Vault
Programmer Guide

Optegra® Release 6

DOC35288-018

Parametric Technology Corporation

Copyright © 2001 Parametric Technology Corporation. All Rights Reserved.
User documentation from Parametric Technology Corporation (PTC) is subject to copyright laws of the United States and other countries and is provided under a license agreement, which restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed user the right to make copies in printed form of PTC user documentation provided on software or documentation media, but only for internal, noncommercial use by the licensed user in accordance with the license agreement under which the applicable software and documentation are licensed. Any copy made hereunder shall include the Parametric Technology Corporation copyright notice and any other proprietary notice provided by PTC. User documentation may not be disclosed, transferred, or modified without the prior written consent of PTC and no authorization is granted to make copies for such purposes.

Information described in this document is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION.

Registered Trademarks of Parametric Technology Corporation or a Subsidiary
Advanced Surface Design, CADDS, CADDShade, Computervision, Computervision Services, Electronic Product Definition, EPD, HARNESSDESIGN, Info*Engine, InPart, MEDUSA, Optegra, Parametric Technology, Parametric Technology Corporation, Pro/ENGINEER, Pro/HELP, Pro/INTRALINK, Pro/MECHANICA, Pro/TOOLKIT, PTC, PT/Products, Windchill, InPart logo, and PTC logo.

Trademarks of Parametric Technology Corporation or a Subsidiary
3DPAINT, Associative Topology Bus, Behavioral Modeler, BOMBOT, CDRS, CounterPart, CV, CVact, CVaec, CVdesign, CV-DORS, CVMAC, CVNC, CVToolmaker, DesignSuite, DIMENSION III, DIVISION, DVSAFEWORK, DVS, e-Series, EDE, e/ENGINEER, Electrical Design Entry, Expert Machinist, Expert Toolmaker, Flexible Engineering, *i*-Series, ICEM, Import Data Doctor, Information for Innovation, ISSM, MEDEA, ModelCHECK, NC Builder, Nitidus, PARTBOT, PartSpeak, Pro/ANIMATE, Pro/ASSEMBLY, Pro/CABLING, Pro/CASTING, Pro/CDT, Pro/CMM, Pro/COMPOSITE, Pro/CONVERT, Pro/DATA for PDGS, Pro/DESIGNER, Pro/DESKTOP, Pro/DETAIL, Pro/DIAGRAM, Pro/DIEFACE, Pro/DRAW, Pro/ECAD, Pro/ENGINE, Pro/FEATURE, Pro/FEM-POST, Pro/FLY-THROUGH, Pro/HARNESS-MFG, Pro/INTERFACE for CADDS 5, Pro/INTERFACE for CATIA, Pro/LANGUAGE, Pro/LEGACY, Pro/LIBRARYACCESS, Pro/MESH, Pro/Model.View, Pro/MOLDESIGN, Pro/NC-ADVANCED, Pro/NC-CHECK, Pro/NC-MILL, Pro/NC-SHEETMETAL, Pro/NC-TURN, Pro/NC-WEDM, Pro/NC-Wire EDM, Pro/NCPOST, Pro/NETWORK ANIMATOR, Pro/NOTEBOOK, Pro/PDM, Pro/PHOTORENDER, Pro/PHOTORENDER TEXTURE LIBRARY, Pro/PIPING, Pro/PLASTIC ADVISOR, Pro/PLOT, Pro/POWER DESIGN, Pro/PROCESS, Pro/REPORT, Pro/REVIEW, Pro/SCAN-TOOLS, Pro/SHEETMETAL, Pro/SURFACE, Pro/VERIFY, Pro/Web.Link, Pro/Web.Publish, Pro/WELDING, Product Structure Navigator, PTC *i*-Series, Shaping Innovation, Shrinkwrap, The Product Development Company, Virtual Design Environment, Windchill e-Series, CV-Computervision logo, DIVISION logo, and ICEM logo.

Third-Party Trademarks
Oracle is a registered trademark of Oracle Corporation. Windows and Windows NT are registered trademarks of Microsoft Corporation. Java and all Java based marks are trademarks or registered trademarks of Sun Microsystems, Inc. CATIA is a registered trademark of Dassault Systems. PDGS is a registered trademark of Ford Motor Company. SAP and R/3 are registered trademarks of SAP AG Germany. FLEX*lm* is a registered trademark of GLOBEtrotter Software, Inc. VisTools library is copyrighted software of Visual Kinematics, Inc. (VKI) containing confidential trade secret information belonging to VKI. HOOPS graphics system is a proprietary software product of, and copyrighted by, Tech Soft America, Inc. All other brand or product names are trademarks or registered trademarks of their respective holders.

UNITED STATES GOVERNMENT RESTRICTED RIGHTS LEGEND
This document and the software described herein are Commercial Computer Documentation and Software, pursuant to FAR 12.212(a)-(b) or DFARS 227.7202-1(a) and 227.7202-3(a), and are provided to the Government under a limited commercial license only. For procurements predating the above clauses, use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or Commercial Computer Software-Restricted Rights at FAR 52.227-19, as applicable.

Parametric Technology Corporation, 140 Kendrick Street, Needham, MA 02494-2714
8 January 2001

Table of Contents

Preface

Related Documents ____ xi
Book Conventions ____ xii
Online User Documentation ____ xiii
Printing Documentation ____ xiii
Resources and Services ____ xiv
Documentation Comments ____ xiv

Overview of Vault Programming

Introduction to Vault Programming ____ 1-2
Programmatic Interface ____ 1-2
Command Triggers ____ 1-2
Prerequisites for Using Vault Programming ____ 1-2
Licensing for Vault Programming ____ 1-3
Vault Commands ____ 1-5
Vault Programming Overview ____ 1-10
Programmatic Interface ____ 1-10
Ways to Use the Programmatic Interface ____ 1-10
Overview of Command Triggers ____ 1-11
A Way to Use Command Triggers before Command Processing ____ 1-12

Data Checks 1-12
Ways to Use Command Triggers after Command Processing 1-13
Restrictions and Cautions 1-13
Using Command Triggers with Programmatic Interface 1-14

Using the Programmatic Interface

Format of the Programmatic Interface 2-2
Introducing the SVedm Programmatic Interface 2-2
Using the Old Programmatic Interface 2-3
Writing a Program Using the Programmatic Interface 2-4
New Programmatic Interface (SVedm) 2-4
Old Programmatic Interface (cpdm) 2-4
Naming Conventions for Structures and Header Files 2-4
Defining the Programmatic Interface Structures 2-5
Control Structure 2-5
Input Structure 2-6
Output Structure 2-7
Running a Sample Program (New SVedm Interface) 2-8
Special Considerations for Local Vault Commands 2-9
Compiling, Linking, and Running a Program 2-10
Compiling the Program 2-10
Linking the Program 2-10
Running the Program 2-11

Compiling and Linking a Programmatic Interface Program

Linking a Program on SGI IRIX 6.5 3-2
Linking on IRIX 6.5 — Client Only 3-2
Linking on IRIX 6.5— Vault Only 3-2
Linking on IRIX 6.5 — Vault with Oracle 8i Release 3 (8.1.7) 3-2

Linking a Program on Compaq Tru64 UNIX 4.0E 3-3
Linking on Compaq Tru64 UNIX 4.0E — Client Only 3-3
Linking on Compaq Tru64 UNIX 4.0E — Vault Only 3-3
Linking on Compaq Tru64 UNIX 4.0E — Vault with Oracle 8i Release 3 (8.1.7) 3-3
Linking a Program on AIX 4.3.3 3-4
Linking on AIX 4.3.3 — Client Only 3-4
Linking on AIX 4.3.3 — Vault Only 3-4
Linking on AIX 4.3.3 — Vault with Oracle 8i Release 3 (8.1.7) 3-4
Linking a Program on HP-UX 11 3-5
Linking on HP-UX 11— Vault Client (Locator) Only 3-5
Linking on HP-UX 11— Vault Only 3-5
Linking on HP-UX 11— Vault with Oracle 8i Release 3 (8.1.7) 3-6
Linking a Program on Solaris 2.6 3-7
Linking on Solaris 2.6 — Vault Client (Locator) Only 3-7
Linking on Solaris 2.6 — Vault Only 3-7
Linking on Solaris 2.6— Vault with Oracle 8i Release 3 (8.1.7) 3-7
Linking a Program on Windows NT 4.0 (With Service Pack 5) 3-8

Using Command Triggers

Command-Trigger Format 4-2
Developing a Command-Triggered Program 4-3
Command-Trigger Structure 4-4
Output from Triggered Process to the Vault 4-6
Command-Trigger Communications Subroutines 4-7
Connect to the Vault 4-7
Wait for Trigger 4-7
Respond to Trigger 4-8
Disconnect from the Vault 4-8
Modifying the NSM Configuration File 4-9

Compiling, Linking, and Running the Program 4-10
Compiling the Program 4-10
Linking the Program 4-10
Running the Program 4-10
Using the chgctl Command 4-12
Modifying the Command-Trigger List 4-13
Using the Command-Line Format 4-14
Examples 4-14
Modifying the Sample Triggered-Program Commands 4-15

Compiling, Linking, and Running a Command-Triggered Program

Creating an Executable Startup Script File (UNIX) 5-2
Creating the File 5-2
Adding Executable Startup Script File to the nsm.config File 5-3
Compiling the Source Programs (UNIX) 5-4
Compiling and Linking a Command-Triggered Program 5-5
Linking the Command-Triggered Program (UNIX) 5-8
Running the Command-Triggered Program (UNIX) 5-9

Customizing the Client

Overview of the edmosrv Library 6-2
edmosrv Functions 6-4
SQL Functions 6-4
Return Codes 6-5
Procedure for SELECT 6-5
Procedure for UPDATES 6-6
Other Functions 6-6
Example 6-7
SCRAMBLE Library 6-8
Functions 6-8
Utilities 6-9

Customizing Through the Perl Interface 6-10
Perl Functions for the edmosrv Library 6-10
Perl Functions for the Scramble Library 6-12

Vault Commands

Command Overview 7-2
Vault Command 7-3

Sample C Program

svedmsample.c A-2

Using the Vault Rules Processor Language

Overview of the Vault Rules Processor Language B-2
Implementing Attributes B-4
Structure of the Language B-6
Variables B-6
Input and Output B-6
Statement Types Supported B-7
Creating Statements and Blocks B-7
Testing for Specific Conditions B-8
Writing Functions B-9
Using Built-in Functions B-10
SQL Statements B-10
Data Validation Statements B-11
Creating a Data Type B-12
Creating a Classification Function B-14

When Simple Classification Functions
Are Too Simple ____ B-16
The Language Specification ____ B-17
Supplied Data ____ B-19
Supplied Classified Functions ____ B-20
Compiling and Loading Rules and Data Types ____ B-23

Sample Command-Triggered Control Program

Sample Command-Triggered Control Program ____ C-2
ctshell.c ____ C-2
adctstr.h ____ C-6

Sample Command-Triggered Application Subroutine

ctsample.c ____ D-2
ctsample.h ____ D-5

Preface

Vault Programmer Guide provides information and instructions for using the Vault programmatic interface and command triggers. This book is for programmers writing applications that call Vault functions.

Use this book to write, compile, link, and run application programs that call Vault routines from the following operating systems:

- IBM AIX
- Compaq Tru64 UNIX
- HP-UX
- Solaris
- SGI IRIX
- Windows NT

Related Documents

The following documents may be helpful as you use *Vault Programmer Guide*:

- *Vault Interactive Query Facility Guide*
- *Vault Command Reference*

Book Conventions

The following table illustrates and explains conventions used in writing about Optegra applications.

Convention	Example	Explanation
EPD_HOME	cd $EPD_HOME/install (UNIX) cd %EPD_HOME%\install (Windows)	Represents the default path where the current version of the product is installed.
Menu selections	Vault > Check Out > Lock	Indicates a command that you can choose from a menu.
Command buttons and options	Mandatory check box, Add button, Description text box	Names selectable items from dialog boxes: options, buttons, toggles, text boxes, and switches.
User input and code	Wheel_Assy_details -xvf /dev/rst0 Enter command> **plot_config**	Enter the text in a text box or on a command line. Where system output and user input are mixed, user input is in bold.
System output	CT_struct.aename	Indicates system responses.
Parameter and variable names	tar -cvf /dev/rst0 filename	Supply an appropriate substitute for each parameter or variable; for example, replace filename with an actual file name.
Commands and keywords	The ciaddobj command creates an instance of a binder.	Shows command syntax.
Text string	"SRFGROUPA" or 'SRFGROUPA'	Shows text strings. Enclose text strings with single or double quotation marks.
Integer	n	Supply an integer for *n*.
Real number	x	Supply a real number for *x*.
#	# mkdir /cdrom	Indicates the root (superuser) prompt on command lines.
%	% rlogin remote_system_name -l root	Indicates the C shell prompt on command lines.
$	$ rlogin remote_system_name -l root	Indicates the Bourne shell prompt on command lines.
>	> copy filename	Indicates the MS-DOS prompt on command lines.
Keystrokes	Return or Control-g	Indicates the keys to press on a keyboard.

Online User Documentation

Online documentation for each Optegra book is provided in HTML if the documentation CD-ROM is installed. You can view the online documentation from an HTML browser or from the HELP command.

You can also view the online documentation directly from the CD-ROM without installing it.

From an HTML Browser:

1. Navigate to the directory where the documents are installed. For example,

   $EPD_HOME`/data/html/htmldoc/` (UNIX)

   %EPD_HOME%`\data\html\htmldoc\` (Windows NT)
2. Click `mainmenu.html`. A list of available Optegra documentation appears.
3. Click the book title you want to view.

From the HELP Command:

To view the online documentation for your specific application, click HELP. (Consult the documentation specific to your application for more information.)

From the Documentation CD-ROM:

1. Mount the documentation CD-ROM.
2. Point your browser to:

   CDROM_mount_point`/htmldoc/mainmenu.html` (UNIX)

   CDROM_Drive:`\htmldoc\mainmenu.html` (Windows NT)

Printing Documentation

A PDF (Portable Document Format) file is included on the CD-ROM for each online book. See the first page of each online book for the document number referenced in the PDF file name. Check with your system administrator if you need more information.

You must have Acrobat Reader installed to view and print PDF files.

The default documentation directories are:

- $EPD_HOME`/data/html/pdf/`doc_number`.pdf` (UNIX)
- %EPD_HOME%`\data\html\pdf\`doc_number`.pdf` (Windows NT)

Resources and Services

For resources and services to help you with PTC (Parametric Technology Corporation) software products, see the *PTC Customer Service Guide*. It includes instructions for using the World Wide Web or fax transmissions for customer support.

Documentation Comments

PTC welcomes your suggestions and comments. You can send feedback in the following ways:

- Send comments electronically to `doc-webhelp@ptc.com.`
- Fill out and mail the PTC Documentation Survey located in the *PTC Customer Service Guide.*

Chapter 1

Overview of Vault Programming

This chapter describes the programmatic interface and command triggers, shows how Vault Programming fits together with the Vault. It also lists the functions you can perform with Vault Programming.

- Introduction to Vault Programming
- Licensing for Vault Programming
- Vault Commands
- Vault Programming Overview

Introduction to Vault Programming

Vault programming provide programming tools that allows you to use Vault according to your requirements.

The Vault programming tools are the programmatic interface and command triggers. Though command triggers and the programmatic interface provide similar functions, they are independent of each other.

Please note: The Vault programmatic interface does not recognize control structures containing the release value PDM 4.2.0 or Vault 5.0.0. This means that previously compiled code cannot be relinked. You have to recompile and relink the application.

Programmatic Interface

The programmatic interface allows you to develop application programs that use the Vault functions within a standard programming environment. An application requiring a Vault service or function can make a program call to a supplied programmatic interface subroutine with the appropriate input arguments.

Command Triggers

Command triggers enable you to insert customized code within the Vault command processing. Your code is contained in a separate command-triggered program that the Vault invokes at the beginning or end of a Vault command.

You control which Vault commands have command triggers with the `chgctl` command.

Prerequisites for Using Vault Programming

To successfully use Vault Programming, you must be clear about what it is you want to accomplish. In addition, you must be knowledgeable about

- C programming language
- Vault command functions
- Host operating system

Please note: You can use Vault Programming with the C programming language.

Licensing for Vault Programming

The license manager works with an internal timer in the license client library. The client library sends a heartbeat to the license manager to indicate that the license has been checked out by an application.

The license timer does not work in the following cases:

- With sleep (3), pclose (3), system (3), SUNVIEW, and Xview
- If the application is using SIGPIPE and SIGALARM alarms

To resolve this problem, ensure that:

- The client application calls the timer routine `call_lm_timer()`.

  This routine is called once every 5 minutes. The routine is defined in the `cedmpi.a` and `edmpi.a` libraries provided for Vault programming.
- The timer routine is called only after the license is checked out.
- If customer applications use handlers for SIGPIPE and SIGALRM (as in the example below) and performing the `fork` or `exec`, the signals must be restored before the `fork` or `exec` is performed and then restored in the parent process.

Please note: If you have a problem calling `call_lm_timer` every five minutes, set a signal handling function SIGALARM. Then call the `call_lm_timer()` function from this signal-handling routine.

A sample application that needs to be merged in the code follows:

```
/* Beginning of the main.c file */

#include <.......h>
#include <.......h>

void sig_alarm(); /* This prototype calls the license heartbeat */

  main(argc, argv)
  int argc;
  char **argv;
  {

  /* Initialize all your variables here */
    /* send the signal for the first time */

      signal(SIGALRM, sig_alarm);

       /* This is set to 5 minutes ( 300 seconds)*/
       alarm(300);
```

```
rc = aw_init_windows(command_file); /* This is our (loop) call */

   /* The following disables the signal alarm before exit */
      alarm (0);

      exit( rc );
  }


  /* Following function needs to be added extra */

  void sig_alarm()
  {

  /* For printing. */
          printf( "sig_alarm called ......\n");

   call_lm_timer();

       /* Set the function and alarm for the next call */
          signal(SIGALRM, sig_alarm);
          alarm( 60 );
  }

  /* End of main file */
```

Vault Commands

You can use the following interfaces to execute Vault commands:

- Command-line format
- Vault Programming

Vault Programming is described in this book.

Please note: If your site does not have Vault, you cannot use the programmatic interface for Vault commands.

The following Vault commands do not have a programmatic interface:

- `addaset`
- `addattr`
- `addmas`
- `addrule`
- `chgmas`
- `delaset`
- `delattr`
- `delrule`
- `iqf`
- `remmas`
- `ubkup`

The following Vault commands do not have command triggers:

- `addadod`
- `addaset`
- `addattr`
- `addmas`
- `addrule`
- `addsub`
- `addvault`
- `chgctl`
- `chgmas`

- delaset
- delattr
- delrule
- delsub
- export
- import
- locate
- iqf
- remmas
- register
- remadod
- remvault
- signoff
- ubkup

The following table shows the Vault programmatic interface commands grouped by function. It lists the name, a three-letter abbreviation, the Vault server process that initiates the command trigger, and a description for each command.

Table 1-1 Programmatic Interface Commands for Command Triggers

Command Name	Command Abbr	Trigger Abbr	Description
User Validation Commands			
signoff	sof	n/a	Sign off from the Vault
signon	son	dn	Sign on to the Vault
User Maintenance Commands			
addu	adu	dn	Add user to the Vault
chgu	cua	dn	Change user attributes
chgupw	cpw	dn	Change user signon password
delu	dlu	dn	Delete a user from the Vault
Command List Maintenance Commands			
addcl	acl	dn	Add a command list to the Vault
chgcl	ccl	dn	Change a command list in the Vault
delcl	dcl	dn	Delete a command list from the Vault

Table 1-1 Programmatic Interface Commands for Command Triggers

Command Name	Command Abbr	Trigger Abbr	Description
Project Maintenance Commands			
addp	adp	dn	Add a project to the Vault
addup	aup	dn	Add a user to project
chgp	cpa	dn	Change project attributes
chgup	cup	dn	Change a project user's attributes
delp	dlp	dn	Delete a Vault project
remup	rup	dn	Delete a user from a project
Status Maintenance Commands			
adds	ads	dn	Add a status level to the Vault
chgs	csa	dn	Change a status level
dels	dls	dn	Delete a status level
Authority Group Maintenance Commands			
addag	aag	dn	Add an authority group to the Vault
admcopy	adm	dn	Copy administrative data
chgag	cag	dn	Change an authority group in the Vault
delag	dag	dn	Delete an authority group from the Vault
Revision Sequence Commands			
addrs	ars	dn	Add a revision sequence
delrs	drs	dn	Delete a revision sequence
Release/Revision Control Subsystem			
addusa	als	dn	Add a review and/or modification list
remusa	rls	dn	Delete a review and/or modification
reqrvw	rvw	dd	Request a review of a file
rsvp	rvp	dn	Respond to a review
Message Subsystem			
readmsg	rmg	dn	Read a message
sendmsg	smg	dn	Send a message

Table 1-1 Programmatic Interface Commands for Command Triggers

Command Name	Command Abbr	Trigger Abbr	Description
User List Commands			
addmul	aml	dn	Add a user list member
addul	aul	dn	Add a user list to the Vault
delul	dul	dn	Delete a user list
remmul	rml	dn	Remove a user list member
File Access Commands			
get	get	dd	Get files from the Vault for modification
listdir	dir	dd	Create a directory listing of the Vault files
read	rea	dd	Read files from the Vault
replace	rep	dd	Replace files in the Vault
reserve	rsv	dd	Reserve a file name in the Vault
reset	rst	dd	Reset files in the Vault
signout	sot	dd	Sign out files in the Vault
store	str	dd	Store a file in the Vault
update	upt	dd	Update files in the Vault
File Maintenance Commands			
chgfa	cfa	dd	Change file’s user-defined attributes
chgfcl	cfc	dd	Change file’s classification
chgfpw	cfp	dd	Change file’s password
chgfrev	cfr	dd	Change the revision of a Vault file
chgfsc	cfs	dd	Change the status of Vault files
chgfsp	fsp	dd	Change file’s storage pool
copy	cpy	dd	Copy files in the Vault
marka	mka	dd	Mark files for archive
markd	mkd	dd	Mark files for deletion
markr	mkr	dd	Mark files for restore
unmark	umk	dd	Unmark files

Table 1-1 Programmatic Interface Commands for Command Triggers

Command Name	Command Abbr	Trigger Abbr	Description
File Set Maintenance Commands			
addfs	afs	dn	Add a file set to the Vault
addmfs	ams	dn	Add members to a Vault file set
remfs	rfs	dn	Remove a file set
remmfs	rms	dn	Remove a file from a Vault file set
System Maintenance Commands			
archive	arc	local	Archive the Vault files onto tape
delete	del	dd	Delete marked files from the Vault
dellog	dlg	dn	Delete entries from the Vault log
load	loa	local	Load files from a FUTIL or SAVFIL
purge	pur	dd	Purge marked files from the Vault
restore	res	local	Restore archived files in the Vault
scantape	scn	local	List contents of a Vault tape
unload	unl	local	Unload files from a FUTIL tape
Database Maintenance Commands			
addsp	asp	local	Add a storage pool
chgsps	cps	dn	Change status of storage pool
chgspt	cpt	dn	Change type of storage pool
delov	dlv	dn	Delete old versions
ibkup	ibu	local	Incremental backup
recsf	rsf	local	Recover a single file
recsp	rsp	local	Recover a storage pool
User-Defined Table Commands			
addt	adt	local	Add a table to the control of the Vault
clst	clt	local	Close a table opened in the Vault
opnt	opt	local	Open a table controlled by the Vault
remt	rmt	local	Revoke a table from Vault control
Command Triggering			
chgctl	ctl	n/a	Change the command trigger list

Vault Programming Overview

Programmatic Interface

The programmatic interface provides you with the ability to call most Vault subroutines directly from a C program.

The programmatic interface bypasses the Vault screen and command interfaces, so the user interface can be customized.

Vault commands are called through a common entry point (`SVedm`), which is used for other Vault user interfaces. Therefore, the routines called by the programmatic interface are identical to the screen menu and command user interface routines and the same logic is used, including default field values and validity checks.

User applications that call the programmatic interface have the same architecture as the Locator process and can be local or remote to the Vault host, depending on the Vault command. The user-defined code must be linked with Locator libraries.

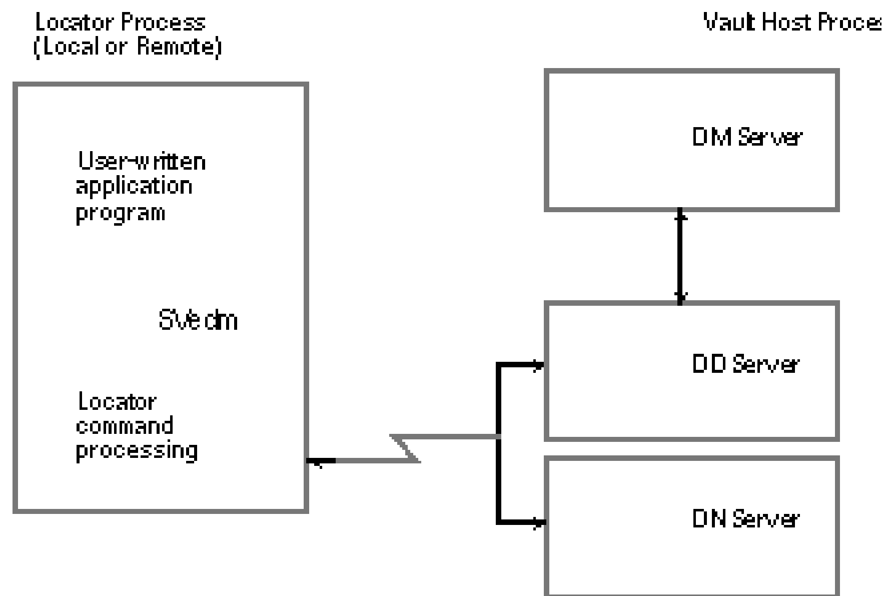
Ways to Use the Programmatic Interface

Use the programmatic interface to

- Create a custom interface to the Vault
- Create custom applications so they initiate Vault functions to
  - Use Vault facilities
  - Combine multiple Vault commands within one presentation to the user (for example, multiple files can be stored and a file set can be created for them with one user command)
  - Invoke one Vault command as a result of the success or failure of a previous command

The following figure illustrates the programmatic interface process in which a user-written application uses the `SVedm` entry point to access an Vault server:

Figure 1-1 Programmatic Interface Operation



Overview of Command Triggers

Command triggers allows you to develop application programs that augment Vault commands. An active command trigger calls a detached (unlinked) exit to perform user-written logic whenever the associated Vault command is executed. The exit is considered to be detached because it is not linked to Vault code.

The triggered process is code that you write in the C language. You can tell the Vault to trigger the process and wait for completion or trigger and proceed without waiting. You can request a command trigger before or after Vault command execution. You can set up triggered processes for all Vault commands except `ubkup`, `signoff`, `iqf`, `chgctl`, and the user-defined attribute commands.

You define which Vault commands include command triggers by using the `chgctl` command to maintain a command trigger list which is stored in the RDBMS. The command trigger list contains the following information:

- Each Vault command that has a command trigger process
- The name of each triggered process
- Trigger point information
- Whether a command trigger is active

Some of the functions where you can use command triggers are

- Augment data validation
- Log additional audit information
- Notify personnel affected by the outcome of a particular Vault command
- Synchronize your data with Vault data

A Way to Use Command Triggers before Command Processing

You can define a command trigger to invoke your process after the Vault performs an access and security check on the user issuing the Vault command but before it processes the command. The command triggers your process regardless of the result of the access and security check. A preprocess command trigger can prevent the Vault from continuing with the command, but it cannot change the input to the Vault command.

Your company may need security checks beyond those included in the Vault. You can define a command trigger before the Vault processes particular commands. A process is invoked that does the additional security checks.

When a security violation occurs, your process can stop the Vault command from being executed.

Data Checks

If your company has specific conventions for user IDs or file names or any other input parameter, you can write code that checks the data against your standards.

For example, many companies have strict conventions for file names. You could write a process that is triggered before the Vault processes the `store` or `reserve` command. Your process can check the Vault file name and prevent the file from

being stored in the database if the name does not comply with your conventions. Your process cannot change the data, but it can stop command execution.

Ways to Use Command Triggers after Command Processing

You can design a command trigger to be invoked after a Vault command has completed processing. While these triggers cannot alter the command just executed, the process can log information or notify appropriate users.

The list below suggests a number of ways command triggers can be useful after command processing.

- Send electronic mail (perhaps after a request for a review)
- Update statistics
- Log additional audit information
- Synchronize Vault data with your site-specific data
- Perform additional security checks

Restrictions and Cautions

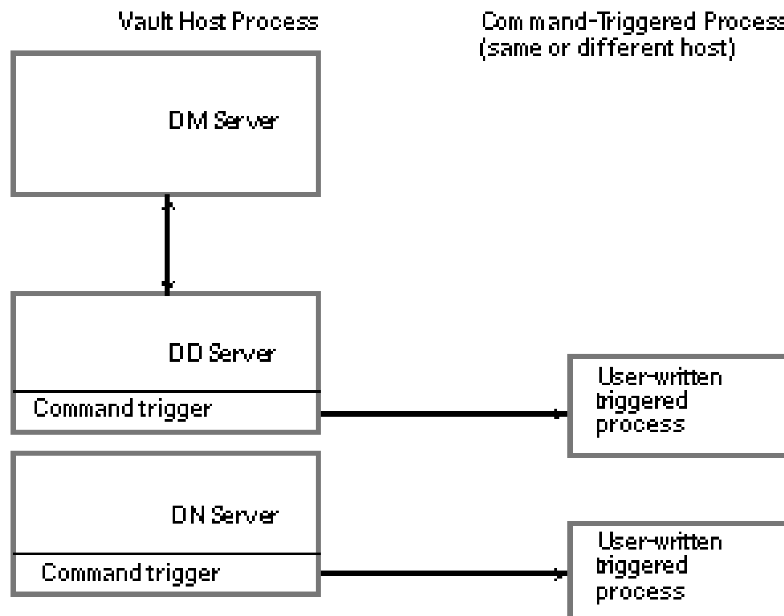
The triggered process cannot alter the input data to the Vault command. When a trigger and wait process is initiated, the server is not available for other Vault users until the process is completed.

Please note: Be cautious about initiating a trigger-and-wait process because the server is unavailable to other clients until the triggered process is completed.

When using Vault command triggers, make sure that you define more than one `pdmadmn (dn)` process and more than one `pdmdd (dd)` process in your NSM configuration file. Vault command trigger transactions may initiate additional Vault commands, if you elect to do so. As a result you must ensure that there are sufficient `PDMADMN` [`DN`] or `PMDDD` [`DD`] servers.

The following figure illustrates the Vault servers that can initiate command triggers.

Figure 1-2 Command-Trigger Operation



Using Command Triggers with Programmatic Interface

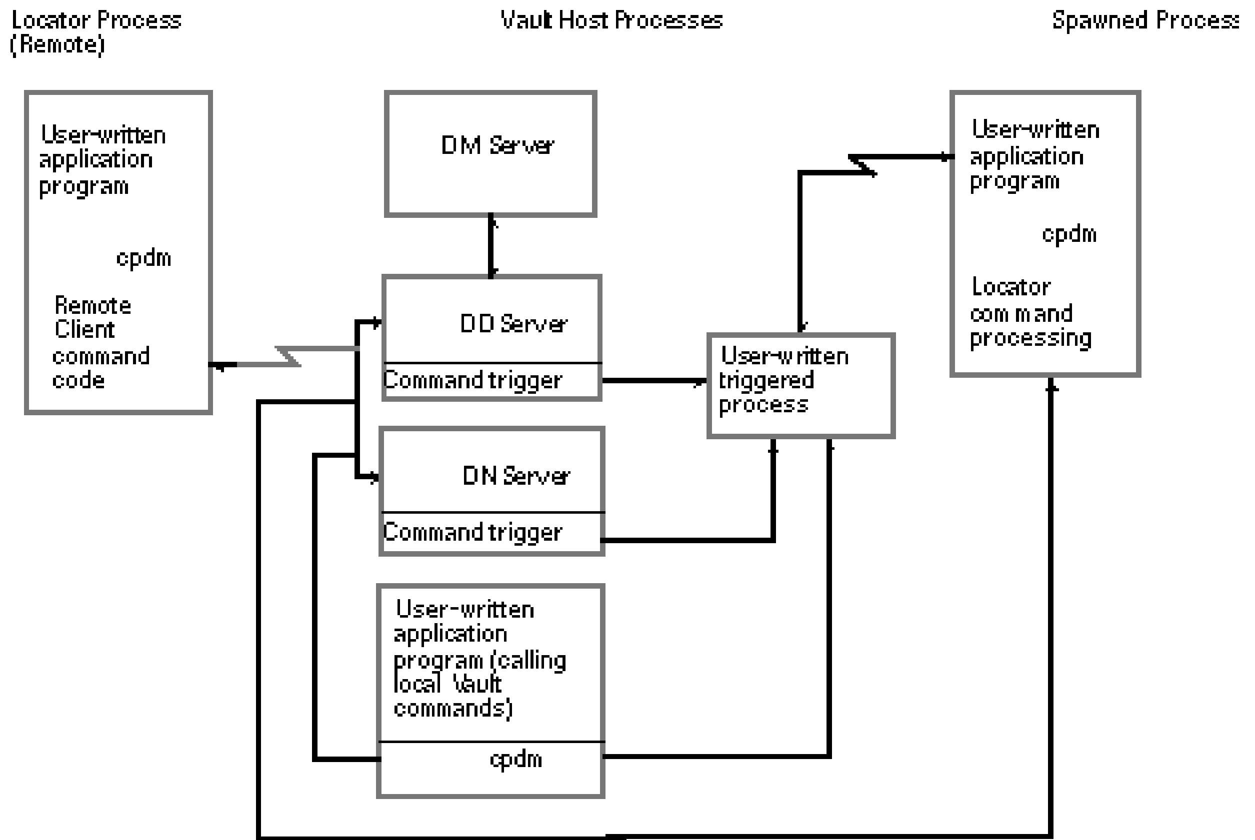
For the programmatic interface to use a triggered process, the programmatic interface must call a Vault command that has a command trigger. When a command includes a trigger, the triggered process is executed no matter which Vault interface (command or programmatic) is used to execute the command.

For a triggered process to use the programmatic interface, the triggered process must communicate with another process that calls the programmatic interface. How this communication takes place depends on your operating system.

Figure 1-3, Command-Trigger and Programmatic Interface Operation, illustrates how the programmatic interface can be used in conjunction with command triggers. In this example, a user-defined application accesses a server on the Vault host which in turn uses a command trigger to activate a triggered process.

Please note: The triggered process spawns another process that calls the programmatic interface, rather than calling the programmatic interface directly, which would be invalid.

Figure 1-3 Command-Trigger and Programmatic Interface Operation



Chapter 2

Using the Programmatic Interface

This chapter describes how to develop, compile, link, and run an application program using the programmatic interface.

- Format of the Programmatic Interface
- Writing a Program Using the Programmatic Interface
- Special Considerations for Local Vault Commands
- Compiling, Linking, and Running a Program

Format of the Programmatic Interface

The programmatic interface allows you to develop application programs that use Vault functions within a standard programming environment. An application requiring a Vault service or function can make a program call to a supplied programmatic interface subroutine with the appropriate input arguments.

Please note: It is recommended that you migrate your application programs to use the new interface. The new interface allows access to all previously available Vault functions. It also provides access to new functionality, such as `Distributed Vault` and binders commands.

Introducing the SVedm Programmatic Interface

The programmatic interface is accessed by the `SVedm` entry point label that uses a variable argument interface. The entry point provides a mechanism for interlanguage communication with the Vault software, which is written in C.

You can use the `SVedm` entry point for C programs on all platforms.

String arguments are used to pass information to the Vault, as well as for the Vault to return information to the calling program.

- Input: Input to `SVedm` is done via a variable string argument mechanism. This is accomplished by first naming the command as a string followed by arguments containing `keyword/value` pairs to be processed by the command The argument list must be terminated with a `NULL` character pointer.

  For example,

  ```
  retcode = SVedm("signon","userid","acctname",
  "userpw","passwd", (char*)0);
  ```

  See the parameter tables in the *Vault Command Reference* for the layout of the input for each Vault command available through the programmatic interface.

- Output: `SVedm` provides a Vault return code as function return. However, a Vault command completion message prefixed by a 10-character Vault message identifier can be obtained using the keyword `message` followed by a user-defined character array to hold the data. At least 256 characters of space is recommended.

  Your application can use the Vault-provided message or you may wish to format your own message. The message number, which is part of the Vault message identifier, is returned in a user-defined character array following the `keyword message`. Your application can use the message number to perform specialized logic.

Using the Old Programmatic Interface

The programmatic interface is accessed by the `cpdm` entry point label, which uses three input-output structures (sometimes called records). The entry point is used to provide a mechanism for interlanguage communication with the Vault software, which is written in C.

You can use the `cpdm` entry point for C programs on all platforms.

Control, input, and output structures are used to pass information to the Vault, as well as for the Vault to return information to the calling program.

To simplify coding, source files for the control structure, each command input structure, and all keywords and length mnemonics are provided with the programmatic interface. These files can be used as include members.

The structures are described below:

- Control Structure: The control structure contains fields that inform the programmatic interface which command is being requested. The Vault also returns status information about the overall status of the programmatic interface call, and the status of the Vault command that was executed. For example, information about whether or not the Vault command completed without any errors, and if errors occurred, the Vault error message number, may be returned by the Vault.

  The same control structure is used for each call to the programmatic interface. Only the contents of the command name field need to change for each programmatic interface call.
- Input Structure: The input structure contains the fields needed by the command to be processed. The fields in the input structure differ for each Vault command. See the parameter tables in the *Vault Command Reference* for the layout of the input structure for each Vault command available through the programmatic interface.
- Output Structure: The output structure contains a Vault command completion message prefixed by a 10-character Vault message identifier.

  Your application can use the Vault-provided message or you may wish to format your own message. The message number, which is part of the Vault message identifier, is also returned in the control structure in a separate field. Your application can use the message number to perform specialized logic.

Writing a Program Using the Programmatic Interface

You can write a program using the interfaces that are discussed in the sections that follow.

New Programmatic Interface (SVedm)

Do the following to write a program that performs Vault commands,

1. Provide the necessary data in the input for the command you wish to execute.
2. Call `SVedm`.
3. Test the results of the command call.

Please note: There is no need for control, input, or output structures or their header files when programming with `SVedm`.

Old Programmatic Interface (cpdm)

Please note: `SVedm` does not affect interface availability through `cpdm` or command triggers. However, it is recommended that you convert your `cpdm` programs to use the `SVedm` interface.

Do the following to write a program that performs Vault commands,

1. Define the three programmatic interface structures (`Control`, `Input`, and `Output`).
2. Provide the necessary data in the input and control structure for the command you wish to perform.
3. Call `cpdm`.
4. Test the results of the command call.

Naming Conventions for Structures and Header Files

Follow the standards of the programming language you are using for bringing header (or source include) files into your program.

For example, source include files for the C programming language are usually specified in lowercase followed by `.h`. Include files for the C programming language are individual files.

The name of the include file for a Vault command is `ad` plus the three-letter abbreviation of the command. See Chapter 1, “Overview of Vault Programming” for the command abbreviations. For example, `adson.h` is the name of the include file for the `edmsignon` command.

The name of the command input structure that is defined using the supplied source files is the command name followed by the characters `struct`. For example, `signon_struct (c)` is the name of the input structure for the `edmsignon` command.

Chapter 7, “Vault Commands” includes the following information for each Vault command available through the programmatic interface:

- The three-character abbreviation
- The source include file name
- The input structure name

Defining the Programmatic Interface Structures

Your application program must allocate memory for each of the three programmatic interface structures — Control, Input, and Output. The source include files provided for the C interface define the mapping of structures and storage locations, but they do not reserve actual storage. Your program must allocate the necessary storage.

Control Structure

The fields for the control structure are defined in the supplied source include file `adpictl.h`.

Also contained within the source include file are symbolic constants for each of the Vault commands and for the current release of the Vault. These symbolic constants can be used to provide the needed values for the fields pi_command and pi_releaseno.

The control structure consists of the fields listed in the following table. These fields appear in lowercase in the C source file.

Table 2-1 Programmatic Interface Control Structure Fields

Field Name	Field Type	Field Description
pdm_retcode	4-byte integer	Return code from Vault command execution
pdm_messageno	4-byte integer	Vault message number
pi_reserved1	4-byte integer	Reserved for Vault use
pi_reserved2	4-byte integer	Reserved for Vault use
pi_reserved3	4-byte integer	Reserved for Vault use
pi_retcode	4-byte integer	Return code from programmatic interface 0 Success 31012 Invalid command name 31014 Unsupported release 31030 Old release, must recompile
pi_command	8-byte character	Vault command name
pi_releaseno	8-byte character	Vault release level
pi_reservedd	6-byte character	Reserved for Vault use
pi_reservedt	6-byte character	Reserved for Vault use
pi_reservedx	10-byte character	Reserved for Vault use

Initialize all control structure fields to blanks or zeros, as appropriate. Null-terminated fields are not used. The pi_command field is initialized with the mnemonic for the Vault command, and the pi_releaseno field with the mnemonic (current_release) for the current release of the Vault. Do not change the release number in the control structure.

Input Structure

Each Vault command has its own source include file that you use whenever you call that command programmatically.

Also provided is an additional include file (`adkeylen.h` for C) that contains length mnemonics for each Vault field name. These mnemonics can be used to fill each input structure field properly.

All fields defined in the input structure are in character format. These fields are initialized with blanks. When user data is entered, any unused portion of the field is blank padded. Null-terminated fields are not used in the input structures.

Output Structure

The output structure consists of a single character field that contains the Vault command completion message prefixed by a Vault message identifier. The message text returned by the Vault is blank-padded and contains no nulls. The length mnemonic ltotalmsg can be used to describe the message text portion of the output structure. The format of the Vault message identifier is

- Vault system ID
- Three-letter abbreviation of the Vault command
- Vault message number
- Severity code

The same three-letter command abbreviations used for the source include files are used in the message identifier. These abbreviations are documented in Chapter 1, "Overview of Vault Programming."

For example, the message identifier returned from a successful Vault signon is CDMSON016I. In this example

- CDM is the Vault system ID
- SON is the code for the `edmsignon` command
- 016 is the Vault message number (duplicating pdm_messageno in the control structure)
- is the severity code (based upon pdm_retcode in the control structure)

Vault severity codes relate to Vault return codes, as shown in the following table:

Table 2-2 Vault Return Codes and Severity Codes

Return Code (pdm_retcode**)**	**Severity Code**	**Meaning**
0	I	Command successful
4	W	Warning
4	T	Message from a command trigger process
8	E	Error
12	V	Security violation
16	F	Fatal error

Any Vault command that returns an E, V, or F severity code has not completed successfully.

Running a Sample Program (New SVedm Interface)

Appendix A, "Sample C Program" contains a sample C program that uses the programmatic interface. The sample programs have been provided to illustrate correct use of the programmatic interface, as discussed in this chapter. The sample programs are also referenced in the following chapters.

This program executes the `signon`, `store`, and `signoff` commands.

To run the SAMPLE program written in C, enter in response to the operating system prompt:

```
% svedmsample edmuserid edmuserpw file_to_store filename_as_stored
```

When executed successfully, the program displays:

```
EDM command      -- SIGNON
 pi return code - 0
 output message - CDMSON016I Sign on to EDM server EDM
                  completed successfully.
                   You have 0 EDM message(s).

EDM command      -- STORE
 pi return code - 0
 output message - CDMSTR177I The file has been stored.

EDM command      -- SIGNOFF
 pi return code - 0
 output message - CDMSOF017I Sign off from EDM completed
                  successfully.
Sample program has finished.
```

A Vault audit file for the `store` command is created in the user's local directory.

Special Considerations for Local Vault Commands

There are two types of Vault commands: local and remote. The local commands are as follows:

- `addadod`
- `addsp`
- `addt`
- `addvault`
- `archive`
- `chgctl`
- `clst`
- `ibkup`
- `load`
- `opnt`
- `recsf`
- `recsp`
- `remadod`
- `remt`
- `remvault`
- `restore`
- `scantape`
- `unload`

Since local commands must access the Vault storage pools or the SQL database or both, they can only be run on the node where the Vault storage pools and SQL database exist.

If your program uses Oracle or SQL/DS and also programmatically calls any of the local Vault commands, you must commit or roll back your database changes prior to calling any Vault local command and you must reconnect to your database after the call. This is necessary since each Vault command is a logical unit of work and local Vault command code resides in the same executable as your application code.

Compiling, Linking, and Running a Program

Follow the steps given in the sections that follow to compile, link, and run a program.

Compiling the Program

Use your standard compile procedure(s) to compile a program that calls the programmatic interface. If your application program calls Oracle or SQL/DS, you must use the required preprocessor supplied with the RDBMS. You should make sure that you have access to the source include files if you are using them to define the programmatic interface structures.

See your system manager for the location of these files.

Linking the Program

Before linking your application you should make sure that you have access to all the necessary object code that is referenced by your application program and by the Vault. This includes RDBMS and networking software, because Locator code makes calls to either the RDBMS or the network. See your system manager for the location of these files and/or libraries.

After successful compilation, the application program must be linked with the Vault object code. The interface to the runtime code is `SVedm`.

On certain operating systems this entry point module is a stub module. The stub module, at runtime, obtains the remainder of the code necessary to run the command. Other operating systems require that the entry point module contain references to the Vault command code for each command called by the application. For these operating systems, a tool is provided that creates an entry point module that contains the needed references.

If the RDBMS is Oracle, a program that uses local Vault commands must be relinked to be moved to another Vault on another machine.

Running the Program

To run an application program using Vault Programming, you must already be signed on to the Vault (with the `signon` command), or your application must execute the `signon` command before any other Vault command. The `signon` command establishes a set of authorities used by the Vault to determine whether a command can be executed and whether a file can be accessed and/or modified.

Before executing your program, ensure that you have access to the Vault software and that your account privileges are correct. This can be verified by executing the Vault command(s) in your application through one of the standard Vault interfaces.

Please note: In case of Windows NT, a sample makefile is provided for building the programmatic Vault interface. Visual C++ 6.0 SP3 is required to build the program. To build the programmatic Vault interface,

1. Change to `%EDM_HOME%\src` directory.
2. Compile using `nmake -f svedmsample.mak`

Chapter 3

Compiling and Linking a Programmatic Interface Program

This chapter describes the process for compiling and linking programs on the following systems as a Vault Client, with Vault, and with Oracle 8*i* Release 3 (8.1.7).

- Linking a Program on SGI IRIX 6.5
- Linking a Program on Compaq Tru64 UNIX 4.0E
- Linking a Program on AIX 4.3.3
- Linking a Program on HP-UX 11
- Linking a Program on Solaris 2.6
- Linking a Program on Windows NT 4.0 (With Service Pack 5)

Linking a Program on SGI IRIX 6.5

Examples of compiling and linking a sample program on a Silicon Graphics workstation (IRIX 6.5) follow. The qualified C and C++ compiler for SGI IRIX 6.5 is version 7.2.1.2m with build option `-n32 MIPS4`.

Linking on IRIX 6.5 — Client Only

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

% cc -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/cedmpi.a \
$EPD_HOME/lib/sqlutil.a -lcvnas $EPD_HOME/lib/cedmpi.a \
-lcvclapi -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt \
-lcvkernel -lcvedmstubs -lcvclientstubs -lC
```

Linking on IRIX 6.5— Vault Only

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

% cc -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a \
$EPD_HOME/lib/sqlutil.a -lcvnas $EPD_HOME/lib/edmpi.a \
-lcvclapi -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt \
-lcvkernel -lcvedmstubs -lcvclientstubs -lC
```

Linking on IRIX 6.5 — Vault with Oracle 8*i* Release 3 (8.1.7)

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

% cc -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a \
$ORACLE_HOME/lib32/libclntshcdk.so \
-lcvnas  $EPD_HOME/lib/edmpi.a -lcvclapi -lcvnsm -lcvhli \
-lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs \
-lC -lgen -lsocket -lnsl -lm  -ldl
```

Linking a Program on Compaq Tru64 UNIX 4.0E

Examples of compiling and linking a sample program on a Compaq Tru64 UNIX 4.0E workstation follow. The qualified C and C++ compilers for Compaq Tru64 UNIX 4.0E are V5.8-009 and V6.2-024 respectively.

Linking on Compaq Tru64 UNIX 4.0E — Client Only

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c
% cc -o svedmsample svedmsample.o -L$EPD_HOME/lib \
-L/usr/lib -L/usr/lib/cmplrs/cxx -L/usr/shlib \
$EPD_HOME/lib/cedmpi.a \
$EPD_HOME/lib/sqlutil.a -lcvnas $EPD_HOME/lib/cedmpi.a \
-lcvclapi \
-lcvnsm -lcvhli -lcvcxx -lcvcaddsrt -lcvkernel \
-lcvedmstubs -lcvclientstubs -lc -lcxx
```

Linking on Compaq Tru64 UNIX 4.0E — Vault Only

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

% cc -o svedmsample svedmsample.o -L$EPD_HOME/lib \
-L/usr/lib -L/usr/lib/cmplrs/cxx -L/usr/shlib \
$EPD_HOME/lib/edmpi.a \
$EPD_HOME/lib/sqlutil.a -lcvnas $EPD_HOME/lib/edmpi.a \
-lcvclapi \
-lcvnsm -lcvhli -lcvcxx -lcvcaddsrt -lcvkernel \
-lcvedmstubs -lcvclientstubs -lc -lcxx
```

Linking on Compaq Tru64 UNIX 4.0E — Vault with Oracle 8*i* Release 3 (8.1.7)

```
cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

/usr/bin/cxx -o svedmsample svedmsample.o -L$EPD_HOME/lib \
-L/usr/lib -L/usr/lib/cmplrs/cxx -L/usr/shlib \
$EPD_HOME/lib/edmpi.a -lcvclapi \
$EPD_HOME/lib/edmpi.a -lcvnsm -lcvnas \
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a \
-lcvnsm -lcvcaddsrt -lcvhli -lcvcxx -lcvkernel \
-L$ORACLE_HOME/lib -lclntsh -lexc \
-lmld -lrt -laio_raw -lm -lcvedmstubs
```

Linking a Program on AIX 4.3.3

Examples of compiling and linking a sample program on a system running AIX 4.3.3 follow. The qualified C and C++ compiler for AIX 4.3.3 is version 3.6.6.0.

Linking on AIX 4.3.3 — Client Only

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

% xlC -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/cedmpi.a \ $EPD_HOME/lib/sqlutil.a -lcvnas \
$EPD_HOME/lib/cedmpi.a \
-lcvclapi -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt -lcvkernel \
-lcvedmstubs -lcvclientstubs -lc -lbsd
```

Linking on AIX 4.3.3 — Vault Only

```
% cc -c -Dunix -I$EPD_HOME/include $EPD_HOME/src/svedmsample.c

% xlC -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a  $EPD_HOME/lib/sqlutil.a \
-lcvnas $EPD_HOME/lib/edmpi.a \
-lcvclapi -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt -lcvkernel \
-lcvedmstubs -lcvclientstubs -lc -lPW -lbsd
```

Linking on AIX 4.3.3 — Vault with Oracle 8*i* Release 3 (8.1.7)

```
cc -c -Dunix -I$EPD_HOME/include svedmsample.c

xlC -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a \
$ORACLE_HOME/lib/ -lclntsh \
-lcvnas $EPD_HOME/lib/edmpi.a -lcvclapi -lcvnsm -lcvhli \
-lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs \
-lm -lc -lPW -lbsd
```

Linking a Program on HP-UX 11

Examples of compiling and linking a sample program on an HP-UX operating system follow. The HP-UX 11 CC Compiler (for compiling) and aCC compiler (for linking) use the ANSI and aCC Compiler version A11.01.02 and A03.15 respectively.

Linking on HP-UX 11— Vault Client (Locator) Only

```
/opt/ansic/bin/cc -DUNIX -DALPHA -D_WANT_SQLCASTORAGE +Z \
-DHPUX -DHP700 \
-DHPUX_10 -DHUGE=HUGE_VAL -Ae -Dunix +DA1.1 +DS1.1 -c \
-DBYTES_LEFT \
-DBITFIELDS_LEFT -Dhpux -I$EPD_HOME/include \
$EPD_HOME/src/svedmsample.c

/opt/aCC/bin/aCC -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/cedmpi.a $EPD_HOME/lib/sqlutil.a -lcvnas \
$EPD_HOME/lib/cedmpi.a -lcvclapi -lcvnsm \
-lcvhli -lcvedmdct -lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs \
-lcvclientstubs -lC -lc -lPW -lepd_cvlms -lm -lXm -lXt -lXext \
-lX11 -lc -lm -lPW -lpthread -lrpcsoc -lnsl
```

Linking on HP-UX 11— Vault Only

```
/opt/ansic/bin/cc -DUNIX -DALPHA -D_WANT_SQLCASTORAGE +Z \
-DHPUX -DHP700 \
-DHPUX_10 -DHUGE=HUGE_VAL -Ae -Dunix +DA1.1 +DS1.1 -c \
-DBYTES_LEFT \
-DBITFIELDS_LEFT -Dhpux -I$EPD_HOME/include \
$EPD_HOME/src/svedmsample.c

/opt/aCC/bin/aCC -o svedmsample svedmsample.o -L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a -lcvnas \
$EPD_HOME/lib/edmpi.a \
-lcvclapi -lcvnsm -lcvhli -lcvedmdct -lcvcxx \
-lcvcaddsrt -lcvkernel \
-lcvedmstubs -lcvclientstubs -lC -lc -lPW -lepd_cvlms -lm \
-lXm -lXt \
-lXext -lX11 -lc -lm -lPW -lpthread -lrpcsoc -lnsl
```

Linking on HP-UX 11— Vault with Oracle 8*i* Release 3 (8.1.7)

```
/opt/ansic/bin/cc -DUNIX -DALPHA -D_WANT_SQLCASTORAGE +Z \
-DHPUX -DHP700 \
-DHPUX_10 -DHUGE=HUGE_VAL -Ae -Dunix +DA1.1 +DS1.1 -c \
-DBYTES_LEFT -DBITFIELDS_LEFT -Dhpux -I$EPD_HOME/include \
$EPD_HOME/src/svedmsample.c

/opt/aCC/bin/aCC -w  -o svedmsample svedmsample.o \
-L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a \
$ORACLE_HOME/lib/libclntsh.sl  -lcvnas \
$EPD_HOME/lib/edmpi.a -lcvclapi \
-lcvnsm -lcvhli -lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs -lC \
-lm -lc -lPW \
-lpthread -lrpcsoc -lnsl
```

Linking a Program on Solaris 2.6

Examples of compiling and linking a sample program on a SUN Microsystems workstation running Solaris 2.6 are shown below. The qualified compiler is SparcWorks 5 (SC 5.0).

Linking on Solaris 2.6 — Vault Client (Locator) Only

```
/opt/SUNWspro/bin/cc -c -Dunix -I$EPD_HOME/include \
$EPD_HOME/src/svedmsample.c

/opt/SUNWspro/bin/cc -o svedmsample svedmsample.o \
-L$EPD_HOME/lib \
$EPD_HOME/lib/cedmpi.a $EPD_HOME/lib/sqlutil.a -lcvnas \
$EPD_HOME/lib/cedmpi.a -lcvclapi -lcvnsm -lcvhli -lcvcxx \
-lcvcaddsrt -lcvkernel -lcvedmstubs -lcvclientstubs \
-lintl -lC -lgen -lsocket -lnsl -lm -ldl
```

Linking on Solaris 2.6 — Vault Only

```
/opt/SUNWspro/bin/cc -c -Dunix -I$EPD_HOME/include \
$EPD_HOME/src/svedmsample.c

/opt/SUNWspro/bin/cc -o svedmsample svedmsample.o \
-L$EPD_HOME/lib \
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a -lcvclapi \
-lcvnas \
$EPD_HOME/lib/edmpi.a -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt \
-lcvkernel -lcvedmstubs -lcvclientstubs \
-lintl -lC -lgen -lsocket -lnsl -lm -ldl
```

Linking on Solaris 2.6— Vault with Oracle 8*i* Release 3 (8.1.7)

```
/opt/SUNWspro/bin/cc -c -Dunix -I$EPD_HOME/include \
$EPD_HOME/src/svedmsample.c

/opt/SUNWspro/bin/CC -L/opt/SUNWspro/SC3.0.1 -o svedmsample \
svedmsample.o \
-L$EPD_HOME/lib $EPD_HOME/lib/edmpi.a \
$EPD_HOME/lib/sqlutil.a \
$ORACLE_HOME/lib/libclntsh.so \
-lcvnas $EPD_HOME/lib/edmpi.a -lcvclapi -lcvnsm -lcvhli \
-lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs \
-lC -lintl -lgen -lsocket -lnsl -lm -ldl
```

Linking a Program on Windows NT 4.0 (With Service Pack 5)

Example of compiling and linking a sample program on a system running on Windows NT 4.0 (with Service Pack 5). Before compiling you should make sure that the environment variable `INCLUDE` is set correctly. Run the following command to define the variable or to append the existing one:

```
set INCLUDE=%include%;%EPD_HOME%\include
```

After defining the variable execute the following command:

```
nmake -f svedmsample.mak
```

Chapter 4

Using Command Triggers

This chapter describes the function and use of command triggers.

- Command-Trigger Format
- Developing a Command-Triggered Program
- Command-Trigger Structure
- Command-Trigger Communications Subroutines
- Modifying the NSM Configuration File
- Compiling, Linking, and Running the Program
- Using the chgctl Command

Command-Trigger Format

Command triggers allows you to develop application programs that augment Vault commands (these are also called Vault commands). When a Vault command is executed, it triggers a user-written process that performs a specific task, such as logging supplemental audit information. A command-triggered process is a detached exit called by a Vault command to perform user-written logic. The exit is considered to be detached because it is not linked to Vault code.

A triggered process is a separate executable that can reside on any platform. It may contain any logics otherwise available on the platform, but must not contain a call to the programmatic interface. A triggered process may, however, invoke another process, not known to Network Services, to call the programmatic interface.

Each Vault command can trigger only one process. Multiple Vault commands may trigger the same process. Communication logics must be included in the triggered process. A description of how they are coded and a sample control program are provided for this purpose.

You must write and test the code to perform your required command-triggered application. When it is ready to be incorporated into the Vault, you must define the process as an Application Entity (AE) in the NSM configuration file.

You must also define the Application Entity to the Vault using the `chgctl` command. It is the `chgctl` command which specifies whether your process is triggered at the beginning or end of the Vault command. The `chgctl` command also specifies whether the Vault is to wait for the triggered process to complete and how long it should wait. You should design your process to keep wait times to a minimum because the process that initiated the trigger is unavailable to other users until the triggered process is complete.

To simplify coding, source files for the control structure, each command input structure, and all keywords and length mnemonics are provided with the programmatic interface. These files can be used as include or copy members for the appropriate language.

Developing a Command-Triggered Program

It is recommended that user applications developed for command triggers be dispatched through a control program that performs the necessary subroutine calls for connects and disconnects to the Vault and for sending messages and waiting for replies. The user application code can then be developed as subroutines, free of network communication concerns.

Appendix C, “Sample Command-Triggered Control Program”, shows the code for a control program (`ctshell.c`) that performs the necessary network communication functions and calls an application subroutine (`ctsample.c`, shown in Appendix D, “Sample Command-Triggered Application Subroutine”). It is designed so the name of the application subroutine called by the control program is identical to the AE name by which Network Services knows the process.

Vault Programming includes the source code for the sample control program (`ctshell.c`), the sample application subroutine (`ctsample.c`), and their include files. It also includes the object code for the four communication subroutines called by the control program.

The control program is meant as a guide and can be modified as necessary (the call in the control program to the sample application subroutine must, at least, be changed to call your subroutine).

One control program can be used for all the user-written command-triggered applications, or multiple control programs can be developed under different names. The number of concurrent command-triggered processes is not limited by the number of main routines employed. Instead, it depends upon parameters specified in the NSM configuration file.

Each application subroutine called by the control program has one input parameter, a pointer to a structure, named `CT_struct`, that contains all of the information communicated between the Vault and the triggered process. This structure is also used by the dispatch program to call the four communication subroutines.

Command-Trigger Structure

The command-trigger structure, `CT_struct`, is defined in the supplied `adctstr.h` source include file.

`CT_struct` consists of the fields listed in the following table:

Table 4-1 Command-Trigger Structure Fields

Field Name	Field Type	Field Description
`aename`	32-byte character	Name of the triggered process (Application Entity name).
`full_aename`	80-byte character	Reserved for use by the Vault. Contains the fully qualified AE name of the triggered process. Format: `node:domain:aename:instance`
`edm_aename`	80-byte character	Reserved for use by the Vault. Contains the fully qualified AE name of the Vault process which initiated the trigger. Format: `node:domain:aename:instance`
`inbuff`	512-byte character	Data from the Vault to the triggered process. See Table 4-2 for more information.
`outbuff`	241-byte character	Data from the triggered process to return to the Vault. See Table 4-3 for more information.

All fields in `CT_struct` are in character format, blank padded where necessary, and not null terminated. The values in `full_aename` and `edm_aename` must not be changed.

The CT_struct field inbuff can contain up to 512 character bytes and consists of the fields listed in the following table:

Table 4-2 CT_struct Field inbuff Contents

Field Name	Field Type	Field Description
wait_flag	1-byte character	0 means trigger and proceed with Vault command. 1 means trigger and wait for triggered process.
location_flag	1-byte character	0 means triggered process connects at beginning of Vault command (after user validation). 1 means triggered process connects at end of Vault command.
severity_code	1-byte character	Severity code showing condition of the Vault at time of trigger. i = Command successful w = Warning e = Error v = Security Violation f = Fatal error A command showing a severity code of e, v, or f does not complete successfully.
releaseno	8-byte character	The release number of the Vault release which is installed on that cpu. The request or command originates from programmatic interface installed on the server.
userid	12-byte character	The programmatic interface userid of the user initiating the Vault command which called this trigger.
command	8-byte character	The Vault command that called this trigger.
keywords	481-byte character (maximum)	The remaining bytes contain the values of the keywords associated with the Vault command. The total length of pdm_inbuff depends on the Vault command. The triggered process cannot alter the contents of Vault command input.

Each Vault command has its own source include file that you can use to define the fields sent to your command-triggered process. The name of the include file for a Vault command is ad plus a three-letter abbreviation of the command. For example, adson.h is the name of the include file for the Vault signon command.

The input structure within the include file is the command name followed by _struct. For example signon_struct is the name of the input structure for the Vault signon command.

Also provided is an additional include file (adkeylen.h) that contains length mnemonics for each Vault field name. These mnemonics can be used to fill each input structure field properly.

Chapter 7, “Vault Commands”, includes the following information for each Vault command:

- The three-character abbreviation
- The source include file name
- The input structure name

Output from Triggered Process to the Vault

The `CT_struct` field `outbuff` contains 241 character bytes and consists of the fields listed in Table 4-3.

Table 4-3 CT_struct Field outbuff Contents

Field Name	Field Type	Field Description
`process_flag`	1-byte character	`0` means the Vault process can continue. `1` means the Vault should stop processing the command. (The Vault only stops processing if this was a beginning trigger, otherwise this flag is ignored.)
`message_text`	240-byte character	The user can send back a message to the Vault process of up to 240 character bytes, blank padded. This message is always written to the Vault audit log.

The Vault command-trigger structure has a field that contains the overall success or failure status of the Vault command. Depending on the selection scope used (`f`=file, `p`=part, `s`=fileset, etc.), the status code could have different interpretations.

For example, if a selection scope of `f` (single file) is used and the store of the file fails, the command status code that is passed to the trigger is unsuccessful.

All or Nothing selection scopes are

- `f` — single file
- `p` — part
- `s` — file set

Command-Trigger Communications Subroutines

The `CT_struct` is used by the four communication subroutines that must be called by the triggered process. They are:

- Connect to the Vault (`ct_connect`) connects the triggered process to the Vault
- Wait for trigger (`ct_waiting_for_pdm`) waits for a command-trigger message to be sent
- Respond to trigger (`ct_sending_response`) responds to the command-trigger input
- Disconnect from the Vault (`ct_disconnect`) disconnects the triggered process from the Vault

The use of these subroutines is shown in the control program in Appendix C, “Sample Command-Triggered Control Program”.

The subroutines, the manner in which they are called, and the `CT_struct` fields that they use, are as follows.

Connect to the Vault

```
int ct_connect( CT_struct )
      CT_struct.aename (input)
```

(Copy the triggered process AE name into the `aename` field of `CT_struct` before calling `ct_connect`.)

```
CT_struct.full_aename (output)
```

(The Vault process sets this field to the full AE name of the triggered process. It should not be altered. Format: `node:domain:aename:instance`)

Wait for Trigger

```
int ct_waiting_for_pdm( CT_struct )
      CT_struct.inbuff (output)
```

The Vault process returns initialized structures to this field. For definitions of these structures, see the file:

```
$EPD_HOME/include/adctstr.h.
      CT_struct.edm_aename (output)
```

The Vault process sets this field to the full AE name of the Vault process which called this trigger. It should not be altered. The format follows:

```
node:domain:aename:instance
```

Respond to Trigger

```
int ct_sending_response( CT_struct )
      CT_struct.outbuff (input)
```

(If trigger and wait specified: You must copy the message you wish to return to the Vault process into this field. The message should be blank padded. If trigger at the beginning of the command: Also set the flag to tell the Vault process whether to continue or not.)

```
CT_struct.edm_aename (input)
```

(This field was previously set to the full AE name of the process which called this trigger.)

Disconnect from the Vault

```
int ct_disconnect( CT_struct )
      CT_struct.full_aename (input)
```

(This field was previously initialized with the full AE name of the triggered process.)

Each of the above subroutines returns a zero (0) to the calling program if it executes correctly. Any value other than 0 is an error, and any error condition should result in termination of the command-triggered process.

Modifying the NSM Configuration File

Applications started by Network Services cannot accept parameters passed to them. Consequently, a UNIX executable script is needed to invoke the command-triggered process.

Refer to Chapter 5, “Compiling, Linking, and Running a Command-Triggered Program” for more information on creating a UNIX script file.

In order for Network Services to invoke your command-triggered process, the AE name of your triggered process must be in the NSM configuration file. Your Vault system administrator should make the required additions to the NSM configuration file. Although these changes can be made at any time, they are not recognized by the process manager until the next time it is started up.

The NSM configuration file can specify that the triggered process be brought up automatically when the Vault is started, or it can specify that the triggered process be brought up manually. In either case, your triggered process can be started any time after the process manager is started, but is best started after all the standard Vault processes.

Below is a list of the application entity attribute keywords that you must specify for a command-triggered process to be started automatically by NSM. It is placed under the node from which you run your triggered process and under the domain name of PDM.

```
AE(aename,start_group_name,start_group_sequence)
PATH(pathname_of_process_to_start)
OWNER(owner_of_started_process)
WORKDIR(working_directory_of_started_process)
MAXINST(maximum_instances_allowed)
CLOSE
GRPCTL(maximum_instances_to_start,
minimum_instances,bind_requirements)
```

The keywords `PATH`, `OWNER`, `WORKDIR`, and `GRPCTL` are required only if the process is to be started automatically by NSM. The `PATH` keyword specifies the executable script (command procedure), not your program name.

When using command-triggering, you should update the NSM configuration file entry for `PDMLOG USER` parameter. This parameter lists severity codes which are logged by the Vault. Command triggers use a severity code of `t` which should be added to the list. The entry under `AE(pdmlog)` to specify logging of all Vault severity codes would be `USER(log=iewtfv)`.

Compiling, Linking, and Running the Program

Perform the following to compile, link, and run a program.

Compiling the Program

Use the standard compiling procedures to compile both the control program and your application subroutines. If your application calls a Relational Database Management System (RDBMS) such as Oracle or SQL/DS, you must use the required preprocessor supplied with the RDBMS. You must have access to the source include files provided with the Vault Programming.

See the appropriate chapter for specific compiling instructions for the operating system that you are using.

Linking the Program

Before linking, make sure you have access to the object code referenced by your application programs, including the object code involving the four communication subroutines and other networking object code.

See the appropriate chapter for specific linking instructions for the operating system that you are using.

Running the Program

Once your command-triggered program is linked, a script or command procedure has been created to run it, and the necessary NSM configuration file changes have been made, your process can be started. If you have specified that Network Services start your process automatically, it is brought up the next time the Vault is started. To start your process manually, you enter the name of the script or command procedure which invokes your program.

See the appropriate chapter for specific run instructions for the operating system that you are using.

Once your command-triggered process is running, it is available for Vault commands to send messages to it and receive replies from it. The sample program is structured so that the control program waits indefinitely for messages, then dispatches them to the appropriate subroutine.

The sample application subroutine, `ctsample`, is called only if a message is directed to the AE name, `ctsample`. These messages should come only from the

store, reserve, and chgfa commands at preprocess time and with wait-for-response requested. The chgctl command, discussed below, is used to enable these commands to send messages to the command-triggered process.

For debugging purposes, it may be useful to start your command-triggered process manually each time you wish to test it. However, if the programmatic interface attempts to send a message to your command-triggered process and the process is not started and cannot be started by Network Services, the Vault command fails. It is not necessary to restart the other Vault processes each time you wish to restart your command-triggered process.

Whether your command-triggered process has been started automatically or manually, you can stop it by using the command nsmstop. To stop the sample command-triggered process, you enter

```
nsmstop ":pdm:ctsample:"
```

If your process crashes without disconnecting from the Vault process, it remains active for 30 minutes before the process manager removes it. If you restart your process, you get another instance of it, up to the MAXINST number of processes specified in the NSM configuration file. To avoid multiple instances of your process, you can use the nsmflush command to delete your process without waiting for the process manager to remove it. To delete the sample command-triggered process, you enter

```
nsmflush ":pdm:ctsample:"
```

The commands nsmflush and nsmstop may have restricted usage. See your system administrator if you need to use these commands. After defining your process to Network Services and starting it, you must enable the command trigger for whichever Vault commands you require by using the chgctl command. The following section, "Using the chgctl Command," explains the use of chgctl.

Using the chgctl Command

Each Vault server has one command-trigger list that includes an entry for each Vault command that can have a command trigger.

When a process is associated with a command on the command-trigger list, you can set a flag that indicates whether the process should or should not be triggered. You determine whether to trigger a process before Vault command processing, after processing, or both. You can also set a maximum number of seconds for the Vault process to wait for a triggered process to return before resuming operation. An application entity name for the triggered process must exist in the NSM configuration file.

The Vault refers to the command-trigger list to know when to trigger a process. Each Vault command on the list has the following information:

- The Vault command name
- Whether or not to trigger the process (that is, whether the process is active)
- Whether or not to trigger the process before command processing
- Maximum number of seconds to wait for a response from the triggered process before command execution
- Whether or not to trigger the process after command processing
- Maximum number of seconds to wait for a response from the triggered process after command execution
- Application entity name of the triggered process in the NSM configuration file

The active flags for all commands are initially disabled. Use the `chgctl` command to specify triggered process attributes for any of the commands in the list, including enabling or disabling the processes.

Use the special command, `all`, to activate or shut down all active processes. When you execute the `chgctl` command and specify Vault command name as `all` and Active as `n`, the Vault disables all active processes. When you specify `y` with the `all` command, the Vault enables all active processes.

The `all` command functions as a master switch. You can use it to turn on or shut off all processes that are set to be triggered without affecting the setting of the individual commands. It is also a toggle. Even after having activated triggers for various Vault commands, nothing happens until the command `all` is referenced in the `chgctl` command. Do a `chgctl` on `all` once to activate it and do it again to deactivate it. No other arguments are required.

Because trigger-and-wait processes initiated from the Vault server make the server unavailable to other clients until the triggered process is completed, extensive wait times are not recommended. If you do use a trigger-and-wait process and your process does not return within the allotted time, the Vault command fails.

Each Vault command can trigger only one process. Multiple Vault commands can trigger the same process.

Modifying the Command-Trigger List

You can enter changed or new data using the command line format. You cannot delete a command from the command-trigger list. You can only modify its values. The following table lists the parameters for the `chgctl` command.

Table 4-4 chgctl Command Parameters

Keyword	Parameter Description
`COMMAND`	Name of a Vault command. Enter 8 or fewer letters or the special command, `all`. The default is None.
`ACTIVE`	Flag that indicates whether or not the Vault should connect to the triggered process. Enter `Y` (yes) or `N` (no). The default is Yes.
`BEGIN`	Flag that indicates whether or not the Vault should connect to the triggered process at the beginning of command execution. Enter `Y` (yes) or `N` (no). The default is None.
`WBTIME`	Maximum number of seconds the Vault process waits for a triggered process to return a response before Vault command execution. Enter `0` to indicate trigger and proceed without waiting. Otherwise enter a number from `1` through `32767`. The default is zero.
`END`	Flag that indicates whether or not the Vault should connect to the triggered process at the end of command execution. Enter `Y` (yes) or `N` (no). The default is None.
`WETIME`	Maximum number of seconds the Vault process waits for a triggered process to return a response after Vault command execution. Enter `0` to indicate trigger and proceed without waiting. Otherwise enter a number from `1` through `32767`. The default is zero.
`AENAME`	Application entity name of the triggered process in the NSM configuration file. Enter as many as 32 letters and/or numbers. The default is None.

Execute the `chgctl` command to modify the attributes for a command in the command-trigger list.

To use the command line format, enter `cichgctl` and the required parameters at the system prompt.

Using the Command-Line Format

To execute `chgctl` at the system prompt, use the following format:

```
cichgctl COMMAND=EPD command name
ACTIVE=connect to triggered process?
BEGIN=connect before command execution?
WBTIME=maximum waiting time at beginning
END=connect after command execution?
WETIME=maximum waiting time at end
AENAME=application entity name
```

You cannot remove commands from the command-trigger list; you can only modify their values.

Examples

```
cichgctl COMMAND=get ACTIVE=y BEGIN=n END=y,
AENAME=signoutlog
```

In this example, a process called `signoutlog` is activated for the `get` command. The `signoutlog` process is executed after the `get` command execution. The Vault does not wait for a response from the triggered process to continue operation.

```
cichgctl COMMAND=reqrvw ACTIVE=n BEGIN=n END=y, AENAME=review
```

In this example, a process called `review` is associated with the `reqrvw` command. The `review` process is not called following `reqrvw` execution until you change its ACTIVE parameter to `y`.

```
cichgctl COMMAND=all ACTIVE=n
```

In this example, the `all` command turns off all active triggered processes. After you execute this command, commands do not trigger any processes regardless of whether those processes are active or not. For the processes to be triggered, you must execute this command again with the `ACTIVE` parameter set to `y`.

Modifying the Sample Triggered-Program Commands

To enable the commands processed by the sample command-triggered program, enter the following information:

```
cichgctl COMMAND=store ACTIVE=y BEGIN=y WBTIME=10, END=N
AENAME=ctsample
cichgctl COMMAND=reserve ACTIVE=y BEGIN=y WBTIME=10, END=n
AENAME=ctsample
cichgctl COMMAND=chgfa ACTIVE=y BEGIN=y WBTIME=10, END=n
AENAME=ctsample
```

After the `chgctl` command is used to activate the command triggers for `store`, `reserve`, or `chgfa`, as shown above, the next time any of these commands is invoked, the command-triggered process for AE name `ctsample` is called.

The `ctsample` subroutine requires that the `USERTYPE` parameter be entered whenever any of these commands is used. If the command is invoked without the required input, the command is rejected, as shown in the following example:

```
cistore SELNAME=test.ctshell SELSCOPE=f, LFNAME=test.file
CDMSTR422T Processing not done. Usertype input required.
```

If the command is invoked correctly, it succeeds.

```
cistore SELNAME=test.ctshell SELSCOPE=f, LFNAME=test.file
USERTYPE=abcdefg
CDMSTR177I The file has been stored.
```

Chapter 5

Compiling, Linking, and Running a Command-Triggered Program

This chapter provides examples of compiling, linking, and running a command-triggered program.

- Creating an Executable Startup Script File (UNIX)
- Compiling the Source Programs (UNIX)
- Compiling and Linking a Command-Triggered Program
- Linking the Command-Triggered Program (UNIX)
- Running the Command-Triggered Program (UNIX)

Creating an Executable Startup Script File (UNIX)

For Network Services to start your command-triggered process, either automatically or manually, you need to make an executable script file.

Creating the File

The following is an example of an executable script named `startCT` which can be used to start the sample command-triggered program:

```
#!/bin/csh
# C-shell script to start Sample command-triggered Process
if ( ! -d /tmp/tp ) mkdir /tmp/tp
cd /tmp/tp
if ( $?PCAPRCNAME != 0 )
   then
   mkdir $PCAPRCNAME
   if ( -d $PCAPRCNAME) cd $PCAPRCNAME
endif

echo ” CTSample running in directory ‘pwd‘ ”
umask 027
/usr1/jdoe/bin/ctshell ctsample
echo ” CTSample has ended ‘date‘ ”
rm *
cd ..
if ( $?PCAPRCNAME != 0 )
   then
   rmdir $PCAPRCNAME
endif

cd /tmp
rmdir /tmp/tp
exit
```

In the previous script, `PCAPRCNAME` is passed from the PCA process.

Adding Executable Startup Script File to the nsm.config File

For the Vault to invoke your command-triggered process, you must define the AE name of your triggered process in the `nsm.config` file.

An example follows of the necessary entry for the sample command-triggered process running on UNIX, invoked by the executable script `startCT` (shown in the previous section), and started automatically by Network Services.

```
AE(ctsample,pdmgrp,4)
  PATH(/usr1/jdoe/bin/startCT)
  OWNER(jdoe)
  WORKDIR(/usr1/jdoe/work)
  MAXINST(10)
  CLOSE
  GRPCTL(1,1,2)
```

Compiling the Source Programs (UNIX)

After you write a control program and an application subroutine that are to be used in a command-triggered process, they must be compiled using a C compiler.

Please note: `$EPD_HOME` is an environment variable created by the Vault installation procedure. Its default is `$EPD_HOME`

To compile programs that use any of the supplied Vault include files, the `$EPD_HOME/include` path name must be listed as an include directory, as shown below in the examples of compiling the control program (`ctshell`) and application subroutine (`ctsample`) for the sample command-triggered program.

```
% /opt/SUNWspro/bin/cc -c -g -I$EPD_HOME/include \
$EPD_HOME/src/ctsample.c
% /opt/SUNWspro/bin/cc -c -g -I$EPD_HOME/include \
$EPD_HOME/src/ctsample.c
```

Compile options precede the source file. The `-c` option means compile only. The `-g` option means preserve debugging information.

Compiling and Linking a Command-Triggered Program

Based on your requirements, you may need to link with the Oracle libraries.You can use either of two methods to build a command trigger.

- Method 1 — Use the Makefile supplied in $EPD_HOME`/src/trig` as a template, modifying it to use the objects required by your trigger.
- Method 2 — Compile the files separately and use the link lines as supplied in the section “Compiling and Linking a Progammatic Interface Program,” modifying them to use the objects required by your trigger.

If you do not have Oracle requirements, use the platform-specific information for the heading “Linking on <Platform Name> — Vault Only” as a template.

If you have Oracle requirements, use the platform-specific information for the heading “Linking on <Platform Name> — Vault with Oracle 8*i* Release 3 (8.1.7)” as a template.

Please note: The variable <Platform Name> refers to any of the supported systems or substituting <Platform Name> for your UNIX Platform.

In general, the link appears similar to the following:

```
% cc -o ctshell ctshell.o ctsample.o dosmcpdm.o \
      edm_libraries \
      system_libraries
```

The following example of Method 2 uses the sample command-triggered program provided. These instructions are for the Solaris 2.6 platform. The link lines need to be adjusted accordingly to your platform.

1. Compile the `ctshell.c` program.

```
% /opt/SUNWspro/bin/cc -c -Dunix -I$EPD_HOME/include
$EPD_HOME/src/ctshell.c
```

2. Compile the `ctsample.c` program.

```
% /opt/SUNWspro/bin/cc -c -Dunix -I$EPD_HOME/include
$EPD_HOME/src/ctsample.c
```

3. Compile `dosmcpdm.c` with one command (`SIGNON`). This is necessary to resolve NAS undefines.

```
/opt/SUNWspro/bin/cc -c -Dunix -DGENCPDM -DSIGNON_ -DNO_CVKERNEL \
-I$EPD_HOME/include $EPD_HOME/src/dosmcpdm.c

% /opt/SUNWspro/bin/CC -o ctshell ctshell.o ctsample.o dosmcpdm.o\
```

```
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a -lcvclapi \
-lcvnas \
$EPD_HOME/lib/edmpi.a -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt \
-lcvkernel \
-lcvedmstubs -lcvclientstubs \
-lintl -lC -lgen -lsocket -lnsl -lm -ldl
```

`edm_libraries` is `edmpi.a` on a client that is local to the Vault, or `cedmpi.a` on a client that is remote from the Vault. When linking on IRIX clients, you are always remote from the Vault.

In the above example, edm_libraries would map to the following:

```
$EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a lcvclapi -lcvnas\
$EPD_HOME/lib/edmpi.a -lcvnsm -lcvhli -lcvcxx -lcvcaddsrt \
-lcvkernel \
-lcvedmstubs -lcvclientstubs
```

and system_libraries would map to:

```
% cc -g -o ctshell ctshell.o ctsample.o \
$EPD_HOME/lib/edm_libraries \system_libraries

-lintl -lC -lgen -lsocket -lnsl -lm -ldl
```

`system_libraries` has the following value on a given platform:

AIX	`-lm -lcurses -ltermcap -lc -lPW`
HP-UX	`-lc -lm -lPW -lpthread -lrpcsoc -lnsl`
IRIX	`-lc -lm`
OSF/1	`-lm -lcurses -ltermcap -lc -lPW`
Solaris	`-lsocket -lnsl`

If Oracle requirements exist, the system_libraries would remain the same, but the above edm_libraries would change to the following:

For sparc_55:

```
-L$EPD_HOME/lib $EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a \
$ORACLE_HOME/lib/libclntsh.so \
-lcvnas $EPD_HOME/lib/edmpi.a -lcvclapi -lcvnsm -lcvhli \
-lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs
```

For alpha_osf1:

```
-L$EPD_HOME/lib $EPD_HOME/lib/edmpi.a $EPD_HOME/lib/sqlutil.a \
-lclntsh  -laio  -lmld \
-lcvnas $EPD_HOME/lib/edmpi.a -lcvclapi -lcvnsm -lcvhli \
```

```
-lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs
```

For other platforms:

```
-L$EPD_HOME/lib -L$ORACLE_HOME/lib $EPD_HOME/lib/edmpi.a \
$EPD_HOME/lib/sqlutil.a \
-lclntsh \
-lcvnas $EPD_HOME/lib/edmpi.a -lcvclapi -lcvnsm -lcvhli \
-lcvcxx -lcvcaddsrt -lcvkernel -lcvedmstubs
```

Please note: For building a command trigger program on Windows NT, use the makefile in `%EDM_HOME%\src\trig` as a template to construct the makefile.Visual C++ 6.0 Service pack 3 is required to build the command trigger program.

Linking the Command-Triggered Program (UNIX)

The following example links the sample command-triggered program to the appropriate libraries:

```
% cc -g -o ctshell ctshell.o ctsample.o \
$EPD_HOME/lib/edm_libraries \
system_libraries
```

`edm_libraries` is `edmpi.a` on a client that is local to the Vault, or `cedmpi.a` on a client that is remote from the Vault. When linking on IRIX clients, you are always remote from the Vault.

`system_libraries` has the following value on a given platform:

AIX	`-lm -lcurses -ltermcap -lc -lPW`
HP-UX	`-lc -lm -lPW -lpthread -lrpcsoc -lnsl`
IRIX	`-lc -lm`
OSF/1	`-lm -lcurses -ltermcap -lc -lPW`
Solaris	`-lsocket -lnsl`

Running the Command-Triggered Program (UNIX)

After adding a command trigger AE to the NSM configuration file and restarting your Vault processes, your command-trigger can be started automatically when the Vault starts.

However, if you would prefer, you can start or restart your command trigger manually by entering.

```
% ctshell ctsample
```

The `startCT` script cannot be used to start the process manually. Each instance of a command-triggered process should run from a different directory to avoid overwriting data files.

Once your command-triggered process is running, it is available for Vault commands to send messages to it and receive replies from it. Messages are sent to your command-triggered process only through Vault commands which have been enabled using the `chgctl` command. You must have your command-triggered process running before using the `chgctl` command to enable Vault commands to direct messages to it.

Chapter 6 Customizing the Client

This chapter describes the functions supported by the `edmosrv` library used for programming and customizing the Vault client.

- Overview of the edmosrv Library
- edmosrv Functions
- SCRAMBLE Library
- Customizing Through the Perl Interface

Overview of the edmosrv Library

The edmosrv library exists as a static library on the AIX operating systems. It exists as a shared library on all the other operating systems as libcvedmosrv.<ext> where ext is a shared library extension. This is edmosrv32.dll on Windows. The client must be linked to edmosrv32.lib.

The files are installed as specified in the table below.

Table 6-1

Library	Windows	UNIX
edmosrv32.dll	Windows system	-
edmosrv32.lib(Connect)	EPD_HOME/lib	-
edmosrv32.lib(Locator sdk)	EPD_HOME/sdk/lib	–
libcvedmosrv.*	-	$EPD_HOME/lib

The include files associated with the library are as follows:

For Windows:

$EPD_HOME/data/edmosrv/edmopub.h.
$EPD_HOME/data/edmosrv/edmopri.h
$EPD_HOME/data/edmosrv/edmcall.h
$EPD_HOME/data/edmosrv/sqlprnt.h

For UNIX:

$EPD_HOME/include/edmosrv/edmopub.h.
$EPD_HOME/include/edmosrv/edmopri.h
$EPD_HOME/include/edmosrv/nokernel.h
$EPD_HOME/include/edmosrv/sqlprnt.h

Please note: Define the variable NO_CVKERNEL in the custom programs before including the header files.

The following libraries are required by the `edmosrv` library for building custom applications:

- UNIX: `cedmpi.a` , `libcvkernel.a`, `liboptscramble.a`

  The library is specific only to the IBM platform. It is necessary to link the executable with the library `liboptscramble.a`.

- Windows: `optscram.lib, edmosrv.lib`

The `edmosrv` client library can scan the ANSPATH for resolving the full domain name of the server.

If the `edmosrv` client and the `edmosrv` server are in different network domains, specify the full name of the node in the `pm.config` file and set the environment variable EDMOANS to 1. On the UNIX platform, set the environment variable using the command `setenv EDMOANS 1`. On the Windows NT platform, set the EDMOANS variable using Start > Settings > Control Panel > System > Environment.

edmosrv Functions

The `edmosrv` library provides the following functions.

SQL Functions

The syntax of all the SQL functions are as follows:

- `EDM_O_STAT edm_o_connect(const char *server, const char *usrid, const char *paswd, *msg)`

  This function connects to Oracle. The function returns `EDM_O_OK` for a successful connection.

- `EDM_O_STAT edm_o_query(const char *query, EDM_O_HANDLE *handle, char *msg)`

  This function assists in issuing a query to the database. The function supports queries like selecting, inserting, updating, or deleting information.

- `EDM_O_STAT edm_o_bindquery(char *query, char *bindvalue1, char *bindvalue2,....,char *0, EDM_O_HANDLE *hand, char *msg)`

  This function assists in issuing a bind variable query to the database. The function supports queries like selecting information. The char *0 argument marks the end of the bind value arguments issued to the database. The function returns integer values. It returns `EDM_O_OK` if the query is executed successfully.

Please note: A maximum of 64 bind variables are supported in the query.

- `EDM_O_STAT edm_o_fetch(EDM_O_HANDLE *handle, char *msg)`

  This function fetches the results of the query, one row at a time. When all the rows are fetched, the function returns `EDM_O_NFOUND`.

- `EDM_O_STAT edm_o_close(EDM_O_HANDLE *handle, char *msg)`

  This function closes the query and removes the memory allocated for the operation.

- `EDM_O_STAT edm_o_disconnect(char *msg)`

  This function assists in disconnecting from the server.

- `EDM_O_STAT edm_o_commit(char *msg)`
- `EDM_O_STAT edm_o_rollback(char *msg)`

  These functions update the changes made to the database and retain the results permanently.

Return Codes

The following table shows the possible return codes and their explanations.

Table 6-2

Return Code	Explanation
EDM_O_OK	Successfully connected to the server
EDM_O_SON	Already connected to the DATABASE
EDM_O_NOUSER	No user name supplied
EDM_O_NOPW	No user password supplied
EDM_O_NOCHAR	Query returned unsupported data type
EDM_O_NSON	Not connected to DATABASE
EDM_O_NOCONNECT	Failed to connect to server
EDM_O_NOSOCK	Bad socket (Communications error on server)
EDM_O_NOFREE	No free RPC (Remote Procedure Call) program numbers
EDM_O_EXPIRED	Licence has expired
EDM_O_NFOUND	No more data to FETCH

In addition to the return codes mentioned in the above table, the Oracle return codes and messages are also returned.

Procedure for SELECT

For SELECT, do the following:

1. Connect to the server using `edm_o_connect(server,usrid,paswd,msg)`. The password is scrambled and passed to the server so that it is not visible to malicious users on the network.
2. Issue a query using `edm_o_query(query,handle,msg)`.
3. Fetch the results one row at a time using `edm_o_fetch(handle,msg)`.
4. Close the query using `edm_o_close(handle,msg)`.
5. Repeat steps 2, 3, and 4 until all the queries are completed.
6. Disconnect from the server using `edm_o_disconnect(msg)`.

Procedure for UPDATES

For UPDATES, do the following:

1. Connect to the server using `edm_o_connect(server,usrid,paswd,msg)`.
2. Issue the query using `edm_o_query(query,handle,msg)`and fetch the results.
3. Commit the changes using `edm_o_commit(msg)`.

   Or

   Roll back using `edm_o_rollback(msg)`.
4. Disconnect from the server using `edm_o_disconnect(msg)`.

Other Functions

- `EDM_O_STAT edm_o_file_query(FILE *, EDM_O_HANDLE *, char *)`

  This function is similar to `edm_o_query`.

Please note: Only one query can be active for a connection at a time.

- `int edm_o_numcol(EDM_O_HANDLE *handle, char *msg)`

  This function returns the number of columns.

- `int edm_o_getnthcol(EDM_O_HANDLE *handle, int col, char *str, int maxlen)`

  This function extracts up to `maxlen` characters from the `col` column and copies the characters to `str`. The `maxlen` includes the terminating null character written to the string.

  The function returns the number of characters written, not including the terminating null. Hence, the maximum value returned is (`maxlen` - 1), but not less than 0.

  This function returns 0 if a problem, such as col larger than the number of available columns, occurs.

- `int32_t edm_o_getnthlen(EDM_O_HANDLE *handle, int col)`

  This function gets the length stored in the B array of long `ints` in the handle. The array is valid only after the query, and it describes the length of the columns.

  The function returns 0 if a problem, such as the value of `col` being larger than the number of available columns, occurs.

Example

The following sample program shows how to sign on to the database and use the `edm_o_bindquery` function to query the interface.

```
EDMOSTAT return_val;
char msg[512];
char ret_param[24]
EDM_O_HANDLE hand;

ret_val = edm_o_connect("scott","tiger","emp_database",msg);
 if(return_val == EDM_O_OK) {
    return_val = edm_o_bindquery("select * from emp where emp_name
=:b1,dept_no =:b2","Scott","801",(char*)0, &hand,msg);
  if(return_val !=EDM_O_OK)
/* Execute failure algorithm */
  else {
   while((return_val=edm_o_fetch(&hand, msg))!=
EDM_O_NFOUND)
{
   /* Do all your operations here */
  if(return_value!=EDM_O_OK) {
/* Execute failure algorithm */
    }
  }
edm_o_close(&hand,msg);
edm_o_disconnect(msg);
}
```

SCRAMBLE Library

This section describes the functions and utilities supported by the `Scramble` library. The `Scramble` library has been implemented as a static library and is also used by the `edmosrv` library. It is used for scrambling or unscrambling strings.

The scramble library gets installed as per the table below.

Table 6-3

Library Name	Windows	Unix (IBM)
`optscram.lib` (EPD.Connect)	$EPD_HOME`/lib`	-
`optscram.lib` (Locator SDK)	$EPD_HOME`/sdk/lib`	-
`liboptscramble.a` (EPD.Connect)	-	$EPD_HOME/`lib`

Please note: The `liboptscramble.a` library is installed with EPD.Connect in UNIX only on the IBM platform. In all the other platforms, it is available through the `edmosrv` library. The include file for this library is located in:

$EPD_HOME`/include/optscram/scramble.h`

Functions

The `Scramble` library provides the following functions:

- `int scramble (unsigned char* string_A, unsigned char* string_B)`

  where `string_A` is a string to be scrambled and `string_B` is the buffer into which the scrambled string is returned.

  The memory for both the strings must be allocated by the caller of the function. The memory allocated for `string_B` must be equal to the following:

  `(3 bytes + (2 x length of string_A))`

  This function supports Japanese characters. The unscramble function always returns 1.

- `int unscramble (unsigned char* string_A, unsigned char* string_B)`

  where `string_A` is a scrambled string, and `string_B` is the buffer in which the unscrambled string is stored.

The memory for both the strings must be allocated by the caller of the function. Memory allocated for `string_B` must be equal to the following:

```
((length of string_A / 2) - 1 byte)
```

The unscrambling algorithm supports Japanese characters. The unscramble function always returns 0.

Utilities

In addition to the preceding functions, the following utility converts a text string to a scrambled string. This utility is useful for manually storing the scrambled string in files or tables.

- `scramexe` (On an operating system based on Unix)
- `scramexe.exe` (Windows)

Usage: `scramexe` string

This utility displays the following output:

```
The password encryption utility for Optegra.
Tiger scrambled to MFHFAKFBCHFK.
```

The client can store the displayed string in a scrambled form. This utility is being shipped with EPD.Connect and Vault in the $EPD_HOME`/bin` directory. The password in the `epdconn.ini` file is also scrambled using this utility.

Customizing Through the Perl Interface

You can use the `edmosrv` library also through the Perl interface to customize the client.

The following sample program shows how you can connect to Oracle and print the employee name from the employee file:

```
#! /usr/CVperl/bin/perl

use Edmosrv;

$ip = "hathi";

$ret = Edmosrv'connect($ip, "scott", "tiger");
if ( $ret == 0 ) {
    print "connected to vault $ip\n";
    $ret = Edmosrv'query("SELECT ENAME from EMP");
    if ( $ret != 0 ) {
        print ("Failed to query\n");
        exit(1);
    }
    while ( !Edmosrv'fetch()) {
        $col = Edmosrv'no_columns();
        for ( $i=0; $i<$col; $i++ ) {
            print "output = ", Edmosrv'column_data($i), "\n";
        }
    }
    $ret = Edmosrv'close();
    if ( $ret != 0 ) {
        print "Could not close the query\n";
    }
    $ret = Edmosrv'disconnect();
    print "disconnected from vault $ip\n";
}
else {
    print "could not connect to vault $ip\n";
}
```

Perl Functions for the edmosrv Library

The syntax of all the Perl functions supported by `edmosrv` is given below:

- `debug_on()`

  This function enables the logging of error messages.

- `debug_off()`

  This function disables the logging of error messages.

- `get_debug()`

  This function displays whether the `DEBUG` mode is On or Off. The function returns an integer value.

- `msgtext()`

  This function returns the last error message, as a string of characters.

- connect(domain, userid, passwd)

  This function connects to the Oracle database. It returns 0 on success.

- `query(query)`

  This function assists in issuing a query to the database. The function supports queries such asselecting, inserting, updating, or deleting information. It returns an integer value.

- `commit()` and `rollback()`

  These two functions update the changes made to the database and retain the results permanently. These functions return integer values.

- `fetch()`

  This function fetches the results of the query, one row at a time. The function returns 0 on success.

- `close()`

  This function closes the query and removes the memory allocated for the operation. The function returns 0 on success.

- `no_columns()`

  This function returns the number of columns.

- `column_data(col)`

  This function extracts characters from the `col` column and copies the characters to a string.

  This function returns 0 if a problem, such as col larger than the number of available columns, occurs.

- `disconnect()`

  This function disconnects the connection to the database.

- `is_connected()`

  This function returns the status of connection to the database.

Perl Functions for the Scramble Library

The `Scramble` library through the Perl interface provides the following functions:

- `scramble(string)`

  This function returns a scrambled string.

- `unscramble(string)`

  This function accepts a scrambled string and returns an unscrambled string.

Chapter 7

Vault Commands

This chapter contains an alphabetical listing of all Vault commands for which there is a programmatic interface.

- Command Overview
- Vault Command

Command Overview

This section contains an alphabetical listing of all Vault commands for which there is a programmatic interface. The entry for each command includes

- Three-letter command abbreviation
- Structure name
- Header file name
- Brief description

Note the following:

- You must consult the *Vault Command Reference* for input parameter information, including field size
- Field size is expressed in bytes
- All unspecified fields in the records must be initialized to blanks
- If you set a field to blanks, its default value (if any) is used
- Alphanumeric characters do not include special characters (# and @, for example)
- Fields that do not occupy the entire allotted space must be blank-filled
- Null terminators are not used anywhere in the records

If you need more information about any command or parameter, use your on-line help facility to locate the command in one of the following books:

- *Vault Manager Guide*
- *Vault Administrator for Windows NT User Guide*
- *Vault Interactive Query Facility Guide*

Please note: If your system administrator has changed the default value for any parameter, the default shown in the *Vault Command Reference* will not be correct for your system.

Vault Command

Table 7-1 Alphabetical Listing of Vault Commands

Command	Abbreviation	Structure Name	Header File	Command Description
ADDAG	AAG	addag_struct	adaag.h	Add an authority group to Vault
ADDCL	ACL	addcl_struct	adacl.h	Add a command list to Vault
ADDFS	AFS	addfs_struct	adafs.h	Add a file set
ADDMFS	AMS	addmfs_struct	adams.h	Add a member to a file set
ADDMUL	AML	addmul_struct	adaml.h	Add members to a user list
ADDP	ADP	addp_struct	adadp.h	Add a project to Vault
ADDRS	ARS	addrs_struct	adars.h	Add a revision code sequence
ADDS	ADS	adds_struct	adads.h	Add a status level to an authority scheme
ADDSP	ASP	addsp_struct	adasp.h	Add a storage pool to Vault
ADDT	ADT	addt_struct	adadt.h	Add a user-defined table to the control of Vault
ADDU	ADU	addu_struct	adadu.h	Add a user to Vault
ADDUL	AUL	addul_struct	adaul.h	Add a user list name
ADDUP	AUP	addup_struct	adaup.h	Add a user to a project
ADDUSA	ALS	addusa_struct	adals.h	Add a user list/status code association
ADMCOPY	ADM	admcopy_struct	adadm.h	Copy administrative data
ARCHIVE	ARC	archive_struct	adarc.h	Move files marked for archiving from Vault to tape
CHGAG	CAG	chgag.struct	adcag.h	Change entries in an authority group
CHGCL	CCL	chgcl_struct	adccl.h	Change entries in a command list
CHGCTL	CTL	chgctl_struct	adctl.h	Change command trigger list
CHGFA	CFA	chgfa_struct	adcfa.h	Change file(s) attributes
CHGFCL	CFC	chgcl_struct	adcfc.h	Change file(s) classification
CHGFPW	CFP	chgfpw_struct	adcfp.h	Change file(s) password
CHGFREV	CFR	chgfrev_struct	adcfr.h	Change file(s) revision code
CHGFSC	CFS	chgfsc_struct	adcfs.h	Change file(s) status code
CHGFSP	FSP	chgfsp_struct	adfsp.h	Change a file’s storage pool
CHGP	CPA	chgp_struct	adcpa.h	Change the attributes of a project
CHGS	CSA	chgs_struct	adcsa.h	Change the attributes of a status level
CHGSPS	CPS	chgsps_struct	adcps.h	Change storage pool status
CHGSPT	CPT	chgspt_struct	adcpt.h	Change storage pool type
CHGU	CUA	chgu_struct	adcua.h	Change an Vault user’s attributes and/or authority
CHGUP	CUP	chgup_struct	adcup.h	Change a user’s project authority
CHGUPW	CPW	chgupw_struct	adcpw.h	Change your Vault user password
CLST	CLT	clst_struct	adclt.h	Close a user-defined table
COPY	CPY	copy_struct	adcpy.h	Copy file(s) to new Vault file(s)
DELAG	DAG	delag_struct	addat.h	Delete an authority group from Vault
DELCL	DCL	delcl_struct	addcl.h	Delete a command list from Vault
DELETE	DEL	delete_struct	addel.h	Delete files marked for deletion
DELLOG	DLG	dellog_struct	addlg.h	Delete Vault audit log entries
DELP	DLP	delp_struct	addlp.h	Delete a project from Vault

Table 7-1 Alphabetical Listing of Vault Commands

Command	Abbreviation	Structure Name	Header File	Command Description
DELS	DLS	dels_struct	addls.h	Delete a status level from an authority scheme
DELU	DLU	delu_struct	addlu.h	Delete a user from the Vault
DELUL	DUL	delul_struct	addul.h	Delete a user list
GET	GET	get_struct	adget.h	Sign out and copy file(s); modifications allowed
IBKUP	IBU	ibkup_struct	adibu.h	Back up new and modified files
LISTDIR	DIR	listdir_struct	addir.h	List file(s) on local or remote Vault node
LOAD	LOA	load_struct	adloa.h	Load file(s) from tape to the Vault
MARKA	MKA	marka_struct	admka.h	Mark file(s) to be archived
MARKD	MKD	markd_struct	admkd.h	Mark file(s) to be deleted
MARKR	MKR	markr_struct	admkr.h	Mark file(s) to be restored
OPNT	OPT	opnt_struct	adopt.h	Open a user-defined table
PURGE	PUR	purge_struct	adpur.h	Delete file(s) not previously marked for deletion
READ	REA	read_struct	adrea.h	Copy file(s); modifications not allowed
READMSG	RMG	readmsg_struct	adrmg.h	Read messages
RECSF	RSF	recsf_struct	adrsf.h	Recover a single file
RECSP	RSP	recsp_struct	adrsp.h	Recover a storage pool
REMFS	RFS	remfs_struct	adrfs.h	Remove a file set
REMMFS	RMS	remmfs_struct	adrms.h	Remove a member from a file set
REMMUL	RML	remmul_struct	adrml.h	Remove a member from a user list
REMT	RMT	remt_struct	adrmt.h	Remove a user-defined table from Vault control
REMUP	RUP	remup_struct	adrup.h	Remove a user from a project
REMUSA	RLS	remusa_struct	adrls.h	Remove a user list/status code association
REPLACE	REP	replace_struct	adrep.h	Save modified file(s); ability to modify ends
REQRVW	RVW	reqrvw_struct	adrvw.h	Request review of file(s)
RESERVE	RSV	reserve_struct	adrsv.h	Define and sign out an Vault file name for future use
RESET	RST	reset_struct	adrst.h	Cancel signing out of file(s); changes are lost
RESTORE	RES	restore_struct	adres.h	Restore files from tape to Vault
RSVP	RVP	rsvp_struct	adrvp.h	Respond to a review request
SCANTAPE	SCN	scantape_struct	adscn.h	List the contents of an Vault tape
SENDMSG	SMG	sendmsg_struct	adsmg.h	Send a message
SIGNOFF	SOF	signoff_struct	adsof.h	Sign off and exit Vault session
SIGNON	SON	signon_struct	adson.h	Sign on to Vault or another Vault user ID
SIGNOUT	SOT	signout_struct	adsot.h	Sign out file(s) to modify; file(s) not copied
STORE	STR	store_struct	adstr.h	Add a new file
UNLOAD	UNL	unload_struct	adunl.h	Unload files to tape
UNMARK	UMK	unmark_struct	adumk.h	Unmark previously marked files
UPDATE	UPT	update_struct	adupt.h	Save modified file(s); continue modifications

Appendix A Sample C Program

This appendix shows a sample C program using the programmatic interface.

- svedmsample.c

svedmsample.c

This appendix shows a sample C program using the programmatic interface. It includes the `svedmsample.c` file, which is the source file for the program.

```
File Name: svedmsample.c

Purpose:  Demonstrates using the EDM Programmatic Interface in C.
Function:  The program executes a SIGNON to EDM using the
           EDM User ID and Password supplied.The named local file
           is STORE'd using an optionally provided selection name
           or the local file name.
           The program executes a SIGNOFF from EDM.
How to Use:Invoke program with the following command line
parameters.

      EDM User ID,
      EDM User Password,
      Local name of file to store,
      EDM selection name for file (optional).

#include <stdio.h>
#include "adkeylen.h"
#define OK
#define PI_ERROR
#define PI_ARGERR "Incorrect number of arguments on command line."
#define PI_COMMAND  " EDM command    -- "
#define PI_DISPLAY1 "  pi return code  - "
#define PI_DISPLAY2 "  output message  - "
#define PI_EXIT_MSG " Sample program has finished.
extern int SVedm();
static int execute_signon();
static int execute_store();
static int execute_signoff();
static void display_returned();
/***  main program  ***/
main(argc, argv)
int     argc;
char    *argv[];

 {
    int  retcode = OK;
  if ((argc < 4) || (argc > 5))
      {
        printf("%s \n", PI_ARGERR);
        retcode = PI_ERROR;
        goto errexit;
      }
retcode = execute_signon(argv[1],argv[2]);
    if (retcode != OK)
        goto errexit;
    if (argc == 5)
      retcode = execute_store(argv[3], argv[4]);
```

```
    else
      retcode = execute_store(argv[3], argv[3]);
    retcode = execute_signoff();
errexit:
    printf("%s \n", PI_EXIT_MSG);
    exit(retcode);
  }
/***  execute_signon procedure  ***/
static int execute_signon (userid, userpw)
char *userid;
char *userpw;
  {
    int retcode = OK;
    char msg_buffer[LTOTALMSG+1];
    printf("%s %s\n", PI_COMMAND, "SIGNON");
    retcode = SVedm("SIGNON", "USERID", userid, "USERPW", userpw,
                "MESSAGE", msg_buffer, (char *) 0);
    display_returned(retcode, msg_buffer);
    return (retcode);
  }
/***  execute_store procedure  ***/
static int execute_store (local_filename, selection_name)
char *local_filename;
char *selection_name;
  {
    int retcode = OK;
    char msg_buffer[LTOTALMSG+1];
    printf("%s %s\n", PI_COMMAND, "STORE");
    retcode = SVedm("STORE", "LFNAME", local_filename, "SELSCOPE",
"F",
            "SELNAME", selection_name, "MESSAGE", msg_buffer,
(char *) 0);
    display_returned(retcode, msg_buffer);
        return (retcode);
  }
/***  execute_signoff procedure  ***/
static int execute_signoff()
  {
    int retcode = OK;
    char msg_buffer[LTOTALMSG+1];
    printf("%s %s\n", PI_COMMAND, "SIGNOFF");
    retcode = SVedm( "SIGNOFF", "MESSAGE", msg_buffer, (char *) 0);
    display_returned(retcode, msg_buffer);
    return (retcode);
  }
/***  display_returned procedure: displays return codes and
message from programmatic interface call  ***/
static void display_returned (retcode, msg_buffer)
  int retcode;
  char* msg_buffer;
  {
    printf("%s %d\n", PI_DISPLAY1, retcode);
    printf("%s %s\n", PI_DISPLAY2, msg_buffer);
  }
```

Appendix B

Using the Vault Rules Processor Language

This appendix presents an overview of the Vault Rules Processor language and provides instructions for creating new data types and classification functions.

- Overview of the Vault Rules Processor Language
- Implementing Attributes
- Structure of the Language
- Creating a Data Type
- Creating a Classification Function
- When Simple Classification Functions Are Too Simple
- Compiling and Loading Rules and Data Types

Overview of the Vault Rules Processor Language

The Vault Rules Processor is an Vault-specific programming language that you can use to determine when and which set of user-defined attributes should be applied to a file or part and to ensure that attribute data conforms to the requirements of your site.

The Vault Rules Processor uses simple constructs common to most programming languages, such as, the “if-then-else statement,” combined with embedded SQL (relational database) code.

You use the Vault Rules Processor language in conjunction with the Vault user-defined attribute commands which follow. See *Vault Manager Guide* for information about using these commands.

- `ADDATTR` defines an attribute
- `ADDASET` defines an attribute set
- `ADDMAS` adds a member to an attribute set. A member can be an attribute or another attribute set
- `ADDRULE` defines a simple rule for applying attributes
- `CHGMAS` changes the characteristics of an attribute set member
- `REMMAS` removes an attribute from a set
- `DELATTR` deletes an attribute
- `DELASET` deletes an attribute set
- `DELRULE` deletes a rule used for applying attributes

After attributes are defined, users assign values to attributes for particular objects when they execute the following commands:

- `STORE`, to store a new file or part in the vault
- `UPDATE`, to save changes made to a file or part
- `REPLACE`, to sign a modified file or part back into the Vault

This appendix provides the following:

- A review of the salient points of attribute management
- An overview of the Vault Rules Processor language
- Instructions for creating a new data type
- Instructions for creating a new classification function
- Description of the Vault Rules Processor language specification
- Source for the supplied data types and classification rules
- Instructions for compiling and loading rules and data types

Implementing Attributes

An attribute is a piece of information related to a file or part. You can set up Vault so that when users execute the `STORE`, `REPLACE`, or `UPDATE` commands, they enter attribute values along with the information required for file transfer.

An attribute has a

- Name, used to refer to the attribute data and consisting of up to 24 characters, including numbers, underscores and hyphens
- Type, a function that you write to enforce data input rules (used to verify the attribute data before it is stored in the database)
- Description, for information only

You use the `ADDATTR` command to define the name of the attribute and the type of data the attribute represents. Users assign the values when they execute the `STORE`, `REPLACE`, or `UPDATE` commands.

The Vault has the following predefined attribute data types:

- CHARACTER accepts from 1 to 240 characters. If your RDBMS is Oracle, the Vault uses the `length` predicate to count the number of characters.
- NUMBER accepts integers. If your RDBMS is Oracle, the Vault uses the `to_number` predicate to validate the input.
- DATE accepts dates in a DD-MMM-YY format. If your RDBMS is Oracle, the Vault uses the `to_date` predicate to validate the input.

The data types are defined in the `attrtype.cg` file. You can modify this file to create additional data types. The data types above can be modified or discarded in favor of others. Subsequent sections of this appendix describes how to create a data type.

For an attribute to be associated with a file or part, that attribute must be a member of an attribute set. A set is a convenient way to group attributes and can include other sets as well as individual attributes. Use the `ADDASET` command to define an attribute set.

When an attribute (or an attribute set) is added to a set, you must specify whether the attribute is optional or required and establish a default value. Default values, other than the reserved "no-default," are verified (by matching data types) before an attribute is added to a set. For required members of a set, the Vault command fails if the user does not enter an attribute value. Use the `ADDMAS` command to add an attribute or attribute set to an attribute set.

After you have defined attributes and attribute sets, you can construct simple rules to determine when to apply them to particular files/parts.

A rule has the following:

- Rule name, used to refer to the rule definition
- Set name, set of attributes applied to the file or part if the function evaluates to TRUE
- Simple classification function name, function written to classify the file or part
- Operator, input to the function
- Test value, input to the function

Use the `ADDRULE` command to add a rule to the database.

The simple classification function names predefined by the Vault follow:

- `USER`, applies the operator and test value to the Vault user
- `PROJID`, applies the operator and the test value to the Vault project identifier
- `STATUS`, applies the operator and test value to the Vault status code
- `CLASS`, applies the operator and test value to the Vault file/part classification
- `USERTYPE`, applies the operator and test value to the Vault file/part user type
- `PARTNUM`, applies the operator and test value to the Vault part number

The simple classification functions are defined in the `attrrule.cg` file. Modify this file to create additional simple classification rules.

The simple classification functions listed above can be modified or discarded in favor of others. Subsequent sections of this appendix describe how to create a simple classification function.

Structure of the Language

The structure of the Vault Rules Processor program can be described in sections as follows:

1. Comments, must be the first section of your program, used to write program modification history, usage notes, etc. (No other portion of the program should contain comments.)
2. Table information, must follow Comments, used to describe compiler output format.
3. Definitions (an optional section), must follow Table information, used to define character sets and lists of valid programming values.
4. Macro (an optional section), must follow Definitions, used to define the tokens used to validate data. Macro is optional if the definition section is not present.
5. Test, last section of program, used to define functions for validating attribute data and classifying the Vault objects.

Variables

The Vault Rules Processor offers the following integer variables that you can set in an assignment statement and test in a conditional statement:

- `SQLCODE`, set by the interpreter after you execute an SQL statement, the value of this variable indicates the correctness of the SQL statement.
- `ROWCOUNT`, set by the interpreter after you execute an SQL statement, the value of this variable indicates the number of relational table rows your SQL statement has affected.
- `RESULT`, set by the interpreter after you execute a data validation statement or set by assignment, the value of this variable indicates the final outcome of the test sequence you designed.

Input and Output

The Vault Rules Processor language does not support input and output from files or terminals. However, the language does allow for the passing of data into a program via an `input stream`.

An `input stream` is a stream of inputs that have the form

```
<keyword>=<data>
```

An input stream is a stream of inputs that have the form `keyword=data`. In order to extract `<data>`, the program refers to `:<keyword>` in an expression in which input substitution is required.

For example,

```
':DM_XCTN_USER_ID' :
```

refers to the Vault user identifier. The quotation marks are used to treat the resulting string as a literal.

The content of the input stream depends on the type of object the user is classifying and on the inputs the user passes via the `attrfile`. For example, for an object of type `FILE`, the contents of the Vault `DM_FILE_DIRECTORY` and the `DM_XCTN_CONTROL` tables is available in the input stream. Also, any attributes the user has written to the `attrfile` is available to the program.

Statement Types Supported

A statement in the Vault Rules Processor Language can be one of the following types:

- Compound statement
- Assignment statement
- Conditional statement
- Function invocation
- Built-in function
- SQL statement
- Validation statement

Creating Statements and Blocks

The Vault Rules Processor language uses the semicolon (;) as a statement terminator, for example

```
result = 0;
sqlcode = 100;
```

Use a `do` and `end` to group simple statements into a compound statement, or "para:0" of code. (A block of code enclosed by a `do` and `end` is treated as a single statement by the Vault Rules Processor language.)

Look at the placement of the `do` and `end` in the block which follows:

```
if expression
then do
statement1;
statement2;
end;
```

The `if-then` statement, nested in the example above, executes only if the conditions established in the `expression` following the `if` clause are evaluated to be true.

Please note: Each nested statement is terminated by a semicolon, as is the entire block.

Testing for Specific Conditions

To test for specific conditions before executing a statement, use an `if-then-else` sequence. If an expression meets the criteria established, the statement following `then` is executed. Otherwise, the statement following the `ELSE` is executed. Whenever an `else` is used, the previous statement is not terminated by a semicolon.

The following is an example `if-then-else` sequence, where `statement1` is terminated by the `else` clause not a semicolon:

```
if expression
then do
statement1
statement2
end
else do
statement3;
statement4;
end;
```

`if-then-else` statements can be nested. The usual cautions regarding nested `if-then-else` statements apply. In the following sequence, for example, an `ELSE` clause exists for every `IF-THEN` clause:

```
if expression1
then if expression2
then if expression3
then ...
else if expression4
then ...
else ...
else ...
else ...
```

Writing Functions

Functions, also called tests, break large tasks into smaller ones. A program written in the Vault Rules Processor language contains at least one function, called `root`, which acts as the starting point for the program.

The following is the general form of a function. `end-test` informs the compiler that it has reached the end of the body of the function:

```
root :
statement-list
end-test
```

The functions you add to the `attrrule.cg` file are the simple classification functions used in the `ADDRULE` command. Similarly, the functions you add to the `attrtype.cg` file are the data type functions that verifies the attribute data.

You can define a function before the body of the function or after. The compiler ensures that the body of the function does exist if a call is made. Functions can not be nested and recursive function calls are not allowed. (The former would be detected by the compiler and the latter would be detected by the interpreter.)

In the following example, the `root` function calls `validate-name` and `validate-name` is defined afterward:

```
root :
validate-name();
if result = 0
then insert-name();
end-test

validate-name :
statements for validating a name

end-test
```

Using Built-in Functions

A built-in function statement instructs the interpreter to call a function that has been precompiled and bound to the interpreter. Built-in functions are written to handle situations that either the interpreter cannot handle easily or that the host SQL language does not support.

In the following example, the built-in function is invoked from within the interpreter's environment.

The interpreter must remain connected as the `EDMATTR SQL` user id for the duration of the program, so it allows only one form of the SQL COMMIT.

```
#commit();
```

SQL Statements

An SQL statement instructs the interpreter to invoke the host SQL processor. Only the `SELECT`, `DELETE`, `INSERT`, and `UPDATE SQL` statements are allowed in the language.

In the following example, the ORACLE RDBMS is being used to compare the input OBJECT-TYPE to the literal 'FILE':

```
exec sql select count(*) from attr_dummy
where upper(':OBJECT-TYPE') = 'FILE';
```

If the SQL statement is correctly formatted, SQLCODE contains `0`. Otherwise, it contains the SQL error code. The `attr_dummy` table has a single row. If the OBJECT-TYPE is 'FILE', then ROWCOUNT is set to `one`. Otherwise, it contains `zero`.

While the interpreter is executing a program, it is connected to the database as `EDMATTR`. Any tables referenced by the program must have the proper permissions granted to `EDMATTR`.

Data Validation Statements

A data validation statement is used to ensure that input data is of the correct format. This statement uses the macros and definitions to make the input data into tokens.

In the following example, the statement is being used to verify that the attribute data is an item of a list:

```
definitions
      letters : 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
      colors : 'RED', 'YELLOW', 'GREEN', 'BLUE', 'BLACK';
end-definitions
macros
      the-colors : (6) letters memberof colors;

end-macros
...
...

exec data validate :$$ATTRDATA using the-colors;

...
...
```

The macro `the-colors` reads up to six letters which forms a token and is compared to each member of the list 'colors'. If `$$ATTRDATA` is a member of the list, the statement sets the `RESULT` variable to zero. Otherwise, the `RESULT` will be nonzero.

Creating a Data Type

The purpose of a data type is to ensure the attribute data meets the requirements of the site before the data is stored in the database. Data types can be simple or complex.

An example of a simple data type is the character data type that is provided. This data type accepts from 1 to 240 characters. The following fragment of code is used to verify the input data is of type character:

```
    exec sql select count(*) from attr_dummy
      where length(':$$ATTRDATA') > 0
      and  length(':$$ATTRDATA') < 241;
```

The Vault Rules processor substitutes each occurrence of `$$ATTRDATA` with the attribute data. The entire statement is then thrown at the SQL RDBMS for evaluation.

The character data type is too simplistic for most uses. For example, for an attribute named COLOR, the site probably wants to restrict the attribute data to some list of valid colors. The following fragments of code illustrate two ways to handle a list of items. In both cases, an attribute has to be added to the database with the type color1 or color2.

This example assumes that a table of widgets has been defined and populated. One of the columns of this table has the list of valid colors.

```
color1 :
    exec sql select count(*) from carlstest.widgets
      where upper(':$$ATTRDATA') = widget_color;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
                  then result = 700
                  else result = 0;
end-test
```

This example uses the list processing of the data validation statement. It assumes that the list of valid colors is small and does not change frequently.

```
definitions
      chars : 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
      colors : 'RED', 'BLUE', 'YELLOW', 'GREEN',
                  'BLACK';
end-definitions
macros
      the-color : (6) chars member of colors;
end-macros
color2 :
    exec data validate :$$ATTRDATA using the-color;
    if result <> 0
      then result = 700;
end-test
```

Creating a Classification Function

The purpose of a classification function is to determine if a set of attributes should be applied to an object. A classification function can be simple or complex. An example of a simple classification function is the USER function that is provided. This function applies the Vault user identifier to a test value using an SQL operator. The following fragment of code represents the body of the function:

```
exec sql select count(*) from attr_dummy
  where ':DM_XCTN_USER_ID' :$$OPERATOR ':$$ATTRDATA'
  ;
```

After a rule definition has been added to the database, the Vault Rules processor substitutes each occurrence of `$$ATTRDATA` with the test value and each occurrence of `$$OPERATOR` with the SQL operator. The entire statement is then *thrown* at the SQL RDBMS for evaluation.

An example of a complex classification function is the `PROJID` function that is provided. Because files are handled differently than parts, the function must have logic to differentiate a file object from a part object. The following fragment is used to evaluate the function:

```
exec sql select count(*) from attr_dummy
  where (upper(':OBJECT-TYPE') = 'FILE'
  and ':DM_FILE_CLASS' = 'PRO'
  and ':DM_FILE_OWNER_ID' :$$OPERATOR ':$$ATTRDATA')
  or (upper(':OBJECT-TYPE') = 'PART'
  and ':DM_PART_CLASS' = 'PRO'
  and ':DM_PART_OWNER_ID' :$$OPERATOR ':$$ATTRDATA')
  ;
```

A classification function can be written that does not evaluate Vault meta data. The `attrfile` used to input the attributes and data can include attributes that are used as inputs to the classification functions. The following fragments of code illustrate this type of rule.

In this example, a set of attributes have been defined that only apply to a specific application. This function assumes that the APPLICATION attribute is set by the application and is written to the input `attrfile`.

```
applicat :
    exec sql select count(*) from attr_dummy
      where upper(':APPLICATION') :$$OPERATOR ':$$ATTRDATA'
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
                  then result = 800
                  else result = 0;
end-test
```

In this example, a set of job accounting attributes are required for every Vault object touched. The attributes apply to everyone. The problem is that the supplied classification rules operate on specific items. In order to ensure this set of attributes is captured, a classification function could be defined that would add the job accounting attribute set to the object's classification if an attribute or attributes were not present in the input stream. (The test value and the operator inputs to `ADDRULE` would not be used.)

```
jobstuff :
    exec sql select count(*) from attr_dummy
      where length(':FAVORITE-FOOD') > 0
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 1
                  then result = 0
                  else
                        do
                        result = #addset(JOB-ACCOUNTING);
                        if result <> 0
                              then result = 998;
                        end;
end-test
```

When Simple Classification Functions Are Too Simple

In each of the prior examples, the data type function names from the `attrtype.cg` file map directly to the data types in the `ADDATTR` command. In addition, the simple classification function names from the `attrrule.cg` file map directly to the simple classification functions in the `ADDRULE` command. For data types, that is all you can do. There might, however, be a situation where the simple classification rules are too simple.

When the attribute server has evaluated all of the simple classification rules, it attempts to evaluate a program called the Complex Classification program. This program is written entirely by you to perform whatever steps you require.

The Complex Classification program is written in the Vault Rules language as are the data types and the simple classification function, with two differences.

- Whereas the table-id for the data types is `'data-validation'`, and the table-id for the simple classification is `'simple-classification'`, the table-id for the complex program is `'classification'`.
- The interpreter begins executing the complex classification program at the `'root'` function and not at the function name.

All of the other rules stated above apply. For example, the input stream is available for input, the contents of the file directory (for file objects) and the part directory (for part objects) are available for use. The file is compiled using the `edmrparser` script, and the interpreter logic is loaded into the database using `LDAMCPLX`. An example using the `carlstest.cg` file begins below and continues to the next page:

```
* This file describe the complex classification program

tableinfo
    tablename : classification;
    tabletype : sql-table;
end-tableinfo
tests
root :
  attribiw();
  if result = 0
     then
      do
      result = #addset(INIT_GRP);
      if result = 0
          then #commit();
      end;
```

```
  if result <> 998
     then result = 0;
end-test
attribiw :
    exec sql select count(*) from attr_dummy
      where (upper(':OBJECT-TYPE') = 'FILE'
                and ':DM_FILE_CLASS' = 'PRO'
                and ':DM_FILE_OWNER_ID' = 'ATTRIB'
            and ':DM_FILE_STATUS_CD' = 'IW')
      or (upper(':OBJECT-TYPE') = 'PART'
                and ':DM_PART_CLASS' = 'PRO'
                and ':DM_PART_OWNER_ID' = 'ATTRIB'
            and ':DM_PART_STATUS_CD' = 'IW')
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
            then result = 800
            else result = 0;
end-test
```

The steps to process this file are

1. edmrparser carlstest.cg
2. ldamcplx carlstest

The Language Specification

The following items are definitions of the symbols used to define the language:

1. Anything between [and] is optional.
   e.g., [optional]
2. Anything between < and > is required.
   e.g., < required >
3. Anything between " and " is a literal and can be either upper- or lowercase. It cannot be mixed case.
   e.g., "literal text"
4. The symbol ::= reads is defined to be.
   e.g., comment ::= [comment text]
5. The symbol | reads "or" and implies a choice.
   e.g., "data-file" | "sql-table"
6. The symbol : separates the name from the definition.
   e.g., "tablename" : <table-id>;
7. The symbol ; is a statement delimiter.
   e.g., "tablename" :<table-id>;

```
comment-section ::= null
      | * [ text of the comment ]
```

```
tableinfo-section ::=
      “tableinfo”  table-specifications  “end-tableinfo”
table-specifications ::=
      “tablename” : <table-id>;
      “tabletype”   : “data-file”
                              | “sql-table”;
table-id ::= 1 to 32 printable characters
                     excluding all of the following ”:’;
```

```
definition-section ::=null
      | “definitions”   definition-list  “end-definitions”
definition-list ::= null
      | definition; definition-list
definition ::= <“definition-id”> : literal-list
definition-id ::= 1 to 32 printable characters
      excluding all of the following ”:’;
literal-list ::= literal
      | literal, literal-list
literal ::= ’<stringofcharacters>’
```

```
macro-section ::= null
      | “macros”  macro-list “end-macros”
macro-list ::= null
      | macro; macro-list
macro ::= <macro-id> : count definition-id qualifier;
macro-id ::= 1 to 32 printable characters
                     excluding all of the following ”:’;()
count ::= ( <number> )
qualifier ::= null
      | “memberof” definition-id
      | “between” range-list
      | “columnof” column-spec
range-list ::= ’<lowerbound>-<upperbound>’
column-spec ::=
      ’<sql-user>.<sql-tablename>.<column-name>’
```

```
test-section ::=
              “tests” test-list
test-list ::= null
      | test “end-test” test-list
test ::= <test-id> : statement-list
test-id ::= 1 to 32 printable characters beginning with
                    a-zA-Z excluding all of the following ”:’;()=#
statement-list ::= null
      | statement; statement-list
statement ::= compound-statement
      | assignment-statement
      | test-invocation
      | builtin-function
```

```
      | "exec sql" sql-statement
      | "exec data" validation-statement
      | conditional-statement
compound-statement ::= "do" statement "end"
assignment-statement ::= identifier "=" rvalue
test-invocation ::= <test-id>()
builtin-function ::= "#commit()"
      | "#rollback()"
      | "#rowid()"
      | "#addset(<literal>)"
      | "#delset(<literal>)"
      | "#cvgmf(<number> <parameter-list:1>)"
parameter-list ::= null
      | , <parameter> <parameter-list:1>
parameter ::= <host-variable>
      | <literal>
sql-statement ::= <insertstatement>
      | <deletestatement>
      | <updatestatement>
      | <selectstatement>
validation-statement ::= "validate" <host-variable>
       "using" validation-spec
validation-spec ::= <macro-id> validation-spec
      | literal  validation-spec
conditional-statement ::=
      "if" condition "then" statement
            [ "else" statement ]
condition ::= identifier operator rvalue
rvalue ::= <number>
      | builtin-function
identifier ::= "sqlcode"
      | "rowcount"
      | "result"
operator ::= "=" | "<>" | "<" | ">" | "<=" | ">="
host-variable ::= ":"<characterstring>
```

Supplied Data

The following data type functions are the supplied data types:

```
tableinfo
      tablename : data-validation;
      tabletype : sql-table;
end-tableinfo
tests
root :
end-test
character :
    exec sql select count(*) from attr_dummy
      where length(':$$ATTRDATA') > 0
      and  length(':$$ATTRDATA') < 241;
    if sqlcode <> 0
```

```
          then result = 998
          else if rowcount = 0
                then result = 700
                else result = 0;
end-test
number :
        exec sql select count(*) from attr_dummy
          where to_number(':$$ATTRDATA') <> 0
          or to_number(':$$ATTRDATA') = 0;
        if sqlcode <> 0
          then result = 998
          else if rowcount = 0
                then result = 700
                else result = 0;
end-test
date :
        exec sql select count(*) from attr_dummy
          where to_date(':$$ATTRDATA') <> to_date('01-JAN-91')
          or to_date(':$$ATTRDATA') = to_date('01-JAN-91');
        if sqlcode <> 0
          then result = 998
          else if rowcount = 0
                then result = 700
                else result = 0;
end-test
```

Supplied Classified Functions

The following are the supplied classification functions:

```
tableinfo
        tablename : simple-classification;
        tabletype : sql-table;
end-tableinfo
tests
root :
end-test
projid :
        exec sql select count(*) from attr_dummy
          where (upper(':OBJECT-TYPE') = 'FILE'
          and ':DM_FILE_CLASS' = 'PRO'
          and ':DM_FILE_OWNER_ID' :$$OPERATOR ':$$ATTRDATA')
          or (upper(':OBJECT-TYPE') = 'PART'
          and ':DM_PART_CLASS' = 'PRO'
          and ':DM_PART_OWNER_ID' :$$OPERATOR ':$$ATTRDATA')
          ;
        if sqlcode <> 0
          then result = 998
          else if rowcount = 0
                then result = 800
                else result = 0;
end-test
```

```
user :
    exec sql select count(*) from attr_dummy
      where ':DM_XCTN_USER_ID' :$$OPERATOR ':$$ATTRDATA'
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
            then result = 800
            else result = 0;
end-test
status :
    exec sql select count(*) from attr_dummy
      where (upper(':OBJECT-TYPE') = 'FILE'
      and ':DM_FILE_STATUS_CD' :$$OPERATOR ':$$ATTRDATA')
      or (upper(':OBJECT-TYPE') = 'PART'
      and ':DM_PART_STATUS_CD' :$$OPERATOR ':$$ATTRDATA')
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
            then result = 800
            else result = 0;
end-test
class :
    exec sql select count(*) from attr_dummy
      where (upper(':OBJECT-TYPE') = 'FILE'
      and ':DM_FILE_CLASS' :$$OPERATOR ':$$ATTRDATA')
      or (upper(':OBJECT-TYPE') = 'PART'
      and ':DM_PART_CLASS' :$$OPERATOR ':$$ATTRDATA')
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
            then result = 800
            else result = 0;
end-test
usertype :
    exec sql select count(*) from attr_dummy
      where (upper(':OBJECT-TYPE') = 'FILE'
      and ':DM_FILE_USER_TYPE' :$$OPERATOR ':$$ATTRDATA')
      or (upper(':OBJECT-TYPE') = 'PART'
      and ':DM_PART_USER_TYPE' :$$OPERATOR ':$$ATTRDATA')
      ;
    if sqlcode <> 0
      then result = 998
      else if rowcount = 0
            then result = 800
            else result = 0;
end-test
partnum :
    exec sql select count(*) from attr_dummy
      where (upper(':OBJECT-TYPE') = 'FILE'
      and ':DM_FILE_PART_NO' :$$OPERATOR ':$$ATTRDATA')
      or (upper(':OBJECT-TYPE') = 'PART'
```

```
        and ':DM_PART_PART_NO' :$$OPERATOR ':$$ATTRDATA')
        ;
      if sqlcode <> 0
        then result = 998
        else if rowcount = 0
              then result = 800
              else result = 0;
end-test
```

Compiling and Loading Rules and Data Types

Compile and load your newly created rules and data types as follows:

1. Compile your program using the `edmrparser` compiler, for example,

   ```
   % edmrparser attrtype.cg
   % edmrparser attrrule.cg
   ```

2. Load your compiled programs into the Vault (server) database using `ldamlogic`.
3. You must shut down the Attribute Server with `nsmstop` to cause the old logic to be flushed from memory and the new logic to replace it.
4. Start the Attribute Server with the `nsmstart` command.
5. Add an attribute to the Vault (server) database. Assign one of your newly created types to the attribute. Give the type a value. And test the rule you created to see if it executes as expected.
6. To further test a newly created rule, run the `ADD RULE`, `STORE`, and `UPDATE` or `REPLACE` commands to see if the rule is being invoked.

Appendix C

Sample Command-Triggered Control Program

This appendix shows the control program for a command-triggered application subroutine.

- Sample Command-Triggered Control Program

Sample Command-Triggered Control Program

Sample Command-Triggered Control Program

The programming examples in the following sections illustrate a shell main program for a command-triggered process, and a program which defines structures and mnemonics for command-triggered programs and communication subroutines.

ctshell.c

```
File Name:  ctshell.c

Purpose:    Shell main program for Command Triggered Process.

Function:   This shell main routine illustrates using the four
            communication subroutines in a user-written command
            triggered process.

            The program connects to EDM using NSM protocols and
            waits for messages to be sent to it.

            Upon the receipt of a message, it checks the EDM
            release level then calls the subroutine corresponding
            to the aename in the CT_struct input message.  This
            shell program checks only for one aename, ctsample.

            After the subroutine completes, if the triggering
            command is waiting for a reply, one is sent.

            If an error condition is detected, the process
            disconnects.

How to Use: To test the functionality of this program:

      -     compile this program and the sample application
            subroutine, ctsample
      -     link them with the four communication subroutines
      -     make the required nsm.config changes
      -     start up the triggered process
      -     use the CHGCTL command to activate a pre-process
            command trigger on the STORE, RESERVE, and/or
            CHGFA commands
      -     test any of these commands using any of the
            EDM interfaces (screen, command, programmatic)
            To develop a customized command triggered process,
            use this routine as a guide, changing the call to
            ctsample, but preserving the four communication
            subroutine calls.

#include <stdio.h>
#include ”adkeylen.h” /* EDM command keyword length mnemonics
```

Sample Command-Triggered Control Program

```
#include ”adpictl.h” /* Command mnenomics and current release
#include ”adctstr.h”  /* Command Trigger structure definition
#include ”ctsample.h” /* Error messages and other mnenomics

extern int ct_connect();
extern int ct_waiting_for_edm();
extern int ct_sending_response();
extern int ct_disconnect();

/***  main program  ***/

main(argc, argv)

int argc;
int *argv[];
      {
       int ct_rc = OK;
       int exit_rc = OK;
       struct  CT_struct   CT_str;

       /***  CONNECT AS THE AENAME PASSED TO THIS MODULE  ***/

       if (argc < 2)
            {
             fprintf(stderr, ”\n %s \n”, CT_ERR01);
             exit_rc = CT_ERROR;
             goto errexit;
            }
/***************************************************************/
/* Connect to EDM:                                             */
/*  int ct_connect(CT_struct)                                  */
/*      CT_struct.aename (input)  aename of the user process   */
/* CT_struct.full_aename (output) Full aename of user process  */
/*                                (node:domain:aename:instance)*/
/***************************************************************/

      memset (CT_str.aename,’ ’,LAENAME);
      memset (CT_str.full_aename,’ ’,LFULLAENAME);
      strncpy(CT_str.aename, argv[1], strlen(argv[1]));
      ct_rc = ct_connect(&CT_str);
      if (ct_rc != OK)
      {
       fprintf(stderr, ”\n %s %d\n”, CT_ERR02, ct_rc);
       exit_rc = CT_ERROR;
       goto errexit;
       }
       /***  LOOP WAITING FOR MESSAGE FROM EDM  ***/
       while (exit_rc == OK)
      {
       memset (CT_str.edm_aename,’ ’,LFULLAENAME);
       memset (CT_str.inbuff.wait_flag,’ ’,sizeof(CT_str.inbuff));
       memset (CT_str.outbuff.message_text,’ ’,LMSGTEXT);
       CT_str.outbuff.process_flag[0] = RESUME_EDM_COMMAND;
```

Sample Command-Triggered Control Program

```
/****************************************************************/
/* Wait for Trigger:                                            */
/* intct_waiting_for_edm(CT_struct)                             */
/* CT_struct.inbuff (output)  Data buffer from EDM              */
/* CT_struct.edm_aename (output) Full aename of EDM process     */
/*                                  triggering this process     */
/*                               (node:domain:aename:instance)*/
/****************************************************************/
      ct_rc = ct_waiting_for_edm(&CT_str);
      if (ct_rc != OK)
        {
            if (ct_rc == STOP_TRIGGER)
              {
                fprintf(stderr,"\n %s \n", CT_STP01, ct_rc);
                exit_rc = CT_SHUTDOWN;
                break;
              }
            else
              {
                fprintf(stderr,"\n %s %d\n", CT_ERR03, ct_rc);
              exit_rc = CT_ERROR;
              break;
              }
        }
        if (RELEASE_CHECK)
          {
        if (strncmp(CT_str.inbuff.releaseno,
                  CURRENT_RELEASE, LRELEASENO) != 0)
                {
                    strncpy(CT_str.outbuff.message_text, CT_ERR12,
                            strlen(CT_ERR12));
                    CT_str.outbuff.process_flag[0] =
STOP_EDM_COMMAND;
                    fprintf(stderr, "\n %s \n", CT_ERR12);
                    fprintf(stderr, " EDM Release: %.*s\n",
LRELEASENO,
                            CT_str.inbuff.releaseno);
                  goto respond_exit;
                }
          }
        /***  CALL SAMPLE APPLICATION LOGIC  ***/
      if (strncmp(CT_str.aename,CT_AE01,LAENAME) != 0)
        {
            strncpy(CT_str.outbuff.message_text, CT_ERR11,
                    strlen(CT_ERR11));
            CT_str.outbuff.process_flag[0] = STOP_EDM_COMMAND;
            fprintf(stderr, "\n %s \n", CT_ERR11);
          exit_rc = CT_ERROR;
          goto respond_exit;
        }
        ct_rc = ctsample(&CT_str);
      if (ct_rc != OK)
          {
            fprintf(stderr,"\n %s %d\n", CT_ERR10, ct_rc);
```

Sample Command-Triggered Control Program

```
            exit_rc = ct_rc;
          goto respond_exit;
          }

/*** CHECK FOR TRIGGER AND WAIT, EVEN WHEN APPLICATION ERROR ***/
    respond_exit:

       if (CT_str.inbuff.wait_flag[0] == TRIGGER_WAIT)
          {

/****************************************************************/
/* Respond to Trigger:                                          */
/*  int   ct_sending_reponse(CT_struct)                         */
/* CT_struct.user_outbuff (input) Message to return, if wait    */
/* CT_struct.edm_aename (output) Full aename of EDM process     */
/*                        triggering this process               */
/*                                 (node:domain:aename:instance)*/
/****************************************************************/
            ct_rc = ct_sending_response(&CT_str);
            if (ct_rc != OK)
            {
                if (ct_rc == STOP_TRIGGER)
                  {
                    fprintf(stderr,"\n %s \n", CT_STP01, ct_rc);
                    exit_rc = CT_SHUTDOWN;
                    break;
                  }
                else
                  {
                  fprintf(stderr,"\n %s %d\n", CT_ERR04, ct_rc);
                  exit_rc = CT_ERROR;
                break;
                  }
              }
           }

      }  /**  End of while loop  **/
    /***  DISCONNECT FROM EDM  ***/
/****************************************************************/
/* Disconnect from EDM:                                         */
/* intct_disconnect(CT_struct)                                  */
/*CT_struct.full_aename (input) Full aename of user process     */
/*                     (node:domain:aename:instance)            */
/****************************************************************/

    ct_rc = ct_disconnect(&CT_str);
    if (ct_rc != OK)
      {
        fprintf(stderr,"\n %s %d\n", CT_ERR05, ct_rc);
      exit_rc = CT_ERROR;
      }
errexit:
    exit(exit_rc);
}
```

Sample Command-Triggered Control Program

adctstr.h

```
     File Name: adctstr.h
     Purpose: Defines structures and mnemonics for Command
                Triggered programs and communication subroutines.
     Notes:   adkeylen.h contains the length mnemonics used here.
                                                                */
/***************************************************************/
#define LFULLAENAME    132
#define LKEYWORDS      481
struct  data_from_edm
   {
      char  wait_flag[LPDMFLAG];
      char  location_flag[LPDMFLAG];
      char  severity_code[LPDMFLAG];
      char  releaseno[LRELEASENO];
      char  userid     [LUSERID];
      char  command     [LCOMMAND];
      char  keywords[LKEYWORDS];
   };
struct  data_to_edm
   {
      char  process_flag[LPDMFLAG];
      char  message_text[LMSGTEXT];
   };
struct  CT_struct
   {
        char    aename[LAENAME];
        char    full_aename[LFULLAENAME];
        char    edm_aename      [LFULLAENAME];
        struct  data_from_edm   inbuff;
        struct  data_to_edm     outbuff;
   };
/**  values used by wait/send routines.
     STOP_TRIGGER is returned if shutdown was requested.  **/
#define    CONTINUE_TRIGGER       0
#define    STOP_TRIGGER           998
/**  values for CT_struct flags  **/

/*   inbuff.wait_flag       */

#define    TRIGGER_PROCEED        '0'
#define    TRIGGER_WAIT           '1'

/*   inbuff.location_flag   */

#define    TRIGGER_AT_START       '0'
#define    TRIGGER_AT_END         '1'

/*   outbuff.process_flag   */

#define    RESUME_EDM_COMMAND     '0'
#define    STOP_EDM_COMMAND       '1'
```

Appendix D

Sample Command-Triggered Application Subroutine

This appendix shows a sample command-triggered application subroutine called by the control program (`ctshell.c`) shown in Appendix C, “Sample Command-Triggered Control Program”. It includes the following files:

- ctsample.c
- ctsample.h

ctsample.c

```
   File Name: ctsample.c
   Purpose: Sample application subroutine for Command Triggered
            Process.
   Function:This sample application subroutine is an aename
            called by the shell main program to perform a
            pre-process check of user-defined attributes which
            may be entered on the STORE, RESERVE, and/or CHGFA
            commands in EDM.  The application merely checks that
            the field USERTYPE has been input to the command.
            If no data is entered, an error message is returned
            to EDM.
            All communication logics are contained in the main
            program, so this subroutine need only perform
            functionality relevant to the application.
   How to Use: See the shell program introduction.
#include  <stdio.h>
#include  "adkeylen.h" /* EDM command keyword length mnenomics */
#include  "adpictl.h"  /* Command mnenomics and current release*/
#include  "adctstr.h"  /* Command Trigger structure definition */
#include  "ctsample.h" /* Error messages and other mnenomics   */
#include  "adstr.h"    /* STORE input structure */
#include  "adrsv.h"    /* RESERVE input structure */
#include  "adcfa.h"    /* CHGFA input structure */
int ctsample(CT_str)
struct   CT_struct    *CT_str;
  {
    int    appl_rc = OK;
    struct store_struct   *sptr;
    struct reserve_struct *rptr;
    struct chgfa_struct   *aptr;
    char   usertype[LUSERTYPE];
    /* If the TRIGGER_INFO flag is set, print out the
       contents of the inbuff structure, excluding the values of
       the keywords.  Then proceed accordingly. */
    if (TRIGGER_INFO)
      {
    fprintf(stdout, "\n");
    fprintf(stdout, "Data from EDM\n");
    fprintf(stdout, "-------------\n");
    fprintf(stdout, " wait_flag: %c\n",
       CT_str->inbuff.wait_flag[0]);
    fprintf(stdout, "location_flag: %c\n",
       CT_str->inbuff.location_flag[0]);
    fprintf(stdout, "severity_code: %c\n",
       CT_str->inbuff.severity_code[0]);
    fprintf(stdout, "   releaseno: %.*s\n", LRELEASENO,
       CT_str->inbuff.releaseno);
    fprintf(stdout, "   userid: %.*s\n", LUSERID,
       CT_str->inbuff.userid);
    fprintf(stdout, "   command: %.*s\n", LCOMMAND,
```

```
                    CT_str->inbuff.command);
          }

        /* First, check that this aename has been correctly triggered
           at pre-process time.  A post-process trigger cannot
           alter the outcome of the EDM command.  This check
           verifies that the CHGCTL command was used correctly
           in setting up the command trigger location. */
        if (CT_str->inbuff.location_flag[0] != TRIGGER_AT_START)
          {

strncpy(CT_str->outbuff.message_text,CT_ERR07,strlen(CT_ERR07));
            CT_str->outbuff.process_flag[0] =
            STOP_EDM_COMMAND; /* Note: process_flag will be ignored */
            appl_rc = CT_AE_ERROR; /* stop triggered process */
            goto ut_exit;
          }
        /* Next, check that this aename has been triggered from one of
           the commands which has user-defined attributes.  The
           tests use the command mnenomics found in adpictl.h.
           This check verifies that the CHGCTL command was used
           only on EDM commands which have the usertype field */
        if (strncmp(CT_str->inbuff.command,STORE,LCOMMAND) == 0)
          {
          sptr = (struct store_struct *) CT_str->inbuff.keywords;
            strncpy(usertype, sptr->usertype, LUSERTYPE);
          }
        else
         if (strncmp(CT_str->inbuff.command,RESERVE,LCOMMAND) == 0)
          {
          rptr = (struct reserve_struct *) CT_str->inbuff.keywords;
            strncpy(usertype, rptr->usertype, LUSERTYPE);
          }
        else
         if (strncmp(CT_str->inbuff.command,CHGFA,LCOMMAND) == 0)
          {
          aptr = (struct chgfa_struct *) CT_str->inbuff.keywords;
            strncpy(usertype, aptr->usertype, LUSERTYPE);
          }
        else     /* Error: this is not an EDM command with USERTYPE */
          {
          strncpy(CT_str->outbuff.message_text,CT_ERR06,
                    strlen(CT_ERR06));
            CT_str->outbuff.process_flag[0] = STOP_EDM_COMMAND;
            appl_rc = CT_AE_ERROR;       /* stop triggered process */
            goto ut_exit;
          }
        /* Now, check value of USERTYPE entered for the command. */
        if ((strncmp(usertype, INVALID_UT, LUSERTYPE) == 0) ||
            (usertype[0] == DELETE_UT))
          {
            strncpy(CT_str->outbuff.message_text,CTAE_ERR01,
                    strlen(CTAE_ERR01));
            CT_str->outbuff.process_flag[0] = STOP_EDM_COMMAND;
```

```
        }
ut_exit:
     return(appl_rc);
  }
```

ctsample.h

```
     File Name:   ctsample.h
     Purpose:     Defines structures and mnemonics for the sample

                  command trigger shell program and application
                  subroutine.
     Notes:       adkeylen.h contains the length mnemonics used
                  here.
     The TRIGGER_INFO constant controls whether information in the
     command trigger structure is displayed by ctsample. A zero
     setting means that the data will not be displayed. A
     non-zero setting means that the data will be displayed.
     The RELEASE_CHECK constant controls whether a check is
     performed against the release number of the EDM command
     which is executing. A zero setting means that the check will
     not be performed.  A non-zero setting means that the check
     will be performed.
     For some commands, if the EDM release of the command differs
     from that of the trigger, there may be new keywords in the
     command trigger structure.  This may require special
     processing, depending on what the trigger does.
     In the ctshell program, if the check is performed, and the
     release numbers of the EDM command and the trigger do not
     match, the trigger stops the command, prints out a message to
     standard error, and waits for the next command.
                                                                */
/***************************************************************/
#define  OK           0
#define  CT_ERROR    12
#define  CT_AE_ERROR  8
#define  CT_SHUTDOWN  4
#define TRIGGER_INFO  0
#define RELEASE_CHECK 0
#define CT_ERR01 ”ctsample stopped. Missing triggered process
aename.”
#define CT_ERR02 ”ctsample stopped. NSM connection failed. NSM rc
= ”
#define CT_ERR03 ”ctsample stopped. NSM wait error. NSM rc = ”
#define CT_ERR04 ”ctsample stopped.NSM respond error. NSM rc = ”
#define CT_ERR05 ”ctsample stopped.NSM disconnect error.NSM rc = ”
#define CT_ERR06 ”ctsample stopped. Invalid triggering command.”
#define CT_ERR07 ”ctsample stopped.Post-process trigger invalid.”
#define CT_ERR10 ”ctsample stopped. Triggered process error.rc = ”
#define CT_ERR11 ”ctsample stopped.Unrecognized AEname from EDM.”
#define CT_ERR12 ”command not executed. EDM release number
mismatch.”
#define CT_STP01 ”ctsample stopped by NSM stop request.”
#define CTAE_ERR01 ”Processing not done.  Usertype input
required.”
#define INVALID_UT ”  ”  /* STORE/RESERVE usertype left blank */
#define DELETE_UT  ’#’    /* CHGFA code for usertype deletion */
#define CT_AE01    ”ctsample                        ”
```

Index

A

adctstr.h file 4-4
ADDAG command 1-7
  programmatic interface 7-3
ADDCL command 1-6
  programmatic interface 7-3
ADDFS command 1-9
  programmatic interface 7-3
ADDMFS command 1-9
  programmatic interface 7-3
ADDMUL command 1-8
  programmatic interface 7-3
ADDP command 1-7
  programmatic interface 7-3
ADDRS command 1-7
  programmatic interface 7-3
ADDS command 1-7
  programmatic interface 7-3
ADDSP command 1-9
  programmatic interface 7-3
ADDT command 1-9
  programmatic interface 7-3
ADDU command 1-6
  programmatic interface 7-3
ADDUL command 1-8
  programmatic interface 7-3
ADDUP command 1-7
  programmatic interface 7-3
ADDUSA command 1-7
  programmatic interface 7-3
adkeylen.h include file 4-5
ADMCOPY command 1-7
  programmatic interface 7-3
ALL keyword
  with command triggers 4-12
Application Entity
  command trigger 4-9
  ctsample 5-3
  defining process as 4-2
ARCHIVE command
  programmatic interface 7-3
  purpose 1-9
Attributes
  user-defined
    constructing rules for B-2
    definition B-4
    rule contruction B-5

B

Blocks
  creating B-7
Builtin functions
  VaultRules Processor B-10

C

C
  sample program A-2
cannot 4-9
CHGAG command 1-7
  programmatic interface 7-3
CHGCL command
  programmatic interface 7-3
CHGCTL command 1-5
  Command Triggering 1-9

execute chgctl 4-14
parameters 4-13
Using the chgctl Command 4-11
CHGFA command 1-8
programmatic interface 7-3
CHGFCL command 1-8
programmatic interface 7-3
CHGFPW command 1-8
programmatic interface 7-3
CHGFREV command 1-8
programmatic interface 7-3
CHGFSC command 1-8
programmatic interface 7-3
CHGFSP command
programmatic interface 7-3
CHGP command 1-7
programmatic interface 7-3
CHGS command 1-7
programmatic interface 7-3
CHGSPS command
programmatic interface 7-3
CHGSPT command
programmatic interface 7-3
CHGU command 1-6
programmatic interface 7-3
CHGUP command 1-7
programmatic interface 7-3
CHGUPW command 1-6
programmatic interface 7-3
Classification function
creating
VaultRules Processor B-14
CLST command 1-9
programmatic interface 7-3
Command Trigger Program 5-5
Command triggers
after command processing 1-13
before command processing 1-12
commands without any 1-5
communication subroutines 4-7
definition 1-11
linking programs 4-10
overview 1-11
format 4-2
restrictions 1-13
running programs 4-10
UNIX 5-9
sample program D-1
starting
HP-UX 5-2
Solaris 5-2
Vault 6 5-2
structure 4-4
with programmatic interface 1-14
compiler, SparcWorks 3-7
Compiling and Linking a Programmatic Interface Program 3-1
Compiling and loading
VaultRules Processor B-23
Compiling programs 4-10
HP-UX 5-4
Solaris 5-4
source programs 5-4
standard procedure
command triggers 4-10
programmatic interface 2-10
ULTRIX 5-4
Control structures, cpdm
description 2-3
fields 2-5
COPY command
in file maintenance 1-8
programmatic interface 7-3
ctsample subroutine 4-10
parameter required 4-15
ctsample.c
compiling 5-4
subroutine 5-4
ctsample.h file D-1
ctshell.c
linking
HP-UX 5-6
Solaris 5-6
Customizing
the Client 6-1

D

Data type
creating
VaultRules Processor B-12
Data validation statements
VaultRules Processor B-11
DELAG command 1-7
programmatic interface 7-3
DELCL command 1-6
programmatic interface 7-3

DELETE command 1-9
  programmatic interface 7-3
DELLOG command 1-9
  programmatic interface 7-3
DELOV command 1-9
DELP command 1-7
  programmatic interface 7-3
DELRS command 1-7
DELS command 1-7
  programmatic interface 7-4
DELU command 1-6
  programmatic interface 7-4
DELUL command 1-8
  programmatic interface 7-4
Documentation, printing from Portable Document Format (PDF) file xiii

E

edmosrv
  bind variables 6-4
edmosrv library 6-1
Example
  AIX 3-4
  HP-UX 3-5
  UNIX 3-3

F

Files
  include 2-4
Functions
  writing
    VaultRules Processor B-9

G

GET command 1-8
  programmatic interface 7-4

I

IBKUP command
  in database maintenance 1-9
  programmatic interface 7-4
if-then-else statement B-8
inbuff
  contents 4-5
Include files
  for C 2-4
Input and output
  VaultRules Processor B-6
Input structures, cpdm
  description 2-3
  include files 2-6
Input structures, SVedm
  description 2-2
IQF
  command 1-5

L

Language specification
  VaultRules Processor B-17
Language structure
  VaultRules Processor B-6
library
  Scramble 6-8
library edmosrv 6-1
Linking a Program on AIX 3-4
Linking a Program on Digital UNIX 3-3
Linking a Program on HP-UX 3-5
Linking a Program on SGI 3-2
Linking a Program on Solaris 3-7
Linking on AIX, Client Only 3-4
Linking on AIX, Vault Only 3-4
Linking on AIX, Vault with Oracle 3-4
Linking on Digital UNIX, Client Only 3-3
Linking on Digital UNIX, Vault Only 3-3
Linking on Digital UNIX, Vault with Oracle 3-3
Linking on HP-UX, Client Only 3-5
Linking on HP-UX, Vault Only 3-5
Linking on HP-UX, Vault with Oracle 3-6
Linking on IRIX, Client Only 3-2
Linking on IRIX, Vault Only 3-2
Linking on IRIX, Vault with Oracle 3-2
Linking on Solaris, Client Only 3-7

Linking on Solaris, Vault Only 3-7
Linking on Solaris, Vault with Oracle 3-7
Linking on Windows NT 3-8
Linking programs
  on SGI 3-2
  on Solaris 3-7
  prerequisites 2-10
  using command triggers 4-10
LISTDIR command 1-8
  programmatic interface 7-4
LOAD command
  programmatic interface 7-4
  system maintenance 1-9
Loading
  VaultRules processor programs B-23
Local
  commands 2-9

M

MARKA command 1-8
  programmatic interface 7-4
MARKD command 1-8
  programmatic interface 7-4
MARKR command 1-8
  programmatic interface 7-4

N

NSM
  commands
    nsmflush 4-11
    nsmstop 4-11
  conditions 4-11
  starting
    processes automatically 4-9
NSM.CONFIG file
  command trigger format 4-3
nsm.config file
  command trigger format 4-2
  modifying 4-9
nsm.config file
  adding executable script files
    SunOS
      Vault 5 5-3
nsm.config file
  adding executable script files
    HP-UX 5-3
    Solaris 5-3
nsmflush command 4-11
nsmstop command 4-11

O

OPNT command 1-9
  programmatic interface 7-4
Output and input
  VaultRules Processor B-6
Output structures, cpdm
  description 2-3
  fields 2-7
Output structures, SVedm
  description 2-2

P

Printing documentation from Portable Document Format (PDF) file xiii
Programmatic interface
  commands
    without any 1-5
  compiling programs 2-10
  how to use 2-2
  linking programs 2-10
  old
    entry point 2-3
    format 2-3
  overview 1-2
  running programs 2-11
  sample C program A-2
  sample program instructions 2-8
  SVedm 1-10
    entry point 2-2
    format 2-2
    overview 2-2
  uses 1-10
  with command triggers 1-14
PURGE command 1-9
  programmatic interface 7-4

R

READ command 1-8
  programmatic interface 7-4
READMSG command 1-7
  programmatic interface 7-4
RECSF command 1-9
  programmatic interface 7-4
RECSP command 1-9
  programmatic interface 7-4
REMFS command 1-9
  programmatic interface 7-4
REMMFS command 1-9
  programmatic interface 7-4
REMMUL command 1-8
  programmatic interface 7-4
Remote
  commands 2-9
REMT command 1-9
  programmatic interface 7-4
REMUP command 1-7
  programmatic interface 7-4
REMUSA command 1-7
  programmatic interface 7-4
REPLACE command 1-8
  programmatic interface 7-4
REQRVW command 1-7
REQRW command
  programmatic interface 7-4
RESERVE command 1-8
  programmatic interface 7-4
RESET command 1-8
  programmatic interface 7-4
RESTORE command 1-9
  programmatic interface 7-4
RESULT variable B-6
ROWCOUNT variable B-6
RSVP command 1-7
  programmatic interface 7-4
Rules
  applying to a file/part B-2
Running programs
  using command triggers 4-10
  using VaultProgramming 2-11

S

Samples
  C program 2-8
    svedmsample.c A-2
  command-triggered subroutine D-1
SCANTAPE command 1-9
  programmatic interface 7-4
Scramble library 6-8
SENDMSG command 1-7
  programmatic interface 7-4
SIGNOFF command 1-6
  programmatic interface 7-4
SIGNON command 1-6
  programmatic interface 7-4
SIGNOUT command 1-8
  programmatic interface 7-4
Solaris 3-7
Source files
  control structures
    command triggers 4-2
    programmatic interface 2-3
Source include files 2-4
SQL statements
  VaultRules Processor B-10
SQLCODE variable B-6
startCT script 5-2, 5-3
Statements
  creating B-7
STORE command
  file access 1-8
  programmatic interface 7-4
Supplied classified functions
  VaultRules Processor B-20
Supplied data
  VaultRules Processor B-19
SVedm programmatic interface 1-10
svedmsample.c file A-2

T

Testing
  for specific conditions
    VaultRules Processor B-8
Triggered processes
  definition 1-11

U

UBKUP command
  without command trigger 1-6
UNLOAD command
  programmatic interface 7-4
  system maintenance 1-9
UNMARK command 1-8
  programmatic interface 7-4
UPDATE command 1-8
  programmatic interface 7-4

V

VaultRules Processor Language B-2

W

when 5-9