CSI_UDP_RPCSERVICE"
#define CSI_CONNECT_AGETIME "CSI_CONNECT_AGETIME"
#define CSI_HOSTNAME         "CSI_HOSTNAME"
#define CSI_RETRY_TIMEOUT    "CSI_RETRY_TIMEOUT"
#define CSI_RETRY_TRIES      "CSI_RETRY_TRIES"
#define CSI_TRACE_VALUE      "TRACE_VALUE"
```

- `CSI_TCP_RPCSERVICE` is used to define whether the CSI will operate as a TCP RPC Server. This variable can be set as follows:

  ```
  "TRUE"
  "FALSE"
  ```

- `CSI_UDP_RPCSERVICE` is used to define whether the CSI will operate as a UDP RPC server. This variable can be set as follows:

  ```
  "TRUE"
  "FALSE"
  ```

- `CSI_CONNECT_AGETIME` defines the value of the maximum age of pending requests in the CSI's request queue. This variable is accessed as a "C" character array (string) datatype, expressed as an integer number of seconds. For example, an entry of `172800` indicates two days.

  Messages that are older than this value are removed from the queue, as it is assumed that they will never be responded to. The CSI sends an entry to the Event Log when this happens. See *Error Detection and Recovery* in *Chapter 5*.

- `CSI_HOSTNAME` defines for the SSI the hostname of its CSI server. Defining this variable is optional.
- `CSI_RETRY_TIMEOUT` and `CSI_RETRY_TRIES` are used together to determine the minimum total time over which the CSI will attempt to send a message. Network transmission failures are detected using a retry algorithm. While the algorithm itself is transparent to the programmer, the functionality of the algorithm is controlled by setting these two environment variables.
  - `CSI_RETRY_TIMEOUT` defines the minimum amount of time, in seconds, that the CSI should wait between attempts at establishing a network connection. Its value is placed in the timeout variable used on either the `clntudp_create()` or `clnttcp_create()` RPC library calls. See *Error Detection and Recovery* in *Chapter 5*.

- `CSI_RETRY_TRIES` defines the number of attempts the CSI should make to transmit a message. Pending messages are discarded if a connection cannot be established within the number of tries defined.

Both of these variables can be set at Storage Server installation, and reset at reconfiguration. See the *UNIX Storage Server System Administrator's Guide* for details on Storage Server installation and reconfiguration. If they are not set, defaults are supplied as defined in the `csi.h` header file. Since the timeout chosen will be a factor of system and network processing loads, these environment variables can be used to tailor CSI response to expected system load before the CSI is initiated.

- `CSI_TRACE_VALUE` is used to enable packet tracing by the CSI. Setting trace to `00800000` before initiating the CSI will cause packet tracing to be enabled. See the *CSI Message Processing* section in *Chapter 5*.

INPUT TO THE CSI FROM THE SSI

Request Structure

Input to the CSI from the NI is in the format of Storage Server request packets which are *deserialized* from XDR format using the StorageTek-supplied XDR interface function, `csi_xdrrequest()`. Use of this high-level XDR interface makes deserialization transparent to the caller; therefore, CSI input is discussed in terms of its format *after* deserialization.

After being deserialized, CSI input has the following format, as defined in the `csi_structs.h` header file:

```
struct {
    CSI_HEADER      csi_header;
    MESSAGE_HEADER  message_header;
                    •
                    •
          (message_data)
                    •
};
```

Note: `CSI_HEADER` and `MESSAGE_HEADER` together define the `CSI_REQUEST_HEADER`.

The structure of the `CSI_HEADER` is described in the following section.

The `MESSAGE_HEADER` specifies ACSLM request information according to a predefined format. See the *Requests* section in *Chapter 3* for the structure and contents of `MESSAGE_HEADER`.

csi_header Structure

Description

CSI_HEADER specifies client system return address information, as defined by a client. Both the CSI and the SSI function as clients, as well as servers. When the CSI receives a message from the SSI, it strips off the CSI_HEADER, saves it, and uses it to route response messages. The values in the CSI_HEADER are returned to the SSI in the CSI's responses.

The structure of the CSI_HEADER is defined in the csi_header.h header file which is compiled directly into CSI program code. The CSI_HEADER definitions are required for the SSI, but the csi_header.h file is not.

Format

```
typedef struct {
    CSI_XID          xid;
    unsigned long    ssi_identifier;
    CSI_SYNTAX       csi_syntax;
    CSI_PROTOCOL     csi_proto;
    CSI_CONNECT      csi_ctype;
    CSI_HANDLE_RPC   csi_handle_rpc;
} CSI_HEADER;
```

Values

xid serves as a transaction identifier. It is used by both the CSI and the SSI to identify duplicate packets. Initialization of this variable is required.

ssi_identifier is reserved for the exclusive use of the SSI and is preserved by the CSI in its responses. It is intended to be a routing identifier which the SSI can use to identify a particular client application, if the client architecture allows multiple client applications to use a single SSI for network services.

csi_syntax is a version-independent variable that specifies the translation syntax the client SSI uses to communicate with the CSI. The CSI uses this identifier to determine the particular type of byte stream to be passed across the NI. It enables the CSI to apply the correct deserialization routines when more than a single translation syntax is supported. Currently only the XDR translation syntax is supported. The *csi_syntax* variable should therefore be set to CSI_SYNTAX_XDR, as defined in the ssi.h header file.

csi_proto is a version-independent variable that specifies the transport protocol used between the SSI and CSI over the NI. The CSI uses this identifier to make the appropriate type of connection when executing a

callback to the client SSI. The current implementation employs TCP/IP and UDP/IP as transport mechanisms. This variable should be set to `CSI_PROTOCOL_TCP` or `CSI_PROTOCOL_UDP`, as defined in `csi_header.h`.

csi_ctype is a version-independent variable that specifies the type of the session layer connection and data transfer interface between the SSI and the CSI. It tells the CSI the structure of the client return address handle, *csi_handle_rpc*. Therefore this variable should be set to `CSI_CONNECT_RPCSOCK`.

csi_handle_rpc contains version-specific return address information that enables the CSI to send responses to the SSI.

Note: Since IP is being used for the network software layers, *csi_handle_rpc* is a `sockaddr_in` Internet address structure. The definition for this structure resides in `netinet/in.h`. The "C" `include` statement for this header file should be:

```
#include <netinet/in.h>.
```

csi_xid Structure

Description

CSI_XID is used in CSI_HEADER to filter out duplicate packets. It uniquely identifies a message by address, process, and sequence number.

Format

```
typedef struct {
    unsigned char addr[CSI_NETADDR_SIZE];
    unsigned int  pid;
    unsigned long seq_num;
} CSI_XID;
```

Values

addr is the SSI's host address.

pid is the SSI's process ID.

seq_num is a sequence number generated by the SSI that identifies the sequencing of messages sent from a particular SSI.

csi_handle_rpc Structure

Description

CSI_HANDLE_RPC is used in CSI_HEADER. By definition, the contents of this structure are specific to the Session Layer network software implementation. The format shown below is used for UNIX operating systems that support RPC and the implementation of 4.3BSD sockets.

Format

```
typedef struct {
    unsigned long        program;
    unsigned long        version;
    unsigned long        proc;
    struct sockaddr_in   raddr;
} CSI_HANDLE_RPC;
```

Values

program is set to the transient port-mapped RPC program number for the SSI, to which the CSI must return Storage Server responses.

version is set to the current version number of SSI server software, per RPC requirements for registration and port mapping. Per csi.h, the TCP version is defined as CSI_TCP_VERSION, and the UDP version is defined as CSI_UDP_VERSION.

proc is set to the RPC procedure number to be executed by the client SSI callback service dispatcher in response to a CSI RPC callback.

raddr is used for Internet addressing and varies according to the RPC implementation. All of the *raddr* information can be directly set to the values of the sockaddr_in structure established when making the transient port-mapping for the client callback service. See *Communications Methodology* in *Chapter 5* for information on transient port-mapping.

sockaddr_in Structure

Description

The following is a definition of the Internet addressing "C" programming structure defined in the `in.h` header file.

Format

```
struct sockaddr_in {
    short           sin_family;
    u_short         sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

Values

sin_family is the transport protocol family to be used in sending responses from the CSI to the SSI. In the current implementation it should be set to `AF_INET` to indicate that Internet addressing is being used.

sin_port is set to the number of the port to which the CSI executes callbacks. If this is `0`, the CSI RPC mechanism will reference the RPC port mapper before initiating network transfers in order to determine the correct port number.

sin_addr is set to the Internet address of the SSI's host machine.

sin_zero should be zeroed out.

INPUT TO THE CSI FROM THE ACSLM

Response Structure

Input from the ACSLM is detected on an ongoing basis when the active file descriptor is of the ACSLM type. This input is always a Storage Server response packet, and has the following structure, as defined in the `lm_structs.h` header file.

```
struct {
    IPC_HEADER          ipc_header;
    MESSAGE_HEADER      message_header;
                  •
                  •
      (message_data)
                  •
};
```

Note: `IPC_HEADER` and `MESSAGE_HEADER` together define the `REQUEST_HEADER`.

The structure of the `IPC_HEADER` is described in the following section.

The `MESSAGE_HEADER` specifies ACSLM request information according to a predefined format. See the *Requests* section in *Chapter 3* for the structure and contents of `MESSAGE_HEADER`.

ipc_header Structure

Description

IPC_HEADER provides IPC "return address" information for requests and responses between Storage Server processes. This allows the number of input sockets for any process to be limited to one. The format shown is for the BSD socket IPC mechanism.

Format

```
typedef struct {
    unsigned long  byte_count
    TYPE           module_type;
    unsigned char  options;
    unsigned long  seq_num
    unsigned char  return_socket[SOCKET_NAME_SIZE];
    unsigned int   return_pid
    unsigned long  ipc_identifier;
} IPC_HEADER;
```

Values

byte_count is the length of the message, including the header, in bytes.

module_type is a "C" enumeration that identifies the process sending the message. See the *Common Variables* section in *Chapter 3* for definitions of these types.

seq_num is a message sequence number used to verify message order and identify duplicate message packets.

options currently has no defined use. It is reserved for future capabilities and is maintained as a space filler.

return_pid is the process ID generated by the sending process.

return_socket specifies the input socket name of the sending process. If the message is a request, this is the socket name to which replies are to be sent. The string size is limited to the file name size limitation of non-BSD UNIX file systems.

ipc_identifier is used optionally by the requesting process for information to assist in synchronizing responses. This field is set by the process originating the request message, and is copied by the receiving process into its response message.

OUTPUT FROM THE CSI TO THE SSI

Response Structure

CSI output to the SSI is in the format of Storage Server response packets which are *serialized* into XDR format using the StorageTek-supplied XDR interface function, `csi_xdrresponse()`. Use of this high-level XDR interface makes serialization transparent to the caller; therefore, CSI output is discussed in terms of its format *before* serialization.

After being serialized, a response packet has the following format, as defined in the `csi_structs.h` header file:

```
struct {
    CSI_HEADER          csi_header;
    MESSAGE_HEADER      message_header;
                •
                •
          (message data)
                •
};
```

Note: `CSI_HEADER` and `MESSAGE_HEADER` together define the `CSI_REQUEST_HEADER`.

The `CSI_HEADER` is preserved by the CSI and passed back to the SSI unaltered. The CSI matches one or more responses to the `CSI_HEADER` address of the originating SSI, and sends the responses to the client SSI. See the *Input to the CSI From the SSI* section in this chapter for the structure and contents of `CSI_HEADER`.

`MESSAGE_HEADER` specifies ACSLM request information generated by the client application. The ACSLM generally copies this information into its corresponding response messages. See the *Requests* section in *Chapter 3* for the structure and contents of `MESSAGE_HEADER`.

MESSAGE TRANSLATION STRUCTURES

csi_msgbuf Structure

Description

CSI_MSGBUF defines the buffer for translated Storage Server message packets. It is used in the clnt_call() and svc_getargs() calls to csi_xdrrequest() and csi_xdrresponse(). See *Appendix C: XDR Translation Functions* for details on these functions.

CSI_MSGBUF should be allocated for the size of its fields plus the size of the data area desired. The minimum allocation size, described as a "C" language expression, is

```
sizeof(CSI_MSGBUF)+MAX_MESSAGE_SIZE
```

MAX_MESSAGE_SIZE is defined in the defs.h header file.

Format

```
typedef struct {
    int          offset;
    int          size;
    int          maxsize;
    int          translated_size;
    CSI_PAKSTAT  packet_status;
    CSI_Q_MGMT   q_mgmt
    char         data[1];
} CSI_MSGBUF;
```

Values

offset is the starting position of data in the buffer, expressed in bytes. The packet data will start *offset* number of bytes from the start of *data*. The CSI_PAK_NETDATAP, supplied in the csi.h header file, can be used to calculate the starting byte of a packet in *data*. See *Appendix D: CSI and SSI Required Files* for the csi.h header file.

size is the size of the message packet, in bytes, before serialization or after deserialization.

maxsize is the maximum size of packet data in the buffer. It should always equal the allocated size of *data* in bytes.

translated_size is the number of bytes successfully translated during serialization or deserialization. Comparisons of *size* to *translated_size* are useful for detecting certain error conditions: for example, *translated_size* might be less than *size* during serialization if part of a Storage Server packet could not be translated but was nevertheless sent.

packet_status is the status of the packet after decoding. Possible values are:

- `CSI_PAKSTAT_XLATE_COMPLETED`, if packet translation was successful.
- `CSI_PAKSTAT_XLATE_ERROR`, if there was a translation error.
- `CSI_PAKSTAT_DUPLICATE_PACKET`, if a duplicate packet was detected; only the CSI request header will be translated.

q_mgmt is a data structure dedicated to tracking the retry state of network packet send attempts for packets residing on the CSI network output queue. Currently, only the number of transmission attempts is tracked in the structure variable *xmit_tries*. *xmit_tries* is incremented on each invocation of `clnt_call()`.

data is an area of memory where the message packet is to be stored. This should be allocated to an area of memory sized `MAX_MESSAGE_SIZE` bytes or larger. See *Common Variables* in *Chapter 3* for a definition of `MAX_MESSAGE_SIZE`.

(INTENTIONALLY LEFT BLANK)

APPENDIX A:
EVENT LOG MESSAGES

EVENT LOG FORMAT

Event Log entries are ASCII text, allowing the Log to be viewed at any terminal or printed to any printer.

All Event Log entries have a consistent format. Each entry contains a one-line prefix, followed by one or more lines of message text. Figure A-1 is an example of some Event Log entries.

mm-dd-yy hh:mm:ss component_name [*nn*]:
module_name: One or more lines of message text ...

```
06-01-89 17:59:15 CSI[0]:
csi_init(); Initiation started

06-01-89 18:01:02 ACSSA[0]:
sa_demux: CAP   0, 0: Place cartridges in CAP.

06-01-89 18:02:13 AUDIT[1]:
au_initiate: audit started
```

Figure A-1. Sample Event Log

The one-line prefix is broken down as follows:

- *mm-dd-yy hh:mm:ss* are the date and time of entry.
- *component_name* is an abbreviation for the originating Storage Server component. This could be ACSLM, ACSSA, CSI etc.
- [*nn*] is the request ID enclosed in square brackets. This ID is generated by the ACSLM when it receives a valid request and is displayed by a `query` request.

The *module_name* that precedes the message text is the name of the Storage Server program module that generated the message. This is included to help a Customer Services Engineer isolate the cause of the

problem; it is not intended to be used by System Administrators or library users.

EVENT LOG ENTRIES

The following Event Log messages are of particular interest to a System Administrator or a programmer. They are not the only ones that may be written to the Event Log, but they are ones most frequently sent. Messages not listed here indicate a low-level error and should be brought to the attention of a Software Support Representative or a Customer Services Engineer.

The messages are grouped by the software component that generates them (the *component_name* in the one-line message prefix). Within software component, the messages are listed alphabetically. Common messages which may be generated by any component comprise the first group.

COMMON ENTRIES – UNSOLICITED MESSAGES

The following are Event Log messages that can occur during Storage Server processing. The *component_name* in their message prefix is the Storage Server component that generated the message. The entries are listed in alphabetical order. **Note:** These entries also appear in the Display Area of the Command Processor as unsolicited messages.

- This message indicates that cartridges are detected in the CAP during initiation or recovery.

  `CAP` *cap_id*`: Cartridges detected in CAP.`

  cap_id is the ID of the CAP.

- This message indicates that the specified CAP is ready to receive cartridges. This message is repeated at approximately one minute intervals until the CAP door is opened.

  `CAP` *cap_id*`: Place cartridges in the CAP.`

  cap_id is the ID of the CAP.

- This message indicates that the specified CAP contains cartridges and is ready for the operator to remove them. This message is repeated at approximately one minute intervals until the CAP door is opened.

  `CAP` *cap_id*`: Remove cartridges from the CAP.`

  cap_id is the ID of the CAP.

- This message indicates that the ACSLM is unable to access the data base. A data base error code, indicating the reason for the failure, will also be written to the Event Log.

  ```
  Data Base Failure.
  EXEC SQL sql_command
  ```

  sql_command is the SQL statement on which the error occurred.
  Note: The second line of the message is *not* included in the Display Area message.

- This message indicates that the library hardware is operable, but with degraded performance.

  type identifier: `Degraded mode` *fault_symptom_code*.

  type is the device type. *identifier* is the device ID. *fault_symptom_code* is a four-character hexadecimal code that provides a Customer Services Engineer with information needed to troubleshoot the problem.

- This message indicates that the specified device has been varied to the diagnostic state and is therefore available for requests submitted through the Command Processor only.

  type identifier: `Diagnostic.`

- This message indicates that the specified drive needs to be cleaned.

  `Drive` *drive_id*: `Clean drive.`

  drive_id is the ID of the library drive.

- This message indicates that the Event Logger is unable to open or write to the Event Log file.

  ```
  Event log access failed.
  ```

- This message indicates that the Event Log has reached the maximum size defined during installation. This message will be displayed at one minute intervals until you reduce the size of the Event Log.

  ```
  Event log is full.
  ```

- This message indicates that the ACSLM or ACSSA cannot communicate with another Storage Server software component.

  `IPC failure on socket` *socket_id*.

  socket_id is the ID of the failing socket.

- This message indicates that a library hardware error occurred.

  type identifier: `Library error` *status.*

  type is the device type. *identifier* is the device ID. *status* is an explanation of the error.

- This message indicates that the library configuration specified in the data base is not the same as that defined in the LMU by a Customer Services Engineer, or if a component appears in the data base but fails to respond to LMU commands. This error causes the Storage Server to terminate.

  `Library configuration error.`

- This message indicates that the specified LSM has failed to recover in-transit cartridges during Storage Server recovery.

  `LSM` *lsm_id*: `In-transit cartridge recovery incomplete.`

  lsm_id is the ID of the LSM containing the in-transit cartridges.

- This message indicates that the specified device has been varied offline. See the `vary` command description for additional details.

  type identifier: `Offline.`

  type is the device type. *identifier* is the device ID.

- This message indicates that the specified device has been varied online. See the `vary` command description for additional details.

  type identifier: `Online.`

  type is the device type. *identifier* is the device ID.

- This message indicates that the ACSLM has been placed in the idle state and is therefore unavailable for requests using library resources.

  `Server System idle.`

- This message indicates that the ACSLM is in an idle-pending state and is therefore unavailable for requests using library resources.

  `Server system idle is pending.`

- This message indicates that a timeout has occurred during network data handling. Data may have been lost.

  `Server System network interface timeout.`

- This message indicates that Storage Server recovery has been completed successfully.

  `Server system recovery complete.`

- This message indicates that Storage Server recovery has failed.

  ```
  Server system recovery failed.
  ```

- This message indicates that Storage Server recovery has been initiated.

  ```
  Server system recovery started.
  ```

- This message indicates that the CSI has encountered a Remote Procedure Call (RPC) failure. Data may have been lost.

  ```
  Server system RPC failure.
  ```

- This message indicates that the ACSLM has been placed in the run state.

  ```
  Server system running.
  ```

COMMON ENTRIES

- This entry indicates that a process has received an invalid status from another process.

  calling_module: *module_called* unexpected status = *status*

  calling_module is the software module making the call. *module_called* is the module that was called. *status* is the invalid status received.

ACSLH

The following messages are generated by the ACS Library Handler (ACSLH) component of the ACSLM. The *component_name* in their message prefix is ACSLH. The entries are listed in alphabetical order.

- This entry indicates that the ACSLH has detected an invalid CAP door status value in an LSM status response from the LMU.

  ```
  bad LSM STATUS cap door status value.
  ```

- This entry indicates that the ACSLH has detected an invalid LSM door status value in an LSM status response from the LMU.

  ```
  bad LSM STATUS door status value.
  ```

- This entry indicates that the ACSLH has detected an invalid hand indicator value in an LSM status response from the LMU.

  bad LSM STATUS hand *n* indicator value.

  n is the number of the robot hand with the invalid indicator. *n* is either 0 or 1.

- This entry indicates that the ACSLH has detected an invalid hand status value in an LSM status response from the LMU.

  `bad LSM STATUS hand` *n* `status value.`

  n is the number of the robot hand with the invalid status. *n* is either `0` or `1`.

- This entry indicates that the ACSLH has detected an invalid line value in an LSM status response from the LMU.

  `bad LSM STATUS line value.`

- This entry indicates that the ACSLH has detected an invalid ready value in an LSM status response from the LMU.

  `bad LSM STATUS ready value.`

- This entry indicates that the ACSLH has experienced an overflow in its message buffer. The LMU will automatically resubmit the message to the ACSLH, so there should be no loss of data. This message should be a cause for concern only if it appears frequently, in which case you should contact your Customer Services Engineer.

  `buffer overrun ... lmu messages will be lost.`

- The ACSLH has received a message from the LMU that it is unable to interpret. This may be a problem with the way the ACSLH is handling LMU messages.

  `Invalid message code received:` *message_code*.

  message_code is the code received.

- The following entry indicates a problem with the RS423 line that runs between the server system and the LMU. This message is issued after the ACSLH has been unable to establish communications with the LMU for approximately five minutes. It is reissued every five minutes thereafter until communications are reestablished.

  `LMU communication failure.`

- The following entry indicates that the Storage Server data base indicates an LSM that does not appear in the library configuration defined in the LMU. This error will most likely occur in response to a library request.

  `lsm:` *lsm_id* `not configured !!!`

  lsm_id is the LSM ID from the data base.

- The following entry indicates that the number of circuits defined in the library configuration exceeds the maximum limit.

  `maximum number of circuits are already open.`

- The following entry indicates that the library configuration defined in the LMU indicates a Pass-Thru Port (PTP) that the LSM robot is unable to locate. This error will most likely occur during Storage Server initialization or recovery.

  ```
  No Pass-Through ports exist.
  ```

- The following entry indicates a problem with the RS423 line that runs between the server system and the LMU. This message is issued after the ACSLH has been unable to establish communications with the LMU for approximately one minute. It is reissued every minute thereafter until communications are reestablished.

  ```
  Possible LMU communication failure.
  ```

- This entry indicates that the ACSLH did not detect a "start of frame" character at the beginning of a message from the LMU. The LMU will automatically resubmit the message to the ACSLH, so there should be no loss of data. This message should be of concern only if it appears frequently, in which case you should contact your Customer Services Engineer.

  ```
  Start of frame character not found.
  ```

- This entry indicates that the ACSLH has detected an invalid error response from the LMU.

  unexpected LMU error response *lmu_response*.

  lmu_response is the response received from the LMU.

- This entry indicates that the ACSLH has detected an invalid response from the LMU.

  unexpected response received (tran_code=*lmu_response*).

  lmu_response is the response code received from the LMU.

ACSLM

The following messages are generated by the ACSLM. The *component_name* in their message prefix is ACSLM. The entries are listed in alphabetical order.

- This entry indicates that the ACSLM has received a message that is too small from a CSI or the ACSSA. The ACSLM does not attempt to interpret the message because it does not have enough information.

  byte count(*byte_count*) too small for min packet size(*min_size*) ignored.

byte_count is the number of bytes in the message. *min_size* is the minimum size of a valid, readable message.

- This entry indicates that the CAP door has been closed after having been opened.

  `CAP Door Closed.`

- This entry indicates that the CAP door has been opened.

  `CAP Door Opened.`

- This entry indicates that the LSM access door has been closed after having been opened.

  `LH_MSG_TYPE_DOOR_CLOSED received for` *lsm_id*.

  lsm_id is the LSM that has been closed.

- This entry indicates that the LSM access door has been opened.

  `LH_MSG_TYPE_DOOR_OPENED received for` *lsm_id*.

  lsm_id is the LSM that has been opened.

- This entry indicates that an LMU has been placed online.

  `LH_MSG_TYPE_LMU_READY received for` *acs_id*.

  acs_id is the ACS to which the LMU is connected.

- This entry indicates that the LSM has been taken offline.

  `LH_MSG_TYPE_LSM_NOT_READY received for` *lsm_id*.

  lsm_id is the LSM that has gone offline.

- This entry indicates that the LSM has been placed online.

  `LH_MSG_TYPE_LSM_READY received for` *lsm_id*.

  lsm_id is the LSM that is online.

- This entry indicates that a port between the server system and the LMU has been taken offline.

  `LH_MSG_TYPE_PORT_OFFLINE received for` *port_id*.

  port_id is the identifier of the port that has gone offline.

- This entry indicates that the ACSLM has encountered a fatal error, such as a data base failure or an inconsistency in the library configuration. This is a fatal error to the ACSLM. The ACSLM will automatically initiate recovery processing if it is able. If recovery does not start automatically, the system must be rebooted.

  `Severe Error (`*status*`), Exiting to ACSSS.`

  status is a message indicating the nature of the severe error.

- This entry indicates that the ACSLM has received a library degraded mode message with an invalid device type.

  ```
  Unexpected LH_ADDR_TYPE(device_type) received on
  DEGRADED MODE Msg.
  ```

  device_type is the invalid device type received in the message.

- This entry indicates that the ACSLM has detected a request with an IPC_HEADER *module_type* not set to TYPE_CSI or TYPE_SA. The ACSLM will only process requests received from a client application through the CSI or from a user through the ACSSA.

  ```
  Unsupported module type module_type detected: discarded
  ```

  module_type is the invalid entry.

ACSSA

The following messages are generated by the ACSSA. The *component_name* in their message prefix is ACSSA. The entries are listed in alphabetical order.

- This entry indicates that the ACSSA has received a message packet with an IPC identifier not found in the request queue. The ACSSA is unable to process the message.

  ```
  Unknown packet received, command = command,
  identifier = ipc_id.
  ```

  command is the entry in the MESSAGE_HEADER. *ipc_id* is the identifier assigned to this message (used to synchronize requests and responses).

AUDIT

The following entries are generated by the audit request. The *component_name* in their message prefix is AUDIT. The entries are listed in alphabetical order.

- This entry indicates that audit processing has been cancelled. The data base may have discrepancies or errant cartridges may not have been ejected, therefore the audit should be rerun.

  ```
  Audit cancelled
  ```

- This entry indicates that audit processing has completed successfully.

  ```
  Audit completed
  ```

- This entry indicates that audit processing has terminated due to some error condition. The description of the error is displayed in the Command Area. The data base may have discrepancies or errant cartridges may not have been ejected, therefore the `audit` should be rerun.

  `Audit failed`

- This entry indicates that `audit` processing has begun.

  `Audit started`

- This entry indicates that the robot has found a cartridge with a duplicate external label.

  `Cartridge` *vol_id* `ejected from location` *cell_id*`, duplicate label.`

  The *vol_id* is the tape cartridge with the duplicate label. The *cell_id* is the storage cell location where the cartridge was found.

- This entry indicates that the robot has encountered a cartridge with an unreadable or nonexistent external label.

  `Cartridge` *vol_id* `ejected from location` *cell_id*`, unreadable label.`

  The *vol_id* is the external label of the tape cartridge; question marks (`?`) are substituted for the characters that the robot is unable to read. The *cell_id* is the storage cell location where the cartridge was found.

- This entry indicates that a tape cartridge *not* listed in the data base is found in the ACS. The cartridge is added to the data base.

  `Cartridge` *vol_id* `found at location` *cell_id*`.`

  The *vol_id* is the external label of the tape cartridge. The *cell_id* is the storage cell location where the cartridge was found.

- This entry indicates that a tape cartridge is not in the location defined by the data base. The cartridge is not moved in the ACS; instead, the data base is updated to the new storage location.

  `Cartridge` *vol_id*`, new location` *cell_id*

  The *vol_id* is the external label of the tape cartridge. The *cell_id* is the assigned storage cell location of the cartridge.

- This entry indicates that a tape cartridge listed in the data base is *not* found in the ACS. The cartridge is removed from the data base.

  `Cartridge` *vol_id* `not found`

  The *vol_id* is the external label of the tape cartridge.

- This entry indicates that another process has reserved a cell record in the data base and the `audit` process is unable to access it after the appropriate number of retries and timeouts. The `audit` continues with the next cell.

  `cell` *cell_id* `reserved by another process.`

  cell_id is the ID of the cell record.

- This entry indicates that the LSM robot has unexpectedly found a cartridge in a CAP cell during ejection of cartridges. This will occur if the operator did not completely empty the CAP during a previous `eject` operation. The `audit` process will issue an unsolicited message to empty the CAP, and will then resume ejecting cartridges after the CAP door is closed.

  `Destination location full: CAP cell` *cell_id*.

  cell_id is the location of the CAP cell.

- This entry indicates that the LSM robot has detected that a storage cell is missing from the LSM. This is a library configuration error and causes the Storage Server to terminate. The `audit` should be rerun after the error has been corrected and the Storage Server has gone through recovery.

  `missing cell` *cell_id* `detected.`

  cell_id is the location of the missing cell.

- This entry indicates that a spawned `audit` process has sent an incomplete or unintelligible message to the parent `audit` process. As a result, some errant cartridges may not be ejected. The `audit` should be rerun, unless the *audit_status* is `Audit complete`.

  audit_status.
  `Not all cartridges were ejected, messages lost.`

  audit_status can be either `Audit cancelled`, `Audit complete`, or `Audit failed`.

- This entry indicates that the `audit` process is unable to eject cartridges. Possible causes are the LSM being forced offline or suffering a failure while the `audit` is ejecting cartridges. The `audit` should be rerun, unless the *audit_status* is `Audit complete`.

  audit_status.
  `Not all cartridges were ejected, status =` *status_msg*.

  audit_status can be either `Audit cancelled`, `Audit complete`, or `Audit failed`. *status_msg* is the reason for the failure.

- This entry indicates that a cartridge marked for ejection is no longer found in its storage cell when the robot goes to move it to the CAP. The `audit` terminates and should be rerun.

  ```
  Source location empty: Cell cell_id.
  ```

  cell_id is the assigned storage cell location of the cartridge.

CSI

The following messages are generated by the CSI. The *component_name* in their message prefix is `CSI`. The entries are listed in alphabetical order. **Note:** Since these messages are of special interest to SSI programmers, the message status code for each is shown under the message text.

- This entry indicates that the CSI has detected a message from the ACSLM but is unable to read it.

  ```
  Cannot read message from the ACSLM: discarded.
  MSG_ACSLM_READ_FAILURE
  ```

- This entry indicates that the CSI is unable to reply to an RPC message because the call to the `svc_sendreply()` function has failed. See the *Sun Network Programming Manual*, "Remote Procedure Call Programming Guide."

  ```
  Cannot reply to RPC message.
  MSG_RPC_CANT_REPLY
  ```

- This entry indicates that the ACSLM IPC mechanism is unable to accept a message from the CSI. The CSI discards the message after the appropriate number of retries with timeouts.

  ```
  Cannot send message to ACSLM: discarded.
  MSG_ACSLM_SEND_FAILURE
  ```

- This entry indicates that the CSI is unable to communicate with the ACSSA. The CSI discards the message after the appropriate number of retries with timeouts.

  ```
  Cannot send message to ACSSA: discarded.
  MSG_SEND_ACSSA_FAILURE
  ```

- This entry indicates that the NI's communications mechanism is unable to accept a message from the CSI. The CSI discards the message after the appropriate number of retries with timeouts.

  ```
  Cannot send message to NI: discarded.
  MSG_SEND_NI_FAILURE
  ```

- This entry indicates that the CSI was unable to put a client's return address on its queue because the call to the `cl_qm_create()` common library function has failed.

  ```
  Can't add member to queue Q-id: queue_id.
  MSG_QUEUE_MEMBADD_FAILURE
  ```

  queue_id is the identifier of the CSI connection queue.

- This entry indicates that the CSI is unable to delete a message in an internal queue. This is a problem with the `cl_qm_mdelete()` common library function.

  ```
  Can't delete Q-id queue_id, Member: member_id
  MSG_DELETE_QMEMBER_FAILURE
  ```

  queue_id is the identifier of the CSI connection queue. *member_id* is the ID of the queue member it is trying to delete.

- This entry indicates that the CSI is unable to get status information because the call to the `cl_qm_mstatus()` common library function has failed.

  ```
  Can't get queue status Q-id: queue_id, Member: member_id
  MSG_QUEUE_STATUS_FAILURE
  ```

  queue_id is the identifier of the CSI connection queue. *member_id* is the ID of the queue member for which the CSI is seeking status information.

- This entry indicates that the CSI is unable to find a specific member in an internal queue. This is a problem with the `cl_qm_maccess()` common library function.

  ```
  Can't locate queue Q-id queue_id, Member: member_id.
  MSG_LOCATE_QMEMBER_FAILURE
  ```

  queue_id is the identifier of the CSI connection queue. *member_id* is the ID of the queue member it is trying to locate.

- This entry indicates that the call to the `svc_register()` function has failed. See the *Sun Network Programming Manual*, "Remote Procedure Call Programming Guide."

  ```
  Can't register RPC TCP service.
  MSG_RPCTCP_SVCREGISTER_FAILED
  ```

- This entry indicates that the call to the `svc_register()` function has failed. See the *Sun Network Programming Manual*, "Remote Procedure Call Programming Guide."

  ```
  Can't register RPC UDP service.
  MSG_RPCUDP_SVCREGISTER_FAILED
  ```

- This entry indicates that the RPC call to the `svctcp_create()` function has failed. See the *Sun Network Programming Manual*, "Remote Procedure Call Programming Guide."

  ```
  Create of RPC TCP service failed,
  Address: address, Port: port
  MSG_RPCTCP_SVCCREATE_FAILED
  ```

  address is the address of the host, expressed as an `unsigned long` integer. *port* is the port number of the client where a connection was attempted.

- This entry indicates that the call to the `svcudp_create()` function has failed. See the *Sun Network Programming Manual*, "Remote Procedure Call Programming Guide."

  ```
  Create of RPC UDP service failed,
  Address: address, Port: port
  MSG_RPCUDP_SVCCREATE_FAILED
  ```

  address is the address of the host, expressed as an `unsigned long` integer. *port* is the port number of the client where a connection was attempted.

- This entry indicates that the CSI was unable to create the network output queue which is used for messages between the CSI and the SSI.

  ```
  Creation of network output queue failed.
  MSG_CREATE_NI_OUTQ_FAILURE
  ```

- This entry indicates that either the CSI has encountered a process that is older than the connection time limit, or that the maximum number of timeouts and retries has been applied to a packet on the CSI network output queue. The connection limit is defined by the `CSI_CONNECT_AGETIME` environment variable in the `rc.acsss` file. The timeout and retry limits are defined by the `CSI_RETRY_TIMEOUT` and `CSI_RETRY_TRIES` environment variables, also defined in the `rc.acsss` file. The CSI drops the packet.

  ```
  Dropping from Queue: Address: address, Port: port_id,
  ssi_identifier: ssi_id, Protocol: protocol
  MSG_QUEUE_ENTRY_DROP
  ```

  address is the return address of the requesting SSI, expressed as an `unsigned long` integer. *port_id* is the identifier of the port on the server system that provides communications with the SSI. *ssi_id* is the identifier for the SSI. *protocol* is type of network transmission protocol.

- This entry indicates that the CSI has received a duplicate IPC packet. It automatically drops the duplicate packet.

  ```
  Duplicate packet from ACSLM detected: discarded.
  MSG_DUPLICATE_ACSLM_PACKET
  ```

- This entry indicates that the CSI has received a duplicate packet from the Network Interface. The duplicate is dropped.

  ```
  Duplicate packet from Network detected: discarded
  address: address, process-id: process_id,
  sequence_number: seq_nbr
  MSG_DUPLICATE_NI_PACKET
  ```

  address is the return address of the packet. *process_id* is the process ID of the packet. *seq_nbr* defines the ordering of packets sent via IPC mechanisms.

- This entry indicates that the call to the `cl_qm_init()` or `cl_qm_create()` common library function has failed while trying to create the internal SSI address connection queue.

  ```
  Initialization of connect queue failed.
  MSG_CREATE_CONNECTQ_FAILURE
  ```

- This entry indicates that CSI initiation has completed successfully. Communications with the ACSLM have been successfully established, and the CSI has been established as an RPC server.

  ```
  Initiation Completed.
  MSG_INITIATION_COMPLETED
  ```

- This entry indicates that CSI initiation has failed.

  ```
  Initiation of CSI Failed.
  MSG_INITIATION_FAILURE
  ```

- This entry indicates that CSI initiation has been started.

  ```
  Initiation Started.
  MSG_INITIATION_STARTED
  ```

- This entry indicates that the CSI has received a request packet from the SSI with an unrecognizable *command* specified in the `MESSAGE_HEADER` portion of the `CSI_REQUEST_HEADER`.

  ```
  Invalid command.
  MSG_INVALID_COMMAND
  ```

- This entry may indicate that neither environment variable for the two available communication services has been defined; these variables are `CSI_TCP_RPCSERVICE` and `CSI_UDP_RPCSERVICE` in the `rc.acsss` file. This entry may also indicate that a request received from the SSI has incorrect values specified in the protocol-dependent portions of the `CSI_HEADER`.

  ```
  Invalid communications service.
  MSG_INVALID_COMM_SERVICE
  ```

- This entry indicates that the CSI has received a message that is too small. The CSI is unable to use this message, therefore it discards it.

  ```
  Invalid message size, size, from NI: discarded.
  MSG_MESSAGE_SIZE
  ```

  size is the size of the message received from the NI.

- This entry indicates that the CSI has received a message that too large. The CSI truncates the message to a valid size and attempts to use it.

  ```
  Invalid message size, size, from NI: truncated.
  MSG_MESSAGE_SIZE_TRUNC
  ```

  size is the size of the message received from the NI.

- This entry indicates that an unsupported network protocol has been passed to the `csi_rpccall()` function.

  ```
  Invalid network protocol.
  MSG_INVALID_PROTO
  ```

- This entry indicates that the calculated network timeout is not a usable number. The timeout is calculated by combining the figures assigned to the `CSI_RETRY_TIMEOUT` and `CSI_RETRY_TRIES` environment variables in the `rc.acsss` file.

  ```
  Invalid network timeout value.
  MSG_INVALID_NI_TIMEOUT
  ```

- This entry indicates that a program is trying to use the CSI but it is not using one of the two valid procedure numbers.

  ```
  Invalid procedure number.
  MSG_RPC_INVALID_PROCEDURE
  ```

- This entry indicates that the CSI has received a packet from the NI with either an unrecognizable `TYPE` in the `IPC_HEADER` portion of the `CSI_REQUEST_HEADER` or an unrecognizable `IDENTIFIER` type in the message packet.

  ```
  Invalid type.
  MSG_INVALID_TYPE
  ```

- This entry indicates that the CSI has detected a message from the ACSLM for an invalid or unknown SSI client. The message is discarded.

  ```
  Message for unknown client discarded.
  MSG_UNDEF_CLIENT
  ```

- This entry indicates that the CSI has encountered an operating system error. This message is indicative of a problem with the operating system itself, not with the CSI or the Storage Server.

  Operating system error. *error_nbr*
  MSG_SYSTEM_ERROR

  error_nbr is the UNIX error number; see your UNIX documentation for a description.

- This entry indicates that the attempted TCP connection is not possible.

  RPC TCP client connection failed: *rpc_error_msg*,
  Address: *address*, Port: *port*
  MSG_RPCTCP_CLNTCREATE

  rpc_error_msg is a detailed error message generated by the RPC service itself. In most cases this message will be `Program number not registered`, which indicates that either the CSI or the SSI is not running. *address* is the address of the client host, expressed as an `unsigned long` integer. *port* is the port number of the client where a connection was attempted.

- This entry indicates that the attempted UDP connection is not possible.

  RPC UDP client connection failed: *rpc_error_msg*,
  Address: *address*, Port: *port*
  MSG_RPCUDP_CLNTCREATE

  rpc_error_msg is a detailed error message generated by the RPC service itself. In most cases this message will be `Program number not registered`, which indicates that the CSI or SSI is not running. *address* is the address of the client host, expressed as an `unsigned long` integer. *port* is the port number of the client where a connection was attempted.

- This entry indicates that The CSI has begun the process of purging old processes from its connection queue. The CSI routinely searches for processes older than `CSI_CONNECT_AGETIME` and purges them.

  Starting cleanup of connection queue, Q-id *queue_id*
  MSG_QUEUE_CLEANING_START

queue_id is the identifier of the CSI connection queue.

- This entry indicates that CSI termination has been completed successfully.

  ```
  Termination Completed.
  MSG_TERMINATION_COMPLETED
  ```

- This entry indicates that CSI termination has been started.

  ```
  Termination Started.
  MSG_TERMINATION_STARTED
  ```

- This entry indicates that the CSI has encountered a message from the ACSLM or the NI that cannot be delivered because of incorrect message format or a CSI failure. The message is discarded.

  ```
  Undefined messaged detected: discarded.
  MSG_UNDEF_MSG
  ```

- This entry indicates that the CSI has received a signal that it did not expect.

  Unexpected signal caught, value: *signal*.
  MSG_UNEXPECTED_SIGNAL

  signal is the signal value the CSI has received.

- This entry indicates that the CSI has been initiated. It notifies you that an RPC number previously assigned to the CSI still exists. The CSI unmaps this number and remaps to a new one as a normal part of the initiation.

  ```
  Unmapped previously registered RPC service.
  MSG_UNMAPPED_RPCSERVICE
  ```

- This entry indicates that the TYPE in the IPC_HEADER is not supported by the CSI. The CSI only recognizes IPC input from the ACSLM, therefore it discards any messages where TYPE is not set to TYPE_LM.

  Unsupported module type *module* detected: discarded.
  MSG_UNDEF_MODULE_TYPE

  module is the TYPE value in the IPC_HEADER.

- This entry indicates that a packet that the CSI XDR translation routines are unable to translate a message *completely* because it has been damaged. The CSI attempts to translate the message up to the point where the error was detected. If at least the CSI_REQUEST_HEADER portion of the message is translatable, the message is forwarded, otherwise it is dropped.

  ```
  XDR message translation failure.
  MSG_XDR_XLATE_FAILURE
  ```

DISMOUNT

The following entries are generated by the `dismount` request. The *component_name* in their message prefix is `DISMOUNT`. The entries are listed in alphabetical order.

- This entry indicates that the storage cell to which a cartridge was to be dismounted is full, although the data base indicates it was empty. The robot will retry the dismount until it finds an available cell.

  `Destination location full:` *cell_id*

  cell_id is the storage cell location indicated in the data base. An `audit` should be performed on this cell location in order to reconcile the data base with the physical contents of the cell.

- This entry indicates that the LSM robot was unable to find a tape cartridge in a tape drive, although the data base indicates that it is in the drive. The request fails.

  `Source location empty:` *drive_id*

  drive_id is the ID of the tape drive.

EJECT

The following entries are generated by the `eject` request. The *component_name* in their message prefix is `EJECT`. The entries are listed in alphabetical order.

- This entry indicates that the LSM robot unexpectedly found a cartridge in a CAP cell. This will occur if the operator did not completely empty the CAP during a previous `eject` operation. The robot will attempt to place the cartridge in the next CAP cell. If it is unable to find an available CAP cell, the `eject` process will issue an unsolicited message to empty the CAP.

  `CAP cell destination location occupied.`

- This entry indicates that a client application submitted an `eject` request while the LSM was in the diagnostic state. The request is rejected; only requests submitted from the Command Processor are processed while the LSM is in the diagnostic state.

  `LSM` *lsm_id* `STATE_DIAGNOSTIC.`

  lsm_id is the ID of the LSM.

- This entry indicates that the LSM is offline and is therefore unavailable for ejecting tape cartridges.

  `LSM` *lsm_id* `STATE_OFFLINE.`

lsm_id is the ID of the LSM.

- This entry indicates that the LSM robot was unable to find a tape cartridge in the location indicated by the data base. The request fails.

  `Source location empty:` *cell_id*

  cell_id is the storage cell location indicated in the data base. If you suspect that the cartridge is in the library an `audit` should be performed on the entire library in order to reconcile the data base with the physical contents of the cell.

ENTER

The following entries are generated by the `enter` request. The *component_name* in their message prefix is ENTER. The entries are listed in alphabetical order.

- This entry indicates that the LSM robot has found a tape cartridge in a location that the data base indicated was empty. The tape cartridge is not entered into the library.

  `Destination location full:` *cell_id*

  cell_id is the storage cell location indicated in the data base.

- This entry indicates that the LSM is offline and is therefore unavailable for entering tape cartridges.

  `LSM` *lsm_id* `= STATE_OFFLINE.`

  lsm_id is the ID of the LSM.

MOUNT

The following entries are generated by the `mount` request The *component_name* in their message prefix is MOUNT. The entries are listed in alphabetical order.

- This entry indicates that the LSM robot has found a cartridge in the tape drive, although the data base indicates that the drive is available. The request fails.

  `Destination location full:` *drive_id*`.`

  drive_id is the ID of the tape drive. An `audit` should be performed in order to reconcile the data base with the physical contents of the library.

- This entry indicates that the LSM robot was unable to find the tape cartridge in the location indicated by the data base. The request fails.

  `Source location empty: cell_id.`

  cell_id is the storage cell location indicated in the data base. If you suspect that the cartridge is in the library an `audit` should be performed on the entire library in order to reconcile the data base with the physical contents of the cell.

STORAGE SERVER INITIATION

The following messages are generated by the Storage Server Initiation process performed by the ACSLM. The *component_name* in their message prefix is ACSSS_DAEMON. The entries are listed in alphabetical order.

- This entry indicates that the daemon has received an unexpected exit status from a Storage Server process.

  `exit status (status), status_code, received from process_id.`

  status is the numeric exit status from the process. *status_code* is the Storage Server status code that was generated as a result of the exit. *process_id* is the Storage Server process.

- This entry indicates that Storage Server initiation has completed successfully.

  `Initiation completed.`

- This entry indicates that Storage Server initiation has begun.

  `Initiation started, acsss_version.`

  acsss_version is the version number of the Storage Server software.

- This entry indicates that a Storage Server process has been automatically restarted.

  process_id `restarted.`

  process_id is the Storage Server process.

- This entry indicates that a Storage Server process has been terminated.

  `signal(signal) terminated process_id.`

  signal is the UNIX signal that caused the termination. *process_id* is the Storage Server process that was terminated.

- This entry indicates that Storage Server termination has begun.

  `Termination invoked, status_code.`

status_code is the Storage Server status code which indicates the reason for the termination.

STORAGE SERVER RECOVERY

The following entries are generated by the Storage Server Recovery process performed by the ACSLM. The *component_name* in their message prefix is RECOVERY. The entries are listed in alphabetical order.

- This entry indicates that the ACS configuration in the Storage Server data base does not match the configuration defined in the LMU. Recovery processing terminates.

  ```
  ACS (acs_id) configuration failed to verify.
  ```

  acs_id is the unique ID of the ACS.

- This entry indicates that an ACS status in the data base is changed from the diagnostic state to offline. The ACS was in the diagnostic state at the time the Storage Server went into recovery, but it will be offline when recovery completes.

  ```
  ACS acs_id in STATE_DIAGNOSTIC, marked STATE_OFFLINE.
  ```

  acs_id is the ACS that was updated.

- This entry indicates that an ACS status in the data base is changed from the recovery state to online. The ACS was in the recovery state at the time the Storage Server went into recovery, but it will be online when recovery completes.

  ```
  ACS acs_id in STATE_RECOVERY, marked STATE_ONLINE.
  ```

  acs_id is the ACS that was updated.

- This entry indicates that the server system is not able to communicate with any ports for the specified ACS. Recovery will continue, but the ACS and its LSMs are marked offline in the data base.

  ```
  ACS acs_id, no ports online, marked offline.
  ```

  acs_id is the ACS that was updated.

- This entry indicates that the server system is not able to communicate with a port to an ACS. The port is marked offline in the data base/

  ```
  ACS acs_id, port <port_id> failed to go online.
  ```

  acs_id is the unique ID of the ACS. *port_id* is the port that failed to go online.

- This entry indicates that the recovery process was unable to successfully verify the drive configuration in the data base against the configuration defined in the LMU. This may be because the LSM is offline or because there is an actual configuration mismatch.

  ```
  DRIVE configuration failed to verify.
  ```

- This entry indicates that a drive has a status of empty, but was marked in use in the data base. The data base is updated to indicate that the drive is available.

  drive (*drive_id*) error, status unloaded, marked empty.

  drive_id is the drive that was updated.

- This entry indicates that a drive for an online LSM has a status of empty or not communicating, but was marked in use in the data base. The data base is updated to indicate that the drive is available, and any volume records referencing the drive are deleted from the data base.

  drive (*drive_id*) marked available.

  drive_id is the drive that was updated.

- This entry indicates that a drive for an offline LSM is updated in the data base to indicate that it is available. The LMU cannot report drive statuses for an offline LSM; therefore all drives for the LSM are marked available in the data base.

  drive (*drive_id*) marked available, LSM offline.

  drive_id is the drive that was updated.

- This entry indicates that a drive has a status of loaded, but was marked available in the data base. The data base is updated to indicate that the drive is in use.

  drive (*drive_id*) marked in use, loaded.

  drive_id is the drive that was updated.

- This entry indicates that a drive has a status of in use and unloaded, but was marked available in the data base. The external label of the cartridge in the drive is readable. The data base is updated to indicate that the drive is in use.

  drive (*drive_id*) readable, marked in use.

  drive_id is the drive that was updated.

- This entry indicates that a drive has a status of in use and unloaded, but was marked available in the data base. In addition, the LSM robot is unable to read the external label of the cartridge in the drive. The data base is updated to indicate that the drive is in use.

  ```
  drive (drive_id) unreadable, marked in use.
  ```

  drive_id is the drive that was updated.

- This entry indicates that a cartridge that was selected for use by a request process is not physically located in the storage cell indicated by the data base. The volume record is deleted from the data base.

  ```
  in-transit volume (vol_id) deleted.
  ```

  vol_id is the volume record that was deleted.

- This entry indicates that a cartridge that was selected for use by a request process is not physically located in the storage cell indicated by the data base. In addition, the LSM is offline. The volume record is deleted from the data base.

  ```
  in-transit volume (vol_id) deleted from location cell
  (cell_id), LSM offline.
  ```

  vol_id is the volume and *cell_id* is the cell location updated.

- This entry indicates that a cartridge that was selected for use by a request process is not physically located in the tape drive indicated by the data base. In addition, the LSM is offline. The volume record is deleted from the data base.

  ```
  in-transit volume (vol_id) deleted from location drive
  (drive_id), LSM offline.
  ```

  vol_id is the volume and *drive_id* is the tape drive updated.

- This entry indicates that a cartridge that was selected for use by a request process is found in the storage cell indicated by the data base. The volume record is updated to indicate that the cartridge is in the storage cell, not in-transit.

  ```
  in-transit volume (vol_id) marked home.
  ```

  vol_id is the volume record that was updated.

- This entry indicates that a cartridge with an unreadable label is found in a storage cell location that the data base indicates contains a cartridge that was reserved by a request process. The volume record is updated to indicate that the selected (in-transit) cartridge is in the storage cell.

  ```
  in-transit volume (vol_id) unreadable, marked home.
  ```

vol_id is the volume record that was updated.

- This entry indicates that the LSM configuration in the data base does not match the configuration defined in the LMU. Recovery processing terminates.

  ```
  LSM configuration failed to verify.
  ```

- This entry indicates that an LSM status in the data base is changed from the diagnostic state to offline. In addition, the ACS is online. The LSM was in the diagnostic state at the time the Storage Server went into recovery, but it will be offline when recovery completes.

  LSM *lsm_id* in STATE_DIAGNOSTIC, marked STATE_OFFLINE.

  lsm_id is the LSM that was updated.

- This entry indicates that an LSM status in the data base is changed from the recovery state to offline. In addition, the ACS is online. The LSM was in the recovery state at the time the Storage Server went into recovery, but it will be offline when recovery completes.

  LSM *lsm_id* in STATE_RECOVERY, marked STATE_ONLINE.

  lsm_id is the LSM that was updated.

- This entry indicates that the server system is not able to communicate with any ACS. Recovery continues, but all ACSs and their LSMs are marked offline.

  ```
  No server ports online.
  ```

- This entry indicates that a cell marked reserved in the data base is found to be empty. The cell record is updated to empty.

  reserved cell (*cell_id*) marked empty.

  cell_id is the cell record that was updated.

- This entry indicates that a cell marked reserved in the data base is updated to empty because the LSM is offline. The recovery process is unable to verify the cell contents of an offline LSM.

  reserved cell (*cell_id*) marked empty, LSM offline.

  cell_id is the cell record that was updated.

- This entry indicates that a cell marked reserved in the data base is found to contain a cartridge with a readable label. The cell record is updated to full.

  reserved cell (*cell_id*) readable, marked full.

  cell_id is the cell record that was updated.

- This entry indicates that a cell marked reserved in the data base is found to contain a cartridge with an unreadable label. The cell record is updated to full.

  `reserved cell (cell_id) unreadable, marked full.`

  cell_id is the cell record that was updated.

- This entry indicates that recovery processing terminates because the ACSLM is unable to receive a response from the ACS Library Handler (ACSLH).

  `timed out awaiting ACSLH response.`

- This entry indicates that an LSM has failed to vary online.

  `unexpected identifier status = status_code for LSM lsm_id`

  status_code is the final status of the LSM. *lsm_id* is the unique ID for the LSM.

- This entry indicates that a drive marked as containing a tape cartridge is found to be empty. The volume record is deleted from the data base.

  `volume (vol_id) not in drive (drive_id), deleted`

  vol_id is the volume record that was deleted. *drive_id* is the tape drive that the data base indicated contained the cartridge.

- This entry indicates that a cell or drive marked reserved is found to contain a tape cartridge that does not exist in the data base. A record is created for the new volume.

  `volume (vol_id) record created.`

  vol_id is the volume record that was created.

- This entry indicates that a cell or drive marked reserved is found to contain a different tape cartridge than the one indicated in the data base. The data base is updated with the correct volume ID.

  `volume (vol_id) record updated.`

  vol_id is the volume record that was updated.

VARY

The following entries are generated by the `vary` request. The *component_name* in their message prefix is `VARY`. The entries are listed in alphabetical order.

- This entry indicates that a client application submitted a `vary` request while the ACS was in the diagnostic state. The request is rejected; only requests submitted from the Command Processor are processed while the ACS is in the diagnostic state.

  `ACS` *acs_id* `incorrect requestor type: vary disallowed`

  acs_id is the ACS in the request.

- This entry indicates that a `vary` request was received while the ACS was in the recovery or offline-pending state. The request is rejected.

  `ACS` *acs_id* `is in transitional state: vary disallowed`

  acs_id is the ACS in the request.

- This entry indicates that a `vary offline` request was received while the ACS was already in the offline state. The request is rejected.

  `ACS` *acs_id* `is offline: vary disallowed.`

  acs_id is the ACS in the request.

- This entry indicates that a request was received to vary offline the last online port for an online ACS. The request is rejected. The ACS must be varied offline before the last online port can be varied offline.

  `Attempted to vary last port` *port_id* `for online ACS: vary disallowed.`

  port_id is the port in the request.

- This entry indicates that a device was forced offline while it was in the offline-pending state. The `vary` request that placed the device in the offline-pending state is overridden.

  `Current vary request overidden by a vary with the FORCE option.`

- This entry indicates that a client application submitted a `vary` request while the drive was in the diagnostic state. The request is rejected; only requests submitted from the Command Processor are processed while the drive is in the diagnostic state.

  `Drive` *drive_id* `incorrect requestor type: vary disallowed.`

drive_id is the drive in the request.

- This entry indicates that a `vary` request was received against a tape drive that the data base indicates is in use. The request is rejected.

  `Drive` *drive_id* `not available: vary disallowed.`

  drive_id is the drive in the request.

- This entry indicates that a request to vary an LSM offline was processed to completion, but the LSM failed to vary offline.

  `LSM:` *lsm_id* `failed to vary offline.`

  lsm_id is the LSM in the request.

- This entry indicates that a request to vary an LSM online was processed to completion, but the LSM failed to vary online.

  `LSM:` *lsm_id* `failed to vary online after an online request.`

  lsm_id is the LSM in the request.

- This entry indicates that a client application submitted a `vary` request while the LSM was in the diagnostic state. The request is rejected; only requests submitted from the Command Processor are processed while the LSM is in the diagnostic state.

  `LSM` *lsm_id* `incorrect requestor type: vary disallowed.`

  lsm_id is the LSM in the request.

- This entry indicates that a `vary` request was received while the LSM was in the recovery or offline-pending state. The request is rejected.

  `LSM` *lsm_id* `is in transition state: vary disallowed.`

  lsm_id is the LSM in the request.

- This entry indicates that in-transit cartridge recovery completed successfully while an LSM was varied online.

  `LSM` *lsm_id*`: recovery complete.`

  lsm_id is the LSM varied online.

- This entry indicates that the LSM robot was unable to dispose of in-transit cartridges while an LSM was varied online. The LSM is successfully brought online, but there may be cartridges left in the robot's hands.

  `LSM` *lsm_id*`: recovery incomplete.`

  lsm_id is the LSM varied online.

- This entry indicates that, during LSM recovery, the robot finds a cartridge in-transit, but the data base indicates that the cartridge is *not* in-transit. The volume record is deleted from the data base, and the robot places the cartridge in the Playground area of the LSM. You must enter the LSM and remove the cartridge.

  `Misplaced tape. Removed volume record for volid` *`vol_id`* `from database.`

  vol_id is the ID of the volume record deleted from the data base.

- This entry indicates that a `vary` request was received while all ports to the ACS were offline. The request is rejected. At least one port must be online in order for a `vary` request to be processed.

  `No port online for ACS` *`acs_id`*`: vary disallowed`

  acs_id is the ACS in the request.

- This entry indicates that a request to vary a port offline was processed to completion, but the port failed to vary offline.

  `port` *`port_id`* `failed to vary offline.`

  port_id is the port in the request.

- This entry indicates that a request to vary a port online was processed to completion, but the port failed to vary online.

  `port` *`port_id`* `failed to vary online.`

  port_id is the port in the request.

(INTENTIONALLY LEFT BLANK)

APPENDIX B:
ACSLM HEADER FILES

OVERVIEW

This Appendix includes the following header files which contain definitions used for ACSLM processing:

- `db_defs.h` – Contains definitions used for data base transactions.
- `defs.h` – Contains general, system-wide definitions. Includes `db_defs.h`.
- `identifier.h` – Contains structure definitions used for data base transactions.
- `structs.h` – Contains general, system-wide data structure definitions. Includes `db_defs.h` and `identifier.h`.

db_defs.h HEADER FILE

```
/*  SccsId      @(#)db_defs.h       1.4 5/18/89 (c) 1988 StorageTek */
#define _DB_DEFS_       1
/*
 *                              StorageTek SECRET
 *                  Property of Storage Technology Corporation
 *                  Do not copy or distribute unpublished work
 *
 *                              Copyright (1988)
 *                         Storage Technology Corporation
 *                               All Rights Reserved
 *
 * Functional Description:
 *
 *      system-wide definitions used in data base transactions.
 *    due to limitations of the INGRES embedded sql front-end,
 *    it is necessary to separate definitions required by the
 *    sql preprocessor because it does not handle all C preprocessor
 *    statements.  this file also gets included by defs.h if
 *    _DB_DEFS_ is not yet defined, therefore any routine needing
 *    both MUST include db_defs.h in a "sql declare" section before
 *    the C #include of defs.h.
 *    NOTE: "unsigned" is not supported by INGRES esqlc
 *
 * Modified by:
 *
 *    D. F. Reed        06-Oct-1988     Original.
 *    D. A. Beidle      21-Apr-1989     Added DB_MULTI_ROW data base error
 *    code definition.
 */

/*
 *      Header Files:
 */

/*
 *    Defines, Typedefs and Structure Definitions:
 */

#define DATA_BASE       "lib1"      /* data base name */

typedef char            ACS;        /* acs number/identifier */
#define MIN_ACS         0
#define MAX_ACS         127         /* really 255, but not in a char */

typedef char            LSM;        /* lsm number */
#define MIN_LSM         0
#define MAX_LSM         15

typedef char            PORT;       /* server-acs communications line */
#define MIN_PORT        0
```

```
#define MAX_PORT        15
#define PORT_NAME_SIZE  32

typedef char            PANEL;      /* lsm panel number */
#define MIN_PANEL       0
#define MAX_PANEL       19

typedef char            DRIVE;      /* transport number */
#define MIN_DRIVE       0
#define MAX_DRIVE       3

typedef char            ROW;        /* row number within a lsm panel */
#define MIN_ROW         0
#define MAX_ROW         14
#define MIN_CAP_ROW     0
#define MAX_CAP_ROW     2

typedef char            COL;        /* column number within a lsm panel */
#define MIN_COL         0
#define MAX_COL         23
#define MIN_CAP_COL     0
#define MAX_CAP_COL     6

#define CAP_SIZE        21          /* cap cartridge capacity (row*col) */

typedef long            FREECELLS;  /* count of unused cells within a */
                                    /* server or acs or lsm */

#define EXTERNAL_LABEL_SIZE 6       /* label characters */

#define SOCKET_NAME_SIZE 14         /* max characters in socket name */

/* cell location codes */
typedef enum {
    LOCATION_FIRST = 0,             /* illegal */
    LOCATION_CELL,
    LOCATION_DRIVE,
    LOCATION_LAST                   /* illegal */
} LOCATION;

/* state codes */
typedef enum {
    STATE_FIRST = 0,                /* illegal */
    STATE_CANCELLED,                /* process state only */
    STATE_DIAGNOSTIC,
    STATE_IDLE,
    STATE_IDLE_PENDING,
                                                /*  5 */
    STATE_OFFLINE,
    STATE_OFFLINE_PENDING,
    STATE_ONLINE,
    STATE_RECOVERY,
    STATE_RUN,
```

```
    STATE_LAST                                          /* illegal */
} STATE;

/* status codes */
typedef enum {
    STATUS_SUCCESS = 0,
    STATUS_ACS_FULL,
    STATUS_ACS_NOT_IN_LIBRARY,
    STATUS_ACS_OFFLINE,
    STATUS_ACSLM_IDLE,
                                                        /*   5 */
    STATUS_ACTIVITY_END,
    STATUS_ACTIVITY_START,
    STATUS_AUDIT_ACTIVITY,
    STATUS_AUDIT_IN_PROGRESS,
    STATUS_CANCELLED,
                                                        /*  10 */
    STATUS_CAP_AVAILABLE,
    STATUS_CAP_FULL,
    STATUS_CAP_IN_USE,
    STATUS_CELL_EMPTY,
    STATUS_CELL_FULL,
                                                        /*  15 */
    STATUS_CELL_INACCESSIBLE,
    STATUS_CELL_RESERVED,
    STATUS_CLEAN_DRIVE,
    STATUS_COMMUNICATION_FAILED,
    STATUS_CONFIGURATION_ERROR,
                                                        /*  20 */
    STATUS_COUNT_TOO_SMALL,
    STATUS_COUNT_TOO_LARGE,
    STATUS_CURRENT,
    STATUS_DATABASE_ERROR,
    STATUS_DEGRADED_MODE,
                                                        /*  25 */
    STATUS_DONE,
    STATUS_DOOR_CLOSED,
    STATUS_DOOR_OPENED,
    STATUS_DRIVE_AVAILABLE,
    STATUS_DRIVE_IN_USE,
                                                        /*  30 */
    STATUS_DRIVE_NOT_IN_LIBRARY,
    STATUS_DRIVE_OFFLINE,
    STATUS_DRIVE_RESERVED,
    STATUS_DUPLICATE_LABEL,
    STATUS_EJECT_ACTIVITY,
                                                        /*  35 */
    STATUS_ENTER_ACTIVITY,
    STATUS_EVENT_LOG_FULL,
    STATUS_IDLE_PENDING,
    STATUS_INPUT_CARTRIDGES,
    STATUS_INVALID_ACS,
                                                        /*  40 */
    STATUS_INVALID_COLUMN,
```

```
    STATUS_INVALID_COMMAND,
    STATUS_INVALID_DRIVE,
    STATUS_INVALID_LSM,
    STATUS_INVALID_MESSAGE,
                                        /*  45 */
    STATUS_INVALID_OPTION,
    STATUS_INVALID_PANEL,
    STATUS_INVALID_PORT,
    STATUS_INVALID_ROW,
    STATUS_INVALID_STATE,
                                        /*  50 */
    STATUS_INVALID_SUBPANEL,
    STATUS_INVALID_TYPE,
    STATUS_INVALID_VALUE,
    STATUS_INVALID_VOLUME,
    STATUS_IPC_FAILURE,
                                        /*  55 */
    STATUS_LIBRARY_BUSY,
    STATUS_LIBRARY_FAILURE,
    STATUS_LIBRARY_NOT_AVAILABLE,
    STATUS_LOCATION_OCCUPIED,
    STATUS_LSM_FULL,
                                        /*  60 */
    STATUS_LSM_NOT_IN_LIBRARY,
    STATUS_LSM_OFFLINE,
    STATUS_MESSAGE_NOT_FOUND,
    STATUS_MESSAGE_TOO_LARGE,
    STATUS_MESSAGE_TOO_SMALL,
                                        /*  65 */
    STATUS_MISPLACED_TAPE,
    STATUS_MULTI_ACS_AUDIT,
    STATUS_NORMAL,
    STATUS_NONE,
    STATUS_NOT_IN_SAME_ACS,
                                        /*  70 */
    STATUS_ONLINE,
    STATUS_OFFLINE,
    STATUS_PENDING,
    STATUS_PORT_NOT_IN_LIBRARY,
    STATUS_PROCESS_FAILURE,
                                        /*  75 */
    STATUS_RECOVERY_COMPLETE,
    STATUS_RECOVERY_FAILED,
    STATUS_RECOVERY_INCOMPLETE,
    STATUS_RECOVERY_STARTED,
    STATUS_REMOVE_CARTRIDGES,
                                        /*  80 */
    STATUS_RETRY,
    STATUS_STATE_UNCHANGED,
    STATUS_TERMINATED,
    STATUS_VALID,
    STATUS_VALUE_UNCHANGED,
                                        /*  85 */
    STATUS_VARY_DISALLOWED,
```

```
    STATUS_VOLUME_ADDED,
    STATUS_VOLUME_EJECTED,
    STATUS_VOLUME_ENTERED,
    STATUS_VOLUME_FOUND,
                                                    /*  90 */
    STATUS_VOLUME_HOME,
    STATUS_VOLUME_IN_DRIVE,
    STATUS_VOLUME_IN_TRANSIT,
    STATUS_VOLUME_NOT_IN_DRIVE,
    STATUS_VOLUME_NOT_IN_LIBRARY,
                                                    /*  95 */
    STATUS_UNREADABLE_LABEL,
    STATUS_UNSUPPORTED_OPTION,
    STATUS_UNSUPPORTED_STATE,
    STATUS_UNSUPPORTED_TYPE,
    STATUS_VOLUME_IN_USE,
                                                    /* 100 */
    STATUS_PORT_FAILURE,
    STATUS_MAX_PORTS,
    STATUS_PORT_ALREADY_OPEN,
    STATUS_QUEUE_FAILURE,
    STATUS_RPC_FAILURE,
                                                    /* 105 */
    STATUS_NI_TIMEDOUT,
    STATUS_INVALID_COMM_SERVICE,
    STATUS_COMPLETE,
    STATUS_AUDIT_FAILED,
    STATUS_NO_PORTS_ONLINE,
                                                    /* 110 */
    STATUS_CARTRIDGES_IN_CAP,
    STATUS_TRANSLATION_FAILURE,
    STATUS_DATABASE_DEADLOCK,
    STATUS_DIAGNOSTIC,
    STATUS_DUPLICATE_IDENTIFIER,
    STATUS_EVENT_LOG_FAILURE,

    STATUS_LAST                                     /* illegal */
} STATUS;

/*  Database Dependent Constants */
#define DB_SUCCESS      0           /* successful completion */
#define DB_TIMEOUT      -17702      /* timeout occurred */
#define DB_LOCK_TIMEOUT 10          /* system lock timeout period (sec.) */
#define DB_NODATA       100         /* no data was found */
#define DB_DEADLOCK     -17700      /* deadlock was detected */
#define DB_INCONSISTENT -38         /* internal database inconsistency */
#define DB_MULTI_ROW    -1320       /* non-cursor select attempted to */
                                    /* return more than one row  */
```

defs.h HEADER FILE

```
/*  SccsId      %W% %G% (c) 1988 StorageTek */
#ifndef _DEFS_
#define _DEFS_
/*
 *                              StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                              Copyright (1988)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Functional Description:
 *
 *      system-wide definitions
 *    includes db_defs.h, unless already defined, to be complete.
 *
 * Modified by:
 *
 *    D. F. Reed            19-Sep-1988     Original.
 */

/*
 *     Header Files:
 */
#ifndef               _DB_DEFS_
#include "db_defs.h"
#endif                _DB_DEFS_

/*
 *    Defines, Typedefs and Structure Definitions:
 */

#define ACSSS_VERSION   "Version 1.0"

#ifndef TRUE
#define FALSE           0
#define TRUE            (!FALSE)
#endif  TRUE

#ifndef NULL
#define NULL            ((char*)0)
#endif  NULL

/* execution trace support definitions */
#define TRACE_ACSSS_DAEMON          0x00000100L
#define TRACE_CSI                   0x00000200L
#define TRACE_ACSLM                 0x00000400L
#define TRACE_MOUNT                 0x00000800L
#define TRACE_DISMOUNT              0x00001000L
```

```
#define TRACE_ENTER                 0x00002000L
#define TRACE_EJECT                 0x00004000L
#define TRACE_AUDIT                 0x00008000L
#define TRACE_QUERY                 0x00010000L
#define TRACE_VARY                  0x00020000L
#define TRACE_RECOVERY              0x00040000L
#define TRACE_ACSSA                 0x00080000L
#define TRACE_CP                    0x00100000L
#define TRACE_LIBRARY_HANDLER       0x00200000L
#define TRACE_EVENT_LOGGER          0x00400000L
#define TRACE_CSI_PACKETS           0x00800000L

#define TRACE(lev) ((trace_value & trace_module) && ((trace_value & 0xff) >=
lev))

/* well-known socket name definitions */
/* uses IP port numbers > 50000 (IPPORT_USERRESERVED) */
#define ACSEL           "50001"
#define ACSLH           "50002"
#define ACSLM           "50003"
#define ACSSA           "50004"
#define ACSSS           "50005"
#define ANY_PORT        "0"

typedef unsigned intBOOLEAN;        /* {TRUE, FALSE} */

typedef unsigned short  MESSAGE_ID; /* request id assigned by acslm */

#define MAX_ID          21          /* max identifier count */
#define MIN_MESSAGE     1           /* min message_id value */
#define MAX_MESSAGE     65535       /* max message_id value */
#define MAX_MESSAGE_SIZE 4096       /* max IPC message size */
#define MAX_RETRY       10          /* max retry lh_request attempts */
#define RETRY_TIMEOUT   2           /* time-out seconds between retries */
#define MAX_ACS_DRIVES  128         /* max drives per acs */
#define MAX_LSM_PTP     4           /* max pass-thru ports per lsm */
#define MAX_PORTS       16          /* max acs communication ports */
#define FD_SETSIZE      32          /* max # file desc. for select */

/* IPC_HEADER option values (bit field) */
#define RETRY           0x01

/* command codes */
typedef enum {
    COMMAND_FIRST = 0,              /* illegal */
    COMMAND_AUDIT,
    COMMAND_CANCEL,
    COMMAND_DISMOUNT,
    COMMAND_EJECT,
                                                    /*  5 */
    COMMAND_ENTER,
    COMMAND_IDLE,
    COMMAND_MOUNT,
    COMMAND_QUERY,
```

```
    COMMAND_RECOVERY,               /* ACSLM internal use only */
                                                    /* 10 */
    COMMAND_START,
    COMMAND_VARY,
    COMMAND_UNSOLICITED_EVENT,
    COMMAND_TERMINATE,              /* ACSLM internal use only */
    COMMAND_ABORT,                  /* ACSLM internal use only */

    COMMAND_LAST                    /* illegal */
} COMMAND;

/* message_option qualifier codes (bit field) */
#define FORCE           0x01
#define INTERMEDIATE    0x02
#define ACKNOWLEDGE     0x04

/* log_option codes */
typedef enum {
    LOG_OPTION_FIRST = 0,           /* illegal */
    LOG_OPTION_EVENT,
    LOG_OPTION_TRACE,
    LOG_OPTION_LAST                 /* illegal */
} LOG_OPTION;

/* type codes */
typedef enum {
    TYPE_FIRST = 0,                 /* illegal */
    TYPE_ACS,                       /* automated cartridge system */
    TYPE_AUDIT,                     /* audit request process */
    TYPE_CAP,                       /* cartridge access port */
    TYPE_CELL,                      /* cell identifier */
                                                    /*  5 */
    TYPE_CP,                        /* ACSSA command process */
    TYPE_CSI,                       /* client system interface */
    TYPE_DISMOUNT,                  /* dismount request process */
    TYPE_EJECT,                     /* eject request process */
    TYPE_EL,                        /* event logger */
                                                    /* 10 */
    TYPE_ENTER,                     /* enter request process */
    TYPE_DRIVE,                     /* library drive */
    TYPE_IPC,                       /* inter-process communication */
    TYPE_LH,                        /* library handler */
    TYPE_LM,                        /* library manager (ACSLM) */
                                                    /* 15 */
    TYPE_LSM,                       /* library storage module */
    TYPE_MOUNT,                     /* mount request process */
    TYPE_NONE,                      /* no identifier specified */
    TYPE_PANEL,                     /* LSM panel */
    TYPE_PORT,                      /* ACS communications line */
                                                    /* 20 */
    TYPE_QUERY,                     /* query request process */
    TYPE_RECOVERY,                  /* recovery request process */
    TYPE_REQUEST,                   /* storage server request */
    TYPE_SA,                        /* system administrator (ACSSA) */
```

```
    TYPE_SERVER,                    /* storage server */
                                                /* 25 */
    TYPE_SUBPANEL,                  /* LSM subpanel */
    TYPE_VARY,                      /* vary request process */
    TYPE_VOLUME,                    /* tape cartridge */

    TYPE_LAST                       /* illegal */
} TYPE;

/* data base field update codes */
typedef enum {
    FIELD_FIRST = 0,                /* illegal */
    FIELD_ACTIVITY,
    FIELD_CAP,
    FIELD_STATE,
    FIELD_STATUS,
    FIELD_LAST                      /* illegal */
} FIELD;

/* query type codes */
typedef enum {
    QUERY_TYPE_FIRST = 0,           /* illegal */
    QUERY_TYPE_ALL,                 /* init sequential read of table */
                                    /* and return first record */
    QUERY_TYPE_NEXT,                /* get next sequential record */
    QUERY_TYPE_ONE,                 /* get record keying off specified id */
    QUERY_TYPE_LAST                 /* illegal */
} QUERY_TYPE;

/* data base write_mode codes */
typedef enum {
    WRITE_MODE_FIRST = 0,           /* illegal */
    WRITE_MODE_CREATE,              /* create a new record */
    WRITE_MODE_UPDATE,              /* update an existing record */
    WRITE_MODE_LAST                 /* illegal */
} WRITE_MODE;

/* volid volume_type codes */
typedef enum {
    VOLUME_TYPE_FIRST = 0,          /* illegal */
    VOLUME_TYPE_DIAGNOSTIC,         /* volid may contain blanks */
    VOLUME_TYPE_STANDARD,           /* volid must be 6 chars, [A-Z][0-9] */
    VOLUME_TYPE_LAST                /* illegal */
} VOLUME_TYPE;

/* cell select option codes */
typedef enum {
    SELECT_OPTION_FIRST = 0,        /* illegal */
    SELECT_OPTION_ACS,              /* if lsm full, try any lsm in acs */
    SELECT_OPTION_LSM,              /* try specified lsm only */
    SELECT_OPTION_LAST              /* illegal */
} SELECT_OPTION;

/*
```

```
 *      Global Variable Declarations:
 */
extern int              errno;          /* system error number */
extern int              sd_in;          /* module input socket descriptor */
extern int              n_fds;          /* number of input descriptors */
extern int              fd_list[FD_SETSIZE];
                                        /* input descriptor list */
extern char             my_sock_name[SOCKET_NAME_SIZE];
                                        /* module input socket name */
extern TYPE             my_module_type;/* executing module's type */
extern TYPE             requestor_type;/* request originator's module type */
extern int              restart_count; /* process failed/restart count */
extern MESSAGE_ID       request_id;     /* associated request ID, or 0 if */
                                        /* not associated with a request */
extern STATE            process_state; /* executing process' state flag */
extern unsigned long    trace_module;  /* module trace define value */
extern unsigned long    trace_value;   /* trace flag value */
extern int              acs_count;     /* number of ACSs configured */
extern int              port_count;    /* number of ports configured */

/*
 *      Procedure Type Declarations:
 */
STATUS          cl_acs_destroy();
STATUS          cl_acs_read();
STATUS          cl_acs_update();
STATUS          cl_acs_valid();
STATUS          cl_acs_write();
STATUS          cl_cap_eject();
STATUS          cl_cap_release();
STATUS          cl_cap_reserve();
STATUS          cl_cap_valid();
STATUS          cl_cel_destroy();
STATUS          cl_cel_next();
STATUS          cl_cel_read();
STATUS          cl_cel_select();
STATUS          cl_cel_update();
STATUS          cl_cel_valid();
STATUS          cl_cel_write();
int             cl_chk_input();
char *          cl_command();
STATUS          cl_db_connect();
STATUS          cl_db_disconnect();
STATUS          cl_drv_destroy();
STATUS          cl_drv_list();
STATUS          cl_drv_read();
STATUS          cl_drv_update();
STATUS          cl_drv_valid();
STATUS          cl_drv_write();
char *          cl_identifier();
STATUS          cl_inform();
STATUS          cl_ipc_create();
STATUS          cl_ipc_destroy();
int             cl_ipc_open();
```

```
STATUS          cl_ipc_read();
STATUS          cl_ipc_send();
STATUS          cl_ipc_write();
STATUS          cl_ipc_xmit();
char *          cl_lh_error();
char *          cl_lh_identifier();
char *          cl_lh_type();
void            cl_log_db_error();
void            cl_log_event();
void            cl_log_trace();
void            cl_log_lh_error();
void            cl_log_unexpected();
STATUS          cl_lsm_destroy();
STATUS          cl_lsm_list();
STATUS          cl_lsm_read();
STATUS          cl_lsm_update();
STATUS          cl_lsm_valid();
STATUS          cl_lsm_write();
STATUS          cl_pnl_valid();
STATUS          cl_proc_init();
STATUS          cl_prt_destroy();
STATUS          cl_prt_read();
STATUS          cl_prt_update();
STATUS          cl_prt_valid();
STATUS          cl_prt_write();
STATUS          cl_req_valid();
STATUS          cl_rp_init();
int             cl_select_input();
TYPE            cl_set_type();
void            cl_sig_hdlr();
char *          cl_status();
STATUS          cl_sub_valid();
void            cl_trace();
char *          cl_type();
STATUS          cl_vol_destroy();
STATUS          cl_vol_read();
STATUS          cl_vol_update();
STATUS          cl_vol_valid();
STATUS          cl_vol_write();
#endif _DEFS_
```

identifier.h HEADER FILE

```
/*  SccsId      @(#)identifier.h    1.2 4/6/89 (c) 1988 StorageTek */
#define _IDENTIFIER_    1
/*
 *                              StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                              Copyright (1988)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Functional Description:
 *
 *      system-wide data structure definitions used in data base transactions.
 *    due to limitations of the INGRES embedded sql front-end,
 *    it is necessary to separate definitions required by the
 *    sql preprocessor because it does not handle all C preprocessor
 *    statements.  this file also gets included by other header files if
 *    _IDENTIFIERS_ is not yet defined.
 *    NOTE: "unsigned" is not supported by INGRES esqlc
 *      definitions of system data structures
 *
 * Modified by:
 *
 *      D. F. Reed          27-Jan-1989     Original.
 */

/*
 *      Header Files:
 */

/*
 *    Defines, Typedefs and Structure Definitions:
 */

typedef struct {
    ACS             acs;
    LSM             lsm;
} LSMID;

typedef LSMIDCAPID;

typedef struct {
    ACS             acs;
    PORT            port;
} PORTID;

typedef struct {
    LSMID    lsm_id;
```

```
    PANEL           panel;
} PANELID;

typedef struct {
    PANELID         panel_id;
    ROW             begin_row;
    COL             begin_col;
    ROW             end_row;
    COL             end_col;
} SUBPANELID;

typedef struct {
    PANELID         panel_id;
    DRIVE           drive;
} DRIVEID;

typedef struct {
    PANELID         panel_id;
    ROW             row;
    COL             col;
} CELLID;

typedef struct {
    char            external_label[EXTERNAL_LABEL_SIZE + 1];
} VOLID;

typedef union {
    ACS             acs_id;
    CAPID           cap_id;
    CELLID          cell_id;
    DRIVEID         drive_id;
    LSMID           lsm_id;
    PANELID         panel_id;
    PORTID          port_id;
    SUBPANELID      subpanel_id;
    VOLID           vol_id;
    char            socket_name[SOCKET_NAME_SIZE];
    long            request;/* really a MESSAGE_ID */
} IDENTIFIER;
```

structs.h HEADER FILE

```
/*  SccsId      @(#)structs.h1.2 4/3/89 (c) 1988 StorageTek */
#ifndef _STRUCTS_
#define _STRUCTS_
/*
 *                              StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                              Copyright (1988)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Functional Description:
 *
 *      definitions of system data structures.  most of these structures
 *    are common to csi and acslm, and are included by both header files.
 *    includes "identifier.h", unless already defined, to be complete.
 *    includes "db_structs.h", unless already defined, to be complete.
 *
 * Modified by:
 *
 *      D. F. Reed          22-Sep-1988     Original.
 */

/*
 *      Header Files:
 */

#include "defs.h"
#ifndef               _IDENTIFIER_
#include "identifier.h"
#endif                _IDENTIFIER_
#ifndef               _DB_STRUCTS_
#include "db_structs.h"
#endif                _DB_STRUCTS_

/*
 *    Defines, Typedefs and Structure Definitions:
 */

typedef struct {
    unsigned short  packet_id;      /* client-specified */
    COMMAND         command;
    unsigned char   message_options;
} MESSAGE_HEADER;

typedef struct {
    unsigned long  byte_count;      /* message length, including header */
    TYPE           module_type;     /* sending module type */
    unsigned char  options;         /* see defs.h */
```

```
    unsigned long  seq_num;          /* message sequence number */
    char           return_socket[SOCKET_NAME_SIZE];
                                     /* sender's input socket name */
    unsigned int   return_pid;       /* sender's PID */
    unsigned long  ipc_identifier;   /* used for message sync */
} IPC_HEADER;

typedef struct {
    IPC_HEADER     ipc_header;
    LOG_OPTION     log_options;
    char           event_message[MAX_MESSAGE_SIZE];
} EVENT_LOG_MESSAGE;

typedef struct {
    STATUS         status;
    TYPE           type;
    IDENTIFIER     identifier;
} RESPONSE_STATUS;

typedef struct {
    IPC_HEADER     ipc_header;
    MESSAGE_HEADER message_header;
    RESPONSE_STATUS message_status;
    unsigned short error;
} UNSOLICITED_MESSAGE;

typedef struct {                     /* volume status sub-structure */
    VOLID          vol_id;
    RESPONSE_STATUS status;
} VOLUME_STATUS;

/*****************************************************************************
 *                      AUDIT REQUEST/RESPONSE COMMON STRUCTURES             *
*****************************************************************************/

typedef struct {                     /* audit ACS identifier status */
    ACS            acs_id;
    RESPONSE_STATUS status;
} AU_ACS_STATUS;

typedef struct {                     /* audit LSM identifier status */
    LSMID          lsm_id;
    RESPONSE_STATUS status;
} AU_LSM_STATUS;

typedef struct {                     /* audit panel identifier status */
    PANELID        panel_id;
    RESPONSE_STATUS status;
} AU_PNL_STATUS;

typedef struct {                     /* audit subpanel identifier status */
    SUBPANELID     subpanel_id;
    RESPONSE_STATUS status;
```

```
} AU_SUB_STATUS;

/*****************************************************************************
 *                        QUERY REQUEST/RESPONSE COMMON STRUCTURES           *

 *****************************************************************************/

typedef enum {                          /* request summary commands */
    AUDIT = 0,
    MOUNT,
    DISMOUNT,
    ENTER,
    EJECT,
    MAX_COMMANDS
} QU_COMMANDS;

typedef enum {                         /* request summary dispostions */
    CURRENT = 0,
    PENDING,
    MAX_DISPOSITIONS
} QU_DISPOSITIONS;

typedef struct {                       /* request summary matrix */
    MESSAGE_ID  requests[MAX_COMMANDS][MAX_DISPOSITIONS];
} REQ_SUMMARY;

typedef struct {                       /* ACS status (one/acs_id) */
    ACS                 acs_id;        /*   ACS for status */
    STATE               state;         /*   ACS state     */
    FREECELLS           freecells;     /*   number of free cells in ACS */
    REQ_SUMMARY         requests;      /*   request summary for ACS */
    STATUS              status;        /*   ACS status    */
} QU_ACS_STATUS;

typedef struct {                       /* CAP status (one/cap_id) */
    CAPID               cap_id;        /*   CAP for status   */
    STATUS              status;        /*   CAP status      */
} QU_CAP_STATUS;

typedef struct {                       /* drive status (one/drive_id)   */
    DRIVEID             drive_id;      /*   drive for status   */
    STATE               state;         /*   drive state     */
    VOLID               vol_id;        /*   volume if STATUS_DRIVE_IN_USE */
    STATUS              status;        /*   drive status    */
} QU_DRV_STATUS;

typedef struct {                       /* LSM status (one/lsm_id) */
    LSMID               lsm_id;        /*   LSM for status */
    STATE               state;         /*   LSM state     */
    FREECELLS           freecells;     /*   number of free cells in LSM */
    REQ_SUMMARY         requests;      /*   request summary for LSM */
    STATUS              status;        /*   LSM status    */
} QU_LSM_STATUS;
```

```
typedef struct {                    /* drive proximity status (one/volid) */
    VOLID               vol_id;     /*   volume for drive proximity list  */
    STATUS              status;     /*   volume status      */
    unsigned short      drive_count;/*   number of drive identifiers      */
    DRIVEID             drive_id[MAX_ACS_DRIVES];
                                    /*   drive list in proximity order    */
} QU_MNT_STATUS;

typedef struct {                    /* port status (one/port_id) */
    PORTID              port_id;    /*   port for status     */
    STATE               state;      /*   port state        */
    STATUS              status;     /*   port status       */
} QU_PRT_STATUS;

typedef struct {                    /* request status (one/request_id) */
    MESSAGE_ID          request;    /*   request for status   */
    COMMAND             command;    /*   command from request_packet   */
    STATUS              status;     /*   request status   */
} QU_REQ_STATUS;

typedef struct {                    /* server status (one)      */
    STATE               state;      /*   ACSLM state     */
    FREECELLS           freecells;  /*   number of free cells in library */
    REQ_SUMMARY         requests;   /*   request summary for library     */
} QU_SRV_STATUS;

typedef struct {                    /* volume status (one/vol_id)  */
    VOLID               vol_id;     /*   volume for status  */
    LOCATION            location_type; /*   LOCATION_CELL or LOCATION_DRIVE */
    union {                         /*   current location of volume  */
        CELLID          cell_id;    /*     if STATUS_VOLUME_HOME  */
        DRIVEID         drive_id;   /*     if STATUS_VOLUME_IN_DRIVE  */
    } location;                     /*     undefined if none of above */
    STATUS              status;     /*   volume status  */
} QU_VOL_STATUS;

/*****************************************************************************
 *                      VARY REQUEST/RESPONSE COMMON STRUCTURES              *

*****************************************************************************/

typedef struct {                    /* ACS status (one/acs_id) */
    ACS                 acs_id;
    RESPONSE_STATUS     status;
} VA_ACS_STATUS;

typedef struct {                    /* drive status (one/drive_id)   */
    DRIVEID             drive_id;
    RESPONSE_STATUS     status;
} VA_DRV_STATUS;

typedef struct {                    /* LSM status (one/lsm_id) */
    LSMID               lsm_id;
    RESPONSE_STATUS     status;
```

```
} VA_LSM_STATUS;

typedef struct {                        /* port status (one/port_id) */
    PORTID              port_id;
    RESPONSE_STATUS     status;
} VA_PRT_STATUS;

/*
 *      Procedure Type Declarations:
 */

#endif _STRUCTS_
```

(INTENTIONALLY LEFT BLANK)

APPENDIX C:
XDR TRANSLATION FUNCTIONS

OVERVIEW

This Appendix includes listings of the high-level XDR translation routines supplied by StorageTek:

- `csi_xdrrequest()` – Used to serialize and deserialize Storage Server request packets.
- `csi_xdrresponse()` – Used to serialize and deserialize Storage Server response packets.

csi_xdrrequest() FUNCTION

```
/*
 *                              StorageTek SECRET
 *                  Property of Storage Technology Corporation
 *                  Do not copy or distribute unpublished work
 *
 *                              Copyright (1989)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Name:
 *
 *    csi_xdrrequest()
 *
 * Description:
 *
 *    CSI high level xdr based routine for serializing and deserializing
 *    storage server request packets.  The routines in this source and those
 *    called from this source support a bi-directional protocol for either
 *    encoding or decoding storage server request packets based on the value
 *    of the XDR handle's "xdrsp->x_op" directional variable.
 *
 *    The data buffer passed, "bufferp->data", is encoded during serialization
 *    for up to "bufferp->size" bytes beginning with the byte number specified
 *    as  "bufferp->offset".   During deserialization,  the  XDR stream
 *    is  translated out of the xdr handle "xdrsp" into "bufferp->data"
 *    beginning at the offset specified by "bufferp->offset".
 *    The number of bytes of the packet that were successfully translated
 *    is returned in "bufferp->translated_bytes".
 *
 *    This routine will only return an error to the rpc layer (return 0)
 *    if the contents of the request header could not be translated.
 *
 *    If the request header was translated but there was a translation error
 *    lower in the packet, "bufferp->packet_status" equals
 *    CSI_PAKSTAT_XLATE_ERROR otherwise it equals CSI_PAKSTAT_XLATE_COMPLETED.
 *
 *    If a duplicate packet was detected during deserialization, this routine
 *    returns "bufferp->packet_status" equal to CSI_PAKSTAT_DUPLICATE_PACKET.
 *    In this case, upon return, only the CSI_REQUEST_HEADER portion of the
 *    packet will have been translated.

 *    SERIALIZATION:
 *    For serialization of a request packet, xdrsp->x_op equals XDR_ENCODE.
 *    Upon entry to this routine, the data buffer description structure of
 *    type CSI_MSGBUF must be initialized as follows:
 *
 *    Upon Entry:
 *    o  bufferp->data             - ptr to memory containing a request packet
 *    o  bufferp->offset           - postion where data starts in buffer
 *    o  bufferp->size             - size of entire request packet in buffer
 *    o  bufferp->translated_size  - 0 or don't care
 *    o  bufferp->packet_status    - don't care or CSI_PAKSTAT_INITIAL
```

```
*     o  bufferp->maxsize          - allocated size of bufferp->data
*
*     Upon Exit:
*     o  bufferp->data             - unaltered
*     o  bufferp->offset           - unaltered
*     o  bufferp->size             - unaltered
*     o  bufferp->translated_size  - bytes of request that xdr could translate
*     o  bufferp->packet_status    - CSI_PAKSTAT_XLATE_ERROR or
*                                    CSI_PAKSTAT_XLATE_COMPLETED
*     o  bufferp->maxsize          - unaltered
*
*     DE-SERIALIZATION:
*     For deserialization of a request packet, xdrsp->x_op equals XDR_DECODE.
*     During deserialization, if "bufferp->data" is NULL, xdr will allocate
*     memory for the packet read in off of the wire.  In this case it is
*     the responsibility of the caller to free that memory.
*
*     Upon entry to this routine, the data buffer description structure of
*     type CSI_MSGBUF must be initialized as follows:
*
*     Upon Entry:
*     o  bufferp->data             - memory where request packet will be put
*                           OR     - NULL for xdr to do allocation for caller
*     o  bufferp->offset           - start byte# where data is placed in buffer
*     o  bufferp->size             - 0 or don't care
*     o  bufferp->translated_size  - 0 or don't care
*     o  bufferp->packet_status    - don't care or CSI_PAKSTAT_INITIAL
*     o  bufferp->maxsize          - allocated size of bufferp->data
*
*     Upon Exit:
*     o  bufferp->data             - contains the translated data
*     o  bufferp->offset           - unaltered
*     o  bufferp->size             - bytes of request that xdr could translate
*     o  bufferp->translated_size  - bytes of request that xdr could translate
*     o  bufferp->packet_status    - CSI_PAKSTAT_XLATE_ERROR or
*                                    CSI_PAKSTAT_XLATE_COMPLETED or
*                                    CSI_PAKSTAT_DUPLICATE_PACKET
*     o  bufferp->maxsize          - unaltered bufferp->data not null on entry
*                           OR     - if xdr allocated, bufferp->translated_size
*
*     ERROR CONDITIONS AND HANDLING DURING TRANSLATION:
*     During translation, portions of a packet might not be translatable
*     for numerous reasons following:
*
*     o  xdr error,
*     o  client packet format error,
*     o  invalid command,
*     o  invalid identifier type,
*     o  invalid count,
*     o  invalid packet size for designated storage server command
*     o  duplicate packet
*
*     If at least the csi_request_header (type CSI_REQUEST_HEADER in
*     csi_structs.h) can be processed, then a partial packet is translated.
```

```
 *    The receiver of the partial packet can determine what the attempted
 *    operation was by analyzing the request header and the downstream
 *    portions of the packet that were translated.  The amount of the
 *    packet that was actually translated and presumably sent on the wire
 *    (barring a higher level Network layer error) is returned in
 *    "bufferp->translated_size".
 *
 * Return Values:
 *
 *      (bool_t)    1    - At least a partial conversion performed.
 *      (bool_t)    0    - Conversion failed.
 *
 * Implicit Inputs:
 *
 *      NONE
 *      bufferp->data            - buffer for packet data
 *      bufferp->offset          - byte# location of start of packet in buffer
 *
 * Implicit Outputs:
 *
 *      bufferp->data            - data is placed here during deserialization.
 *      bufferp->size            - size of data placed during deserialization
 *      bufferp->packet_status   - describes various translation errors
 *      bufferp->translated_size - bytes of data that could be translated
 *
 * Considerations:
 *
 *    This routine may return a partial packet.  Return code will be (1)
 *    if at least a request header can be serialized/deserialized.
 *
 *    The value of size and translated_size is undefined when 0 return code
 *    (error) is returned.
 *

/*
 *      Header Files:
 */
#include <rpc/rpc.h>
#include "structs.h"
#include "defs.h"
#include "csi.h"
#include "csi_xdr_xlate.h


bool_t
csi_xdrrequest(xdrsp, bufferp)
XDR          *xdrsp;        /* XDR handle */
CSI_MSGBUF   *bufferp;      /* data buffer description structure */
{
```

csi_xdrresponse() FUNCTION

```
/*
 *                              StorageTek SECRET
 *                  Property of Storage Technology Corporation
 *                  Do not copy or distribute unpublished work
 *
 *                              Copyright (1989)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Name:
 *
 *csi_xdrresponse()
 *
 * Description:
 *
 *    CSI high level xdr based routine for serializing and deserializing
 *    storage server response packets.  The routines in this source and those
 *    called from this source support a bi-directional protocol for either
 *    encoding or decoding storage server response packets based on the value
 *    of the XDR handle's "xdrsp->x_op" directional variable.
 *
 *    The data buffer passed, "bufferp->data", is encoded during serialization
 *    for up to "bufferp->size" bytes beginning with the byte number specified
 *    as "bufferp->offset".   During  deserialization, the  XDR stream
 *    is  translated out of  the xdr handle "xdrsp" into "bufferp->data"
 *    beginning at the offset specified by "bufferp->offset".
 *    The number of bytes of the packet that were successfully translated
 *    is returned in "bufferp->translated_bytes".
 *
 *    This routine will only return an error to the rpc layer (return 0)
 *    if the contents of the request header could not be translated.
 *
 *    If the request header was translated but there was a translation error
 *    lower in the packet, "bufferp->packet_status" equals
 *    CSI_PAKSTAT_XLATE_ERROR otherwise it equals CSI_PAKSTAT_XLATE_COMPLETED.
 *
 *    If a duplicate packet was detected during deserialization, this routine
 *    returns "bufferp->packet_status" equal to CSI_PAKSTAT_DUPLICATE_PACKET.
 *    In this case, upon return, only the CSI_REQUEST_HEADER portion of the
 *    packet will have been translated.
 *
 *    SERIALIZATION:
 *    For serialization of a response packet, xdrsp->x_op equals XDR_ENCODE.
 *    Upon entry to this routine, the data buffer description structure of
 *    type CSI_MSGBUF must be initialized as follows:
 *
 *    Upon Entry:
 *    o  bufferp->data            - ptr to memory containing a response packet
 *    o  bufferp->offset          - postion where data starts in buffer
 *    o  bufferp->size            - size of entire response packet in buffer
 *    o  bufferp->translated_size - 0 or don't care
 *    o  bufferp->packet_status   - don't care or CSI_PAKSTAT_INITIAL
```

```
*     o  bufferp->maxsize          - allocated size of bufferp->data
*
*     Upon Exit:
*     o  bufferp->data             - unaltered
*     o  bufferp->offset           - unaltered
*     o  bufferp->size             - unaltered
*     o  bufferp->translated_size  - bytes of response that xdr could translate
*     o  bufferp->packet_status    - CSI_PAKSTAT_XLATE_ERROR or
*                                    CSI_PAKSTAT_XLATE_COMPLETED
*     o  bufferp->maxsize          - unaltered
*
*     DE-SERIALIZATION:
*     For deserialization of a response packet, xdrsp->x_op equals XDR_DECODE.
*     During deserialization, if "bufferp->data" is NULL, xdr will allocate
*     memory for the packet read in off of the wire.  In this case it is
*     the responsibility of the caller to free that memory.
*
*     Upon entry to this routine, the data buffer description structure of
*     type CSI_MSGBUF must be initialized as follows:
*
*     Upon Entry:
*     o  bufferp->data             - memory where a response packet will be put
*                           OR     - NULL for xdr to do allocation for caller
*     o  bufferp->offset           - start byte# where data is placed in buffer
*     o  bufferp->size             - 0 or don't care
*     o  bufferp->translated_size  - 0 or don't care
*     o  bufferp->packet_status    - don't care or CSI_PAKSTAT_INITIAL
*     o  bufferp->maxsize          - allocated size of bufferp->data
*
*     Upon Exit:
*     o  bufferp->data             - contains the translated data
*     o  bufferp->offset           - unaltered
*     o  bufferp->size             - bytes of response that xdr could translate
*     o  bufferp->translated_size  - bytes of response that xdr could translate
*     o  bufferp->packet_status    - CSI_PAKSTAT_XLATE_ERROR or
*                                    CSI_PAKSTAT_XLATE_COMPLETED or
*                                    CSI_PAKSTAT_DUPLICATE_PACKET
*     o  bufferp->maxsize          - unaltered bufferp->data not null on entry
*                           OR     - if xdr allocated, bufferp->translated_size
*
*     ERROR CONDITIONS AND HANDLING DURING TRANSLATION:
*     During translation, portions of a packet might not be translatable
*     for numerous reasons following:
*
*     o  xdr error,
*     o  client packet format error,
*     o  invalid command,
*     o  invalid identifier type,
*     o  invalid count,
*     o  invalid packet size for designated storage server command
*     o  duplicate packet
*
*     If at least the csi_request_header (type CSI_REQUEST_HEADER in
*     csi_structs.h) can be processed, then a partial packet will be encoded.
```

```
 *    The receiver of the partial packet can determine what the attempted
 *    operation was by analyzing the request header and the downstream
 *    portions of the packet that were translated.  The amount of the
 *    packet that was actually translated and presumably sent on the wire
 *    (barring a higher level Network layer error) is returned in
 *    "bufferp->translated_size".
 *
 * Return Values:
 *
 *    bool_t    - 1 successful xdr conversion
 *    bool_t    - 0 xdr conversion failed
 *
 * Implicit Inputs:
 *
 *      NONE
 *
 * Implicit Outputs:
 *
 *      bufferp->data            - data is placed here during deserialization.
 *      bufferp->size            - size data put here during deserialization.
 *      bufferp->translated_size - bytes of data that could be translated.
 *      bufferp->packet_status   - describes various translation errors
 *
 *      bufferp->data            - buffer for packet data
 *      bufferp->offset          - byte# location of start of packet in buffer


 * Considerations:
 *
 *      This routine may return a partial packet. Return code will be (1) if
 *      at least a request header can be serialized/deserialized.
 *
 *      The value of packet size returned is undefined when 0 return code
 *      (error) is returned.
 *


/*
 *      Header Files:
 *
 *    See csi.h/rpc.h for other include files.
 */
#include <rpc/rpc.h>
#include "structs.h"
#include "defs.h"
#include "csi.h"
#include "csi_xdr_xlate.h


bool_t
csi_xdrresponse(xdrsp, bufferp)
XDR          *xdrsp;                    /* XDR handle */
CSI_MSGBUF   *bufferp;                  /* data buffer description structure
*/
{
```

(INTENTIONALLY LEFT BLANK)

APPENDIX D:
CSI AND SSI REQUIRED FILES

OVERVIEW

This Appendix includes the following files which are required by the CSI and the SSI:

- `csi_header.h` header file – Contains `CSI_HEADER` definitions.
- `csi.h` header file – Includes `csi_header.h`, `csi_structs.h`, and `csi_msg.h`. This file should be included or duplicated for CSIs and SSIs.
- `csi_structs.h` header file – Contains CSI data structure definitions. This file parallels the `lm_structs.h` header file, with the following differences:
  - `lm_structs.h` uses the `IPC_HEADER` return address structure, specifying a return socket name.
  - `csi_structs.h` uses the `CSI_HEADER` return address structure, specifying RPC and Internet return address information.

  Changes to this file must be reflected in `lm_structs.h`, and vice versa.
- `csi_msg.h` header file – Defines access numbers to each CSI message.
- `csi_getmsg.c` source file – Defines the contents of each CSI message.

csi_header.h HEADER FILE

```
/*  SccsId      @(#)csi_header.h    1.4 6/8/89 (c) 1988 StorageTek */
#ifndef _CSIHEADER_
#define _CSIHEADER_

/*
 *                          StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                          Copyright (1988)
 *                     Storage Technology Corporation
 *                          All Rights Reserved
 *
 * Functional Description:
 *
 *      Header file containing all CSI_HEADER structure-specific definitions.
 *
 * Modified by:
 *
 *      J. W. Wishner   30-Jan-1989.    Original.
 *                      J. A. Wishner   05/01/89.TIME STAMP-POST CUSTOMER
INITIAL RELEASE
 */

/*
 *      Header Files:
 */

/*
 *      Defines, Typedefs and Structure Definitions:
 */

/*
 *      Procedure Type Declarations:
 */
#define CSI_NETADDR_SIZE      6     /* #of bytes in a network address */

typedef struct {
    unsigned long       program;    /* callback program number */
    unsigned long       version;    /* version number */
    unsigned long       proc;       /* procedure number to call back to */
    struct sockaddr_in  raddr;      /* return internet address */
} CSI_HANDLE_RPC;

typedef enum {
    CSI_SYNTAX_NONE     = 0,        /* default transfer syntax is none */
    CSI_SYNTAX_XDR,                 /* XDR used as transfer syntax */
} CSI_SYNTAX;

typedef enum {
    CSI_PROTOCOL_TCP    = 1,        /* transport protocol used is TCP/IP */
    CSI_PROTOCOL_UDP    = 2,        /* transport protocol used is TCP/IP */
} CSI_PROTOCOL;

typedef enum {
    CSI_CONNECT_RPCSOCK = 1,        /* type of connection defined by CSI */
} CSI_CONNECT;

typedef struct {
    unsigned char       addr[CSI_NETADDR_SIZE]; /* sender network address */
    unsigned int        pid;        /* sender process id */
```

```
    unsigned long       seq_num;    /* sender sequence number */
} CSI_XID;

/*
 * Note:  the xid must stay at the very top of CSI_HEADER in order for
 * duplicate packet comparisons to work in csi_xdrrequest() &
csi_xdrresponsed()
 */
typedef struct {
    CSI_XID xid;                    /* transaction id=net address,pid,seq#*/
    unsigned long   ssi_identifier; /* identifier for use by SSI */
    CSI_SYNTAX      csi_syntax;     /* type of transfer syntax */
    CSI_PROTOCOL    csi_proto;      /* protocol used */
    CSI_CONNECT     csi_ctype;      /* type connection management used */
    CSI_HANDLE_RPC  csi_handle;     /* return handle of client */
} CSI_HEADER;
#endif
```

csi.h HEADER FILE

```
/*  SccsId      @(#)csi.h     1.11 6/21/89 (c) 1988 StorageTek */
#ifndef _CSI_
#define _CSI_
/*
 *
 *                          StorageTek SECRET
 *              Property of Storage Technology Corporation.
 *              Do not copy or distribute unpublished work.
 *
 *                      Copyright (1988)
 *                Storage Technology Corporation
 *                      All Rights Reserved
 * Name:
 *    csi.h
 *
 * Functional Description:
 *
 *    CSI/SSI interface include file for the client system interface.
 *
 * Modified by:
 *
 *    J. A. Wishner     12/02/88.Created.
 *    J. A. Wishner     05/01/89.TIME STAMP-POST CUSTOMER INITIAL RELEASE
 *    J. A. Wishner     05/15/89.Added csi routine level tracing
 *                      CSI_XDR_TRACE_LEVEL definition.
 *                      Cleaned out redundant #includes.
 *    J. A. Wishner     05/30/89Changed data portion allocation on
 *                      CSI_MSGBUF.  Was "char *", now data[1].
 *                      External for csi_netbuf goes to char *.
 *                      External for csi_netbuf_data goes away.
 *    J. A. Wishner     06/16/89Took limits off of queue sizes.
 *
 *
 */


#include <sys/types.h>
#include <rpc/rpc.h>
#include "cl_qm_defs.h"
#include "cl_qm.h"
#include "lm_structs.h"/*includes defs.h structs.h db_structs.h identifier.h*/
#include "csi_header.h"
#include "csi_structs.h"
#include "csi_msg.h"
/*
#undef DEBUG
*/

/*
 *    IPC related definitions
 *    Including workaround (for common library) for Sun IPC bug
 */
#ifndef INETSOCKETS
#define CSI_INPUT_SOCKET      "./to_CSI"        /* csi input socket name */
#define CSI_ACSLM_SOCKET      "./to_ACSLM"      /* acslm input socket name */
#else
#define CSI_INPUT_SOCKET ANY_PORT    /* csi input socket name */
#define CSI_ACSLM_SOCKET ACSLM      /* see defs.h acslm input socket name */
#endif

/*
```

```
 *    Miscellaneous CSI definitions
 */
#define CSI_XDR_TRACE_LEVEL          5     /* xdr routine level tracing */
#define CSI_DEF_CONNECTQ_AGETIME     172800/* seconds time connection aging */
#define CSI_SELECT_TIMEOUT           2     /* seconds time timeout */
#define CSI_DEF_RETRY_TIMEOUT        3     /* seconds per network send try */
#define CSI_DEF_RETRY_TRIES          5     /* number of times network retry */
#define CSI_HOSTNAMESIZE             32    /* size of name of host csi is on */
#define CSI_NO_CALLEE        (char *) NULL /* no fail function name passed to
                                            * to csi_logevent() */
typedef void (*CSI_VOIDFUNC)();           /* function as a variable on call */
#define CSI_NO_LOGFUNCTION           (CSI_VOIDFUNC) NULL
#define CSI_NO_SSI_IDENTIFIER 0   /* csi_header-no value in ssi_identifier */
#define CSI_ISFINALRESPONSE(opt) (0 == (INTERMEDIATE & opt) && \
                                  0 == (ACKNOWLEDGE & opt) ? TRUE : FALSE)
#define CSI_MAX_MESSAGE_SIZEMAX_MESSAGE_SIZE


/*
 *  packet transfer direction used in packet tracing routine(s)
 */
#define CSI_TO_ACSLM                 0     /* packet direction csi_ptrace() */
#define CSI_FROM_ACSLM               1     /* packet direction csi_ptrace() */


/*
 *  RPC variables specifically for a CSI.  Note:  The program numbers are
 *  effective until post-beta when numbers will be obtained from Sun.
 */
#define CSI_PROGRAM            0x200000fe  /* CSI RPC program number */
#define CSI_UDP_VERSION        1           /* RPC UDP server version# */
#define CSI_TCP_VERSION        2           /* RPC TCP server version# */
#define CSI_ACSLM_PROC         1000        /* RPC server procedure# */
#define CSI_DEF_TCPSENDBUF0                /* size tcp rpc send buffer */
#define CSI_DEF_TCPRECVBUF0                /* size tcp rpc receive buffer*/

/* network send options for routine csi_net_send() */
typedef enum {
    CSI_NORMAL_SEND,                       /* regular send of packet */
    CSI_FLUSH_OUTPUT_QUEUE,                /* flush network send queue */
} CSI_NET_SEND_OPTIONS;


/*
 *                       Environment Variables
 */
#define CSI_TCP_RPCSERVICE  "CSI_TCP_RPCSERVICE" /* registers as tcp server */
#define CSI_UDP_RPCSERVICE  "CSI_UDP_RPCSERVICE" /* registers as udp server */
#define CSI_CONNECT_AGETIME "CSI_CONNECT_AGETIME"/* # secs connection aging */
#define CSI_HOSTNAME        "CSI_HOSTNAME"       /* name of host csi is on */
#define CSI_RETRY_TIMEOUT   "CSI_RETRY_TIMEOUT"  /* per try NI timeout */
#define CSI_RETRY_TRIES     "CSI_RETRY_TRIES"   /* number of retrys */
#define CSI_TRACE_VALUE     "TRACE_VALUE"       /* trace label for getenv()*/


/*
 *                       Connection queue related defines for saving
csi_header return addresses
 */
#define CSI_MAXMEMB_LM_QUEUE         0            /* unlimited size LM Q */
#define CSI_MAXMEMB_NI_OUT_QUEUE     0            /* unlimited size NI out Q */
#define CSI_MAXQUEUES                2            /* max # of csi queues */
#define CSI_CONNECTQ_NAME "connection queue"      /* name of connection Q */
#define CSI_NI_OUTQ_NAME  "network output queue" /* name of net output Q */
```

```
#define CSI_QCB_REMARKS   "master control block" /* name of Q control block */


/*
 *   csi message buffer description structure passed to csi_xdrrequest() and
 *   csi_xdrresponse() XDR translation routines.
 */
/*   data offsets into packet buffer */
#define CSI_PAK_NETOFFSET (sizeof(CSI_HEADER) > sizeof(IPC_HEADER))\
                              ? 0 : sizeof(IPC_HEADER) - sizeof(CSI_HEADER)
#define CSI_PAK_LMOFFSET  (sizeof(CSI_HEADER) > sizeof(IPC_HEADER))\
                              ? sizeof(CSI_HEADER) - sizeof(IPC_HEADER) : 0
#define CSI_PAK_NETDATAP(bufp) ((bufp)->data) + ((char *)CSI_PAK_NETOFFSET)
#define CSI_PAK_LMDATAP(bufp)  ((bufp)->data) + ((char *)CSI_PAK_LMOFFSET)

/* packet buffer status for buffer of type CSI_MSGBUF */
typedef enum {
    CSI_PAKSTAT_INITIAL =     0,    /* currently testing packet */
    CSI_PAKSTAT_XLATE_COMPLETED,    /* packet translation completed */
    CSI_PAKSTAT_XLATE_ERROR,        /* translate error incomplete packet */
    CSI_PAKSTAT_DUPLICATE_PACKET,   /* packet in buffer is duplicate */
} CSI_PAKSTAT;

/* queue management */
typedef struct csi_q_mgmt {
    unsigned short      xmit_tries; /* number of attempts at transmission */
} CSI_Q_MGMT;

/* packet buffer */
typedef struct {
    int                 offset;     /* starting offset of packet data */
    int                 size;       /* size of the data in buffer */
    int                 maxsize;    /* maximum size of the data in buffer */
    int                 translated_size;/* size valid data xdr translatable */
    CSI_PAKSTAT         packet_status;  /* success/failure of translation */
    CSI_Q_MGMT          q_mgmt;     /* for managment of queueing */
    char                data[1];    /* starting address of data storage */
} CSI_MSGBUF;

#define CSI_MSGBUF_MAXSIZE(sizeof(CSI_MSGBUF) + CSI_MAX_MESSAGE_SIZE)



/*
 *   Merged request/response types used so data structures can be accessed
 *   at their top layer (common layered structure) to determine their type
 *   and handling.  Listed below, these are:
 */
/*
 *   packet structure definitions for requests sent from csi to the acslm
 */
typedef union {
    REQUEST_HEADER      req_header;
    AUDIT_REQUEST       audit_req;
    ENTER_REQUEST       enter_req;
    EJECT_REQUEST       eject_req;
    VARY_REQUEST        vary_req;
    MOUNT_REQUEST       mount_req;
    DISMOUNT_REQUEST    dismount_req;
    QUERY_REQUEST       query_req;
    CANCEL_REQUEST      cancel_req;
    START_REQUEST       start_req;
    IDLE_REQUEST        idle_req;
    EJECT_ENTER         eject_enter_req;
```

```
} CSI_LM_REQUEST;

/*
 *   packet structure definitions for responses sent from acslm to the csi
 */
typedef union {
    REQUEST_HEADER       req_header;
    ACKNOWLEDGE_RESPONSE ack_res;
    AUDIT_RESPONSE       audit_res;
    ENTER_RESPONSE       enter_res;
    EJECT_RESPONSE       eject_res;
    VARY_RESPONSE        vary_res;
    MOUNT_RESPONSE       mount_res;
    DISMOUNT_RESPONSE    dismount_res;
    QUERY_RESPONSE       query_res;
    CANCEL_RESPONSE      cancel_res;
    START_RESPONSE       start_res;
    IDLE_RESPONSE        idle_res;
    EJECT_ENTER          eject_enter_res;
} CSI_LM_RESPONSE;

/*
 *   packet structure definitions for requests sent from SSI/NI to the csi
 */
typedef union {
    CSI_REQUEST_HEADER   csi_req_header;
    CSI_AUDIT_REQUEST    csi_audit_req;
    CSI_ENTER_REQUEST    csi_enter_req;
    CSI_EJECT_REQUEST    csi_eject_req;
    CSI_VARY_REQUEST     csi_vary_req;
    CSI_MOUNT_REQUEST    csi_mount_req;
    CSI_DISMOUNT_REQUEST csi_dismount_req;
    CSI_QUERY_REQUEST    csi_query_req;
    CSI_CANCEL_REQUEST   csi_cancel_req;
    CSI_START_REQUEST    csi_start_req;
    CSI_IDLE_REQUEST     csi_idle_req;
    CSI_EJECT_ENTER      csi_eject_enter_req;
} CSI_REQUEST;

/*
 *   packet structure definitions for responses sent from the csi to SSI/NI
 */
typedef union {
    CSI_REQUEST_HEADER       csi_req_header;
    CSI_ACKNOWLEDGE_RESPONSE csi_ack_res;
    CSI_AUDIT_RESPONSE       csi_audit_res;
    CSI_ENTER_RESPONSE       csi_enter_res;
    CSI_EJECT_RESPONSE       csi_eject_res;
    CSI_VARY_RESPONSE        csi_vary_res;
    CSI_MOUNT_RESPONSE       csi_mount_res;
    CSI_DISMOUNT_RESPONSE    csi_dismount_res;
    CSI_QUERY_RESPONSE       csi_query_res;
    CSI_CANCEL_RESPONSE      csi_cancel_res;
    CSI_START_RESPONSE       csi_start_res;
    CSI_IDLE_RESPONSE        csi_idle_res;
    CSI_EJECT_ENTER          csi_eject_enter_res;
} CSI_RESPONSE;



/*
 *   external declarations for global variables
 */
extern QM_QID     csi_lm_qid;          /* connection queue to acslm */
```

```
extern QM_QID      csi_ni_out_qid;       /* network output queue */
extern long        csi_lmq_lastcleaned;  /* time acslm connect queue cleaned */
extern long        csi_connect_agetime;  /* aging time for connection */
extern int         csi_rpc_tcpsock;      /* rpc tcp service socket */
extern int         csi_rpc_udpsock;      /* rpc udp service socket */
extern BOOLEAN     csi_udp_rpcsvc;       /* TRUE if using RPC UDP server */
extern BOOLEAN     csi_tcp_rpcsvc;       /* TRUE if using RPC TCP server */
extern CSI_MSGBUF *csi_netbufp;          /* network packet buffer */
extern SVCXPRT     *csi_udpxprt;         /* CSI UDP transport handle */
extern SVCXPRT     *csi_tcpxprt;         /* CSI TCP transport handle */
extern QM_QID      csi_lm_qid;           /* ID of CSI connection queue */
extern CSI_HEADER  csi_ssi_address;      /* CSI header to build ssi packets */
extern IPC_HEADER  csi_ipc_header;       /* IPC header used to build packets */
extern int         csi_retry_timeout;    /* seconds per network send try */
extern int         csi_retry_tries;      /* number of times network retry */
extern unsigned char csi_netaddr[];      /* address of this host */
extern char        csi_hostname[];       /* name of this host */
extern int         csi_pid;              /* process id this program */


/*
 *   external declarations for csi internal routines
 */
extern void                    csi_fmtlmq_log();
extern void                    csi_fmtniq_log();
extern STATUS                  csi_freeqmem();
extern char                    *csi_getmsg();
extern STATUS                  csi_getiaddr();
extern STATUS                  csi_hostaddr();
extern STATUS                  csi_init();
extern STATUS                  csi_lminput();
extern STATUS                  csi_netbufinit();
extern STATUS                  csi_net_send();
extern STATUS                  csi_process();
extern void                    csi_ptrace();
extern STATUS                  csi_qclean();
extern STATUS                  csi_qget();
extern STATUS                  csi_qinit();
extern STATUS                  csi_qput();
extern STATUS                  csi_rpccall();
extern STATUS                  csi_rpctinit();
extern                         csi_rpcdisp();
extern int                     csi_rpcinput();
extern unsigned long           csi_rpctransient();
extern STATUS                  csi_rpcuinit();
extern int                     csi_sighdlr();
extern int                     csi_ssicmp();
extern STATUS                  csi_svcinit();
extern int                     csi_xidcmp();


/*
 *   external declarations for XDR conversion routines
 */
extern bool_t csi_xackresponse();
extern bool_t csi_xacs();
extern bool_t csi_xcap_id();
extern bool_t csi_xcell_id();
extern bool_t csi_xcol();
extern bool_t csi_xcommand();
extern bool_t csi_xcsi_hdr();
extern bool_t csi_xdrive_id();
extern bool_t csi_xdrrequest();
extern bool_t csi_xdrresponse();
```

```
extern bool_t csi_xeject();
extern bool_t csi_xeject_enter();
extern bool_t csi_xfreecells();
extern bool_t csi_xidentifier();
extern bool_t csi_xipc_hdr();
extern bool_t csi_xlocation();
extern bool_t csi_xlsm();
extern bool_t csi_xlsm_id();
extern bool_t csi_xmsg_hdr();
extern bool_t csi_xpnl();
extern bool_t csi_xpnl_id();
extern bool_t csi_xport();
extern bool_t csi_xport_id();
extern bool_t csi_xqu_response();
extern bool_t csi_xreq_hdr();
extern bool_t csi_xreqsummary();
extern bool_t csi_xres_status();
extern bool_t csi_xrow();
extern bool_t csi_xsockname();
extern bool_t csi_xspnl_id();
extern bool_t csi_xstate();
extern bool_t csi_xstatus();
extern bool_t csi_xtype();
extern bool_t csi_xvol_id();
extern bool_t csi_xvol_status();

#endif
```

csi_structs.h HEADER FILE

```
/*  SccsId      %W% %G% (c) 1988 StorageTek */
#ifndef _CSI_STRUCTS_
#define _CSI_STRUCTS_
/*
 *                              StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                              Copyright (1988)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Functional Description:
 *
 *    definitions of CSI data structures.
 *    includes "defs.h", unless already defined, to be complete.
 *    includes "identifier.h", unless already defined, to be complete.
 *    includes "db_structs.h", unless already defined, to be complete.
 *    includes "structs.h", unless already defined, to be complete.
 *
 *    NOTE:
 *    the structures defined here have corresponding definitions for the
 *    ACSLM in lm_structs.h.  any modifications to this file MUST be
 *    reflected in lm_structs.h as well.
 *
 * Modified by:
 *
 *    D. F. Reed          29-Jan-1989     Original.
 *    J. A. Wishner       30-Jan-1989     Added definitions for CSI_HEADER.
 */

/*
 *      Header Files:
 */

#include "defs.h"
#ifndef  _IDENTIFIER_
#include "identifier.h"
#endif   _IDENTIFIER_
#ifndef  _DB_STRUCTS_
#include "db_structs.h"
#endif   _DB_STRUCTS_
#include "structs.h"
#include "csi_header.h"

/*
 *    Defines, Typedefs and Structure Definitions:
 */

typedef struct {            /* fixed portion of request_packet */
    CSI_HEADER        csi_header;
    MESSAGE_HEADER    message_header;
} CSI_REQUEST_HEADER;

typedef struct {            /* intermediate acknowledgment */
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
    MESSAGE_ID         message_id;
} CSI_ACKNOWLEDGE_RESPONSE;
```

```
/*****************************************************************************
 *                        AUDIT REQUEST/RESPONSE STRUCTURES
 *
 *****************************************************************************/

typedef struct {                            /* audit_request
*/
    CSI_REQUEST_HEADER csi_request_header;
    CAPID              cap_id;        /*   CAP for ejecting cartridges      */
    TYPE               type;          /*   type of identifiers              */
    unsigned short     count;         /*   number of identifiers            */
    union {                           /*   list of homogeneous ids to audit */
        ACS            acs_id[MAX_ID];
        LSMID          lsm_id[MAX_ID];
        PANELID        panel_id[MAX_ID];
        SUBPANELID     subpanel_id[MAX_ID];
    } identifier;
} CSI_AUDIT_REQUEST;

typedef struct {                            /* audit_response                   */
    CSI_REQUEST_HEADER  csi_request_header;
    RESPONSE_STATUS     message_status;
    CAPID               cap_id;           /*   CAP for ejecting cartridges   */
    TYPE                type;             /*   type of identifiers           */
    unsigned short      count;            /*   number of audited identifiers */
    union {                               /*   list of ids audited w/status  */
        AU_ACS_STATUS   acs_status[MAX_ID];
        AU_LSM_STATUS   lsm_status[MAX_ID];
        AU_PNL_STATUS   panel_status[MAX_ID];
        AU_SUB_STATUS   subpanel_status[MAX_ID];
    } identifier_status;
} CSI_AUDIT_RESPONSE;

typedef struct {                           /* eject_enter intermediate response */
    CSI_REQUEST_HEADER  csi_request_header;
    RESPONSE_STATUS     message_status;
    CAPID               cap_id;         /*   CAP for ejecting cartridges     */
    unsigned short      count;          /*   no. of volumes ejected/entered  */
    VOLUME_STATUS       volume_status[MAX_ID];
} CSI_EJECT_ENTER;

/*****************************************************************************
 *                         EJECT REQUEST/RESPONSE STRUCTURES                 *
 *****************************************************************************/

typedef struct {                            /* eject request */
    CSI_REQUEST_HEADER csi_request_header;
    CAPID              cap_id;            /* CAP used for ejection */
    unsigned short     count;             /* Number of cartridges */
    VOLID              vol_id[MAX_ID];    /* External tape cartridge label */
} CSI_EJECT_REQUEST;

typedef CSI_EJECT_ENTER    CSI_EJECT_RESPONSE;

/*****************************************************************************
 *                         ENTER REQUEST/RESPONSE STRUCTURES                 *
 *****************************************************************************/

typedef struct {                            /* eject request */
    CSI_REQUEST_HEADER csi_request_header;
    CAPID              cap_id;            /* CAP used for entry */
} CSI_ENTER_REQUEST;
```

```
typedef CSI_EJECT_ENTER    CSI_ENTER_RESPONSE;
/*****************************************************************************
 *                        MOUNT REQUEST/RESPONSE STRUCTURES                  *
 *****************************************************************************/

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    VOLID              vol_id;
    unsigned short     count;
    DRIVEID            drive_id[1];
} CSI_MOUNT_REQUEST;

typedef struct  {
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
    VOLID              vol_id;
    DRIVEID            drive_id;
} CSI_MOUNT_RESPONSE;

/*****************************************************************************
 *                        DISMOUNT REQUEST/RESPONSE STRUCTURES               *
 *****************************************************************************/

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    VOLID              vol_id;
    DRIVEID            drive_id;
} CSI_DISMOUNT_REQUEST;

typedef struct  {
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
    VOLID              vol_id;
    DRIVEID            drive_id;
} CSI_DISMOUNT_RESPONSE;

/*****************************************************************************
 *                        QUERY REQUEST/RESPONSE STRUCTURES                  *
 *****************************************************************************/

typedef struct {                        /* query_request                */
    CSI_REQUEST_HEADER csi_request_header;
    TYPE               type;            /*   type of query                */
    unsigned short     count;           /*   number of identifiers          */
    union {                             /*   list of homogeneous ids to query */
    ACS                acs_id[MAX_ID];
    LSMID              lsm_id[MAX_ID];
    CAPID              cap_id[MAX_ID];
    DRIVEID            drive_id[MAX_ID];
    VOLID              vol_id[MAX_ID];
    MESSAGE_ID         request[MAX_ID];
    PORTID             port_id[MAX_ID];
    } identifier;
} CSI_QUERY_REQUEST;

typedef struct {                          /* query_response          */
    CSI_REQUEST_HEADER  csi_request_header;
    RESPONSE_STATUS     message_status;
    TYPE                type;             /*   type of query          */
    unsigned short      count;            /*   number of identifiers      */
    union {                               /*   list of ids queried w/status */
        QU_SRV_STATUS   server_status[MAX_ID];
        QU_ACS_STATUS   acs_status[MAX_ID];
        QU_LSM_STATUS   lsm_status[MAX_ID];
```

```
        QU_CAP_STATUS    cap_status[MAX_ID];
        QU_DRV_STATUS    drive_status[MAX_ID];
        QU_MNT_STATUS    mount_status[MAX_ID];
        QU_VOL_STATUS    volume_status[MAX_ID];
        QU_PRT_STATUS    port_status[MAX_ID];
        QU_REQ_STATUS    request_status[MAX_ID];
    } status_response;
} CSI_QUERY_RESPONSE;

/*****************************************************************************
 *                          VARY REQUEST/RESPONSE STRUCTURES                 *
 *****************************************************************************/

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    STATE              state;
    TYPE               type;
    unsigned short     count;
    union {                              /*   list of homogeneous ids to vary */
    ACS                acs_id[MAX_ID];
    LSMID              lsm_id[MAX_ID];
    DRIVEID            drive_id[MAX_ID];
    PORTID             port_id[MAX_ID];
    } identifier;
} CSI_VARY_REQUEST;

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
    STATE              state;
    TYPE               type;
    unsigned short     count;
    union {                              /*   list of ids varied w/status */
    VA_ACS_STATUS      acs_status[MAX_ID];
    VA_LSM_STATUS      lsm_status[MAX_ID];
    VA_DRV_STATUS      drive_status[MAX_ID];
    VA_PRT_STATUS      port_status[MAX_ID];
    } device_status;
} CSI_VARY_RESPONSE;

/*****************************************************************************
 *                          CANCEL REQUEST/RESPONSE STRUCTURES               *
 *****************************************************************************/

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    MESSAGE_ID         request;
} CSI_CANCEL_REQUEST;

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
    MESSAGE_ID         request;
} CSI_CANCEL_RESPONSE;

/*****************************************************************************
 *                          START REQUEST/RESPONSE STRUCTURES                *
 *****************************************************************************/

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
} CSI_START_REQUEST;

typedef struct {
```

```
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
} CSI_START_RESPONSE;


/*****************************************************************************
 *                          IDLE REQUEST/RESPONSE STRUCTURES                 *
 *****************************************************************************/

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
} CSI_IDLE_REQUEST;

typedef struct {
    CSI_REQUEST_HEADER csi_request_header;
    RESPONSE_STATUS    message_status;
} CSI_IDLE_RESPONSE;


/*
 *      Procedure Type Declarations:
 */

#endif _CSI_STRUCTS_
```

csi_msg.h HEADER FILE

```
/*  SccsId      @(#)csi_msg.h       1.9 6/19/89 (c) 1988 StorageTek */
#ifndef _CSIMSG_
#define _CSIMSG_              /* where MODULE == header module name */
/*
 *                              StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                              Copyright (1988)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Functional Description:
 *
 *      functional description of objects defined in header file.
 *
 * Modified by:
 *
 *      J. A. Wishner   18-Jan-1988.    Created.
 *      J. A. Wishner   05/01/89.TIME STAMP-POST CUSTOMER INITIAL RELEASE
 *
 */

/*
 *      Header Files:
 */

/*
 *    Defines, Typedefs and Structure Definitions:
 *
 *          Enumerated type for csi messages.
 *
 *    Considerations:
 *
 *          Must be kept in sync with the message declarations in
 *          csi_getmsg.c.
 */

typedef enum {
    MSG_FIRST = 0,/* invalid */
    MSG_UNMAPPED_RPCSERVICE,
    MSG_RPCTCP_SVCCREATE_FAILED,
    MSG_RPCTCP_SVCREGISTER_FAILED,
    MSG_RPCUDP_SVCCREATE_FAILED,
    MSG_RPCUDP_SVCREGISTER_FAILED,
    MSG_INITIATION_STARTED,
    MSG_INITIATION_COMPLETED,
    MSG_INITIATION_FAILURE,
    MSG_CREATE_CONNECTQ_FAILURE,
    MSG_CREATE_NI_OUTQ_FAILURE,
    MSG_LOCATE_QMEMBER_FAILURE,
    MSG_DELETE_QMEMBER_FAILURE,
    MSG_SYSTEM_ERROR,
    MSG_UNEXPECTED_SIGNAL,
    MSG_RPC_INVALID_PROCEDURE,
    MSG_RPC_INVALID_PROGRAM,
    MSG_RPC_CANT_REPLY,
    MSG_RPCTCP_CLNTCREATE,
    MSG_RPCUDP_CLNTCREATE,
    MSG_INVALID_PROTO,
    MSG_QUEUE_CREATE_FAILURE,
```

```
    MSG_QUEUE_STATUS_FAILURE,
    MSG_QUEUE_MEMBADD_FAILURE,
    MSG_QUEUE_CLEANING_START,
    MSG_UNDEF_MSG,
    MSG_UNDEF_MSG_TRUNC,
    MSG_UNDEF_MODULE_TYPE,
    MSG_UNDEF_CLIENT,
    MSG_MESSAGE_SIZE,
    MSG_MESSAGE_SIZE_TRUNC,
    MSG_ACSLM_SEND_FAILURE,
    MSG_ACSLM_READ_FAILURE,
    MSG_SEND_NI_FAILURE,
    MSG_SEND_ACSSA_FAILURE,
    MSG_INVALID_COMM_SERVICE,
    MSG_XDR_XLATE_FAILURE,
    MSG_RPC_CANT_FREEARGS,
    MSG_QUEUE_ENTRY_DROP,
    MSG_UNDEF_HOST,
    MSG_TERMINATION_STARTED,
    MSG_TERMINATION_COMPLETED,
    MSG_DUPLICATE_ACSLM_PACKET,
    MSG_INVALID_NI_TIMEOUT,
    MSG_DUPLICATE_NI_PACKET,
    MSG_NI_TIMEDOUT,
    MSG_UNEXPECTED_FAILURE,
    MSG_INVALID_COMMAND,
    MSG_INVALID_TYPE,
    MSG_LAST,                          /* invalid */
} CSI_MSGNO;

/*
 *      Procedure Type Declarations:
 */
/* external procedure declarations */
#endif _CSIMSG_
```

csi_getmsg.c SOURCE FILE

```
#ifndef lint
static char SccsId[] = "@(#)csi_getmsg.c  1.16 6/19/89 (c) 1989 StorageTek";
#endif
/*
 *                              StorageTek SECRET
 *                Property of Storage Technology Corporation
 *                Do not copy or distribute unpublished work
 *
 *                              Copyright (1989)
 *                        Storage Technology Corporation
 *                              All Rights Reserved
 *
 * Name:
 *
 *    csi_getmsg()
 *
 * Description:
 *
 *    Functions returns a pointer to a csi message, accessed via  defined
 *    value (defined in csi_msg.h).
 *
 * Return Values:
 *
 *      (char *)- pointer to a static message string
 *
 * Implicit Inputs:
 *
 *      st_msgtab- static message table
 *
 * Implicit Outputs:
 *
 *
 *
 * Considerations:
 *
 *    Messages must be kept in sync with the enumerated type CSI_MSGNO
 *    in csi_msg.h.  Also, table must be fully populated and messages must
 *    at least be defined as an empty string (at a minumum).
 *
 *
 * Module Test Plan:
 *
 *    NONE
 *
 * Revision History:
 *
 *      J. A. Wishner       25-Jan-1989.    Created.
 *
 *
 */


/*
 *      Header Files:
 */
#include "csi.h"



/*
 *    Defines, Typedefs and Structure Definitions:
```

```
 */

/*
 *    Global and Static Variable Declarations:
 */
static char             *st_src = __FILE__;
static char             *st_module = "csi_getmsg()";

static struct           st_msg {
    CSI_MSGNO           msgno;
    char                *msg;
} st_msgtab [] = {
MSG_FIRST,                      "Invalid Message MSG_FIRST",
MSG_UNMAPPED_RPCSERVICE,        "Unmapped previously registered RPC service.",
MSG_RPCTCP_SVCCREATE_FAILED,    "Create of RPC TCP service failed",
MSG_RPCTCP_SVCREGISTER_FAILED,"Can't register RPC TCP service",
MSG_RPCUDP_SVCCREATE_FAILED,    "Create of RPC UDP service failed",
MSG_RPCUDP_SVCREGISTER_FAILED,"Can't register RPC UDP service",
MSG_INITIATION_STARTED,         "Initiation Started",
MSG_INITIATION_COMPLETED,       "Initiation Completed",
MSG_INITIATION_FAILURE,         "Initiation of CSI Failed",
MSG_CREATE_CONNECTQ_FAILURE,    "Creation of connect queue failed",
MSG_CREATE_NI_OUTQ_FAILURE,     "Creation of network output queue failed",
MSG_LOCATE_QMEMBER_FAILURE,     "Can't locate queue Q-id:%d, Member:%d",
MSG_DELETE_QMEMBER_FAILURE,     "Can't delete Q-id:%d, Member:%d",
MSG_SYSTEM_ERROR,               "Operating system error %d",
MSG_UNEXPECTED_SIGNAL,          "Unexpected signal caught, value:%d",
MSG_RPC_INVALID_PROCEDURE,      "Invalid procedure number",
MSG_RPC_INVALID_PROGRAM,        "Invalid RPC program number",
MSG_RPC_CANT_REPLY,             "Cannot reply to RPC message",
MSG_RPCTCP_CLNTCREATE,          "RPC TCP client connection failed,
%s\nAddress:%lu, Port:%d",
MSG_RPCUDP_CLNTCREATE,          "RPC UDP client connection failed,
%s\nAddress:%lu, Port:%d",
MSG_INVALID_PROTO,              "Invalid network protocol",
MSG_QUEUE_CREATE_FAILURE,       "Queue creation failure",
MSG_QUEUE_STATUS_FAILURE,       "Can't get queue status Q-id:%d, Member:%d",
MSG_QUEUE_MEMBADD_FAILURE,      "Can't add member to queue Q-id:%d",
MSG_QUEUE_CLEANING_START,       "Starting cleanup of connection queue, Q-id %d",
MSG_UNDEF_MSG,                  "Undefined message detected: discarded",
MSG_UNDEF_MSG_TRUNC,            "Invalid message contents from NI: truncated",
MSG_UNDEF_MODULE_TYPE,          "Unsupported module type %d detected:
discarded",
MSG_UNDEF_CLIENT,               "Message for unknown client discarded",
MSG_MESSAGE_SIZE,               "Invalid message size, %d, from NI: discarded",
MSG_MESSAGE_SIZE_TRUNC,         "Invalid message size, %d, from NI: truncated",
MSG_ACSLM_SEND_FAILURE,         "Cannot send message to ACSLM: discarded",
MSG_ACSLM_READ_FAILURE,         "Cannot read message from the ACSLM: discarded",
MSG_SEND_NI_FAILURE,            "Cannot send message to NI: discarded,
%s\nAddress:%lu, Port:%d",
MSG_SEND_ACSSA_FAILURE,         "Cannot send message to ACSSA: discarded",
MSG_INVALID_COMM_SERVICE,       "Invalid communications service",
MSG_XDR_XLATE_FAILURE,          "XDR message translation failure",
MSG_RPC_CANT_FREEARGS,          "Cannot decode to free memory allocated by XDR",
MSG_QUEUE_ENTRY_DROP,           "Dropping from Queue: Address:%lu, Port:%d,
ssi_identifier:%d, Protocol:%d, Connect type:%d",
MSG_UNDEF_HOST,                 "Undefined hostname",
MSG_TERMINATION_STARTED,        "Termination Started",
MSG_TERMINATION_COMPLETED,      "Termination Completed",
MSG_DUPLICATE_ACSLM_PACKET,     "Duplicate packet from ACSLM detected:
discarded",
MSG_INVALID_NI_TIMEOUT,         "Invalid network timeout value",
MSG_DUPLICATE_NI_PACKET,        "Duplicate packet from Network detected:
discarded\naddress:%lu, process-id:%d, sequence number:%lu",
```

```
MSG_NI_TIMEDOUT,                "Network timeout",
MSG_UNEXPECTED_FAILURE,         "Unexpected failure detected: errno=%d",
MSG_INVALID_COMMAND,            "Invalid command",
MSG_INVALID_TYPE,               "Invalid type",
MSG_LAST,                       "Invalid Message MSG_LAST"
}; /* end of table declaration */


/*
 *      Procedure Type Declarations:
 */

char *
csi_getmsg(msgno)
CSI_MSGNO   msgno;                  /* message number */
{

            int i;                  /* message number */

#ifdef DEBUG
    if TRACE(0)
        cl_trace(st_module,         /* routine name */
        1,                          /* parameter count */
        (unsigned long) msgno);     /* argument list */
#endif DEBUG


    i = (int) msgno;

                                    /* make sure status if valid */
    if (i <= (int) MSG_FIRST || i >= (int) MSG_LAST) {
      cl_log_unexpected(st_module, st_module, STATUS_INVALID_MESSAGE);
      return("error: unknown message");
    }

                                    /* return valid message */
    return(st_msgtab[i].msg);

}
```

(INTENTIONALLY LEFT BLANK)

GLOSSARY OF TERMS

A

AC — Alternating Current.

ACS — See *Automated Cartridge System.*

ACSEL — See *ACS Event Logger.*

ACS Event Logger (ACSEL) — The Storage Server software component that receives messages from other Storage Server components and writes them to an Event Log.

ACS ID — A unique identifier for an ACS.

ACSLH — See *ACS Library Handler.*

ACS library — A library is composed of one or more ACSs, attached 4480 tape cartridge drives, and cartridges residing in the ACSs.

ACS Library Handler (ACSLH) — The part of the ACS Library Manager that communicates directly with the LMU.

ACS Library Manager (ACSLM) — The Storage Server software component that validates and routes library requests and responses.

ACSLM — See *ACS Library Manager.*

ACSSA — See *ACS System Administrator.*

ACS System Administrator (ACSSA) — The Storage Server software component that provides a screen interface enabling library operators and users to monitor and control Storage Server operations.

Automated Cartridge System (ACS) — The library subsystem consisting of one LMU, and one to sixteen LSMs connected to that LMU.

Automated library — See *Library.*

B

Bar code — A code consisting of a series of bars with varying widths. This code appears on the external label attached to the spine of a cartridge, and its value is equivalent to the volume serial number. This code is read by the robot's machine vision system.

Beginning of Tape — The location on a tape where written data begins.

BOT — See *Beginning of Tape.*

BSD — Acronym for Berkeley Software Distribution, a version of the UNIX operating system.

C

CAP — See *Cartridge Access Port.*

CAP ID — A CAP ID uniquely identifies the location of a CAP by the LSM on which it resides. A CAP ID consists of the ACS ID and the LSM number.

Cartridge — A plastic housing containing a length of data recording tape. It is approximately 4 inches (100mm) by 5 inches (125 mm) by 1 inch (25mm). The tape is threaded automatically when loaded in a transport. A plastic leader block is attached to the tape for automatic threading. The spine of the cartridge contains an OCR/Bar Code label listing the volume ID.

Cartridge Access Port (CAP) — A bidirectional port, built into the door panel of a LSM, which provides for the manual entry or automatic ejection of tape cartridges.

Cartridge drive — A device containing two or four cartridge transports and their associated power and pneumatic supplies.

Cartridge transport — An electromechanical device that moves tape from a cartridge over a head that writes data on and reads data from the tape. A transport is distinct from the power and pneumatic sources that supply the electricity and air it needs to function. See *Cartridge drive.*

Cell — A receptacle in the LSM in which a cartridge is stored.

Central Support Remote Center (CSRC) — An installation whose operators can access and test StorageTek products over telephone lines.

Channel — A device that connects the host and main storage with the input and output control units.

Checkpoint — A static backup of a data base.

Client applications — Software applications that manage tape cartridge contents. They access tape cartridges by interacting with the Storage Server. Any number of client applications can be resident on a client system.

Client System Interface (CSI) — The Storage Server software component that translates and routes messages between the ACS Library Manager and the Storage Server Interfaces.

Client system user — A person who executes applications on a client system.

Command Processor — The screen interface of the ACS System Administrator. The Command Processor performs basic syntax validations on user input.

Control Unit — A microprocessor-based unit logically situated between a channel and up to sixteen cartridge transports. It translates channel commands into transport commands and sends transport status to the channel.

CSE — Acronym for Customer Services Engineer.

CSI — See *Client System Interface.*

CSRC — See *Central Support Remote Center.*

CU — See *Control Unit.*

D

Data base — A collection of interrelated data files.

Data base catalog — A file that keeps track of data base files.

Data base management system (DBMS) — The process that accesses, controls, organizes, and modifies a data base.

DC — Direct Current.

E

EOT — Acronym for End of Tape.

EPO — Acronym for Emergency Power Off.

Event Log — A file, maintained by the ACS Event Logger, that contains messages describing significant library and Storage Server events; these events include errors.

External label identifier — A six-character alphanumeric label adhered to an outside edge of a tape cartridge. It is used to identify a physical

tape volume. It may consist of upper case letters A through Z, numerals 0 through 9, and blanks.

F

FIFO — First In/First Out

G

H

Home location — The cell in an LSM in which a cartridge is currently stored.

I

ID — Identifier or identification.

Initial Program Load (IPL) — A process that activates a machine reset, initiates wake up diagnostics (from EPROMs) and, upon completion of wake up, loads functional code from a floppy disk.

Inline diagnostics — Routines that test components of a subsystem while operating on a time-sharing basis with the functional microcode in the subsystem component.

I/O — Input/Output

IPC — Acronym for Interprocess Communication

IPL — See *Initial Program Load.*

J

Journal — A sequential log of changes made to the data base since the last checkpoint.

K

L

LAN — See *Local Area Network.*

LCU — See *Library Control Unit.*

Library — A library is composed of one or more ACSs, attached 4480 cartridge drives, volumes placed in to the ACSs, the Storage Server software that controls and manages the ACSs, and the data base that describes the states of the ACSs.

Library Control Unit (LCU) — The portion of the LSM that controls the picking, mounting, dismounting, and replacing of tape cartridges.

Library drive — A cartridge transport that is attached to an LSM and is connected to, and controlled by, a client system. Library drives interact with the LCU during automated tape cartridge mount and dismount operations. Library drives interact with a client application during tape data transfer operations. Library drives are individually addressable by the ACSLM and are individually accessible by client applications. See *Cartridge Transport.*

Library Management Unit (LMU) — The portion of an ACS that manages the LSM, allocates its resources, and communicates with the Storage Server.

Library Storage Module (LSM) — The portion of an ACS that provides the storage area for cartridges and the robot necessary for moving them.

Light Emitting Diode (LED) — A light emitting device that uses little energy and is used mainly to indicate on/off conditions.

LMU — See *Library Management Unit.*

Local Area Network (LAN) — A computer network in which any component in the network can access any other component. This is the type of interface between an LMU and attached LSMs.

LSM — See *Library Storage Module.*

LSM ID — A unique identifier for an LSM. The LSM ID consists of the ACS ID and the LSM number.

M

Machine Initiated Maintenance (MIM) — A unique feature of the 4400 ACS in which an expert system monitors conditions and performance of the subsystem and requests attention before a problem becomes serious enough to impact operations. The customer can set threshold levels.

MIM — See *Machine Initiated Maintenance.*

N

Network Adapter — Equipment that provides an electrical and logical interface between a network and specific attached equipment.

Network Interface (NI) — An interface between the server system and the client systems that maintains network connections and controls the exchange of messages. A Network Interface is resident on the server system and each client system.

NI — See *Network Interface.*

O

OCR — Optical Character Recognition.

OCR label — An external label attached to the spine of a cartridge that is both human- and machine-readable.

OSI — Acronym for Open Systems Interconnection, a software architecture model of the International Organization for Standardization. The OSI model provides standards for the interconnection of data processing systems.

P

Pass-Thru Port (PTP) — Mechanism that allows a cartridge to be passed from one LSM to another in a multiple LSM ACS.

PTP — See *Pass-Thru Port.*

Q

R

Relational data base — A data base that is organized and accessed according to relationships between the data items; relationships are represented by tables.

RPC — Acronym for Remote Procedure Call.

S

Server system — The part of the library that is the residence for the Storage Server software. The server system acts as an interface between a library and any number of client systems.

Servo — A system that uses feedback to control a process.

Server system user — A person who invokes ACS Storage Server commands, utilities or procedures, on the server system. Server system users are generally site and maintenance personnel (for

example, library operators, tape librarians, system administrators, CSEs, and systems personnel).

SQL — See *Structured Query Language*.

SSI — See *Storage Server Interface*.

SSR — Acronym for Software Support Representative.

Storage Server — The software that interprets library commands from client applications or library operators and routes them to the appropriate LMU. The Storage Server consists of the following software components: ACS Library Manager (ACSLM), ACS System Administrator (ACSSA), Client System Interface (CSI), ACS Event Logger (ACSEL), Network Interface (NI), and Storage Server data base.

Storage Server data base — A data base used by the Storage Server to track the library configuration and the locations and IDs of all tape cartridges in the library.

Storage Server Interface (SSI) — A software component, resident on a client system, that translates and routes messages between client applications and the Client System Interface.

Structured Query Language (SQL) — A language used to define, access, and update data in a data base.

SVID — Acronym for System V Interface Definition.

T

TCP — Acronym for Transport Connect Protocol.

TLMS — Acronym for Tape Library Management System, a type of client application.

Transport — An electromechanical device capable of threading tape from a cartridge, moving the tape across a read/write head, and writing data onto or reading data from the tape.

U

UNIX — An operating system originally developed by Bell Laboratories and used by a variety of computer systems.

UDP — Acronym for User Datagram Protocol.

V

Volume ID — A six-character string that uniquely identifies a tape cartridge to the data base.

Volume serial number — A synonym for external label identifier.

W

X

XDR — Acronym for External Data Representation.

Y

Z

INDEX

Numerics

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

X