OpenVault™ Infrastructure Programmer's Guide

Document Number 007-3305-002

CONTRIBUTORS

Written by Bill Tuthill
Production by Allen Clardy
Engineering contributions by Loellyn Cassell, Curtis Anderson, and Joshua Toub

OpenVault™ Infrastructure Programmer's Guide
Document Number 007-3305-002

Contents

List of Figures

List of Tables

About This Guide

OpenVault software allows multiple applications to manage, mount, and unmount removable media. This product supports a wide range of removable media libraries and drives. OpenVault software simplifies the engineering of device support for new removable media systems, and streamlines the release of revised versions and modules. It also allows sharing of storage devices by applications such as backup, archive, hierarchical storage management (HSM), and direct tape access.

This document describes how to program the control program components that manage removable media drives and libraries. In OpenVault, the media library manager (MLM) fulfills requests from multiple client applications, directing media operations such as mount and unmount that are performed by control programs.

The *OpenVault Applications Programming Guide* describes the client side of OpenVault, showing how applications can make OpenVault requests in a prescribed format.

Intended Audience

This document is intended for system programmers who are adding support for removable media libraries or drives. By conforming to the standard OpenVault infrastructure, developers can eliminate the need to write custom interfaces for each removable media library and drive in the marketplace.

What This Guide Contains

Here is an overview of the material in this book:

- Chapter 1, “OpenVault Overview,” contains a thumbnail sketch of components.
- Chapter 2, “Common Implementation Issues,” covers topics you should know about before constructing an OpenVault control program.

- Chapter 3, “Abstract Library Interface (ALI) Language,” describes the language used for library control programs.
- Chapter 4, “Programming a Library Control Program (LCP),” offers a tutorial introduction to creating a library control program.
- Chapter 5, “Abstract Drive Interface (ADI) Language,” describes the language used for drive control programs.
- Chapter 6, “Programming a Drive Control Program (DCP),” offers a tutorial introduction to creating a drive control program.
- Appendix A, “Sample Implementations,” contains control program source code.
- Appendix B, “Return Values and Ready States,” lists these by control program.
- Appendix C, “LCP and DCP Syntax,” specifies control program syntax.
- “Glossary” and index are included at the end.

Conventions Used in This Guide

These are the typographic conventions used in this guide:

Purpose	Example
Names of keywords and functions	The **lcp_init()** function initializes LCP data structures.
Names of shell commands	The *umsh* command provides access to OpenVault volumes.
Titles of manuals	Refer to the *OpenVault Applications Programming Guide*.
A term defined in the glossary	The unit of OpenVault storage is a *cartridge*.
Filenames and pathnames	The control path to the drive is */dev/rmt/tps0d4*.
What you type; variables in italic	**`cc -g`** *sourcename*`.c -lmlm -lali`
Exact quotes of computer output	`Error: device not connected`

Chapter 1

OpenVault Overview

OpenVault helps simplify the engineering of software to control removable media libraries, by providing standard interfaces for robotic libraries, loadable drives, client applications, and library administration.

This chapter describes in more detail what this product provides and why it is useful, and gives an overview of OpenVault architecture and its standard interfaces.

What OpenVault Does

OpenVault is a package of mediation software that helps other applications manage removable media. This facility can support a wide range of removable media libraries, as well as a variety of drives interfaced to these libraries. The modular design of OpenVault eases the task of adding support for new robotic libraries and drives.

A unit of removable media is called a *cartridge*. This could be a tape reel, a tape cartridge, an optical disc, a removable magnetic disk, or a videotape.

OpenVault itself does not provide an end-user interface, nor does it generally become involved in I/O operations to cartridges loaded in drives. User interfaces are provided by OpenVault client applications, which perform I/O to drives using system facilities after control programs have mounted and loaded a cartridge for the application.

The following tertiary storage applications can all benefit from OpenVault:

- tape access, for example with *tar* or *cpio*
- backup, to guard against system crash or accidental data loss
- archive, for long-term storage of unused data
- hierarchical storage management (HSM)
- CD-ROM jukeboxes or information libraries
- broadcast libraries containing videotapes

Why OpenVault Is Needed

Because of the proliferation of data, many information professionals have trouble putting their fingers on the data they want. Secondary storage on disk drives is usually near capacity, and is generally devoted to system overhead and working files. Tertiary storage often contains the desired data, but is reachable only after expenditure of time and effort. Attentive management of removable media libraries can enhance the availability of information without significantly increasing overall system cost.

The traditional way of dealing with robotic libraries is with specialized applications that interface to particular libraries and drives. Generally, devices are monopolized by a single application. This approach has several shortcomings:

- Manufacturers of robotic libraries and drives have to develop device drivers for each new product on all supported system platforms.
- Software vendors must develop additional code to integrate new robotic libraries and drives, resulting in product support delays.
- Computer system providers have a difficult time offering a complete range of robotic libraries and applications when customers want them.
- Users and administrators have no access to the removable media library except as granted by a specialized application—sharing is not possible.

OpenVault solves these problems by providing a set of standard interfaces that raise the level of abstraction, enabling rapid deployment of removable media libraries, drives, systems, and client applications.

OpenVault as Middleware

Software that mediates between operating systems and application programs is called *middleware*. Middleware creates a common language so that users can access data in a variety of formats or using devices from different vendors. OpenVault is middleware in the sense that it mediates between client applications and device control programs, making it possible for different users to share a removable media library.

Middleware can often improve release independence. With its modular architecture, OpenVault assists vendors in adding support for new removable media libraries and drives and delivering upgraded client applications, without requiring rerelease of other OpenVault components.

OpenVault Architecture

OpenVault is organized as a set of cooperating components, as shown in Figure 1-1.

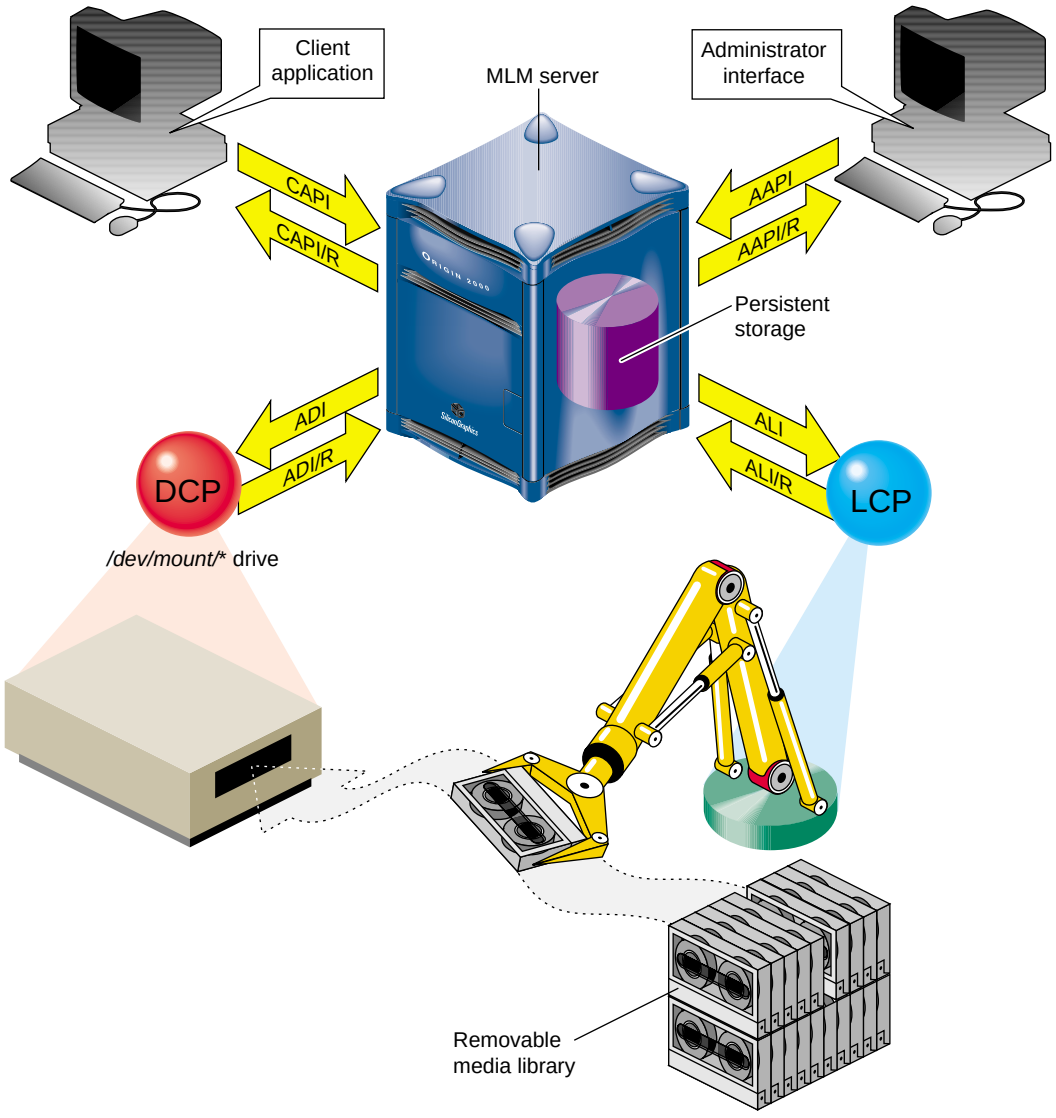

Figure 1-1 OpenVault Architecture

The central mediation component is the media library manager (MLM), a multithreaded process that accepts client connections and fulfills access requests by forwarding them to appropriate library and drive control programs. The MLM server maintains persistent storage containing information about cartridges in the system, and descriptions of authorized applications, libraries, and drives.

OpenVault consists of the following pieces:

1. One MLM server process mediates among other components.
2. Any number of client applications can make requests using the client application programming interface, CAPI; the MLM server replies in CAPI response (CAPI/R).
3. An administrative interface makes system requests in a similar but less restricted administrative API, AAPI; the MLM server replies in AAPI response (AAPI/R).
4. Persistent storage (a database) tracks cartridges and system components.
5. A library control program (LCP) is required for each removable media library controlled by the MLM server.

   The MLM server talks to an LCP using the abstract library interface (ALI), and receives answers in ALI response (ALI/R). An LCP translates from ALI to the actual library control interface, and replies in ALI/R.
6. A drive control program (DCP) is required for each drive controlled by the MLM server. Some removable media libraries contain multiple drives, in which case each drive has its own DCP. Drives need not be associated with a robotic library.

   The MLM server talks to a DCP using the abstract drive interface (ADI), and receives answers in ADI response (ADI/R). A DCP translates from ADI to the actual drive control interface, and replies in ADI/R.

The OpenVault languages consist entirely of ASCII strings.

MLM Server

The MLM server accepts requests from applications, and forwards commands to an LCP and DCP, which translate them into low-level robotic and drive control operations to serve that request. MLM also schedules competing requests from different applications, creates and enforces cartridge groups for each application, and maps logical cartridge names (used by applications) to physical cartridge labels (used by libraries).

The MLM server manages cartridges, directing LCP and DCP to mount and unmount a cartridge. Often, cartridges store data. After requesting that a cartridge be mounted, the client application may read and write the media using POSIX® standard I/O interfaces. Cartridges can also store audio-video streams for broadcast. In either case, MLM is not directly involved in I/O operations.

Client applications, libraries, and drives may be added to a live MLM server. The system administrator installs new programs on the appropriate hosts, and issues administrative commands on a live system to inform the MLM server that these new programs exist.

Cartridge Naming

Client applications may choose their own names for cartridges. Because OpenVault client applications operate in separate name spaces, different applications may use the same name for different cartridges. Moreover, cartridges used by one application are not visible to or accessible from another application, unless the system administrator permits specific cartridges to be moved from one application to another.

Some robotic libraries can interpret barcodes and labels affixed to cartridges. It is the responsibility of the LCP to pass any physical cartridge label (PCL) information to the MLM server.

Communication Paths

The OpenVault languages CAPI, CAPI/R, AAPI, AAPI/R, ALI, ALI/R, ADI, and ADI/R are expressed exclusively in text strings, which travel between components by means of TCP sockets. The underlying communications layer is encapsulated in a C library, so OpenVault developers need not worry about the details.

OpenVault Interfaces

This section describe the various OpenVault programming interfaces.

CAPI for Client Applications

CAPI (client application programming interface) is the language client applications use to communicate with the MLM server.

The command-response format is semi-asynchronous. After submitting each command, the application waits for the server to acknowledge receiving the command, but need not wait for results before sending the next command. CAPI communications libraries can also work synchronously if this makes implementation more convenient.

Access to the server is session-oriented. The application initiates a session with the *hello* command, and ends with a *goodbye*. Meanwhile, the application may send commands to the server to mount and unmount removable media, or to change attributes of media.

Here is a list of CAPI commands organized alphabetically:

- *allocate* requests volumes for use by this application.
- *attribute* sets attribute-value pairs associated with OpenVault volumes.
- *deallocate* returns volumes to the free pool.
- *mount* asks the MLM server to provide volumes for data access.
- *reject* tells the server to recategorize a volume.
- *rename* declares a new name for a volume.
- *show* displays information about OpenVault volumes.
- *unmount* says that volumes are no longer needed for data access.
- *unwelcome* informs the client of an MLM server version mismatch.
- *welcome* tells the client which version of the MLM server is responding.

The *OpenVault Application Programming Guide* describes how to program CAPI.

AAPI for Administrative Applications

AAPI (administrative API) is the language that administrative applications use to communicate with the MLM server. AAPI commands and responses are ASCII strings. As with CAPI, the command-response format is semi-asynchronous, and access to the server is session-oriented. AAPI is a superset of CAPI.

Here is a list of AAPI commands organized alphabetically:

- *attribute* sets attribute-value pairs associated with OpenVault volumes.
- *create* establishes a volume or object in the OpenVault database.
- *delete* removes a volume or object from the OpenVault database.
- *eject* pushes a cartridge out of a library into the operator's hand.
- *export* removes a volume from the OpenVault database.
- *inject* allows the operator to insert a cartridge into a library.
- *mount* tells the MLM server to provide data access to a volume.
- *move* relocates a cartridge from one slot in a library to another.
- *rename* declares a new name for a volume.
- *show* displays information about OpenVault volumes.
- *unwelcome* informs the client of an MLM server version mismatch.
- *unmount* says that volumes are no longer needed for data access.
- *welcome* tells the client which version of the MLM server is responding.

The *OpenVault Application Programming Guide* describes how to program the AAPI.

Abstract Library Interface (ALI)

A library control program (LCP) is a part of OpenVault that deals with low-level details of a removable media library and its configuration and control procedures. There is at least one LCP associated with each MLM-managed library. The purpose of an LCP is to expose library configuration to the MLM server, and to control a library as requested.

The MLM server issues directives to the LCP in a language called ALI. The LCP replies to the MLM server in a language called ALI response (ALI/R).

ALI/R implements a different command set from ALI, reflecting different needs of an LCP and the MLM server. The ALI language is primarily a library control interface, whereas ALI/R constitutes a status reporting interface with support for administration and configuration. Like CAPI, ALI and ALI/R are semi-asynchronous.

If you are developing a library control program, your program must be able to read ALI from, and write ALI/R to, the MLM server. The OpenVault infrastructure developer's kit includes an ALI parser and ALI/R generator. The parser and generator, as well as the communications layer, are delivered with a C language interface.

Here is a list of ALI commands organized alphabetically:

- *activate disable* forces the LCP to stop talking to the library.
- *activate enable* forces the LCP to resynchronize its internal information with the physical state of the library, and keep it synchronized.
- *attribute* sets and unsets named attributes in the LCP.
- *barrier* tells the LCP to complete all asynchronous commands before continuing.
- *cancel* revokes a command that the LCP has queued but not yet started.
- *eject* pushes a cartridge out of the library immediately, or queues a cartridge to be pushed out of the library (if queueing is implemented).
- *exit* tells the LCP to store state information, clean up, and exit.
- *mount* asks the LCP to put cartridges into drives.
- *move* requests transfer of a cartridge from one physical slot into another.
- *openPort* instructs the LCP to open the library door, so that cartridges can be added to or removed from the library.
- *reset* instructs the LCP to reinitialize its library.
- *scan* has the LCP ask its library to verify physical labels of cartridges in the library.
- *show* obtains the current value of an attribute.
- *unmount* tells the LCP to take cartridges out of drives.

Here is a list of ALI/R commands organized alphabetically:

- *attribute* sets and unsets named attributes in the OpenVault database.
- *cancel* prevents execution of a command that has been queued but not yet started.
- *config* copies information (such as slot state) from the LCP to the MLM server.
- *goodbye* asks MLM to end this session (vice versa for ALI).
- *message* sends a message of a specified severity level to an operator or logfile.
- *ready* tells the MLM server about library status for cartridge operations.

- *response* indicates success or failure of an ALI command, and returns results.
- *show* obtains values of attributes stored in the OpenVault database.

The *OpenVault Infrastructure Programming Guide* describes the ALI and ALI/R languages, and offers an introduction to creating library control programs.

Abstract Drive Interface (ADI)

A drive control program (DCP) manages the configuration of drives, and performs the drive control tasks associated with CAPI mount and unmount requests. There is at least one DCP associated with each MLM-managed drive. The purpose of DCP is to expose the drive configuration to the MLM server, and to control drives as requested.

The MLM server issues directives to the DCP in a language called ADI. The DCP replies to the MLM server in a language called ADI response (ADI/R).

ADI/R implements a different command set from ADI, reflecting different needs of a DCP and the MLM server. The ADI language is primarily a drive control interface, whereas the ADI/R language constitutes a status reporting interface with support for administration and configuration. Like CAPI, ADI and ADI/R are semi-asynchronous

If you are developing a drive control program, your program must be able to read ADI from, and write ADI/R to, the MLM server. The OpenVault infrastructure developer's kit includes an ADI parser and ADI/R generator. The parser and generator, as well as the communications layer, are delivered with a C language interface.

Here is a list of ADI commands organized alphabetically:

- *activate disable* forces the DCP to store persistent state and stop communicating with its hardware.
- *activate enable* forces the DCP to resynchronize with its drive hardware, ensuring that the DCP has the current drive state.
- *attach* selects the appropriate access method, and binds it to a drive handle.
- *attribute* sets and unsets named attributes in the DCP.
- *barrier* tells the DCP to complete all asynchronous commands before continuing.
- *cancel* requests the DCP to stop execution of a command, if possible.
- *detach* removes the access method binding created by an *attach* command.

- *exit* tells the DCP to store state information, clean up, and exit.
- *load* pushes a cartridge into the drive and engages media at the media access point (read/write head), or verifies that the drive is loaded.
- *reset* instructs the DCP to attempt drive reinitialization.
- *show* asks the DCP to return state or configuration information.
- *unload* rewinds if necessary, disengages media from the media access point, and returns media to its cartridge.

Here is a list of ADI/R commands organized alphabetically:

- *attribute* stores persistent state in the OpenVault database.
- *cancel* tells OpenVault to prevent execution of a particular command, if possible.
- *config* tells OpenVault about access modes, form factors, and media formats.
- *goodbye* asks MLM to end this session (vice versa for ADI).
- *message* sends a message of some severity level to an operator or logfile.
- *ready* informs OpenVault of the status of the DCP's connection to the drive.
- *response* indicates success or failure of an ADI command, and returns results.
- *show* queries persistent state stored in the OpenVault database.

The *OpenVault Infrastructure Programming Guide* describes the ADI and ADI/R languages, and offers an introduction to creating drive control programs.

Administrative Commands

OpenVault can be administered with commands given from the system prompt. Most of these commands cause MLM to forward library or drive requests to a particular LCP or DCP. Most OpenVault commands produce helpful usage messages when invoked with the wrong syntax or with the **-help** option. For a list of OpenVault commands, type:

```
man -k ov_
```

The user mount shell, *umsh*, is a system command that provides user and administrator access to OpenVault volumes. See the umsh(1M) reference page for details.

Chapter 2

Common Implementation Issues

This chapter presents information you must know before implementing an LCP or DCP. Please read these sections whether you are implementing an LCP, a DCP, or both:

- "Booting OpenVault Components" shows how OpenVault starts its modules.
- "Persistent Storage" on page 12 tells how OpenVault tracks information.
- "Communication Protocols" on page 12 describes how the modules communicate.

Booting OpenVault Components

Because it is composed of different modules working together, OpenVault booting is critical for correct operation. This section describes how OpenVault assembles itself, either at system boot time or when recovering from partial failure of the system.

The MLM server initiates a sequence to bootstrap a functioning OpenVault system. Each component boots independently, reading its own configuration file, which contains just enough information to initialize that particular component. Remaining information is derived from the state of a device, persistent storage, or from parameters compiled into a particular component. Configuration files vary greatly from component to component. The session initiation sequence is the same for all components, and allows a component to identify itself by name, type, and the language versions that it supports.

MLM Server Booting

The MLM server should be the first component to initialize itself. If the MLM server reboots, all LCP and DCP connections to it are lost. The MLM server must:

1. Read its configuration file.

   The LCP or DCP developer does not need to be concerned about this file.

2. Accept connections from booting DCPs and LCPs.

The communications layer establishes TCP *keepalive* sockets. If the connection is lost, the MLM server tries to re-establish the connection every two minutes.

3. Service other client connections and AAPI or CAPI requests.

   The MLM server accepts client connections as they arrive. AAPI and CAPI requests are fulfilled if the resources needed to service them are available.

LCP and DCP Booting

Each LCP and DCP must also initialize itself. For details on LCP booting, see "LCP Booting" on page 38. For details on DCP booting, see "DCP Booting" on page 72.

Persistent Storage

The OpenVault persistent store is implemented as a database subsystem that resides in the MLM server. This is a multiuser, in-memory relational database subsystem whose clients are the modules that make up core OpenVault services. Each OpenVault module is linked with a C library to handle

- constructing queries and other data update operations
- assembling and disassembling the data update structures

One important OpenVault process is the Catalog Manager, which handles database startup and recovery, manages the on-disk transactional log file, and takes periodic snapshots of the database.

The LCP or DCP developer does not need to be concerned about details of the OpenVault database. The MLM server handles database operations triggered by LCP and DCP events or by CAPI requests from client applications transparently. LCPs and DCPs interact with the persistent store through the ALI/R or ADI/R language.

Communication Protocols

The OpenVault interfaces ALI, ADI, CAPI, and AAPI are based on message passing. Only ASCII strings travel across the sockets. OpenVault client and control program processes communicate with the MLM server through TCP/IP sockets. The *hello-welcome* sequence establishes an IPC connection based on a TCP socket.

Once an IPC connection has been established, the entity at either end of the connection may send and receive commands compatible with the negotiated language and version. The sender of a command generates a unique task ID for that command. The task ID is used in subsequent responses to that command. The sender may also use the task ID to cancel the original command or check command status.

Version Negotiation Language

To allow partial upgrades and peaceful coexistence of different language versions, OpenVault includes a session initiation facility to negotiate language version. When connecting to the MLM server, a client or control program announces which language it uses, and which versions of the language it understands. The MLM server then selects one version and tells the client which one to use for the current session.

hello — A client or control program uses the *hello* command to announce itself to the MLM server. The client includes in that command the name of the language it would like to speak, a list of the different language version numbers it supports, a name for itself as an application, and a name for a particular instance of that application. An LCP or DCP should use the OpenVault name of the device it controls as its application name.

welcome — After the client announcement, the MLM server responds with a *welcome* command, telling the client which version to use. This version is one that the client enumerated in the *hello* command. At this point, a session is established between the client and MLM server, implemented by an underlying TCP/IP connection.

unwelcome — The *unwelcome* command tells the client that none of the combinations of language and language version it provided are supported by this MLM server. After the external client has announced itself to the MLM server, the server may respond with an *unwelcome* command if the language name is unknown, or if none of the language versions supported by the client are supported by the server.

LCP and DCP programmers working in the C language can use a library routine that encapsulates the *hello* and *welcome* exchange to establish a session. For an LCP, version negotiation is built into the **ALIR_initiate_session()** function. For a DCP, version negotiation is built into the **ADIR_initiate_session()** function.

The OpenVault session is demarcated by version negotiation (*hello* and *welcome*) at the beginning, and close of session (*goodbye*) at the end.

Authentication Requests

Before a session can be established between the initiator and its recipient, authentication is needed. OpenVault employs public key session verification to provide a modicum of security while still avoiding export restrictions.

As an example, assume that Alice represents the client that initiates communication with the MLM server (the client could be a DCP, LCP, or client application). Bob represents the MLM server. The authentication process begins with Alice sending her name to Bob. Bob replies by generating a 32-bit random number (R1) and sending it to Alice as a challenge. Upon receiving this number, Alice encrypts it with the key she shares with Bob and sends this value, along with another 32-bit random number she has generated herself (R2) to Bob. After checking to make sure that Alice has successfully encrypted R1, Bob then encrypts R2 and generates a third random number (R3). Bob now sends the encrypted R2 and R3 to Alice. Alice verifies that R2 has been properly encrypted and then decrypts R3 and stores it as the session key.

Infrastructure developers do not need to be concerned about details of the OpenVault authentication method. The OpenVault transport layer handles authentication requests from client applications transparently.

Command Phases

A communication session between the MLM server and a client or control program employs a stylized sequence of phases. Since the interface is a full-duplex bidirectional peer-to-peer interface, this applies to both directions of a session. The phases are:

command — In this phase, the sender transmits the text of the command, plus a task ID it assigns to the command, to help track responses.

ack — The receiver sends back an intermediate response indicating that it accepted a command with the given task ID. The receiver may send back an *unacceptable* response if the command was incorrectly constructed, in which case there is no data phase. The sender cannot transmit another command until it receives an accepted or unaccepted response.

data — The receiver of the command sends back a final response, including the task ID, so as to identify the original command, a return value, which could be an indication of success or failure, and possibly some data.

Associated ALI/R or ADI/R commands may intervene between transmission of a command and receipt of the corresponding final response.

Since sessions are full-duplex, each endpoint must be prepared both to read and write on a session without blocking for either. For example, if the LCP is sending but the MLM server is not responding and its buffers are full, the LCP must still be prepared to accept incoming data from the MLM server. The only permitted blocking I/O operation is a **select()** call. This requirement helps reduce the likelihood of deadlocks.

Protocol Layers

Figure 2-1 shows OpenVault communication layers, which are described in this section.

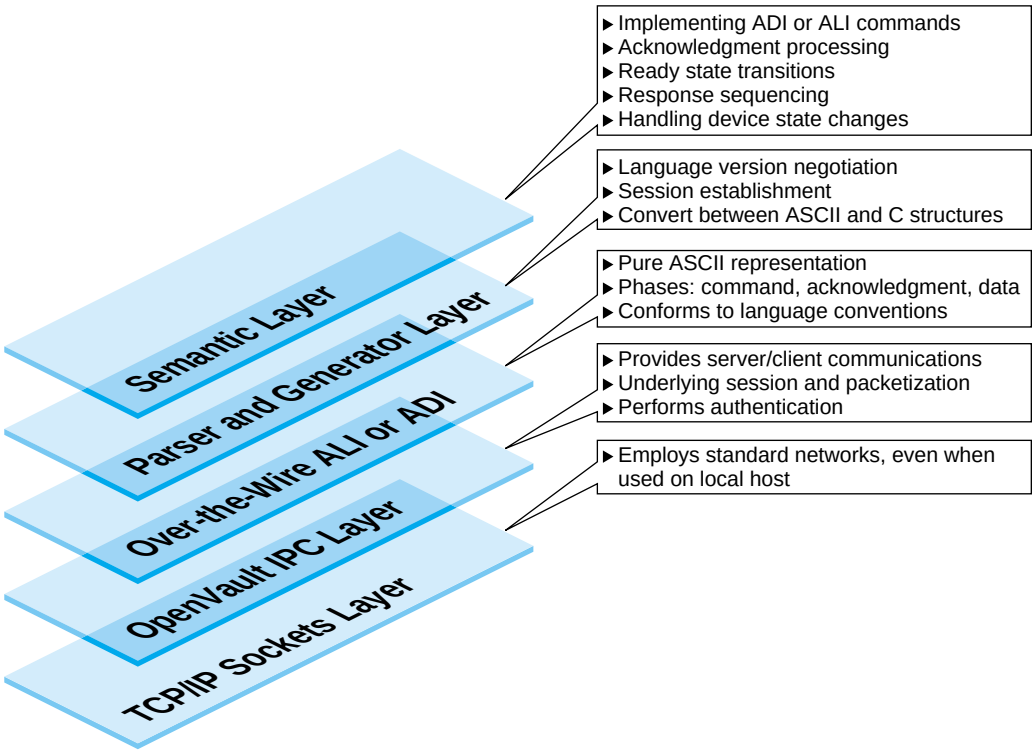

Figure 2-1 Communication Layers

The function of the semantic layer is the same for both ALI and ADI. It is responsible for

- implementation of ALI and ADI commands
- *ack* processing—synchronizing commands by ensuring that a command is not sent until an acknowledgment is received for the previous command
- ready state processing—see “Ready State Transition Rules” on page 95
- response sequencing

  If an ALI or ADI command results in ALI/R or ADI/R commands being sent, in addition to the normal ALI/R or ADI/R responses for acknowledgment and final response, the intervening ALI/R or ADI/R commands should be sent in between the *ack* and final responses. For example, an *activate enable* command to a DCP usually results in the series ADIR_reponse for acknowledgment, ADIR_config, ADIR_ready, and finally ADIR_response for final response.
- detection and handling of device state changes

  This can range from full asynchronous notification by a device or device controller to a control program (no examples at this time) to periodic polling of a device by the control program to detect changes. With SCSI, the device raises a *unit attention* condition, and sends a *unit attention* notification piggy-backed on a response from the SCSI device, which indicates that some device state has changed. The control program can then send additional SCSI commands to determine what state has changed, and to clear the *unit attention* condition.

  When the control program detects state changes that affect the control program’s ready state or configuration from the MLM server’s point of view (for example, the library may have gone offline, or the library contents may have been altered if the library front door was detected to be opened and then closed), then the control program should update ready state and configuration information, as appropriate, and push the new ready state and configuration up to the MLM server.

The parser and generator layer uses the POSIX compliant GNU utilities Bison and Flex, and is responsible for

- language version negotiation and session establishment

  The source files involved are *ovsrc/include/hello.h* and *ovsrc/libs/hellor/**.
- converting commands between C data structures and ASCII representations

  The ALI source files involved are *ovsrc/include/{ali,lcp}.h* and *ovsrc/libs/ali/**.

  The ADI source files involved are *ovsrc/include/{adi,dcp}.h* and *ovsrc/libs/adi/**.

The over-the-wire ALI and ALI/R or ADI and ADI/R layer employs nothing but ASCII strings, and is responsible for

- transitioning between command phases (*command*, *ack*, *data*)
- conforming to language conventions (the parser enforces this)

The OpenVault IPC layer is responsible for

- providing OpenVault interprocess communication between clients and the server
- implementing underlying session connections for OpenVault processes, including the packetization of over-the-wire ASCII commands
- authentication

The TCP/IP socket layer employs standard networks to aid portability.

Language Conventions

All commands are designed so that the basic arguments of the command may be entered in any order. For example, these two commands are equivalent:

```
mount slot["#12", "vol.001", "sideA"] drive["DLT2"];
mount drive["DLT2"] slot["#12", "vol.001", "sideA"];
```

OpenVault strings are composed of ASCII characters in the range 32 to 126 (decimal). Strings must be quoted with either a double-quote or single-quote (“ or ‘). OpenVault considers these different quote characters to be identical. To include either quote character in a string, precede it with backslash (\). To include a single backslash character in a string, put two backslash characters in a row.

For example:

```
"This string contains a backslash \\ and a double quote \" character."
```

Potential return value types depend on the command issued. In general, when a command is successful, the return value specification is the following:

`response success` *text* [*retValue(s)*]

When a command is unsuccessful, the error return value conforms to the following specification:

`response error` *errorSpec*

Convenience Routines for Developers

The following modules are provided in the source code tree as an aid to LCP and DCP developers:

- a generic linked list queue in *ovsrc/include/queue.h*
- a command queuing facility and state machine in *ovsrc/include/cctxt.h* and *ovsrc/libs/common/cctxt.c*
- shared LCP or DCP data structures and functions in *ovsrc/include/[ld]cp_lib.h* and *ovsrc/[ld]cp/common/util.c*

  These are intended to provide a basic framework for developing DCPs and LCPs, and also reusable software for common control program operations such as *ack*, *attribute*, error, and ready-state processing. This framework will evolve.

LCP and DCP templates will be provided, which developers can use to start coding.

Conformance Suites

An LCP conformance suite is in *ovsrc/clients/conformance/lcp*, and a DCP conformance suite is in *ovsrc/clients/conformance/dcp*. Developers should test each LCP and DCP against a conformance suite to assure compliance with OpenVault specifications. Although there is no formal LCP or DCP certification program, this is the next best thing.

Each conformance suite simulates the MLM server's interaction with an LCP or a DCP, and attempts to find certain logical errors in a control program, such as allowing ejection from an empty slot or unloading of an empty drive. See the respective *README* files for specific information about running an LCP or DCP conformance suite.

Chapter 3

Abstract Library Interface (ALI) Language

This chapter provides programmers with an introduction to the OpenVault languages for controlling removable media libraries, and includes the following sections:

- "Abstract Library Interface—ALI" describes the language in which the MLM server sends directives to an LCP, and responds to requests sent by an LCP.
- "ALI Response—ALI/R" on page 30 tells how an LCP sends configuration and status to the MLM server, and responds to directives from the MLM server.

Abstract Library Interface—ALI

The following sections describe the abstract library interface (ALI), including objects, object attributes, naming conventions, and the ALI command repertoire.

About ALI

ALI is a language that provides an abstraction of a removable media library that is managed by OpenVault. ALI hides details of the underlying library and control methods without compromising the ability of OpenVault as a whole to manage its resources effectively. The MLM server communicates with an LCP using the ALI.

ALI Object Definitions

The ALI language manipulates the following objects:

library control program (LCP)
: Each LCP knows the details of a removable media library, including its configuration and control procedures. An LCP is responsible for accomplishing tasks that the MLM server asks it to perform, primarily managing library resources. An LCP communicates with its library using some device-specific language.

An LCP can be seen as a black-box language translator, or a device management module. See Chapter 4, "Programming a Library Control Program (LCP)" for details about writing an LCP.

removable media library
: A library contains one or more housing units, called bays, for storing cartridges. Bays contain storage locations for cartridges, optional attached drives, and one or more transfer agents for moving cartridges between storage locations in the same or different bay (using the *move* command), or between storage locations and drives in the same or a different bay (using the *mount* and *unmount* commands).

  A library provides some way to read or verify external labels affixed to cartridges. A removable media library also provides some means for inserting cartridges into and removing cartridges from the library.

  Each library has a specific control method. For automated libraries, this is typically some physical control connection from a host. For a human operated library, this might be a connection to an operator console.

  Typically, a library is a single automated device, with some sort of robotic transfer agent to move cartridges between storage locations and drives. Larger devices may include a number of bays attached with pass-through ports. A human operated vault, where tapes are stored on racks and transported between racks and drives by people, is another type of removable media library.

cartridge
: A physical container for storage media. Each cartridge in the OpenVault system should have some kind of external identifying label (a physical cartridge label) that the library or an operator can verify. Part of the external label should be human readable. For automated libraries, another part of the label is machine readable—typically a barcode label that a laser scanner can interpret.

  Cartridges can have multiple sides. If they do, their containing library should be able to move or mount cartridges to achieve a particular orientation, for example, "side A" up.

bay
: A location for cartridges, with locality determined by similar access (mount) time. Typically, a bay is a physical grouping of cartridges in a common unit of housing, where cartridges are stored. A bay contains storage locations for cartridges, optional drives, and one or more transfer agents to move cartridges between storage locations and drives or other storage locations.

In a multibay library, each bay in the library is attached to at least one other bay in the same library. For each cartridge in the library, there is some path for moving that cartridge from its current bay to any other bay, with one or more transfer agents to move that cartridge.

slot
: A storage location for a cartridge. It has a shape, or form factor, that determines which kinds of cartridges it can hold.

drive
: A device for accessing media inside a cartridge that has been mounted.

port
: A door or opening where cartridges may be inserted into or removed from the library.

command
: ALI commands are objects as far as ALI is concerned. When the MLM server sends an ALI command, it associates a task ID with that command. The sender may refer to that command later by using the same task ID, but only to cancel the command. When an LCP receives a command, it includes the task ID in command responses.

Attributes and Object Properties

OpenVault requires an LCP to maintain library configuration attributes and notify the MLM server when they change. LCPs use the ALI/R *config* and *ready* commands to do this. These commands send properties back to the MLM server, where configuration information is kept in the MLM server persistent store. It is potentially recoverable by the LCP using the ALI/R *show* command. Here are the required configuration attributes:

- LCP ready state (see “Ready States” on page 94)
- library nominal cartridge exchange time (see Table 4-3)
- element maps for slot, bay, and drive (see “Element Maps” on page 22)
- cartridge form factor associated with slots, ports, and drives
- number of free slots in each bay, by form factor

Note: Currently, OpenVault does not support recovery of any attribute or property information stored in the MLM server persistent store by an LCP. However, this may be supported in a future version of OpenVault.

arbitrary attributes
: These are LCP private attributes. Developers may devise arbitrary attributes, and store them to and recover them from the MLM server persistent store. These attributes are opaque to the MLM server.

mandatory attributes
: These are attributes that an LCP is required to support. Developers may store the *logLevel* mandatory attribute in the MLM server persistent store, so the LCP can recover it and resume logging at the same level across reboots.

ALI expresses LCP attributes using the tuple: *object type*, *object name*, *attribute name*. Table 3-1 shows the mandatory attributes, not including the configuration attributes.

Table 3-1 Mandatory LCP Attributes

Object Type	Object Name	Attribute Name	Command
LCP	""	name	ALI show
LCP	""	supportPCLs	ALI show
LCP	""	vendor	ALI show
LCP	""	logLevel	ALI show, ALI attribute set

Element Maps

Element maps are kept in the OpenVault persistent store and refreshed by the LCP when appropriate. There are element maps for the following objects:

baymap
: A list of bays in the library, with information on whether each bay is accessible or not.

slotmap
: The slotmap is an array of elements, one per slot, provided by the LCP to help the MLM server operate and administer the library, including:

- physical cartridge label (PCL); for instance, a barcode
- *bayID* for the bay the slot is in
- *slotID* for the name of the slot
- *formFactor* of the slot
- whether a slot is full or empty (PCL is NULL if a slot is empty)
- slot accessibility information (PCL is NULL if this is false)

drivemap
: A list of drives in the library, with information on whether there is a cartridge in each drive, and whether it is accessible.

Table 3-3 shows element map objects that an LCP supports.

Table 3-2 Element Map Components

Object Type	Object Name	Attribute Name	Command
bay	*bayID*	description	ALI show
slot	*slotID*	slot description	ALI show
drive	*driveID*	description	ALI show

ALI Object Naming

These names refer to specific ALI objects:

LCP name — Each LCP is uniquely named by a value pair including an OpenVault client name and an OpenVault instance name.

client name — The OpenVault client name refers to a specific removable media library. This is the name by which a client identifies itself in a *hello* command to the MLM server. For ALI clients, this is name that the MLM server associates with the library that is managed by the associated LCP.

instance name — The OpenVault instance name is arbitrary, but is needed in case there are multiple LCPs controlling the same library, so as to distinguish between LCPs with the same client (library) name.

PCL — A physical cartridge label (PCL) refers to a cartridge. It is some form of identification on the outside of the cartridge, as opposed to being stored on media inside the cartridge. A PCL may contain a machine-readable label (barcode), but it must also contain a human-readable text portion.

bay ID — A text string provided by the LCP, which refers to a bay in the library. An LCP should choose bay IDs that are easy for a human operator to interpret. For multibay libraries, the bay ID is usually consistent with the device name or address for a bay.

slot ID — A text string provided by the LCP, which refers to a slot in the library. The slot ID must uniquely identify any slot under control of that LCP, and should be easy for a human operator to interpret. For libraries with explicit slot locations, slot ID is usually consistent with the device name or address for that slot.

drive name — Refers to an OpenVault removable media device.

port name — Currently, there is no ALI support for port names. Port naming may be supported in a future version of OpenVault.

task ID — Uniquely identifies a sender-generated command.

Attribute naming in ALI is different than for CAPI and AAPI, in which an attribute is given as *TableName.ColumnName*; attributes are just columns in a relational table. In ALI and ALI/R, attributes are named with a tuple:

objectType, objectName, attrName

ALI Commands

The MLM server speaks ALI to the LCP, which in turn speaks ALI/R to the MLM server. The ALI language includes the following commands:

activate — The *activate* command and its variations are used to start and stop LCP interactions with the library. Note that once the LCP has established a session with the MLM server using the *hello-welcome* sequence, it may begin accepting ALI commands from the server. However, until it has successfully been *activate enabled* and is in ready state, it will resend *ready lost* state and fail ALI commands requiring access to its library with the error ALI_E_READY. The LCP uses one of the ALI/R *ready* command variations after processing the current command.

These are the variations of the *activate* command:

activate enable — The *activate enable* command forces the LCP to resynchronize its internal information with the physical state of the library, and keep it synchronized. For example, with a SCSI-based sighted robot, the LCP could do a barcode inventory and resume status polling.

Performing this command will probably result in the LCP modifying slotmap information in the MLM server for this library, pushing the slotmap to the MLM server using *config*, and possibly accessing LCP-private attributes stored in the MLM database. The LCP reports *ready* when all its internal resynchronization operations have completed (for example, when the barcode scan is done).

activate disable — The *activate disable* command forces the LCP to stop talking to the library. For example, on a SCSI-based robot the LCP may be in the habit of polling the device for status changes; this command would stop that polling.

An *activate disable* should complete or cancel ALI commands that require access to the library, and store any persistent library state in the MLM server.

The LCP requires an *activate enable* command before it can talk to the library again.

The LCP reports *ready lost* when all its state update operations have completed and any internal machinery has been shut down. Performing this command may not result in the library's cartridges becoming inaccessible if there is an alternate LCP and the library is connected to multiple hosts.

attribute — The *attribute* command sets and unsets named attributes in the LCP. You can think of attributes in an LCP as named memory locations that may cause operations to happen as a side effect of setting or reading them. Some attributes defined by an LCP may be read-only to the MLM server.

A list of mandatory attribute names appears in Table 3-1.

barrier — The *barrier* command forces the LCP to complete work on all commands received prior to the *barrier* command, before it begins working on any commands that might follow. This may require special processing for queued *eject* and *openPort* commands. For example, if an LCP normally flushes ejects with the *openPort* command, *barrier* should be rejected if the LCP has not already received an *openPort* command.

In general, an LCP is free to execute the commands it receives in any convenient order. Since there might be circumstances where the MLM server requires an explicit order for executing a sequence of commands, the *barrier* command can be employed to force ordering.

cancel — The *cancel* command prevents execution of a command that has been queued in the LCP but which the device has not yet started. The LCP may choose to cancel already started jobs on a best-effort basis.

Note: The *cancel* and *response* commands may not be cancelled.

eject

The *eject* command, in conjunction with the *openPort* command, pushes cartridges out of the library. It takes a (slot ID, PCL) pair for the cartridge that is to be operated on. The LCP should send the corresponding changes in its slot and drive maps to the MLM server.

The implementation of the *eject* command may vary from LCP to LCP, but there are three basic cases, as listed in Table 3-3.

Table 3-3 Three Cases of Eject

Operator Interaction Required	LCP Becomes not ready	Likely Semantics and Effect
No	No	The *eject* command causes the given cartridge to be immediately pushed out of the library. The *openPort* command is a successful no-op. The library continues operation uninterrupted. The ATL2640 is one example of a library in this class. It has a bin where exported cartridges simply pile up. No operator interaction with the LCP is required.
Yes	No	The *eject* command causes the given cartridge to be marked as needing to be pushed out of the library, but the cartridge is not yet pushed out. The LCP is free to move the cartridge if it needs to. An *openPort* command tells the LCP that the operator is ready to physically take the cartridges out of the library. The library continues operation uninterrupted. A StorageTek silo is an example of this library type. The silo has a port with slots on the inside where the LCP can move cartridges when they are ejected. The *openPort* command unlocks access port(s) and allows the operator to remove the cartridge(s).

Table 3-3 (continued) Three Cases of Eject

Operator Interaction Required	LCP Becomes not ready	Likely Semantics and Effect
Yes	Yes	The *eject* command causes the given cartridge to be marked as needing to be pushed out of the library, but the cartridge is not yet pushed out. The LCP is free to move the cartridge if it needs to. An *openPort* command tells the LCP to put the library into the "ready not" state and prepare it to allow the operator easy access to those cartridges marked for ejection. The Exabyte 210 is an example of this type of library. It must be taken offline to physically remove cartridge(s). The *openPort* command puts the library into *ready not* state and unlocks the access door, allowing the operator to remove ejected cartridge(s). When an LCP determines that the access door has been opened and closed, it should lock the door, reinventory the library, complete affected ejects, inform the MLM server of slotmap changes, and transition to *ready* state.

When a cartridge is physically ejected, it must immediately disappear from the OpenVault slotmap maintained by the LCP. This implies that an LCP that cannot immediately push a cartridge out of the library must be prepared to inform OpenVault that a particular slot ID (and therefore PCL) has been marked for ejection. The LCP should mark this slot as inaccessible and push the information to the MLM server.

The LCP should recall this information from the OpenVault database upon booting, when OpenVault supports retrieval of LCP attributes from the MLM server's persistent storage.

exit — The *exit* command tells the LCP to clean up and exit.

The LCP should store any persistent LCP or library information in the OpenVault database, complete or cancel any pending ALI commands, send or abort any pending ALI/R commands, do shutdown processing as required by its interface to the library, send *ready lost* and *goodbye* commands to the MLM server, and exit.

goodbye — The *goodbye* command tells the communicating LCP to end this session.

mount

The *mount* command places a cartridge into a drive. The arguments to mount are a list of tuples (slot ID, PCL, side) and a drive name. The operation involves taking a cartridge from one of the given slots and putting it into the specified drive. For multisided cartridges, placement is according to a specified side orientation, for example "side A" up. The slot list may have just one element; if more than one is specified, the LCP decides which slot to use. The LCP should send the corresponding changes in its slot and drive maps to the MLM server.

Note: Multisided cartridges are not supported in OpenVault version 1.

move

The *move* command transfers a cartridge from one physical slot in the library to another physical slot. The move source is a slot ID, PCP pair, and the destination is a slot ID. The LCP should send the corresponding changes in its slot and drive maps to the MLM server.

openPort

The *openPort* command, in conjunction with the *eject* command, removes or allows cartridges to be removed from the library. It may also be used on its own to allow cartridges to be inserted into the library. The function of the *openPort* command is to prepare the library for an operator to gain physical access to cartridges. Once access is granted, cartridges may be removed from and inserted into the library. The implementation of *openPort* may vary from LCP to LCP, and a given library might be in a different class for export than for import, but there are three basic cases, as listed in Table 3-4.

Table 3-4 Three Cases of OpenPort

Operator Interaction Required	LCP Becomes not ready	Likely Semantics and Effect
No	No	New cartridges are simply inserted into the library. For example, the ATL2640 has a cartridge insert door and a request button next to it. Pressing the request button is all that is required to prepare the library to accept a new cartridge.
Yes	No	The LCP must prepare to accept a new cartridge. For example, the StorageTek silo may be told to unlock port(s) so that the operator can add new cartridges.

Table 3-4 (continued) Three Cases of OpenPort

Operator Interaction Required	LCP Becomes not ready	Likely Semantics and Effect
Yes	Yes	The LCP unlocks the library door and puts the library into *ready not* state when it detects a door open. When it detects the door is closed again, it reexamines cartridge inventory to see what has been added or removed, and returns the library to *ready* state. For example, the EXABYTE-210 must have its main door unlocked before the operator can add cartridges.

See the description of the *ready* and *ready not* commands under ALI/R for more information on how an LCP becomes not ready, permitting its library to be temporarily not available during an *openPort* operation.

reset — A *reset* command asks the LCP to try and force the library to reinitialize. This may cause the library to perform internal diagnostics.

If a reset makes the library unavailable to process other requests for an extended time, the LCP should use the *ready not* command to tell the MLM server that its library is temporarily not available, followed by a *ready* command when the library becomes available again.

response — The *response* command acknowledges and indicates success or failure of an ALI/R command. The optional text portion of the response contains error details or command results.

scan — The *scan* command forces the LCP to verify or recheck the PCLs of all cartridges in the library. These are variations of the *scan* command:

- scan all — The LCP should rescan the entire contents of the library in order to resynchronize its internal information with the physical state of the library. It should send changes in content information to the MLM server.
- scan from to — The LCP should rescan all slots represented by slot IDs lexicographically between the *from* slot and the *to* slot. The LCP may rescan more slots than listed for some implementation dependent reason. It should send changes in content information to the MLM server.

If the library will be unavailable to process other requests during this time, the LCP should use the *ready not* command to tell the MLM server that the library is temporarily not available for other motion commands (such as *mount*, *unmount*, *move*, or *eject*), followed by a *ready* command when the library becomes available again.

show — The *show* command obtains the current value of an attribute. Some of the attributes defined by an LCP may be write-only to the MLM server.

For more information about LCP attributes, see "Attributes and Object Properties" on page 31.

unmount — The *unmount* command takes a cartridge out of a drive and returns it to a slot. The arguments to unmount are a drive name and a slot ID. The operation involves taking the cartridge from the drive and putting it into the given slot. Optionally, you can specify "any" for slot ID, and let the LCP choose where to return a cartridge. The LCP should send the corresponding changes in its slot and drive maps to the MLM server.

ALI Response—ALI/R

The following sections describe the ALI/R language, including objects, object attributes, naming conventions, and the ALI/R command repertoire.

About ALI/R

ALI/R is primarily the response language for ALI. In addition to giving the matching acknowledgment and final response to an ALI command, ALI/R provides the means for an LCP to send its configuration and status to the MLM server.

ALI/R Object Definitions

The ALI/R language manipulates the following objects:

command — ALI/R commands are objects as far as ALI/R is concerned. When an LCP sends an ALI/R command, it associates a task ID with that command. The sender may refer to that command later by using the same task ID, but only to cancel the command. When the MLM server receives a command, it includes the task ID in command responses.

message — A text message to be entered into an MLM server-managed log, and perhaps displayed on some console by the MLM server, or one of its administrative applications.

Messages are associated with a severity level, or a level of urgency, which determines (along with site policy) whether the message text is stored in the MLM server logs, displayed on a library or OpenVault console for the operator, or both.

Attributes and Object Properties

Currently, ALI/R attributes are not supported by OpenVault, except for attributes stored by the ALI/R *config* and *ready* commands in the MLM server persistent store. Currently, OpenVault supports setting and unsetting of *config* and *ready* attributes only.

ALI/R Object Naming

These names refer to specific ALI/R objects:

message ID — Refers to a text message of a given severity level.

task ID — Uniquely identifies a sender-generated command.

ALI/R Command Descriptions

The LCP reads ALI commands from the MLM server, and replies to the server in ALI/R. The ALI/R language includes the following commands:

attribute — The *attribute* command sets and unsets named attributes in the MLM server, thereby creating persistent storage for whatever the LCP deems necessary. The MLM server simply stores these attributes; there are never any side effects of setting them. For background, see "Attributes and Object Properties" on page 21.

cancel — The *cancel* command prevents execution of a command that has already been queued in the MLM server but not yet started. The cancelled command returns *response cancelled* status, and the response for the *cancel* command itself follows.

Note: The *cancel* and *response* commands may not be cancelled.

config

The *config* command copies configuration information, especially about element map changes, from the LCP to the MLM server.

The MLM server stores a non-authoritative copy of all the element map information for all the LCPs it controls. Each LCP must use the *config* command in ALI/R to update the MLM server's copy of the element map information whenever it changes. The element map should change only as a result of Administrator or Operator actions.

In the *full scope* option, all information that the MLM server associates with the LCP is deleted and replaced with information listed in the *config* command. In the *partial scope* option, the MLM server replaces only pieces of LCP information that are listed in the *config* command.

Normally, the full scope option is employed at startup and when major changes to the library configuration occur, whereas partial scope is employed when a cartridge movement operation happens. Very large libraries can initially use a partially populated full scope command followed by a series of partial scope commands, if this proves easier.

Use the *config* command to:

- Copy the list of slots to the MLM server, including information on which bay slots are in, the PCL of the cartridge in a slot, what form factor of cartridge is in (or could be in) that slot, whether the slot is occupied or not, and whether the slot is accessible or not.
- Copy the list of drives to the MLM server, including information on whether there is a cartridge in the drive (it may not have been loaded, so the DCP might not see it) and whether it is accessible.
- Copy the list of bays to the MLM server, including information on whether each bay is accessible or not. It is possible for a single bay in a multi-bay library to be inaccessible or temporarily broken.
- Copy a list and count of free slots in each form factor inside all library bays to the MLM server. Some libraries have no name for empty slots, and bays sometimes contain several form factors, so we need a count of the number of free slots of each type.
- Provide some approximate performance information to the MLM server for the library. The MLM server may use that information when choosing which library to use. For example, a library with an expected cartridge mount time of 10 seconds may be preferable over one with an expected mount time of 24 hours.

goodbye — The *goodbye* command tells the MLM server to end this session and clean up its end of the session. This protects against the accumulation of idle connections, since the MLM server has no way of detecting that an LCP exited other than the TCP/IP *keepalive* option. *Keepalive* helps recover from process failures, but an LCP should send a *goodbye* before exiting to prevent unnecessary continuation of connection resources.

message — The *message* command sends a message of some severity level to the MLM server. The LCP *logLevel* attribute determines a limit on the severity level of messages sent to the MLM server. This command provides a mechanism for the LCP to send messages that the MLM server can convey to an operator and possibly a system administrator.

Note: This mechanism may change in future releases of OpenVault.

ready — The *ready* command and its variations tell the MLM server the current status of the library, and whether it is available for cartridge operations. Like the *config* command, the *ready* command is just a shorthand way of conveying attributes about ALI objects to the MLM server.

These are variations of the *ready* command:

ready	The LCP has resynchronized its internal information with the physical state of its device, and is prepared to accept commands that require it to access its device.
ready not	The library is temporarily unavailable for motion operations, such as ALI *mount*, *unmount*, *move*, and *eject*, or ADI *load* and *unload*.
ready lost	The LCP has lost contact with its device.
ready broken	The LCP detected that its device hardware is reporting a hard failure and is nonfunctional.

See “Ready States” on page 94 for more information about ready states.

response — The *response* command acknowledges and indicates success or failure of an ALI command. The optional text portion of the response contains error details or command results.

show — The *show* command obtains the value of an attribute that the LCP previously stored in the MLM server.

Ordering of ALI Response Text

For some ALI commands, the matching ALI/R response command for a successful response contains a text portion, which must have a particular format and ordering. This section describes these requirements.

Response Text for ALI_show Command

The text portion of a successful response to show depends on the specified mode for the show, and on the number of attributes to be queried. There are three possible modes:

`ALI_show_name`	*show name only*
`ALI_show_value`	*show value only*
`ALI_show_namevalue`	*show name and value, in that order*

For each attribute to be queried, the text portion of the response includes name-value information, as dictated by this mode, and is ordered according to the specified attribute list. So, for example, if a show command requested a query of LCP *logLevel* and vendor attributes, with mode ALIR_show_namevalue, the corresponding text portion of the response would look something like this:

```
text[ 'logLevel' 'debug' 'vendor' 'EXABYTE' ]
```

Response Text for ALI_mount and ALI_unmount Commands

The text portion of a success response for mount and unmount includes the value tuple *source slotID*, *PCL*, *OpenVault drive name*. The values are not tagged with a name, and must appear in this order. The corresponding text portion of the response would look something like this for source slotID “slot 1” PCL “AB1234” drive “fred”:

```
text[ 'slot 1' 'AB1234' 'fred' ];
```

Response Text for ALI move command

The text portion of a success response for a move command includes the value tuple *source slotID*, *PCL*, *destination slotID*. The values are not tagged with a name, and must appear in this order. The corresponding text portion of the response would look something like this for source slotID “slot 2” PCL “AB5432” destination slot “slot 5”:

```
text[ 'slot 2' 'AB5432' 'slot 5' ];
```

Response Text for ALI_eject Command

The text portion of a success response for an eject command includes the value pair *slotID, PCL*. The values are not tagged with a name, and must appear in this order. The corresponding text portion of the response would look something like this for slotID "slot 10" PCL "AB9999":

```
text[ 'slot 10' 'AB9999' ];
```

Other Information

See Appendix B for a list of return values and detailed information about ready states.

Chapter 4

Programming a Library Control Program (LCP)

This chapter provides a tutorial to LCP programming, and includes the following topics:

- "Initialization Issues" on page 38 talks about starting up a control program.
- "LCP Development Framework" on page 42 describes the LCP subroutine libraries.
- "Example LCP Implementation" on page 51 discusses layout of sample source code.
- "Defined Tokens List" on page 54 presents tables of OpenVault tokens for an LCP.

About the LCP

A library control program (LCP) translates between the OpenVault ALI and the actual device control interface for its library, and between device responses and ALI/R. The LCP does what is necessary to affect the required ALI semantics. It keeps the MLM server's "cache" (persistent store) up to date regarding LCP configuration, library configuration, and ready state information. To do this, the LCP sends *config* and *ready* commands when it detects changes in state, on a best-effort basis.

Use of Persistent Storage

Currently, the library configuration and state is moved in one direction only, from an LCP to the MLM server persistent store. The MLM server uses this information to assist with library and drive selection for cartridge and volume mounts. In future revisions of OpenVault, the LCP might recover some state from the persistent store, so that state and configuration information can flow in both directions. However, the LCP and library are always considered the authoritative source for information about the LCP or its library.

LCP Configuration

In sample implementations, LCP configuration is stored in a configuration file that is local to each LCP. See "Configuration File" on page 38 for more information.

Initialization Issues

Each LCP must initialize itself in order to contact the MLM server.

LCP Booting

Removable media libraries may be connected to multiple hosts and thus have multiple control paths. There may be one LCP associated with each control path. Only one LCP at a time can be active for any library; the MLM server arbitrates which LCP is active.

For example, an LCP could be on an inactive library connection. The LCP boot sequence must not interfere with another LCP with an active connection. The MLM server is the arbitrator of control for multiconnected libraries and drives. An LCP should not assume that it controls a library until the MLM server says so.

Configuration File

Each LCP should have a configuration file containing at least the following information:

- The address of the controlling MLM server.

  This allows the LCP to initiate contact with the controlling MLM server. It is the name of the system, or its numeric IP address. The MLM server is usually available at well-known port number on that system, by default 44444.

- The OpenVault name for the managed library.

  The MLM server uses this name as an identifier for this physical library. This is the name of the device that it is managing, not the name of the particular instance of LCP. All names must be unique within an OpenVault domain so that the server can detect multiconnected libraries (multiple LCPs controlling the same library).

- The LCP instance name.

  The instance name is arbitrary, but is required for cases where there are multiconnected libraries.

- The control path to the library (for example, */dev/scsi/sc0d2l0*).

  This is how an LCP talks to the hardware. This information is not visible to the MLM server. Some library implementations are not controlled in this fashion, but all LCP implementations need something equivalent.

- The OpenVault name for the drives contained in this library.

  The MLM server uses this information to determine relationships between libraries and drives (between LCPs and DCPs). The "contained in" relationship is helpful when deciding into which drives a cartridge can be placed, based on which library contains the cartridge. Each library has some method for addressing each drive inside that library. The LCP's name-to-drive address mapping takes the form: the OpenVault drive named "dlt1" corresponds to library drive 1, while the drive named "dlt2" corresponds to library drive 2.

For easy editing, LCP configuration files should be composed of readable ASCII text.

LCP Boot Sequence

When an LCP boots or re-boots, it must:

1. Allocate internal data structures.
2. Refrain from talking to the library.

   The LCP boots into *activate disable* state, and must wait for the MLM server to tell it when to talk to the library. If the library is dual-ported with another LCP actively controlling it, that session should not be interrupted! The MLM server issues an *activate enable* command when conditions permit your LCP to control the library. If the library is single-ported, *activate enable* is issued almost immediately.
3. Read its configuration file.
4. Establish a session with the MLM server.

   The LCP sends the "hello" message upon opening the connection. In this example, *name* is the OpenVault name for the library, and *inst* is the LCP instance name. If connection fails, retry ever two minutes. The LCP blocks until it receives a *welcome* command telling it which language version to use during this session.

   `hello language["ALI"] version["1.0"] client["`*name*`"] instance["`*inst*`"];`

When the MLM server is first contacted by an LCP it will:

1. Integrate the library into its list of managed devices.

   The MLM server checks for other LCPs managing that physical library. If this LCP is the first, OpenVault allows this LCP to proceed. This sequencing implies that LCPs are given control of their associated library on a first-come-first-served basis.
2. (Eventually) issue an *activate enable* command to the LCP.

When the MLM server says to *activate enable*, the LCP must:

1. Reply to the MLM server with a *ready no* command.

   The LCP informs the MLM server that it has started to come up, but is not yet ready to accept cartridge movement commands.

2. Talk to the library to determine:

   - That the library is supported by this LCP ("ATL-2640" is supported).
   - Whether or not the library supports PCLs (barcodes), true or false.
   - List of supported cartridge form factors ("DLT"); may be compiled into the LCP.
   - Total number of slots for each formFactor.
   - Total number of used slots for each formFactor.
   - Import/export port configuration.
   - The slotmap (all the barcodes and occupancy info for the library).
   - Any other information that may be relevant to library or LCP operation.

3. Collect any state or configuration information from the MLM server.

   The LCP can store state or configuration information in the OpenVault persistent store. For example, the LCP probably needs to retrieve the *loglevel* attribute so it can resume logging only the messages that the system administrator wants logged.

4. Push all the slotmap and drive information up into the MLM server.

   The LCP owns the slotmap and therefore needs to update the MLM server's copy of the slotmap whenever required. The LCP needs to tell the MLM Core when it is ready to accept cartridge movement commands.

5. Send a *ready* command to the MLM server.

   The LCP is now ready to accept cartridge movement commands.

6. Respond *success* to the original *activate enable* command.

   This is defined to be the last step as a convenience to the MLM server, so that the server can block until it receives a response from its *activate enable* command rather than continually polling for arrival of the *ready* command.

After receiving slotmap and drive information from the LCP, the MLM server will:

1. Crosscheck the list of drives.

   The MLM server crosschecks the list of contained drive names with the list of drives controlled by known DCPs. Not all DCPs may have checked in before the LCP does. The MLM server keeps a list of known DCPs that have not yet checked in so that it can flag them as possible hardware failures.

2. Crosscheck the list of PCLs (barcodes).

   The MLM server crosschecks the LCP's list of PCLs against the previously known contents of the library, looking for new or missing cartridges. A message is sent to the system administrator and/or a logfile if any changes are detected.

3. Store all the slot and drive information in persistent store.

   The MLM server stores all the information that the LCP provided in its database. That information is the basis for choosing drives and cartridges on behalf of CAPI or AAPI clients.

When the MLM server gets a successful response to *activate enable*, it will:

1. Mark the library as being available for cartridge mounts.

   The library is ready to accept cartridge mount, unmount, and movement requests. This implies that cartridges in that library are no longer filtered out of the list of candidates for mount operations because they are not accessible to OpenVault.

Activation Sequence

When an LCP receives an *activate enable* command from the MLM server, and the LCP is in *ready lost* state, it should perform these steps:

1. Access its library to acquire or verify device-specific configuration and state.

   For example, an LCP may consult its library to determine:

   - if the library is supported by this LCP (for instance, "ATL-2640" is supported)
   - whether the library is in a usable state by this LCP
   - whether the library supports verification of PCLs (has a barcode reader)
   - supported cartridge form factors (for instance, "DLT")
   - total number of slots for each *formFactor*
   - total number of free slots for each *formFactor*

- import and export port configuration
- element maps (slot, drive, bay, port)

2. Push configuration information to the MLM server.

   For example, configuration information includes: free slots, element maps, and performance information. The LCP is responsible for updating the MLM server's copy of element maps whenever it detects a change in map information.

3. Transition to *ready* state, and push this new state to the MLM server.

While in *ready lost* state, the LCP should service the *activate* command, and any ALI commands in the session that do not require device access. The LCP should return a ready error (ALI_E_READY) and resend *ready lost* state for other ALI commands.

LCP Development Framework

The infrastructure developer's kit includes a framework for writing an LCP that helps ease the development, porting, and maintenance effort for new devices. The framework provides general processing of ALI and ALI/R commands, thus freeing the developer to focus on the idiosyncrasies of a particular device, and on developing suitable support for a new removable media library.

This section describes the general source tree layout.

OpenVault Client-Server IPC

OpenVault clients and servers communicate with a custom interprocess communication (IPC) layer. LCP modules that deal directly with ALI and ALI/R must include the following header file, and be loaded with the following C library:

ovsrc/include/ov_lib.h
: C data structures, macros, and subroutine prototypes for IPC

ovsrc/libs/comm/libov_comm.so
: C library containing IPC subroutines

ALI Parser and ALI/R Generator

The framework includes language parsers and generators. LCP modules using these facilities must include the following header files, and be linked with these C libraries:

ovsrc/include/ali.h
: Supported ALI and ALI/R language version, ALI standard errors, and C data structures for ALI and ALI/R command representation.

ovsrc/include/lcp.h
: Parser and generator subroutine prototypes.

ovsrc/include/hello.h
: C data structures for *HELLO* and *WELCOME* command representation.

ovsrc/libs/hellor/libov_hello.so
: C library that contains HELLO parser-generator subroutines.

ovsrc/libs/ali/libov_ali.so
: C library that contains ALI parser-generator subroutines.

LCP C Library Routines

The LCP(3) reference page documents the ALI and ALI/R lexical library routines that you employ when writing a LCP. Table 4-1 offers a summary of these routines.

Table 4-1 ALI and ALI/R Lexical Library Routines

Purpose of Activity	LCP Function	Short Description
Initiate session with MLM server	**ALIR_initiate_session()**	Begins session with a specific MLM server, including HELLO version negotiation
Parse ALI command from MLM server	**ALI_receive()**	Parses an ALI command and returns an ALI_command structure
Acknowledge ALI command	**ALI_acknowledge()**	Informs MLM server that the LCP received an ALI_command
Send ALI/R command to MLM server	**ALIR_alloc_cmd()**	Allocates ALIR_command structure
	ALIR_alloc_ready()	Allocates ALIR_command for *ready* command
	ALIR_alloc_message()	Allocates ALIR_command for ALIR_message
	ALIR_alloc_slotinfo()	Inserts slot map info for *config* command
	ALIR_alloc_bayinfo()	Inserts bay map info for *config* command
	ALIR_alloc_driveinfo()	Inserts drive map info for *config* command

Table 4-1 ALI and ALI/R Lexical Library Routines

Purpose of Activity	LCP Function	Short Description
Send final response for ALI command to MLM server	**ALIR_alloc_response()**	Allocates ALIR_response structure
	ALI_alloc_string()	Allocates string for response, error, data results
	ALIR_send()	Transmits ALIR_command to MLM server
	ALIR_free()	Deallocates ALIR_command structure
Free ALI command	**ALI_free()**	Deallocates ALI_command structure

LCP Common Framework

The infrastructure developer's kit includes common utility code for writing an LCP. To use this code, include the following header files, and read the following C module:

ovsrc/include/queue.h
: Generic queue and linked list implementation.

ovsrc/include/cctxt.h
: Generic command queuing mechanism.

ovsrc/include/maps.h
: Generic element map representation.

ovsrc/include/lcp_lib.h
: Generic representation of LCP and library state, generic representation of an attribute, common LCP fixed and programmable entry points, and common LCP utility subroutine prototypes.

ovsrc/clients/lcp/common/util.c
: LCP common fixed-entry points and utility subroutines.

Generic Representation of a Library—lcp_lib.h

Much of an LCP's representation of LCP and library state can be represented generically. However, the LCP developer needs a way to customize this representation for a particular library and implementation.

The LCP framework provides a private data area and programmable LCP entry points as a means for the developer to customize the LCP's representation of LCP and library state. The private data area allows the developer to maintain additional information about the LCP and library; the programmable entry points allow the developer to

customize actions associated with ALI command dispatch, deactivation (transition to *ready lost* state), graceful shutdown, and ALI/R command task ID generation. This arrangement allows the shared framework to invoke these entry points as appropriate.

Here is the framework's generic representation for a library:

```
struct libinfo
{
    /* elements from LCP config file. */
    char *client;                   /* MLM name of this library.     */
    char *instance;                 /* Client instance.              */
    char *mlmhost;                  /* MLM host.                     */
    int mlmport;                    /* MLM port.                     */
    int pollinterval;               /* seconds between library polls */
    char *addr;                     /* library control address.      */

    /* elements initiated by LCP. */
    char *type;                       /* Type of library.            */
    enum ALIR_msg_severity loglevel;  /* Log level for LCP messages  */
    enum ALIR_ready_type readystatus; /* ready, not r_, disconnected */
    int supportPCLs;                  /* 1 if barcode scanner, or 0  */
    char *vendor;                     /* Library vendor name.        */
    queue_t ALI_cmd_queue;            /* ALI command queue.          */
    queue_t ALIR_cmd_queue;           /* ALIR command queue.         */
    int waiting_for_ack;              /* 1 if waiting for ack, or 0  */
    char *taskid_for_ack;             /* TaskID of last ALIR command */

    void(*lcp_deactivate)(struct libinfo *libi);      /* deactivate  */
    void(*lcp_exit)(struct libinfo *libi, int abnormal); /* shutdown */
    void(*lcp_dispatch)(struct libinfo *libi, struct ALI_command *cmd);
    char *(*lcp_taskid)(struct libinfo *libi);  /* taskid generation */

    /* element map info, shared by do- and control-layers */
    element_map_t slotmap;              /* Slot map                  */
    element_map_t drivemap;             /* Drive map                 */
    element_map_t portmap;              /* Port map                  */
    element_map_t baymap;               /* Bay map                   */

    void *private;                      /* LCP private libary info   */
};
```

Common LCP Entry Point

An LCP that makes use of this developer framework must call the following subroutine to initialize the generic and private data areas for LCP and library information, and set the programmable LCP entry points:

```
void lcp_init(struct libinfo *libi,
        void lcp_init_private(),
        void lcp_deactivate(),
        void lcp_exit(),
        void lcp_dispatch(),
        char *lcp_taskid(),
        void slot_private(),
        void drive_private(),
        void bay_private(),
        void port_private());
```

Programmable LCP Entry Points

This entry point is called one time only from **lcp_init()**, so the *libinfo* structure does not store it. Required entry point for LCP private data area allocation and initialization:

```
void lcp_init_private(struct libinfo *libi);
```

Remaining entry points are stored in the *libinfo* structure. Required entry point for LCP private actions to *activate disable*:

```
void lcp_deactivate(struct libinfo *libi);
```

Required entry point for LCP private actions to shut down gracefully and exit:

```
void lcp_exit(struct libinfo *libi, int abnormal);
```

Required entry point for ALI command dispatch from the command state machine:

```
void lcp_dispatch(struct libinfo *libi, struct ALI_command *cmd);
```

Required entry point for LCP to generate a task ID for ALI/R commands:

```
void char *lcp_taskid(struct libinfo *libi);
```

Optional entry points for element map allocation and initialization (may be NULL):

```
void slot_private(queue_t *q, int initflag);
void drive_private(queue_t *q, int initflag);
void bay_private(queue_t *q, int initflag);
void port_private(queue_t *q, int initflag);
```

Generic Representation of Element Maps

Much of the information that an LCP needs to maintain about library elements, including slots, drives, bays, and ports, may be generically represented. However, LCP developers must be able to customize element information that the LCP maintains.

For example, typical information that an LCP needs to maintain about a slot includes the *slotID*, the device-specific address for this slot, the name of the bay in which this slot is located, whether the slot is accessible and occupied, the PCL of the cartridge that is currently occupying this slot (if any), and the name of the drive where the cartridge that was last in this slot is currently mounted (if any).

For typical slot information, the framework provides an extension to the common information by means of an LCP private data area and programmable entry points for allocating and deallocating this data area.

An example of how an LCP might use its private slot data area is for multi-sided media, where the library can mount the cartridge either “side A” up, or “side B” up. In addition to the typical slot information, an LCP for such a library would probably maintain the current orientation of a cartridge in its private data area for that slot.

The element map header file, *maps.h*, is separated from the LCP common header file, *lcp_lib.h*, so that the generic element map representation and subroutines may be used separately from the generic library piece. In the sample implementations, this permits the ALI semantic layer and the control layer modules for an LCP to share the element map representation, without both having to include the generic library piece. The control layer needs the generic element map piece, but not the generic library piece.

Here are the common representations for slot, drive, bay, and port:

```
typedef struct slot {
    char       *name;        /* Slot id.                          */
    char       *addr;        /* Hardware address.                 */
    char       *bayid;       /* Name of bay where slot resides    */
    char       *shape;       /* Of cartridges fitting this slot   */
    int        access;       /* T/F: is slot accessible?          */
    int        occupied;     /* T/F: is slot occupied?            */
    char       *PCL;         /* Label of cartridge in slot, if any  */
    char       *driveid;     /* Drive with slot's cartridge, if any */
    void       *private;     /* LCP private data area.            */
    queue_t    queue;        /* To next/prev slots.               */
} slot_t;
```

```
typedef struct drive {
    char       *name;        /* Name of drive.                       */
    char       *addr;        /* Hardware address.                    */
    char       *bayid;       /* Name of bay where drive resides      */
    char       *shape;       /* Of cartridges that fit in this drive */
    int        access;       /* T/F: is drive accessible?            */
    int        occupied;     /* T/F: is drive occupied?              */
    char       *PCL;         /* Label of cartridge in drive, if any  */
    char       *slotid;      /* Slot from where cartridge mounted    */
    void       *private;     /* LCP private data area                */
    queue_t    queue;        /* To next/prev drives                  */
} drive_t;

typedef struct bay {
    char       *name;        /* Name of bay                          */
    char       *addr;        /* Hardware address                     */
    int        access;       /* T/F: is bay accessible?              */
    void       *private;     /* LCP private data area                */
    queue_t    queue;        /* To next/prev bays                    */
} bay_t;

typedef struct port {
    char       *name;        /* Name of port                         */
    char       *addr;        /* Hardware address                     */
    char       *bayid;       /* Name of bay where port resides       */
    int        access;       /* T/F: is port accessable?             */
    element_map_t slots;     /* Separately addressable slots in port */
    void       *private;     /* LCP private data area                */
    queue_t    queue;        /* To next/prev ports                   */
} port_t;
```

Convenience Routines for Element Maps

The element map header file is separated from generic library representation, to allow element maps to be shared between potentially different layers of an LCP, for instance between the ALI semantic layer and the device access layer. In sample implementations, the device layer fills in some of this information and the ALI semantic layer fills in the rest, then passes element maps to the MLM server with an ALI/R *config* command.

The following convenience routines are provided in module *ovsrc/include/util.c* to handle LCP element maps. See the file *ovsrc/include/maps.h* for subroutine prototypes.

void **map_init()**	Initialize element map of a given type
void **map_free()**	Free an element map
void **map_move()**	Swap two element maps
slot_t ***slotmap_add()**	Add an entry to the slot map
void **slotmap_del()**	Delete a slot map entry
slot_t ***slotmap_find_name()**	Find the entry for a given name in the slot map
slot_t ***slotmap_find_addr()**	Find the entry for a given address in the slot map
slot_t ***slotmap_find_PCL()**	Find the entry for a given PCL in the slot map
slot_t ***slotmap_find_empty()**	Find an empty slot, if one exists
int **slotmap_compare()**	Compare two slot map entries for equivalence
void **slotmap_mount()**	
void **slotmap_unmount()**	
void **slotmap_move()**	
void **slotmap_inject()**	
void **slotmap_eject()**	
drive_t ***drivemap_add()**	Add an entry to the drive map
void **drivemap_del()**	Remove an entry from the drive map
drive_t ***drivemap_find_name()**	Find entry for a given drive name in the drive map
drive_t ***drivemap_find_addr()**	Find entry for a given drive address in the drive map
int **drivemap_compare()**	Compare two drive map entries for equivalence
void **drivemap_inject()**	
bay_t ***baymap_add()**	Add an entry to the bay map
void **baymap_del()**	Remove an entry from the bay map
bay_t ***baymap_find_name()**	Find entry for a given bay name in the bay map
bay_t ***baymap_find_addr()**	Find entry for a given bay address in the bay map
int **baymap_compare()**	Compare two bay map entries for equivalence

port_t ***portmap_add()**	Add an entry to the port map
void **portmap_del()**	Remove an entry from the port map
port_t ***portmap_find_name()**	Find entry for a given port name in the port map
port_t ***portmap_find_addr()**	Find entry for a given port address in the port map
int **portmap_compare()**	Compare two port map entries for equivalence

LCP Utility Functions

This section summarizes convenience routines available in module *ovsrc/include/util.c*, grouped by purpose. The following functions are provided for ALI command queuing and the state machine:

queue_t ***ali_command()**	Enqueue ALI command, and initialize command state.
void **ali_next()**	Send next ALI command.
void **ali_complete()**	ALI command finished, so update state and dequeue it.
void ***ali_context()**	Set and return private command context.
enum cmd_state **ali_state()**	Return ALI command state.

The following functions are provided for ALI/R command queuing and MLM server acknowledgment processing:

queue_t ***alir_command()**	Enqueue ALI/R command for sending.
void **alir_next()**	Dispatch next ALI/R command.
void **alir_abort()**	Dequeue pending ALI/R commands.
int **ali_response()**	Match ALI response to ALI/R command, and vice-versa.

The following function is provided for LCP ready state processing:

void **readystate_change()**	LCP standard ready state processing.

The following functions are provided for handling ALI error responses:

void **attribute_error()**	Handle attribute or show error.
void **ready_error()**	Handle ready state error.

The following functions are provided for mandatory attribute and show processing:

int **attribute_()**	LCP generic attribute and show processing.
int **lcp_attr()**	Attribute and show for generic LCP attribute.
int **bay_attr()**	Attribute and show for generic bay attribute.
int **drive_attr()**	Attribute and show for generic drive attribute.
int **slot_attr()**	Attribute and show for generic slot attribute.
int **bay_description()**	Attribute and show for bay description attribute.
int **drive_description()**	Attribute and show for drive description attribute.
int **slot_description()**	Attribute and show for slot description attribute.
int **lcp_name()**	Attribute and show LCP name attribute.
int **lcp_supportPCLs()**	Attribute and show LCP supportPCLs attribute.
int **lcp_vendor()**	Attribute and show LCP vendor attribute.
int **lcp_loglevel()**	Attribute and show LCP logLeval attribute.

The following functions are provided for debugging:

void **print_stringlist()**	Print ALI_stringlist.
void **print_attrlist()**	Print ALI_attrlist.

Example LCP Implementation

The Exabyte 210/220/440/480 libraries are SCSI-2 medium changers. The Exabyte 210 is a model with 10 slots and one or two Exabyte 8505XL drives. It is comparatively simple to operate—the LCP source code for this autochanger is less than 4000 lines long. The Exabyte 220 is similar but has 20 slots. The Exabyte 440 has 40 slots and up to four drives. The Exabyte 480 is similar but has 80 slots. (In Exabyte model numbers, the first digit describes the maximum number of drives, while the remaining digits describe the number of available slots.)

The Exabyte 210 LCP may be used in conjunction with the Exabyte 8505XL DCP.

IRIX Implementation

Calls to the pass-through SCSI driver are made with the IRIX C library for generic SCSI operations; see the dslib(3X) reference page. Direct SCSI access is by means of this device special file:

`/dev/scsi/sc`*`C`*`d`*`U`*`l`*`L`*

In this filename, *C* is the SCSI controller number, *U* is the unit number, and *L* is the logical unit number (*lun*) for accessing library control. This information may be determined on IRIX systems by using the *hinv* command.

Source Code Organization

This section describes the LCP source and run-time configuration modules.

Configuration Processing

The *ovsrc/clients/lcp/EXABYTE-210/config* file describes both the library and MLM server:

```
localhost          # MLM server host name
739                # MLM server TCP socket
wilma              # OpenVault name for library
host-bedrock       # LCP instance name
/dev/scsi/sc0d510  # SCSI drive control access path
60                 # Library polling interval
fred:82            # Map OpenVault drive name to library address
barney:83          # Do likewise for second drive
```

The *ovsrc/clients/lcp/EXABYTE-210/config.c* module parses this file and fills in library information in both the LCP generic and private data areas.

Device Access Layer

The *ovsrc/clients/lcp/EXABYTE-210/control.h* header file contains the device access layer device representation, and declares subroutine entry points for the ALI semantics layer to access the device. The *ovsrc/clients/lcp/EXABYTE-210/control.c* module implements these subroutines.

ALI Semantic Do* Layer

This layer, named after its many functions starting with "do," is where the LCP interprets ALI commands. The programmer customizes this layer, based on the generic library methods that are provided as part of the LCP developer framework.

The *ovsrc/clients/lcp/EXABYTE-210/main.h* header file contains the LCP private data area of a generic library representation, and associated macros and subroutine prototypes, including four programmable LCP entry points used by the framework, and semantic support routines. The *ovsrc/clients/lcp/EXABYTE-210/main.c* module is where ALI semantic handling routines are implemented, and where ALI commands are dispatched to the appropriate semantic handling routine. For example, the ALI_mount command would be dispatched to the **do_mount()** function.

Representing Private Element Map Entries

The Exabyte 210 LCP does not require custom element maps, because the developer framework provides an adequate generic representation. Other LCPs may require customization. Programmers can customize element map representation by creating the *ovsrc/clients/lcp/NAME/maps_private.h* and *ovsrc/clients/lcp/NAME/maps_private.c* files, where *NAME* represents the LCP name.

Future LCP Implementations

There is potential for a single, shared SCSI-2 media changer LCP. An additional device module would be required only for vendor-dependent processing, or for possible departures from the standard. The infrastructure developer's kit was developed on IRIX, and has been ported to an increasing list of operating system platforms.

Parallel Execution and Complex Mappings

Certain media libraries may perform parallel (instead of serial) execution of commands, and complex (not simple) mappings of ALI to underlying library control. Some libraries, such as SCSI-2 media changers, execute device control commands in a blocking or serial fashion. For most of these devices, there is a one-to-one mapping between an ALI command and the underlying SCSI-2 request. The LCP implementation for such a device may be trivial. For this sort of device, the LCP implementor may simply implement all ALI commands in a serial fashion. No extension of the framework is needed.

Other libraries, such as the StorageTek ACSLS and the IBM 3494, provide some parallelism in control command execution. Optimal use of these devices requires some extra work on the part of the LCP developer to extend the framework. These controllers tend to be more complex than SCSI-2, and require one ALI command to be mapped to potentially multiple underlying control requests. This requires a command execution state machine. Also, developers must understand command dependencies and how the underlying library or controller executes commands, to ensure proper sequencing.

Defined Tokens List

This section documents the predefined strings that are relevant to LCP development.

Cartridge Form Factors

The ALI interface lets the LCP describe to the MLM server what shapes of cartridges it can accept, and what capabilities it can offer with cartridges of that shape. Table 4-2 shows the tokens used for the currently existing cartridge shapes.

Table 4-2 Predefined Cartridge Form Factor Tokens

Token	Description or Usage
8mm	Any generic 8 mm shell
3480	For example: IBM 3480/3490/3495, STK 4480/4490, and so forth
DLT	Digital linear tape (Quantum)
DAT	4 mm digital audio tape (DDS1 and DDS2)
D2-S	Small DST cartridges (25 GB capacity)
D2-M	Medium DST cartridges (75 GB capacity)
D2-L	Large DST cartridges (165 GB capacity)
DTF	20 GB cartridges from Sony
VHS	For example: Metrum
QIC	Quarter inch cartridge
CD-ROM	Compact disk read-only memory

Attribute Names (LCP)

Table 4-3 shows one attribute used in an LCP, where it is used, and what it means.

Table 4-3 Predefined Attribute Name Tokens (LCP)

Attribute Name	Where Used	Possible Values	Required?	Description
ExchangeTime	ALI config command, perf clause	numeric, in seconds	yes	The approximate time it takes for the library to move a cartridge from its home location to a drive, or back, not including drive load/unload time.

Chapter 5

Abstract Drive Interface (ADI) Language

This chapter provides programmers with an introduction to the OpenVault languages for controlling removable media drives, and includes the following sections:

- "Abstract Drive Interface—ADI" describes the language in which the MLM server sends directives to a DCP, and responds to requests sent by a DCP.
- "ADI Response—ADI/R" on page 66 tells how a DCP sends configuration and status to the MLM server, and responds to directives from the MLM server.

Abstract Drive Interface—ADI

The following sections describe the abstract drive interface (ADI), including objects, object attributes, naming conventions, and the ADI command repertoire.

About ADI

ADI is a language that provides an abstraction of removable media drives managed by OpenVault. ADI hides details of the underlying drive and control without compromising the ability of OpenVault as a whole to manage its resources efficiently.

ADI Object Definitions

The ADI language manipulates the following objects:

drive control program (DCP)
: Each DCP manages the configuration of drives, and performs drive control tasks associated with mount and unmount requests from OpenVault client applications. The main purposes of a DCP are to expose drive configuration to the MLM server, and to control drives that have an OpenVault accessible control interface.

  See Chapter 6 for a tutorial on DCP programming.

drive — A place where a cartridge may be mounted and its media loaded for read/write access.

access method instance — The instantiation of a drive access method—the implementation of a particular set of capabilities that describe a mode of access to the drive; this is equivalent to a UNIX® device special file or *dev* node.

partition — A region on the cartridge media that has a physically marked beginning and end, both of which the drive recognizes.

command — ADI commands become objects as far as ADI is concerned. When the MLM server sends an ADI command, it associates a task ID with that command. The sender may refer to that command later by using the same task ID, but only to cancel the command. When a DCP receives a command, it includes the task ID in command responses.

Abstraction of a Drive

The most important object managed by a DCP is the drive, which has the following traits:

capabilities and mode of access — A drive has an associated set of capabilities, which describe specific feature settings. Capabilities determine which device driver to use for control and data requests, what device settings to select, and device state to check. Particular combinations of capabilities represent a particular mode of access. A drive has a configurable set of legal access modes, each of which represents a logical instance of the drive with underlying control and access methods. The use of canonical capabilities and modes of access is what permits a DCP to hide implementation details such as the underlying local control and access methods for the device.

media — Recordable surface(s) upon which data are read or written. A cartridge may contain one or more pieces of media. Associated with this and the drive is a bit format, which determines the recording format. Together with media type, bit format determines media storage capacity.

mount point — The physical drive opening where a cartridge may be placed, often called the drive door. A cartridge must be present at the mount point before the cartridge and the media it contains can be engaged at the media access point. When the media is disengaged from the media access point (and returned to its cartridge, as necessary), the cartridge is returned to the mount point.

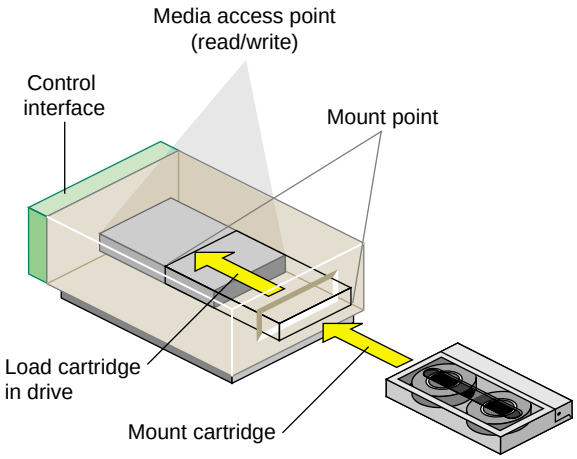

Figure 5-1 Conceptual View of a Drive

media access point
: A cartridge must be moved and the media it contains engaged at a media access point before the media can be read/write accessed. This is the component that reads and writes the media contained by a cartridge. The cartridge, media, or media access point may physically restrict access to the media. For instance physical access may be restricted to read only. Once media is engaged at the media access point, media data may be accessed through the drive data path.

control path
: A control interface to the drive that is accessible by the DCP, and possibly by OpenVault client applications on the DCP host. Typically, a drive is connected to a host by a local channel or bus. This connection represents the control path to the drive.

data path
: A connection between the DCP host and the drive media access point that may be accessible by the DCP and MLM client applications on the DCP host. Drives with a control path typically also have a data path, with control and data paths sharing the same connection, with access through a local device driver. For drives with a data path, DCP may require access to that data path, for example to identify a partition. A drive media access point may lack a data path. For example, a set of RGB lines attaching a video drive to a display device lacks a host connection, so applications do not have access to it.

drive handle
: A local binding between a name, such as a device pathname, and a logical instance of the drive, such as a device node that corresponds to a particular mode of access. The name is called a drive handle. For drives that have a data path, the drive handle may be passed to and used by an OpenVault client application to send drive control or access the media. When the binding is removed, the drive handle is invalidated.

Attributes and Object Properties

OpenVault requires a DCP to maintain drive configuration attributes and notify the MLM server when they change. DCPs use the ADI/R *config* and *ready* commands to do this. These commands send attributes back to the MLM server, where configuration information is kept in the MLM server persistent store. It is potentially recoverable by the DCP using the ADI/R *show* command. Here are the required configuration attributes:

- DCP ready state (see "Ready States" on page 94)
- drive capability configuration (see "Drive Capabilities" on page 86)
- additional drive attributes (see "Attribute Names (DCP)" on page 87)

Note: Currently, OpenVault does not support recovery of any attribute or property information stored in the MLM server persistent store by a DCP. However, this will be supported in a future version of OpenVault, and developers will be encouraged to use it.

arbitrary attributes
: A DCP developer may also maintain arbitrary attributes, and store them in and recover them from the MLM server persistent store. These attributes are opaque to the MLM server.

mandatory attributes
: A DCP developer may store the *logLevel* mandatory attribute in the MLM server persistent store, so it can recover the attribute and resume logging at the same level across reboots.

ADI expresses DCP attributes using the tuple *object type*, *object name*, *attribute name*. Table 5-1 shows the mandatory attributes, not including the configuration attributes.

Table 5-1 Mandatory DCP Attributes

Object Type	Object Name	Attribute Name	Command
DCP	""	name	ADI show
DCP	""	logLevel	ADI show, ADI attribute set

ADI Object Naming

These names refer to specific ADI objects:

DCP name	Each DCP is uniquely named by a value pair including an OpenVault client name and an OpenVault instance name.
client name	Refers to a specific drive, and is the name by which a client identifies itself in a HELLO command to the MLM server. This is the name the MLM server associates with the device managed by the associated DCP.
instance name	This name is arbitrary, but is needed if multiple DCPs control the same drive, so as to differentiate DCPs with the same client (drive) name.
drive name	An OpenVault drive name refers to a removable media drive.
drive handle	Refers to a particular drive access method instance.
partition name	References the name for a media partition.
task ID	Uniquely identifies a sender-generated command.

Attribute naming in ADI is different from that for CAPI and AAPI, in which an attribute is named with *TableName.ColumnName*; attributes are just columns in a relational table. In ADI and ADI/R, attributes are named with a tuple:

objectType, objectName, attrName

ADI Commands

The MLM server speaks ADI to the DCP, which in turn speaks ADI/R to the MLM server. The ADI language includes the following commands:

activate

The *activate* commands start and stop the DCP and drive interactions. Once the DCP has established a session with the MLM server with a *hello-welcome* sequence, it may begin accepting ADI commands from the server. However, until it has successfully been *activate enabled* by the server, and is in ready state, it should resend *ready lost* state and fail any ADI commands that require drive access, with an ADI_E_READY error.

The DCP should issue one of the *ready* command variations when it finishes processing the activate request. *Activate* is supported for all drives managed by OpenVault, but is a no-op for drives that lack an OpenVault control interface.

activate enable

The *activate enable* command forces the DCP to resynchronize with its drive hardware, ensuring that the DCP has current drive state. This helps support drives that are attached to multiple hosts. If drive control switches from one DCP to another, the *activate* command ensures that the controlling DCP has up-to-date drive status.

In cases where multiple DCPs are associated with one drive (that drive is attached to multiple hosts), the MLM server ensures that only one DCP at a time is actively controlling the drive.

The DCP reports *ready* when it has successfully resynchronized with its drive.

activate disable

The *activate disable* command forces the DCP to stop communicating with its drive hardware. The DCP requires an *activate enable* command before it can talk to its drive again. This arrangement supports drives that are attached to multiple hosts. If drive control switches from one DCP to another, the *activate disable* command ensures that the DCP that loses control does not interfere with another one.

Performing this command should cause the DCP to complete or cancel any ADI commands that require access to the drive, store persistent drive state in the MLM server, stop communicating with the drive, and send a *ready lost* command to the MLM server.

Note that *activate* may require the DCP to have access to a drive data path, in addition to a control path; otherwise, it may be a no-op.

attach

Selects the appropriate logical instance of a drive according to the access mode specified by the MLM server. *Attach* instantiates this access method as needed, and binds an opaque drive handle to the logical instance (on UNIX systems, this means linking to a device node). The drive handle must be unique on the DCP host, and may be generated by the DCP, or specified by the MLM server. The DCP returns an error if it detects that the drive handle is already in use on the local host.

Generally, the MLM server invokes *attach* as part of drive selection for a CAPI *mount*, after loading. This command is supported for all drives managed by OpenVault, but is a no-op for drives that lack a data path.

In the case of partitioned drives (such as two-sided optical disc units), the drive handle that is created may be dependent on the partition. For example, most disks on UNIX systems have the partition to be accessed encoded in the drive handle (the device node). The loaded media is positioned to the specified partition. Partition names may be defined; see "Defined Tokens List" on page 83 for a list of partition names.

Typically, drives have a shared control and data path. In this case, the drive handle that is passed back to the MLM server is ultimately passed back to an OpenVault client application. The application uses the drive handle to establish access to the drive control/data path.

The *attach* command allows the MLM server to change drive access mode multiple times, without changing any names from the client application's perspective. However, the application must reestablish access after each *attach* for the change to affect the application.

The MLM server ensures that drive access mode is consistent with drive capabilities.

attribute

Attributes are a mechanism by which information that is not contained in normal configuration data passed to the MLM server can be accessed. Examples include data that is unique to a drive type, or data that varies over time. Attributes may read/write or read-only. If the attributes represent internal information or settings associated with the drive itself, the DCP sends corresponding requests to the drive, then returns that information. See "Attributes and Object Properties" on page 60.

cancel

The *cancel* command attempts to stop execution of a command sent to the DCP. The DCP is free to continue the execution of the command if the command has proceeded too far to cancel.

Note: The *cancel* and *response* commands may not be cancelled.

detach	A *detach* command removes the logical instance as necessary, and the binding created by an *attach* command (on UNIX systems, this means unlinking a device node associated with the device). *Detach* invalidates the drive handle created by a previous *attach* command. For drives with a shared control and data path, this disables a client application from establishing access to drive control and data paths through this handle. Generally, the MLM server invokes this command as part of drive deselection for a CAPI *unmount*, before unloading. This command is supported for all drives managed by OpenVault, but is a no-op for drives that lack a data path. The MLM server and the DCP should try to ensure that applications do not continue to access drive control or data through a drive handle that has been invalidated by *detach*. Note that *detach* and *attach* may have no immediate impact on an application that was already accessing the drive control and data paths. Once the application has established its access, it may proceed to access the drive control and data paths, without being affected by subsequent invocations of *attach* and *detach*. A case in point is with random access media and UNIX applications that perform an open (read/write) and close system call sequence in which the drive handle is passed by the application as an argument only to **open()**. In this case, the effects of the *attach* or *detach* command may occur only during the **open()** call. The *detach* and *attach* commands may have no effect on any reads and writes that are made between open and close.
exit	The *exit* command tells the DCP to store any persistent DCP and drive information to the MLM server, complete or cancel pending ADI commands, complete or abort pending drive operations, do shutdown processing as required, send *ready lost* and *goodbye* commands to the MLM server, and exit.
goodbye	The *goodbye* command tells the communicating DCP to end this session.
load	If a cartridge is present at the drive mount point, the *load* command moves that cartridge to the media access point, and engages it, making it accessible at the media access point. The drive is then called loaded.

Minimally, *load* verifies that the drive is loaded. This command does not identify which media is engaged. It is invoked by the MLM server as part of a CAPI mount request. Normally, the ALI *mount* command associated with the CAPI mount loads the drive (the library causes the load to occur), so DCP *load* needs to verify only that the drive is loaded.

This command is supported for all drives managed by OpenVault, but is a no-op for drives that lack an OpenVault control interface.

reset — The *reset* command tells the DCP to force the drive to reinitialize. This may also cause the drive to execute self-diagnostics. This is a best-effort type of command. If it is possible to reset a drive only by resetting the whole SCSI bus, thereby interrupting other transfers on that bus, the DCP is free to treat this command as a no-op.

If *reset* is a prolonged drive activity, the DCP should send a *ready not* command to indicate that its drive is temporarily not available, followed by a *ready* command when the drive becomes available again.

response — The *response* command acknowledges and indicates success or failure of an ADI/R command. The optional text portion of the response contains error details or command results.

show — This is the attribute query mechanism. Note that *show* commands that query information directly from a drive may require that a DCP have access to a data path with the drive, and otherwise may return an error. See “Attributes and Object Properties” on page 60.

unload — If the drive is loaded, the *unload* command disengages the media, returns it to the cartridge as necessary, and returns the cartridge to the drive mount point. The drive is said to be unloaded at this point. This command rewinds media before disengaging, as necessary. It is invoked by the MLM server as part of a CAPI unmount. Minimally, it detects whether the drive is already unloaded.

This command is supported for all drives managed by OpenVault, but is a no-op for drives that lack an OpenVault control interface.

The ADI/R *response* is responsible for returning drive usage and error statistics as transmitted on pages 2 through 5 of the SCSI log:

```
response whichtask["A"] success
    text ["bytes written" "32768" "softerrors" "0" ... ];
```

There is no *barrier* command in ADI—OpenVault assumes that ADI commands are executed serially by the DCP and its drive.

ADI Response—ADI/R

The following sections describe the ADI response language (ADI/R), including objects, object attributes, naming conventions, and the ADI/R command repertoire.

About ADI/R

ADI/R is primarily the response language for ADI. In addition to giving the matching acknowledgment and final response to an ADI command, ADI/R provides the means for a DCP to send its configuration and status to the MLM server.

ADI/R Object Definitions

The ADI/R language manipulates the following objects:

command — ADI/R commands become objects as far as ADI/R is concerned. When a DCP sends an ADI/R command, it associates a task ID with that command. The sender may refer to that command later by using the same task ID, but only to cancel the command. When the MLM server receives a command, it includes the task ID in command responses.

message — A text message to be entered into an MLM server-managed log, and perhaps displayed on some console by the MLM server, or one of its administrative applications. Messages are associated with a severity level, or a level of urgency, which determines (along with site policy) whether the message text is stored in the MLM server logs, displayed on a library or OpenVault console for the operator, or both.

Attributes and Object Properties

Currently, ADI/R attributes are not supported by OpenVault, except for attributes stored by the ADI/R *config* and *ready* commands in the MLM server persistent store. Currently, OpenVault supports only setting and unsetting of these attributes. See “Attributes and Object Properties” on page 60.

ADI/R Object Naming

These names refer to specific ADI/R objects:

message ID — Refers to a text message of a given severity level.

task ID — Uniquely identifies a sender-generated command.

ADI/R Command Descriptions

The DCP reads ADI commands from the MLM server, and replies to the server in ADI/R. The ADI/R language includes the following commands:

attribute — With an *attribute* command, the DCP stores persistent state in the OpenVault database.

cancel — The *cancel* command tells the MLM server to prevent execution of a particular command, if possible.

Note: The *cancel* and *response* commands may not be cancelled.

config — The *config* command tells the MLM server what access modes are supported, with the form factors, media formats, and performance characteristics for each. *Config* copies configuration information, such as the capabilities of a drive, from the DCP to the MLM server. The MLM server stores a nonauthoritative copy of all such information for all the DCPs it controls. Each DCP must use the *config* command to update configuration information whenever it changes.

The *config* command is shorthand for sending attributes about a drive to the MLM server. See "Attributes and Object Properties" on page 60, and "Defined Tokens List" on page 83 for more information.

In a *full scope*, all information associated with the DCP should be deleted and replaced with the information listed by *config*. By contrast, in a *partial scope*, only the pieces of information about the DCP that are listed by *config* should be replaced. Normally *full scope* is used only at startup time, or when making major changes to drive configuration.

The *config* command gives the MLM server a list of access modes that the drive offers. Each mode has a name and additional characteristics:

- Supported cartridge form factors. Some drives support different shapes of cartridges, and offer different capabilities when using one shape instead of another.

- Supported media bit formats. Some drives support different formats for the bits on the media, and offer different capabilities when using one bit format instead of another.
- Supported capabilities. Each access mode provides a certain set of capabilities to the application, and each capability has a name.
- Performance characteristics. Some drives are able to handle different form factors or media bit formats, and offer different performance characteristics when using one form factor or bit format instead of another. The MLM server may use that information when choosing which drive to use. For example, a drive with a read bandwidth of greater than 1 MB per second may be required for a particular application.
- Whether or not a drive is occupied by a cartridge.

goodbye

The *goodbye* command tells the MLM server to end this session and clean up its end of the session. This protects against the accumulation of idle connections, since the MLM server has no way of detecting that a DCP exited other than the TCP/IP *keepalive* option. *Keepalive* helps recover from process failures, but a DCP should send a *goodbye* before exiting to prevent unnecessary continuation of connection resources.

message

This is a method for the DCP to send a message to the operator or to a log file. It contains a list of uninterpreted character strings.

ready

With a *ready* command, the DCP informs the MLM server about the current status of its drive connection. Like the *config* command, the *ready* command is shorthand for sending drive attributes to the MLM server.

These are variations of the *ready* command:

ready yes
: Tells the MLM server that the DCP is ready to process commands.

ready no
: Informs the MLM server that the DCP is not prepared to process commands at this time.

ready lost
: Informs the MLM server that the DCP has lost communication with its drive. It might be appropriate for OpenVault to try another control path (another DCP) connected to the drive.

ready broken
: The hardware reports a fatal error, so there is no point in trying an alternate control path.

See “Ready States” on page 94 for more detailed information.

response — The *response* command acknowledges and indicates success or failure of an ADI command. The optional text portion of the response contains error details or command results.

show — With a *show* command, the DCP queries persistent state it has stored in the OpenVault database.

Ordering of ADI Response Text

For some ADI commands, the matching ADI/R response command for a successful response contains a text portion, which must have a particular format. This section describes the required format.

Response Text for ADI_show Command

The text portion of a successful response to show depends on the specified mode for the show, and on the number of attributes to be queried. There are three possible modes:

`ADI_show_name` — show name only

`ADI_show_value` — show value only

`ADI_show_namevalue` — show name and value, in that order

For each attribute to be queried, the text portion of the response includes name-value information, as dictated by this mode, and is ordered according to the specified attribute list. So, for example, if a show command requested a query of DCP *logLevel* and vendor attributes, with mode ADIR_show_namevalue, the corresponding text portion of the response would look something like this:

```
text[ 'logLevel' 'debug' 'vendor' 'EXABYTE' ]
```

Response Text for ADI_attach Command

The text portion of a success response for an ADI *attach* command includes the value *drive-handle*. Suppose an attach caused the drive handle */tmp/mlm/handleXXX* to be bound to the instantiation of a drive access method. The corresponding text portion of the response would look something like this:

```
text[ '/tmp/mlm/handleXXX' ]
```

Chapter 6

Programming a Drive Control Program (DCP)

This chapter provides a tutorial to DCP programming, and includes the following topics:

- "Initialization Issues" on page 72 talks about starting up a control program.
- "DCP Development Framework" on page 76 describes DCP subroutine libraries.
- "Example DCP Implementation" on page 80 discusses sample source code layout.
- "Defined Tokens List" on page 83 presents tables of OpenVault tokens for a DCP.

About the DCP

A DCP (drive control program) translates between the OpenVault ADI and the actual device control interface for its drive, and between device responses and ADI/R. The DCP does what is necessary to affect the required ADI semantics. It keeps the MLM server's "cache" (persistent store) up to date regarding DCP configuration, drive configuration, and ready state information. To do this, the DCP sends *config* and *ready* commands when it detects changes in state, on a best-effort basis.

Use of Persistent Storage

Currently, the drive configuration and state is moved in one direction only, from a DCP to the MLM server persistent store. The MLM server uses this information to assist with drive selection for cartridge and volume mounts. In future revisions of OpenVault, the DCP may recover some state from the persistent store, so that configuration and state information can flow in both directions. However, the DCP and drive are always considered the authoritative source for state information about a DCP or its drive.

DCP Configuration

In sample implementations, DCP configuration is stored in a configuration file that is local to each DCP. See "Configuration File" on page 72 for more information.

Initialization Issues

Each DCP must initialize itself in order to contact the MLM server.

DCP Booting

Drives may be connected to multiple hosts and thus have multiple control paths. There can be one DCP associated with each control path. Only one DCP at a time may be active for any drive; the MLM server arbitrates which DCP is active.

For example, a DCP could be on the inactive side of a multiconnected library. The DCP boot sequence must not interfere with the active side of a multiconnected library. The MLM server is the arbitrator of control for multiconnected libraries and drives. A DCP should not assume that it is controlling a drive until the MLM server says so.

Configuration File

Each DCP should have a configuration file containing at least the following information:

- The address of the controlling MLM server.

  This allows the DCP to initiate contact with the controlling MLM server. It is the name of the system, or its numeric IP address. The MLM server is usually available at well-known port number on that system, by default 44444.

- The OpenVault name for the managed drive.

  The MLM server uses this name as an identifier for this physical drive. This is the name of the device that it is managing, not the name of the particular instance of DCP. All names must be unique within an OpenVault domain so that the server can detect multiconnected libraries (multiple LCPs controlling the same library).

- The DCP instance name.

  The instance name is arbitrary, but is required for cases where there are multiconnected libraries.

- The control path to the drive (for example, */dev/rmt/tps0d3*).

  This is how a DCP talks to the hardware. This information is not visible to the MLM server. Some drives are not controlled in this way (VHS videocassette players, for instance), but all DCP implementations need something equivalent.

- A list of access mode names and access capabilities for this drive.

  This is completely implementation-dependent, and sometimes might not even exist, but some way is needed for administrators to control the capabilities that the DCP advertises to the MLM server. In IRIX implementations, the configuration file lists tag names and their associated capabilities and performance parameters. Capabilities are simply agreed-upon text strings. The MLM server does not care what they are, it simply compares them for equality when looking for a drive to satisfy user requirements. Device tags are text strings (pathnames) the MLM server uses to inform the DCP which combination of capabilities it wants when mounting the next cartridge. For example:

```
name      capabilities list for that handle          device pathname

base                                                 /dev/rmt/tps0d4nrv
r         rewind                                     /dev/rmt/tps0d4v
f         fixedblock                                 /dev/rmt/tps0d4nr
c         compressed,                                /dev/rmt/tps0d4cnrv
cr        compressed, rewind                         /dev/rmt/tps0d4cv
cf        compressed, fixedblock                     /dev/rmt/tps0d4cnr
rf        rewind, fixedblock                         /dev/rmt/tps0d4
crf       compressed, rewind, fixedblock             /dev/rmt/tps0d4c
stat      status                                     /dev/rmt/tps0d4stat
all       allmodes                                   /dev/rmt/tps0d4
```

  In this example, a UNIX device pathname is included so as to avoid having the DCP understand the format of a *dev_t* minor number, or the equivalent on some other operating system. The DCP can replicate the path (copy the *dev_t*) when it needs to create a handle for that combination of drive and access mode. OpenVault defines the default capabilities of a drive, so the DCP must specify what capabilities it offers in terms of changes to that default set.

For easy editing, DCP configuration files should be composed of readable ASCII text.

DCP Boot Sequence

When a DCP boots or reboots, it must:

1. Allocate internal data structures and initialize state.
2. Refrain from talking to the drive.

   The DCP boots into *activate disable* state, and must wait for the MLM server to tell it when to talk to the drive. If the drive is dual-ported with another DCP actively controlling it, that session should not be interrupted!

The MLM server issues an *activate enable* command when conditions permit your DCP to control the drive. If the library is single-ported, *activate enable* is issued almost immediately.

3. Read its configuration file.
4. Establish a session with the MLM server.

   The DCP sends the "hello" message upon opening the connection. In this example, *name* is the OpenVault name for the drive, and *inst* is the DCP instance name. If connection fails, retry every two minutes. The DCP blocks until it receives a *welcome* command telling it which language version to use during this session.

   `hello language["ADI"] version["1.0"] client["`*name*`"] instance ["`*inst*`"];`

When the MLM server is first contacted by a DCP it will:

1. Integrate the drive into its list of managed devices.

   The MLM server checks for other DCPs managing that physical drive. If this DCP is the first, OpenVault allows this DCP to proceed. This sequencing implies that DCPs are given control of their associated drive on a first-come-first-served basis.
2. (Eventually) issue an *activate enable* command to the DCP.

When the MLM server says to *activate enable*, the DCP must:

1. Reply to the MLM server with a *ready no* command.

   The DCP informs the MLM server that it has started to come up, but is not yet ready to accept drive control commands.
2. Talk to the drive to determine:
   - That the drive is supported by this DCP.
   - The supported media formats (for example: EXABYTE-8mm-5GB).
   - Whether or not the drive can support the listed access modes.
   - If the drive is loaded or in use at this time.
   - Any other information that may be relevant to drive or DCP operation.
3. Collect any state or configuration information from the MLM server.

   The DCP can store state or configuration information in the OpenVault persistent store. For example, the DCP probably needs to retrieve the *loglevel* attribute so it can resume logging only the messages that the system administrator wants logged.

4. Push all the capability information up into the MLM server.

   The DCP needs to update the MLM server's copy of the capability list at boot time, before the DCP has been activated. This is different from an LCP, which must be activated in all cases. By contrast, it is unnecessary to activate all the DCPs that might control a given drive just to determine their capability set.

   The DCP takes all its compiled-in settings and information from its configuration file to generate a *config* command for the MLM server. There is a possibility that the offered capabilities might change once the DCP has had a chance to talk to the drive hardware, but the MLM server must deal with this if it happens.

5. Send a *ready* command to the MLM server.

   The DCP is now ready to accept drive control commands.

6. Respond *success* to the original *activate enable* command.

   This is defined to be the last step as a convenience to the MLM server, so that the server can block until it receives a response from its *activate enable* command rather than continually polling for arrival of the *ready* command.

Activation Sequence

When a DCP receives an *activate enable* command from the MLM server, and the DCP is in *ready lost* state, it should perform these steps:

1. Access its drive to acquire or verify device-specific configuration and state.

   For example, a DCP may consult its drive to determine:

   - if the drive is supported by this DCP
   - whether the drive is in a usable state for this DCP
   - optimal block size

2. Push configuration information to the MLM server.

   For example, configuration information includes: supported form factors, media types, bit formats, media capacity, block size, nominal drive load time, drive read and write bandwidth, and drive capabilities. See the tables in the section "Defined Tokens List" on page 83 for particulars.

3. Transition to *ready* state, and push this new state to MLM server.

DCP Development Framework

The infrastructure developer's kit includes a framework for writing a DCP that helps ease the development, porting, and maintenance effort for DCPs. This section describes the general source tree layout.

OpenVault Client-Server IPC

OpenVault clients and servers communicate with a custom interprocess communication (IPC) layer. DCP modules that deal directly with ADI and ADI/R need to include the following header file, and be loaded with the following C library:

ovsrc/include/ov_lib.h
: C data structures, macros, and subroutine prototypes for IPC

ovsrc/libs/comm/libov_comm.so
: C library containing IPC subroutines

ADI Parser and ADI/R Generator

OpenVault includes language parsers and generators. DCP modules using these facilities need to include the following header files, and be loaded with the following C libraries:

ovsrc/include/adi.h
: Supported ADI and ADI/R language version, ADI standard errors, and C data structures for ADI and ADI/R command representation.

ovsrc/include/dcp.h
: Parser and generator subroutine prototypes.

ovsrc/include/hello.h
: C data structures for *HELLO* and *WELCOME* command representation.

ovsrc/libs/hellor/libov_hello.so
: C library that contains HELLO parser-generator subroutines.

ovsrc/libs/adi/libov_adi.so
: C library that contains ADI parser-generator subroutines.

DCP C Library Routines

The DCP(3) reference page documents the ADI and ADI/R lexical library routines that you employ when writing a DCP. Table 6-1 offers a summary of these routines.

Table 6-1 ADI and ADI/R Lexical Library Routines

Purpose of Activity	DCP Function	Short Description
Initiate session with MLM server	**ADIR_initiate_session()**	Begins session with a specific MLM server, including HELLO version negotiation
Parse ADI command from MLM server	**ADI_receive()**	Parses an ADI command and returns an ADI_command structure
Acknowledge ADI command	**ADI_acknowledge()**	Informs MLM server that the DCP received an ADI_command
Send ADI/R command to MLM server	**ADIR_alloc_cmd()** **ADIR_alloc_ready()** **ADIR_alloc_message()** **ADIR_alloc_capinfo()** **ADIR_alloc_attr()**	Allocates ADIR_command structure Allocates ADIR_ready structure Allocates memory for ADIR_message Puts drive capability info into ADIR_capinfo Allocates attribute name and value pair
Send final response for ADI command to MLM server	**ADIR_alloc_response()** **ADI_alloc_string()** **ADIR_send()** **ADIR_free()**	Allocates ADIR_response structure Allocates string and links into ADI_stringlist Transmits ADIR_command to MLM server Deallocates ADIR_command structure
Free ADI command	**ADI_free()**	Deallocates ADI_command structure

DCP Common Framework

The infrastructure developer's kit includes common utility code for writing a DCP. To use this code, include the following header files, and read the following C module:

ovsrc/include/cctxt.h
: Generic command queuing mechanism.

ovsrc/include/dcp_lib.h
: Generic representation of DCP and drive state, generic representation of an attribute, common DCP fixed and programmable entry points, and common DCP utility subroutine prototypes.

ovsrc/include/queue.h
: Generic queue and linked list implementation.

ovsrc/clients/dcp/common/util.c
: DCP common fixed-entry points and utility subroutines.

Generic Representation of a Drive—dcp_lib.h

Much of a DCP's representation of DCP and drive state can be represented generically. However, the DCP developer needs a way to customize this representation for a particular drive and implementation.

The framework provides a private data area and programmable entry points so the developer can customize the representation of DCP and drive state. The private data area allows the developer to maintain additional information about the DCP and drive; programmable entry points allow the developer to customize actions associated with initialization (booting), deactivation (transition to *ready lost* state), and shutdown. This arrangement allows the shared framework to invoke these entry points as appropriate.

Here is the framework's generic representation for a drive:

```
struct driveinfo {
    /* elements from DCP config file. */
    char *client;               /* MLM name of this drive.          */
    char *instance;             /* Client instance.                 */
    char *mlmhost;              /* MLM host.                        */
    int mlmport;                /* MLM port.                        */
    int timeout;                /* ADI receive timeout.             */
    char *addr;                 /* Drive access path for DCP.       */

    /* elements initiated by DCP. */
    enum ADIR_ready_type readystatus; /* ready, not r_, disconnected */
    enum ADIR_msg_severity loglevel;  /* Log level for DCP messages. */
    char *vendor;                     /* Drive vendor name.          */
    queue_t ADI_cmd_queue;            /* ADI command queue.          */
    queue_t ADIR_cmd_queue;           /* ADIR command queue.         */
    int waiting_for_ack;              /* 1 if waiting for ack, or 0  */
    char *taskid_for_ack;             /* TaskID of last ADIR command */
    void(*dcp_deactivate)(struct driveinfo *drivei);   /* deactivate */
    void(*dcp_exit)(*drivei, int abnormal);              /* shutdown */
    void(*dcp_dispatch)(*drivei, struct ADI_command *cmd);
    char *(*dcp_taskid)(*drivei);              /* taskid generation */
    void *private;                       /* DCP private libary info */
};
```

Common DCP Entry Point

A DCP that makes use of this developer framework must call the following subroutine to initialize the generic and private data areas for DCP and drive information, and set the programmable DCP entry points:

```
void dcp_init(struct driveinfo *drivei,
              void dcp_init_private(),
              void dcp_deactivate(),
              void dcp_exit(),
              void dcp_dispatch(),
              void dcp_taskid());
```

Programmable DCP Entry Points

This entry point is called one time only from **dcp_init()**, so the *driveinfo* structure does not store it. Required entry point for DCP private data area allocation and initialization:

```
void dcp_init_private(struct driveinfo *drivei);
```

Remaining entry points are stored in the *libinfo* structure. Required entry point for DCP private actions to *activate disable*:

```
void dcp_deactivate(struct driveinfo *drivei);
```

Required entry point for DCP private actions to shut down gracefully and exit:

```
void dcp_exit(struct driveinfo *drivei);
```

Required entry point for ADI command dispatch from within command state machine:

```
void dcp_dispatch(struct driveinfo *drivei, struct ADI_command *cmd);
```

Required entry point for DCP to generate a task ID for ADI/R commands:

```
void char *dcp_taskid(struct driveinfo *drivei);
```

DCP Utility Functions

The following functions are provided for ADI command queuing and the state machine:

queue_t ***adi_command()**	Enqueue ADI command, and initialize command state.
void **adi_next()**	Send next ADI command.
void **adi_complete()**	ADI command finished, so update state and dequeue it.

void ***adi_context()**	Set and return private command context.
enum cmd_state **adi_state()**	Return ADI command state.

The following functions are provided for ADI/R command queuing and MLM server acknowledgment processing:

queue_t ***adir_command()**	Enqueue ADI/R command for sending.
void **adir_abort()**	Dequeue pending ADI/R commands.
void **adir_next()**	Send next ADI/R command.
int **adi_response()**	Match ADI response to ADI/R command.

The following function is provided for DCP ready state processing:

void **readystate_change()**	DCP standard ready state processing.

The following functions are provided for handling ADI error responses:

void **attribute_error()**	Handle attribute or show error.
void **ready_error()**	Handle ready state error.

The following functions are provided for mandatory attribute and show processing:

int **attribute_()**	DCP generic attribute and show processing.
int **dcp_attr()**	Attribute and show for generic DCP attribute.
int **dcp_name()**	Attribute and show DCP name attribute.
int **dcp_loglevel()**	Attribute and show DCP logLevel attribute.

The following functions are provided for debugging:

void **print_stringlist()**	Print ADI_stringlist.
void **print_attrlist()**	Print ADI_attrlist.

Example DCP Implementation

The EXB-8505XL drive is a SCSI-2 tape device that accepts 8 mm media.

The DCP for an Exabyte 8505XL drive may be used in combination with the LCP for an Exabyte 210 media changer.

IRIX Implementation

Control access is three-part, and includes use of the local filesystem, a pass-through SCSI driver, and IRIX magnetic tape interface (MTIO) **ioctl()** operations.

Use of Local Filesystem

This implementation uses a set of drive instance prototypes, which are represented by a set of existing device special files, for example */dev/rmt/tps0d6*. So drive instances are already instantiated. Attach and detach commands simply bind a drive handle to an existing instance, or device special file. Creating and removing a binding is done using the local filesystem **mknod()** and **unlink()** operations.

Direct SCSI Commands

Calls to the pass-through SCSI driver are made with the IRIX C library for generic SCSI operations; see the dslib(3X) reference page. Direct SCSI access is by means of this device special file:

`/dev/scsi/sc`*C*`d`*U*`l`*L*

In this filename, *C* is the SCSI controller number, *U* is the unit number, and *L* is the logical unit number (*lun*) for accessing drive control. This information may be determined on IRIX systems by using the *hinv* command.

Calls to *dslib* are used to get mode sense information directly from the drive, to check for information such as whether the drive supports partitions, and to issue mode select commands, such as those for moving the tape to a particular position.

MTIO Operations

MTIO calls are made by sending **ioctl()** calls directly to the tape driver associated with the control access path for a particular drive instance. MTIO operations perform load verification and unload.

Source Code Organization

This section describes the DCP source and run-time configuration modules.

Configuration Processing

The *ovsrc/clients/dcp/EXB-8505XL/config* file describes traits of the drive and MLM server:

```
localhost          # MLM server host name
739                # MLM server TCP socket
fred               # OpenVault name for drive
dcpfred            # DCP instance name
/dev/rmt/tps0d6    # MTIO drive control access path
/dev/scsi/sc0d610  # SCSI drive control access path
60                 # Communications timeout
```

Remaining lines include supported drive instance prototypes, including mode name, form factor, media type, bit format, capacity, and control capabilities.

The *ovsrc/clients/dcp/EXB-8505XL/config.c* module parses this file and fills in drive information in both the DCP generic and private data areas.

SCSI Control Access

The *ovsrc/clients/dcp/EXB-8505XL/control.h* header file contains definitions, data type declarations, and subroutine prototypes for control access by means of the pass-through SCSI driver; see dslib(3X).

The *ovsrc/clients/dcp/EXB-8505XL/control.c* module contains convenience routines that make SCSI library calls to get mode sense, check for partition support, and change tape partition. Since partition support is currently not implemented, the latter is implemented as a no-op.

Otherwise, device access is made directly from the main ADI semantic module by means of MTIO **ioctl()** operations.

ADI Semantic Do* Layer

This layer, named after its many functions starting with “do,” is where a DCP interprets ADI commands. The programmer customizes this layer, based on the generic drive methods that are provided as part of the DCP developer framework.

The *ovsrc/clients/dcp/EXB-8505XL/main.h* header file contains the DCP private data area portion of a generic drive representation, as well as macros and subroutine prototypes, including four programmable DCP entry points for use by the framework and semantic support routines.

The *ovsrc/clients/dcp/EXB-8505XL/main.c* module is where ADI semantic handling routines and entry points are implemented, and where ADI commands are dispatched to the appropriate semantic handling routine. For example, the ADI_load command would be dispatched to the **do_load()** function.

Future DCP Implementations

There is potential for a single, shared SCSI-2 DCP. An additional device module would be required only for vendor-dependent processing, or for departures from the standard.

More thought and changes to ADI and the DCP framework are needed to support non host-attached devices, such as broadcast video.

The infrastructure developer's kit was developed on IRIX systems, and has yet to be ported to other platforms. The DCP framework does not yet support partitions.

Defined Tokens List

This section documents the predefined strings that are relevant to DCP development.

Cartridge Form Factors

For a list of predefined cartridge form factors, see "Cartridge Form Factors" on page 54.

Cartridge Types

Table 6-2 shows tokens used to describe media inside a cartridge.

Table 6-2 Predefined Media Type Tokens

Token	Product Name or Description
8mm-12m	12 meter 8 mm
8mm-60m	60 meter 8 mm
8mm-90m	90 meter 8 mm
8mm-112m	112 meter 8 mm
8mm-160m	160 meter 8 mm
mammoth	Exabyte mammoth
3480	IBM 3480
3490	IBM 3490
3490E	IBM 3490E
3495	IBM Magstar native
4480	STK Timberline native
4490	STK Redwood native
DLT2000	Quantum DLT2000
DLT2000XT	Quantum DLT2000XT
DLT4000	Quantum DLT4000
DLT7000	Quantum DLT7000
DDS1	DAT 60 meter
DDS2	DAT 90 meter
DDS3	DAT 120 meter
D2-S	Ampex DST-310 small format
D2-M	Ampex DST-310 medium format
D2-L	Ampex DST-310 165GB large format
DTF	Sony GY-10
QIC	Quarter-inch cartridge tape
ISO9660	CD-ROM

Media Bit Formats

The format of bits recorded on media is independent of external cartridge appearance. One well-known case is the Exabyte 8200 versus Exabyte 8500 format, both being recorded on 8 mm media.

Table 6-3 shows tokens for each bit format, what form factors use it, and a description of how the format is generated.

Table 6-3 Predefined Bit Format Tokens

Token	Form Factor	Description
8200	8 mm	Exabyte 8200 native
8200c	8 mm	Exabyte 8200 compressed
8500	8 mm	Exabyte 8500 native
8500c	8 mm	Exabyte 8500 compressed
mammoth	8 mm	Exabyte mammoth native
mammothc	8 mm	Exabyte mammoth compressed
3480	3480	3480 native
3490	3480	3490 native
3490E	3480	3490E native
3495	3480	IBM Magstar native
4480	3480	STK Timberline native
4490	3480	STK Redwood native
DLT2000	DLT	DLT2000 native
DLT2000c	DLT	DLT2000 compressed
DLT4000	DLT	DLT4000 native
DLT4000c	DLT	DLT4000 compressed
DLT7000	DLT	DLT7000 native
DLT7000c	DLT	DLT7000 compressed
DDS1	DAT	Digital data storage 1.3 GB
DDS2	DAT	Digital data storage 2.0 GB
DDS3	DAT	Digital data storage 4.0 GB
D2	D2-[SML]	Ampex® DST-310

Table 6-3 (continued) Predefined Bit Format Tokens

Token	Form Factor	Description
DTF	DTF	Sony GY-10
QIC80	QIC	Quarter-inch cartridge 80 MB
QIC100	QIC	Quarter-inch cartridge 100 MB
QIC150	QIC	Quarter-inch cartridge 150 MB
QIC525	QIC	Quarter-inch cartridge 525 MB
QIC1024	QIC	Quarter-inch cartridge 1024 MB
ISO9660	CD-ROM	DOS-like (8.3) filesystem on CD-ROM

Drive Capabilities

OpenVault assumes that there is a default set of drive capabilities. Table 6-4 shows the tokens that describe changes from a standard drive.

Table 6-4 Predefined Mount Tokens

Token	Description
read	The mount point does not allow writing to the media
write	The mount point allows writing to the media
rewind	Rewind the media on close of the mount point
compression	Attempt compression of the data stream
fixedblock	Blocks on the media are a fixed size
variable	Blocks on the media are variable sized
status	A status-only mount point is also created (in a directory created for the session)
audio	Mount point allows playing audio data from media (often unimplemented)

Drive capabilities are entirely extensible, so this list is not exhaustive.

Partition Names

The ADI interface assumes that there is a standard set of names used for partitioned media. Table 6-5 shows the tokens used for naming partitions.

Table 6-5 Predefined Partition Name Tokens

Token	Description
PART 1	The first partition on the media. For magneto-optical or two-sided optical disc, this would be side one or side A.
PART 2	The second partition on the media. On linear media such as a tape, PART 2 immediately follows PART 1. On non-linear media such as a disk, PART 2 is the second-lowest numbered or lettered partition. Note that PART 2 does not refer to the next partition that is in use, it refers to the next partition.

Attribute Names (DCP)

Table 6-6 shows attributes used in OpenVault, where they are used, and what they mean.

Table 6-6 Predefined Attribute Name Tokens (DCP)

Attribute Name	Where Used	Possible Values	Required?	Description
ReadBandwidth	ADI config command, perf clause	numeric, in bytes per second	yes	The total effective bandwidth that an application should be able to sustain when reading from that drive using the given capability set.
WriteBandwidth	ADI config command, perf clause	numeric, in bytes per second	yes	The total effective bandwith that an application should be able to sustain when writing to that drive using the given capability set.
Capacity	ADI config command, perf clause	numeric, in bytes	yes	The total storage capacity of the cartridge that an application should be able to expect when accessing that drive using the given capability set.

Table 6-6 (continued) Predefined Attribute Name Tokens (DCP)

Attribute Name	Where Used	Possible Values	Required?	Description
BlockSize	ADI config command, perf clause	numeric, in bytes	yes	The I/O size that would best use the drive/cartridge combination with that drive with the given capability set.
LoadTime	ADI config command, perf clause	numeric, in seconds	yes	The number of seconds between the time a cartridge is first inserted into a drive and the time that the drive is ready to read/write data.
SlotTypeName	ADI config command, config clause	Cartridge FormFactor token (see Table 4-2)	yes	A supported form factor when the drive is using the given capability set.
CartridgeTypeName	ADI config command, config clause	MediaType token	yes	A supported media type, usually indicating tape length.
BitFormat	ADI config command, config clause	Bit Format token	yes	A supported recording format when the drive is using the given capability set.
NominalLoad	ALI config command, perf clause	numeric, in seconds	yes	Approximate time it takes for the library to move a cartridge from its home location to a drive, or back, not including drive load/unload time. This is analogous to "nominal seek time" of a disk drive. It is defined as the total real time to execute a large number of cartridge move-load operations randomly spread through the physical space of a library, divided by the number of such operations performed.

Appendix A

Sample Implementations

This appendix tells where to find sample code for an LCP or a DCP, and describes how to make and test the OpenVault source code.

LCP Sample Code

The sample code in the directories under *ovsrc/clients/lcp* might give you an idea of how to code an LCP for a new removable media library (*ovsrc* depends on where you installed the OpenVault developer's kit).

Source code outside the *ovsrc/clients/lcp* hierarchy is not really important to you, because the SCSI framework, underlying communication and authentication layer, ALI parser, and ALI/R generator are all integrated into the developer's framework.

Odetics ATL 2640

Working source code for the Odetics ATL 2640 autochanger is in the following directory:

ovsrc`/clients/lcp/ATL2640`

Exabyte SCSI Media Changers

Working source code for the Exabyte 210, 220, 440, and 480 is in the following directory:

ovsrc`/clients/lcp/EXABYTE-210`

DCP Sample Code

The sample code in the directories under *ovsrc/clients/dcp* might give you an idea of how to code a DCP for a new removable media library.

Source code outside the *ovsrc/clients/dcp* hierarchy is not really important to you, because the SCSI framework, underlying communication and authentication layer, ADI parser, and ADI/R generator are all provided by the developer's framework.

DLT 2000

Working source code for the Quantum DLT 2000 drive is in the following directory:

ovsrc`/clients/dcp/DLT2000`

Exabyte 8505XL

Working source code for the Exabyte 8505 XL drive is in the following directory:

ovsrc`/clients/dcp/EXB-8505XL`

Compiling and Installing OpenVault

The OpenVault source code distribution can be copied anywhere onto a system with at least 125 MB of free disk space. On IRIX systems the c_dev and compiler_dev product images must be installed. On other systems you need a C compiler, standard libraries, and C build tools. The GNU Flex parser and Bison lexical compiler are also required.

Tip: If your IRIX system does not have "n32" libraries, install the *c_eoe.sw32.lib* and *compiler_eoe.sw32.lib* subsystems to make the compiler options **-n32** and **-mips3** work. On other systems you must edit the *make.defs* file to provide correct CC and LD options.

To compile and install the OpenVault programs, run the *make install* command in the top-level source code directory:

```
% cd OVsrc
% make install
```

This installation does not modify any directories outside the OpenVault source tree. To uninstall OpenVault, run the *make clobber* command.

Running and Testing OpenVault

When the source code build finishes, change to the *install.pseudo* directory and run the pre-configured synthetic OpenVault system:

```
% cd install.pseudo
% ./STARTUP
```

The ./STARTUP script brings up nine windows, each in a different directory. There should be one binary executable in each directory. Run the binary that is in each window. Start with the lower left window, the one labelled MLM-CORE, and run *ovcore*. Then run the other binaries in any order. Cartridges refuse to mount unless the correct subset of binaries is running, so you should probably start them all. The two windows on the lower right contain a set of administrative command-line utilities (*ov_stat* and so forth); the last one is the user mount shell, *umsh*.

On IRIX systems you can use the right mouse button to "clone" any of the existing windows and get the same environment. You can use the *testclient* command to run random tests, or you can turn on verbose mode in the administration commands (using the **-v** option) to see what AAPI commands they send and receive from OpenVault.

The top-level source code directory contains several *testenv.** scripts. Each generates an *install.** directory containing a pre-configured environment for that machine. Currently only "xfs8" and "pseudo" are provided.

Appendix B

Return Values and Ready States

This appendix lists error codes and response types, then discusses ready state processing.

ALI Error and Return Values

The following list shows the error codes for an LCP:

```
#define ALI_E_NOSLOT    "ALI_E_NOSLOT"    /* unknown slot */
#define ALI_E_NOPCL     "ALI_E_NOPCL"     /* unknown PCL */
#define ALI_E_NOBAY     "ALI_E_NOBAY"     /* unknown bay */
#define ALI_E_NODRIVE   "ALI_E_NODRIVE"   /* unknown drive */
#define ALI_E_NOATTR    "ALI_E_NOATTR"    /* unknown attribute */
#define ALI_E_NOTYPE    "ALI_E_NOTYPE"    /* unknown type */
#define ALI_E_NOCMD     "ALI_E_NOCMD"     /* unknown command */
#define ALI_E_NOTASK    "ALI_E_NOTASK"    /* unknown task ID */
#define ALI_E_ACCESS    "ALI_E_ACCESS"    /* access denied or object inaccessible */
#define ALI_E_BADVAL    "ALI_E_BADVAL"    /* bad attribute value */
#define ALI_E_SRCFULL   "ALI_E_SRCFULL"   /* source location full */
#define ALI_E_SRCEMPTY  "ALI_E_SRCEMPTY"  /* source location empty */
#define ALI_E_DSTFULL   "ALI_E_DSTFULL"   /* destination location full */
#define ALI_E_DSTEMPTY  "ALI_E_DSTEMPTY"  /* destination location empty */
#define ALI_E_AGAIN     "ALI_E_AGAIN"     /* retry recommended */
#define ALI_E_READY     "ALI_E_READY"     /* target not ready */
#define ALI_E_PCL       "ALI_E_PCL"       /* PCL mismatch */
#define ALI_E_SEQUENCE  "ALI_E_SEQUENCE"  /* command sequence error */
#define ALI_E_ABORT     "ALI_E_ABORT"     /* command aborted by LCP */
#define ALI_E_LIBRARY   "ALI_E_LIBRARY"   /* library or device driver failure */
#define ALI_E_SHAPE     "ALI_E_SHAPE"     /* cartridge-drive fungibility error */
```

The following list shows the response types for ALI response:

```
ALI_response_accepted,          /* command queued */
ALI_response_unacceptable       /* command not queued */
ALI_response_success,           /* command worked */
ALI_response_error,             /* command failed */
ALI_response_cancelled          /* command cancelled */
```

ADI Error and Return Values

The following list shows the error codes for a DCP:

```
#define ADI_E_PART     "ADI_E_PART"     /* unknown or unsupported partition */
#define ADI_E_MODE     "ADI_E_MODE"     /* unknown or unsupported mode */
#define ADI_E_HANDLE   "ADI_E_HANDLE"   /* unknown or in use handle */
#define ADI_E_NOATTR   "ADI_E_NOATTR"   /* unknown attribute */
#define ADI_E_NOTYPE   "ADI_E_NOTYPE"   /* unknown type */
#define ADI_E_NOCMD    "ADI_E_NOCMD"    /* unknown command */
#define ADI_E_NOTASK   "ADI_E_NOTASK"   /* unknown task ID */
#define ADI_E_ACCESS   "ADI_E_ACCESS"   /* access denied or object inaccessible */
#define ADI_E_BADVAL   "ADI_E_BADVAL"   /* bad attribute value */
#define ADI_E_AGAIN    "ADI_E_AGAIN"    /* retry recommended */
#define ADI_E_READY    "ADI_E_READY"    /* target not ready */
#define ADI_E_SEQUENCE "ADI_E_SEQUENCE" /* command sequence error */
#define ADI_E_DRIVE    "ADI_E_DRIVE"    /* drive or device failure */
```

The following list shows the return values for ADI response:

```
ADI_response_accepted,          /* command queued */
ADI_response_unacceptable,      /* command not queued */
ADI_response_success,           /* command worked */
ADI_response_error,             /* command failed */
ADI_response_cancelled          /* command cancelled */
```

Ready States

Ready state describes the condition of the OpenVault connection with a device. Whenever the ready state changes, the library or drive control program should save changes and also send them to the MLM server, by means of the *ready* command.

When the control program is in "ready yes" state, that means it can talk to its device. If not in this state, the control program can still accept ALI or ADI commands, but will fail to execute any ALI or ADI commands requiring that it to talk to its device.

The following terms define state for both libraries and drives, defining how changes in the underlying device and API state can affect control-program ready status.

Device connected — The control program can communicate with its device by means of the formal device API.

Device not connected	The control program cannot communicate with its device by means of the formal device API.
Device online	The control program has a connection to its device, and the device is able to accept commands.
Device not online	The control program has a connection to its device, but the device is unable to accept commands because it is in some unusable state. (For a library, controller software might be down, and hardware might be offline, or in diagnostic state.)
Device ready	The control program has a connection to its device, which reports “device online” and is ready to accept commands.
Device not ready	The control program has a connection to the device, which reports “device online” but is temporarily not ready to accept commands.

Ready State Transition Rules

Table B-1 describes the initial ready states, the actions that trigger them to change, the new ready state for each condition, and the control program action for state transitions (not including the need to send ready state to the MLM server for each transition).

Table B-1 Ready State Transitions

Initial State	Action Triggering Change	New State	Control Program Action
lost	MLM server sends “activate enable” command. Control program is unable to connect to device.	lost	
lost	MLM server sends “activate enable” command. Control program is able to connect to device and finds it online and ready.	yes	Get device state and send full *config* command to the MLM server.
lost	MLM server sends “activate enable” command. Control program is able to connect to device but finds it online not ready.	no	
lost	MLM server sends “activate enable” command. Control program is able to connect to device but finds device not online.	broken	

Table B-1 (continued) Ready State Transitions

Initial State	Action Triggering Change	New State	Control Program Action
yes	MLM server sends "activate disable" or "exit" command to control program, or control program finds that its connection to device is lost.	lost	Stop communicating with device. Force pending device requests to completion, or cancel. Forget device state. If exiting, force pending ALI or ADI commands to completion, or cancel them, and complete or abort pending ALI/R or ADI/R commands. Also do shutdown processing.
yes	MLM server sends "activate enable" to control program.	yes	Resend full *config* command to the MLM server.
yes	Control program about to send command to device that will effectively block or reject all other commands to device until this one completes, or control program finds device is online but not ready.	no	
yes	Control program finds that device is not online.	broken	Stop communicating with device. Force pending device requests to completion, or cancel. Forget device state.
no	MLM server sends "activate disable" or "exit" to control program, or control program finds its connection to device is lost.	lost	Stop communicating with device. Force pending device requests to completion, or cancel. Forget device state.
no	A device command issued by the control program that effectively blocked all other device commands has now completed, or the control program finds that the device is now online and ready.	yes	
no	MLM server sends ALI or ADI command to control program that requires use of the device.	no	

Table B-1 (continued) Ready State Transitions

Initial State	Action Triggering Change	New State	Control Program Action
no	Control program finds that device is not online.	broken	Stop communicating with device. Force pending device requests to completion, or cancel. Forget device state.
broken	MLM server sends “activate disable” or “exit” to control program, or control program finds its connection to device is lost.	lost	
broken	Control program finds its device is online and ready.	yes	
broken	Control program finds its device is online, but not ready.	no	
broken	MLM server sends ALI or ADI command to control program that requires use of the device.	broken	

Ready State Responses

The MLM action in response to control program ready state changes are as follows:

yes — The control program can be selected for use. May not activate another control program for the same device until this one is disabled.

no — Temporarily do not send ALI or ADI commands that require device access to the control program. May not activate another control program for the same device until this one is disabled.

broken — The device associated with control program has failed. Do not try to activate another control program for this device, because the device itself is broken. Some recovery technique is needed, such as notifying the operator to take corrective action. For instance, the operator can choose to disable the current control program and start a separate one in manual mode, or switch the current control program into manual mode.

lost — The control program is not ready for use. If no other control program is currently active for this device, the MLM server may try to activate this or a different control program for the device, as needed.

These ASCII tokens are associated with each ready state:

lost	“lost”
yes	“”
broken	“broken”
no	“not”

The following list gives more information about control program actions in response to ready state changes:

- Once it has established a connection with the MLM server, a control program should initialize its ready state to *lost*, and send this to the server.
- Once it has established a connection with the MLM server, a control program should accept and process ALI or ADI commands. If it is in *ready lost*, *no*, or *broken* state, and it receives a command that requires it to access its device, then the control program should resend its ready state to the server and fail the command with a *ready* error (for example, ALI_E_READY or ADI_E_READY).

  The exception to this is that the LCP should process *activate enable*, as usual, if in *ready lost* or *broken* state.
- If a control program is already in *ready yes* state, and receives another *activate enable* command, it should resend its full configuration, including its ready state, and send a success response to the server.
- Before transitioning to *ready lost* or *broken* state, a control program must process all pending ALI or ADI commands to completion, either by normal completion along with the appropriate response, or by aborting commands that it cannot complete along with a *cancelled* response.

Appendix C

LCP and DCP Syntax

This appendix documents ALI and ADI syntax, expressed in abstract form. Words in bold font represent literals, as do square brackets and semicolons. Words in regular font are substitutable syntax elements.

ALI Syntax Specification

The MLM server communicates with an LCP using the abstract library interface (ALI), while the LCP communicates with the MLM server using ALI response (ALI/R).

ALI Language

Table C-1 provides a syntax specification for the ALI language.

Table C-1 ALI Language Syntax

Syntactic Element	Valid Syntax Statements
commands	mountStmt unmountStmt moveStmt ejectStmt openportStmt scanStmt activateStmt barrierStmt resetStmt exitStmt attributeStmt showStmt cancelStmt responseStmt
mountStmt	**mount** mountArgs ;

Table C-1 (continued) ALI Language Syntax

Syntactic Element	Valid Syntax Statements
mountArgs	/* empty */ **task** [string] mountArgs **drive** [string] mountArgs **slot** [string string string] mountArgs
unmountStmt	**unmount** unmountArgs ;
unmountArgs	/* empty */ **task** [string] unmountArgs **drive** [string] unmountArgs **slotid** [string] unmountArgs **any** unmountArgs
moveStmt	**move** moveArgs ;
moveArgs	/* empty */ **task** [string] moveArgs **from** [string string] moveArgs **to** [string] moveArgs
ejectStmt	**eject** ejectArgs ;
ejectArgs	/* empty */ **task** [string] ejectArgs **slot** [string string] ejectArgs
scanStmt	**scan** scanArgs ;
scanArgs	/* empty */ **task** [string] scanArgs **all** scanArgs **from** [string] scanArgs **to** [string] scanArgs
openportStmt	**openport task** [string] ;
activateStmt	**activate** activateArgs ;
activateArgs	/* empty */ **task** [string] activateArgs **enable** activateArgs **disable** activateArgs
barrierStmt	**barrier** task [string] ;

Table C-1 (continued) ALI Language Syntax

Syntactic Element	Valid Syntax Statements
resetStmt	**reset** task [string] ;
exitStmt	**exit** task [string] ;
attributeStmt	**attribute** attributeArgs ;
attributeArgs	/* empty */ **task** [string] attributeArgs **type** [string] attributeArgs **name** [string] attributeArgs **set** [string string] attributeArgs **unset** [string] attributeArgs
showStmt	**show** showArgs ;
showArgs	/* empty */ **task** [string] showArgs **type** [string] showArgs **name** [string] showArgs **report** [listOfStrings] showArgs **reportmode** [string] showArgs
cancelStmt	**cancel** cancelArgs ;
cancelArgs	/* empty */ **task** [string] cancelArgs **whichtask** [string] cancelArgs
responseStmt	**response** responseArgs ;
responseArgs	/* empty */ **whichtask** [string] responseArgs **accepted** responseArgs **unacceptable** responseArgs **success** responseArgs **error** [string] responseArgs **cancelled** responseArgs **text** [listOfStrings] responseArgs
listOfStrings	/* empty */ STRING listOfStrings
string	STRING

ALI/R Language

Table C-2 provides a syntax specification for the ALI/R language.

Table C-2 ALI/R Language Syntax

Syntactic Element	Valid Syntax Statements
commands	responseStmt messageStmt configStmt readyStmt attributeStmt showStmt cancelStmt
messageStmt	**message** messageArgs ;
messageArgs	/* empty */ **task** [string] messageArgs **who** [string] messageArgs **severity** [string] messageArgs **text** [listOfStrings] messageArgs
configStmt	**config** configArgs ;
configArgs	/* empty */ **task** [string] configArgs **scope** [string] configArgs **slot** [string string string string string string] configArgs **bay** [string string] configArgs **drive** [string string string string string] configArgs **freeslots** [string string string] configArgs **delslots** [string] configArgs **perf** [string string] configArgs
readyStmt	**ready** readyArgs ;
readyArgs	/* empty */ **task** [string] readyArgs **disconnected** readyArgs **broken** readyArgs **not** [listOfStrings] readyArgs
attributeStmt	**attribute** attributeArgs ;

Table C-2 (continued) ALI/R Language Syntax

Syntactic Element	Valid Syntax Statements
attributeArgs	/* empty */ **task** [string] attributeArgs **type** [string] attributeArgs **name** [string] attributeArgs **set** [string string] attributeArgs **unset** [string] attributeArgs
showStmt	**show** showArgs ;
showArgs	/* empty */ **task** [string] showArgs **type** [string] showArgs **name** [string] showArgs **report** [listOfStrings] showArgs **reportmode** [string] showArgs
cancelStmt	**cancel** cancelArgs ;
cancelArgs	/* empty */ **task** [string] cancelArgs **whichtask** [string] cancelArgs
responseStmt	**response** responseArgs ;
responseArgs	/* empty */ **whichtask** [string] responseArgs **accepted** responseArgs **unacceptable** responseArgs **success** responseArgs **error** [string] responseArgs **cancelled** responseArgs **text** [listOfStrings] responseArgs
listOfStrings	/* empty */ STRING listOfStrings
string	STRING

ADI Syntax Specification

The MLM server communicates with a DCP using the abstract drive interface (ADI), while the DCP communicates with the MLM server using ADI response (ADI/R).

ADI Language

Table C-3 provides a syntax specification for the ADI language.

Table C-3 ADI Language Syntax

Syntactic Element	Valid Syntax Statements
commands	attachStmt detachStmt loadStmt unloadStmt activateStmt resetStmt exitStmt attributeStmt showStmt cancelStmt responseStmt
attachStmt	**attach** attachArgs ;
attachArgs	/* empty */ **task** [string] attachArgs **modename** [string] attachArgs **drivehandle** [string] attachArgs **partition** [string] attachArgs
detachStmt	**detach** detachArgs ;
detachArgs	/* empty */ **task** [string] detachArgs **drivehandle** [string] detachArgs
loadStmt	**load task** [string] ;
unloadStmt	**unload task** [string] ;
activateStmt	**activate** activateArgs ;

Table C-3 (continued) ADI Language Syntax

Syntactic Element	Valid Syntax Statements
activateArgs	/* empty */ **task** [string] activateArgs **enable** activateArgs **disable** activateArgs
resetStmt	**reset task** [string] ;
exitStmt	**exit task** [string] ;
attributeStmt	**attribute** attributeArgs ;
attributeArgs	/* empty */ **task** [string] attributeArgs **type** [string] attributeArgs **name** [string] attributeArgs **set** [string string] attributeArgs **unset** [string] attributeArgs
showStmt	**show** showArgs ;
showArgs	/* empty */ **task** [string] showArgs **type** [string] showArgs **name** [string] showArgs **report** [listOfStrings] showArgs **reportmode** [string] showArgs
cancelStmt	**cancel** cancelArgs ;
cancelArgs	/* empty */ **task** [string] cancelArgs **whichtask** [string] cancelArgs
responseStmt	**response** responseArgs ;
responseArgs	/* empty */ **whichtask** [string] responseArgs **accepted** responseArgs **unacceptable** responseArgs **success** responseArgs **error** [string] responseArgs **cancelled** responseArgs **text** [listOfStrings] responseArgs

Table C-3 (continued) ADI Language Syntax

Syntactic Element	Valid Syntax Statements
listOfStrings	/* empty */ string listOfStrings
string	STRING

ADI/R Language

Table C-4 provides a syntax specification for the ADI/R language.

Table C-4 ADI/R Language Syntax

Syntactic Element	Valid Syntax Statements
commands	configStmt messageStmt readyStmt attributeStmt showStmt cancelStmt responseStmt
configStmt	**config** configArgs ;
configArgs	/* empty */ **task** [string] configArgs **scope** [string] configArgs **config** [string] configArgs **cap** [string configCapArgs] configArgs
configCapArgs	/* empty */ **attr** [string string] configCapArgs **caplist** [listOfStrings] configCapArgs
messageStmt	**message** messageArgs ;
messageArgs	/* empty */ **task** [string] messageArgs **who** [string] messageArgs **severity** [string] messageArgs **text** [listOfStrings] messageArgs

Table C-4 (continued) ADI/R Language Syntax

Syntactic Element	Valid Syntax Statements
readyStmt	**ready** readyArgs ;
readyArgs	/* empty */ **task** [string] readyArgs **disconnected** readyArgs **not** [listOfStrings] readyArgs
attributeStmt	**attribute** attributeArgs ;
attributeArgs	/* empty */ **task** [string] attributeArgs **type** [string] attributeArgs **name** [string] attributeArgs **set** [string string] attributeArgs **unset** [string] attributeArgs
showStmt	**show** showArgs ;
showArgs	/* empty */ **task** [string] showArgs **type** [string] showArgs **name** [string] showArgs **report** [listOfStrings] showArgs **reportmode** [string] showArgs
cancelStmt	**cancel** cancelArgs ;
cancelArgs	/* empty */ **task** [string] cancelArgs **whichtask** [string] cancelArgs
responseStmt	**response** responseArgs ;
responseArgs	/* empty */ **whichtask** [string] responseArgs **accepted** responseArgs **unacceptable** responseArgs **success** responseArgs **error** [string] responseArgs **cancelled** responseArgs **text** [listOfStrings] responseArgs

Table C-4 (continued) ADI/R Language Syntax

Syntactic Element	Valid Syntax Statements
listOfStrings	/* empty */ STRING listOfStrings
string	STRING

Glossary

ALI and ALI/R

Abstract library interface and ALI response, languages for communicating between the media library manager (MLM) server and a library control program.

ADI and ADI/R

Abstract drive interface and ADI response, languages for communicating between the media library manager (MLM) server and a drive control program.

barcode

A machine-readable representation of a physical cartridge label (PCL).

barcode reader

A laser-optical reader that scans a barcode and then uses logic to translate from a scanned barcode to a human-readable representation, such as volume serial number.

bay

A physical grouping of slots in a common unit of housing where cartridges are stored. Usually a bay contains storage locations for cartridges, optional drives, and one or more transfer agents to move cartridges around.

cartridge

A cartridge is the unit of physical operation and management within a library. A cartridge contains one or more pieces of media, and has a certain form factor. The most common forms of cartridge are for magnetic tape and laser- and magneto- optical disk.

DCP (drive control program)

An OpenVault software component that mediates between the media library manager (MLM) server and the actual drive control interface.

drive

A magnetic or optical device for accessing media inside a cartridge mounted in a slot.

LCP (library control program)

An OpenVault software component that mediates between the media library manager (MLM) server and the actual library control interface.

partition

A region on the recording surface of a piece of media that has a physical beginning and ending that can be accessed by a drive. Typically, each piece of media has a single partition, which spans the entire recordable surface of the media. However, there are drives that support partitioning of this recordable surface, such as DDS2 and D2 tape, such that a single piece of media may contain multiple partitions.

PCL (physical cartridge label)

Some form of identification on the outside of the cartridge, as opposed to being stored on media inside the cartridge. A PCL may contain a machine-readable label (barcode), but it must also contain a human-readable text portion.

port

A door or opening where cartridges may be inserted into or removed from the library.

removable media library

A robotic device (usually) with storage slots and drives for accessing multiple cartridges.

side

For tape cartridges containing one piece of recording media, with all recording surfaces accessible when loaded in a drive, the cartridge contains one side. For a multi-sided cartridge, access to a side requires that the cartridge be mounted in a drive with a particular orientation (for side A of optical disk, the cartridge must be positioned for mount with side A up).

slot

A storage location for a cartridge, with a form factor that determines which kinds of cartridges it can hold.

slotmap

A persistent table associated with a single library. For each cartridge contained by that library, this table maps the physical cartridge label (PCL) to a slot within the library.

Index

B

C

I

L

M

N

O

P

Q

R

S

T

U

V

W

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3305-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389