HP DECset for OpenVMS

Cookbook for an Integrated Project Development Environment

January 2005

This document is a collection of articles written by various individuals in the DECset Engineering group. It describes some of the main features of the DECset software and how the individual tools can be used to form an integrated development environment.

Revision/Update Information: This is a revised document.

Software Version: DECset Version 12.6 for OpenVMS

Hewlett-Packard Company
Palo Alto, California

© Copyright 2005 Hewlett-Packard Development Company, L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Intel and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java is a US trademark of Sun Microsystems, Inc.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Printed in the US

Contents

1 Introduction

1.1 Main DECset Tools 1–1
1.1.1 CMS 1–2
1.1.2 LSE 1–2
1.1.3 SCA 1–3
1.1.4 MMS 1–3
1.1.5 DTM 1–3
1.1.6 PCA 1–4

2 The Environment Manager

2.1 Accessing the Environment Manager 2–1
2.2 Setting Up a Project Context Database and Contexts 2–3
2.2.1 Project Definition and Initial Setup 2–3
2.2.2 Context Database Creation 2–5
2.2.3 Project Context Creation 2–5
2.2.4 Team Context Creation 2–8
2.2.5 Personal Context Creation 2–10
2.2.6 Applying and Using a Context 2–12

3 The MMS Description File Generator

3.1 MMS Builder and Generator 3–1
3.2 Basic Description File Generation 3–2
3.3 More Complex Description File Generation 3–6
3.3.1 Generating a Description File for Pascal 3–6
3.3.2 Generating a Description File for COBOL 3–7
3.3.2.1 Creating the Description File from the CMS Library 3–8
3.3.2.2 Creating the Description File from the Command Line 3–9
3.3.2.3 Building the Application 3–9

4 DECset Component Integration

4.1 Integration With the OpenVMS Environment and Language Compilers 4–1
4.2 Integration Among the DECset Tools 4–2
4.2.1 CMS 4–2
4.2.2 LSE 4–3
4.2.3 LSE and SCA 4–4
4.2.4 MMS 4–6
4.2.5 DTM 4–8
4.2.6 PCA 4–8

5 CMS and Software Configuration Management

5.1 Software Configuration Management 5–1
5.2 The Goals of CMS 5–2
5.3 Common CMS Commands for Configuration Management 5–2
5.3.1 Generate a CMS History File (SHOW HISTORY) 5–2
5.3.2 Report on a Single Element (DIFFERENCES and ANNOTATE) 5–3
5.3.3 Track File Development (CREATE or MODIFY ELEMENT/REVIEW) 5–3
5.4 CMS Callable Routines 5–5
5.5 CMS Library Security 5–6
5.5.1 CMS Library Access Control Mechanisms 5–6
5.5.1.1 OpenVMS File Access Controls 5–6
5.5.1.2 CMS Access Control 5–7
5.5.1.3 OpenVMS Protected Subsystems 5–8
5.5.2 Performance Considerations 5–8
5.5.3 Sample CMS Library Configuration 5–10
5.5.3.1 Access Policy 5–10
5.5.3.2 Library Creation 5–10
5.5.3.3 Security Policy and Implementation 5–10
5.5.4 Implementing a CMS Library as a Protected Subsystem 5–12
5.5.5 Using CMS ACEs for Event Handling 5–13

6 DECwindows Motif Testing Using DTM

6.1 Test Organization 6–1
6.2 Creating Reproducible Tests 6–2
6.2.1 Configuring System Options 6–2
6.2.2 Preventing Problems with Multi-Click Operations 6–3
6.2.3 Hiding Copyright Notices 6–3
6.3 Improving Test Performance 6–3
6.4 Using an Alternate Display for Testing 6–4
6.5 Using DTM on the Display Under Test 6–4
6.6 Starting the Application Under Test 6–5
6.7 Using the Play/Record User Interface 6–6

A CMS Callable Routine Examples

A.1 Reserving and Replacing Elements A–1
A.2 Showing and Formatting Reservation Information A–4
A.3 Showing and Filtering Element Information A–7

B CMS Event Handling Example

Examples

A–1 CMS Callable Routines: RESERVE and REPLACE A–1
A–2 CMS Callable Routines: SHOW RESERVATION A–4
A–3 CMS Callable Routines: SHOW ELEMENT A–7
B–1 CMS_EVENT.C B–1
B–2 CMS_ACTION.COM B–4

Figures

2–1	DECset Tools Selection Pull-down menu	2–2
2–2	Environment Manager Context Window and Icon	2–2
2–3	Sample Project Directory Structure	2–3
2–4	Creating an SCA and a CMS Library	2–4
2–5	Creating a Test Library	2–4
2–6	Environment Manager New Database Dialog Box	2–5
2–7	Environment Manager New Context Dialog Box	2–6
2–8	Defining Logical Names for a Context	2–6
2–9	Defining a Test Library for a Context	2–7
2–10	Sample Context File for PROJECT_EG	2–7
2–11	Environment Manager New Context Dialog Box	2–8
2–12	Environment Manager Select Option File Dialog Box	2–9
2–13	Sample Context File for TEAM_Z	2–9
2–14	DECSET SHOW CONTEXT Command Example	2–10
3–1	MMS Main Window	3–2
3–2	MMS Sources Dialog Box	3–3
3–3	Automatically Generated Description File	3–4
3–4	MMS Sources Dialog Box: Pascal Example	3–7
3–5	MMS Sources Dialog Box: COBOL Example	3–8
3–6	MMS Main Window: COBOL Example	3–9
3–7	MMS Build Definitions/Directives Options Dialog Box	3–10
4–1	DECset Tools Integration	4–2
4–2	CMS: Fetching an Element	4–3
4–3	CMS & LSE: Element Fetched into an LSE Buffer	4–3
4–4	LSE & SCA: Goto Declaration in Same File	4–4
4–5	LSE & CMS: Goto Declaration in a CMS Element	4–5
4–6	LSE & CMS: Element Fetched into an LSE Buffer	4–5
4–7	SCA & LSE: From Data Structures Results to LSE Buffer	4–6
4–8	MMS & LSE: MMS Description File in LSE	4–7
4–9	MMS & LSE: Compilation Errors During an MMS Build	4–7
4–10	MMS & LSE: Correcting Compilation Errors	4–8
5–1	CMS Callable Routines Interfaces	5–5
5–2	CMS Event Handling Flow Diagram	5–14

Tables

5–1	CMS Review Commands	5–4

1
Introduction

The DECset for OpenVMS (DECset) software set consists of the following core components:

- Code Management System for OpenVMS (CMS)
- Language-Sensitive Editor for OpenVMS (LSE)
- Source Code Analyzer for OpenVMS (SCA)
- Module Management System for OpenVMS (MMS)
- Digital Test Manager for OpenVMS (DTM)
- Performance and Coverage Analyzer for OpenVMS (PCA)

In addition to these components, DECset also includes the DECset Environment Manager for OpenVMS, which is a tool that provides a single mechanism for tailoring the execution environment across the DECset tools.

To provide support for the Windows programming environment, client applications are also available for CMS and MMS. The HP DECset Clients for CMS and MMS enable PC desktop access to CMS libraries as well as frequently-used MMS and CMS commands. The DECset Clients provide convenient access to CMS and MMS features through the Windows interface, the API, or through Microsoft Visual Studio.

1.1 Main DECset Tools

The DECset tools run on the HP OpenVMS operating system and have both a Digital Command Language (DCL) command-line interface and a DECwindows Motif graphical interface.

LSE/SCA, CMS, and DTM also provide an application programming interface (API) with a library of routines that can be called from the majority of programming languages supported by OpenVMS.

1.1.1 CMS

CMS is a code management tool that provides an efficient way to store files and track all changes to those files.

CMS can be use to archive any file type supported by OpenVMS Record Management Services (RMS). CMS tracks changes to the archived files and stores only the successive change data in ASCII text files. This not only reduces the amount of disk space used for storing multiple versions of files, but also allows CMS to reconstruct any previous version and to identify the changes made between any two versions. In addition to storing change data, CMS also maintains historical records detailing file operations and library access.

Detailed information on CMS can be found in:

- *HP DECset for OpenVMS Software Product Description*
- *Code Management System for OpenVMS Release Notes* (SYS$HELP:CMS*.RELEASE_NOTES)
- CMS DECwindows Motif and DCL command-line online help
- *Guide to DIGITAL Code Management System for OpenVMS Systems*
- *DIGITAL Code Management System Reference Manual*
- *DIGITAL Code Management System Callable Routines Reference Manual*

1.1.2 LSE

LSE is a text editor built on top of the DEC Text Processing Utility (DECTPU). It contains built-in knowledge of a variety of programming languages supported by the OpenVMS operating system. LSE enables application developers to extend and customize their DECTPU editing environment to match specific programming and style preferences. From within LSE, developers can edit, compile, and debug applications as well as create low-level program designs (by embedding pseudocode in source code). They can also create different views of the source code, at various levels of detail, by replacing a sequence of source lines with a single overview line.

Detailed information on LSE can be found in:

- *HP DECset for OpenVMS Software Product Description*
- *Language-Sensitive Editor for OpenVMS Release Notes* (SYS$HELP:LSE*.RELEASE_NOTES)
- LSE (and SCA) DECwindows Motif and DCL command-line online help. LSE has both Portable and VMSLSE command language help.
- *Guide to DIGITAL Language-Sensitive Editor for OpenVMS Systems*
- *DIGITAL Language-Sensitive Editor/Source Code Analyzer for OpenVMS Reference Manual*
- *DIGITAL Language-Sensitive Editor Command-Line Interface and Callable Routines Manual*

1.1.3 SCA

SCA is an interactive cross-reference and static source code analysis tool that works with a variety of languages supported by the OpenVMS operating system. In addition to syntax and reference checking, it also provides navigation capabilities that help developers check the consistency of their source code as well as find and view specific code components.

Detailed information on SCA can be found in:

- *HP DECset for OpenVMS Software Product Description*
- *Source Code Analyzer for OpenVMS Release Notes* (SYS$HELP:SCA*.RELEASE_NOTES)
- SCA (and LSE) DECwindows Motif and DCL command-line online help. Like LSE, SCA also has both portable and VMSLSE versions of its help files.
- *Guide to DIGITAL Source Code Analyzer for OpenVMS Systems*
- *DIGITAL Language-Sensitive Editor/Source Code Analyzer for OpenVMS Reference Manual*
- *DIGITAL Source Code Analyzer Command-Line Interface and Callable Routines Manual*

1.1.4 MMS

MMS automates and simplifies the building of applications in the OpenVMS environment. It rebuilds only those components (and their dependencies) that have changed since an application was last built, which helps to optimize the build process. MMS can also generate description files for supported languages automatically (see Chapter 3).

Detailed information on MMS can be found in:

- *HP DECset for OpenVMS Software Product Description*
- *Module Management System for OpenVMS Release Notes* (SYS$HELP:MMS*.RELEASE_NOTES)
- MMS DECwindows Motif and DCL command-line online help
- *Guide to DIGITAL Module Management System for OpenVMS Systems*

1.1.5 DTM

DTM organizes and automates the software regression testing process. It is used to run, review, and store software regression tests and test results. Two types of testing are currently supported:

- Non-interactive, terminal-oriented or workstation testing, based upon command scripts and output capturing
- Interactive, terminal-oriented or DECwindows Motif record-and-play testing

Detailed information on the DTM can be found in:

- *HP DECset for OpenVMS Software Product Description*
- *Digital Test Manager for OpenVMS Release Notes* (SYS$HELP:DTM*.RELEASE_NOTES)

- DTM DECwindows Motif and DCL command-line online help
- *Guide to DIGITAL Test Manager for OpenVMS Systems*
- *DIGITAL Test Manager for OpenVMS Reference Manual*

1.1.6 PCA

PCA helps analyze the performance of software applications. It can measure where an application spends its time, causes page faulting, and performs I/O operations.

PCA consists of two components: the Collector and the Analyzer. The Collector gathers performance and coverage data from a running application and writes that data to a performance data file. The Analyzer then processes and displays the data graphically in the form of histograms and tables.

Detailed information on PCA can be found in:

- *HP DECset for OpenVMS Software Product Description*
- *Performance and Coverage Analyzer OpenVMS Release Notes* (SYS$HELP:PCA*.RELEASE_NOTES)
- *Guide to DIGITAL Performance and Coverage Analyzer for OpenVMS Systems*
- PCA DECwindows Motif and DCL command-line online help
- *DIGITAL Performance and Coverage Analyzer for OpenVMS Reference Manual*
- *DIGITAL Performance and Coverage Analyzer Command-Line Interface Guide*

2
The Environment Manager

The DECset Environment Manager for OpenVMS (Environment Manager) provides a common interface for defining the execution environment for all the DECset tools displaying on a particular DECwindows Motif display device.

The DECwindows Motif interface is provided by the Environment Manager. The command-line interface is provided by DCL DECset commands. Both of these interfaces are described in the *Using DECset for OpenVMS Systems*. Another good source of information is the online help for DECset and the Environment Manager.

Each tailored execution environment is known as a **context**, which consists of a named set of values that customize the DECset tools for a specific activity. The data for each context is stored in a separate ASCII text file. A listing of related contexts is also available in ASCII format, which is known as the **context database**.

Context definitions can be nested, that is, one context can be based upon another context. The base context is known as the **parent**, and the new context is known as the **child**. A child context inherits all information from its parent.

The rest of this chapter describes how to:

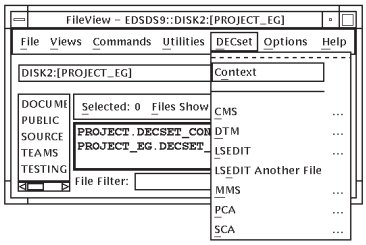
- Access the Environment Manager
- Setup a context database
- Define a series of contexts
- Determine what may be set by a context
- Setup context inheritance

2.1 Accessing the Environment Manager

You can access the Environment Manager from one of the following:

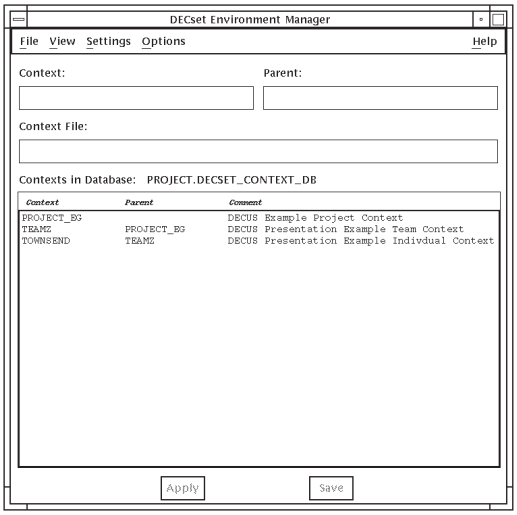
- DECset tool selection or context management pull-down menus available from the DECwindows Motif Session Manager or FileView windows (see Figure 2–1)
- DECwindows Motif interface to a DECset application

Figure 2–1 DECset Tools Selection Pull-down menu



Depending on how the Environment Manager is invoked, it is displayed either as an icon or in its expanded form (see Figure 2–2).

Figure 2–2 Environment Manager Context Window and Icon



From this initial screen you can set up context databases and create and edit contexts. A context allows you to define settings for the default directory, CMS libraries, DTM library, logical names, MMS options, SCA libraries, source directories, and symbols.

Note

Be careful when changing the default directory setting. This setting affects the default directory setting for all DECset components. Once this value is defined or changed in the context, it replaces the user's current default directory setting.

2.2 Setting Up a Project Context Database and Contexts

There are a number of steps involved in setting up a project context:

1. Project definition and initial set up.
2. Context database creation.
3. Project context creation.
4. Team context creation.
5. Individual context creation.
6. Applying and using a context.

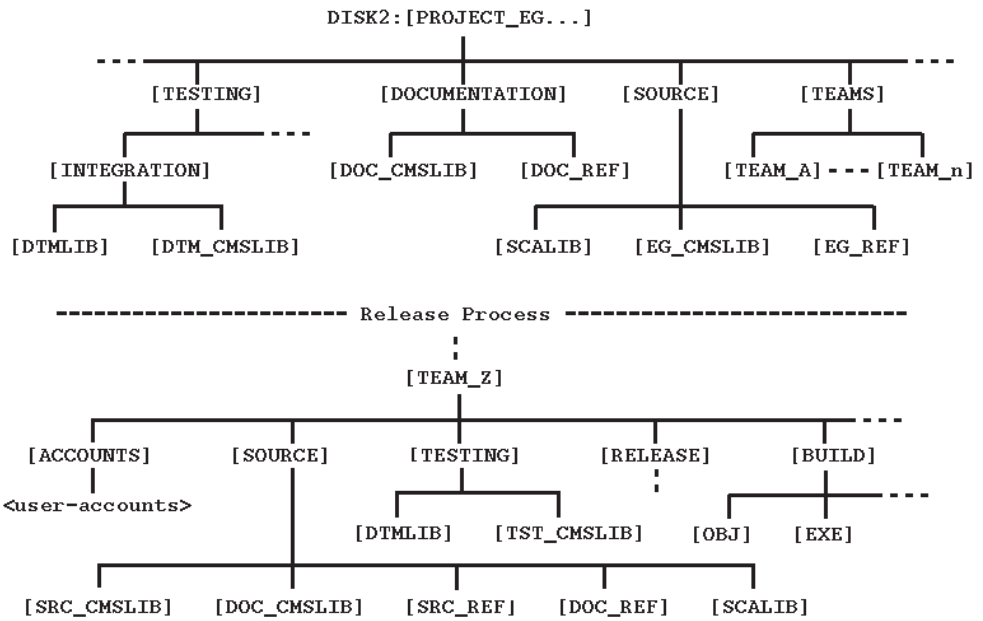
2.2.1 Project Definition and Initial Setup

Some initial groundwork is required before creating project contexts. You need to first specify how the project directories and files are to be organized, how the DECset tools are to look and behave, and what definitions and symbols need to be available.

Most of this information is project- or company-specific, but for the purposes of these examples, the project is broken down by teams. Each team defines their directory and library structure; and the team members tailor their own contexts.

Figure 2–3 illustrates the sample directory and library structures used in this chapter.

Figure 2–3 Sample Project Directory Structure



In this sample:

- The entire project resides on one disk (DISK2).
- All project files reside under the main project directory (PROJECT_EG).
- The team directories and libraries reside under a team directory (TEAMS).

- All the Digital Test Manager libraries use CMS libraries.
- All the non-Digital Test Manager CMS libraries have reference directories.
- There is also an undefined release process to handle the transfer of files from a team to the project. Otherwise, each team is responsible for its own code management.

The examples in this chapter show the setting up of three contexts:

- Project context (PROJECT_EG)
- Team context (TEAM_Z)
- User context (TOWNSEND)

In reality, many other contexts may also be required: contexts for QA, source code librarians, project managers, documentation, release testing, and so on.

Once the project structure has been defined, the next step is to create all the directories and libraries. Figure 2–4 and Figure 2–5 show the creation of SCA and CMS libraries for PROJECT_EG and a test library for TEAM_Z. Subsequent examples assume that all the directories and libraries have been created previously.

Figure 2–4 Creating an SCA and a CMS Library

```
$ CREATE/DIRECTORY DISK2:[PROJECT_EG.SOURCE.SCALIB]
$ SCA CREATE LIBRARY DISK2:[PROJECT_EG.SOURCE.SCALIB]
%SCA-S-NEWLIB, SCA Library created in DISK2:[PROJECT_EG.SOURCE.SCALIB]
$ CREATE/DIRECTORY DISK2:[PROJECT_EG.SOURCE.EG_CMSLIB]  ! library
$ CREATE/DIRECTORY DISK2:[PROJECT_EG.SOURCE.EG_REF]     ! reference directory
$ SET DEFAULT DISK2:[PROJECT_EG.SOURCE]
$ CMS CREATE LIBRARY [.EG_CMSLIB] /REFERENCE_COPY=[.EG_REF]
_Remark: PROJECT_EG CMS Library
%CMS-S-CREATED, CMS Library DISK2:[PROJECT_EG.SOURCE.EG_CMSLIB] created
```

Figure 2–5 Creating a Test Library

```
$ CREATE/DIRECTORY DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING.DTMLIB]
$ CREATE/DIRECTORY DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING.TST_CMSLIB]
$ SET DEFAULT DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING]
$ CMS CREATE LIBRARY [.TST_CMSLIB] "Testing CMS Library"
%CMS-S-CREATED, CMS Library ...TST_CMSLIB] created
$ DTM
DTM> CREATE LIBRARY [.DTMLIB] "Test DTM Library"
%DTM-S-CREATED, Digital Test Manager library ...DTMLIB] created
DTM> SET TEMPLATE_DIRECTORY CMS$LIB ""
%DTM-S-NEWDEF, ...TST_CMSLIB] is the new default collection template directory
DTM> SET BENCHMARK_DIRECTORY CMS$LIB ""
%DTM-S-NEWDEF, ...TST_CMSLIB] is the new default collection benchmark directory
DTM> EXIT
```

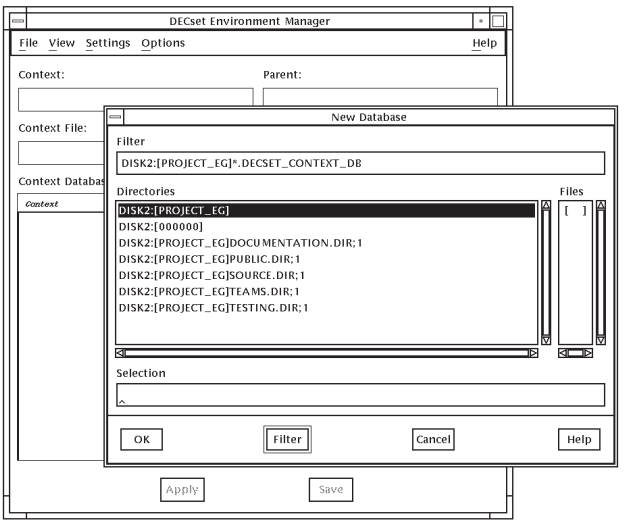
2.2.2 Context Database Creation

After the creation of the directories and libraries, the next step is to create the project context database. This can be achieved by using the Environment Manager, as follows:

1. Select New Database... from the File pull-down menu.
2. Enter the appropriate information (see Figure 2–6).
3. Click the OK button.

Although the database is selected, the database file is not created until the first context is defined and saved.

Figure 2–6 Environment Manager New Database Dialog Box

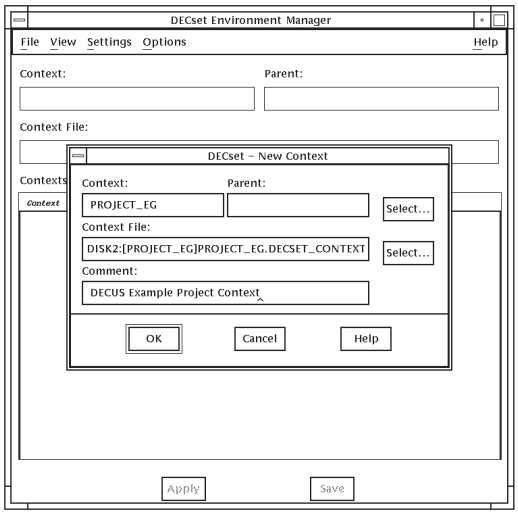


2.2.3 Project Context Creation

A project context can be created from with the Environment Manager, as follows:

1. Select New Context... from the File pull-down menu.
2. Enter the appropriate information (see Figure 2–7).
3. Click the OK button and the context is created.

Figure 2–7 Environment Manager New Context Dialog Box



All the examples throughout this chapter use the logical name DISK2 to define the disk. However, as the Environment Manager expands all the logical names used in directory specifications, in many of the examples, the translation of the logical for DISK2 (1DIA3) is displayed.

4. Define the context settings. As this process is both straight-forward and described in some detail in the Environment Manager's documentation, examples are limited to the definition of a few logical names (see Figure 2–8) and a test library (see Figure 2–9).

Figure 2–8 Defining Logical Names for a Context

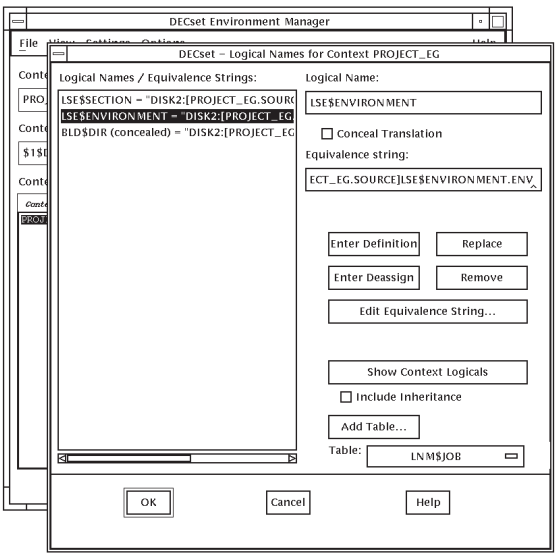


Figure 2–9 Defining a Test Library for a Context

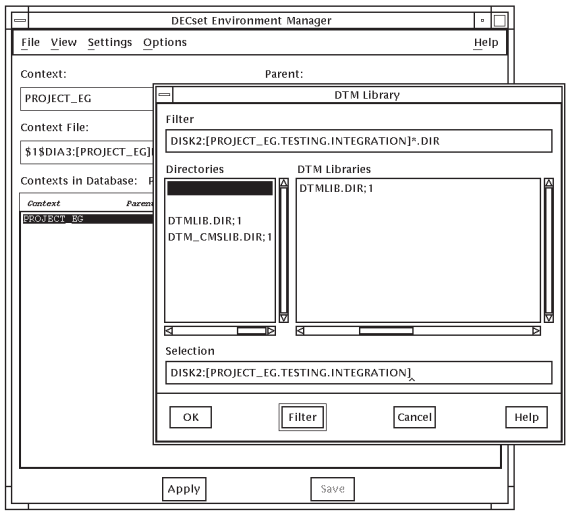


Figure 2–10 contains extracts from a defined project context file.

Figure 2–10 Sample Context File for PROJECT_EG

```
!
! Context file $1$DIA3:[PROJECT_EG]PROJECT_EG.DECSET_CONTEXT
! Written by DECset Environment Manager at 31-OCT-1995 14:17:37.03
! Do not edit this file
!

DEFINE/TABLE=LNM$PROCESS DTM$LIB -
 DISK2:[PROJECT_EG.TESTING.INTEGRATION.DTMLIB]
DEFINE/TABLE=LNM$PROCESS CMS$LIB -
 DISK2:[PROJECT_EG.SOURCE.EG_CMSLIB],DISK2:[PROJECT_EG.DOCUMENTATION.DOC_CMSLIB]
DEFINE/TABLE=LNM$PROCESS SCA$LIBRARY -
 DISK2:[PROJECT_EG.SOURCE.SCALIB]
         :
SET_DIRECTORY SOURCE -
 DISK2:[PROJECT_EG.SOURCE]

CMS SET LIBRARY -
 DISK2:[PROJECT_EG.SOURCE.EG_CMSLIB], -
 DISK2:[PROJECT_EG.DOCUMENTATION.DOC_CMSLIB]

SCA SET LIBRARY -
 DISK2:[PROJECT_EG.SOURCE.SCALIB]

DTM SET LIBRARY DISK2:[PROJECT_EG.TESTING.INTEGRATION.DTMLIB]

SET_MMS RULES
    :
SET_MMS NOLOG
```

Although a comment within each file recommends not editing the content, since a context file is an ASCII text file, it can be edited to remove or add context information. For example, you might find it quicker to add numerous symbols manually instead of using the symbol definition screen. The online help describes the format of a context file and is available by entering the following command:

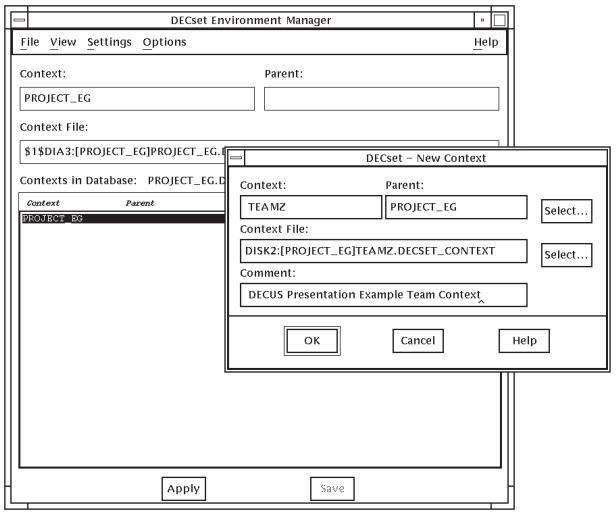
```
$  HELP DECSET CONTEXT_FILE
```

2.2.4 Team Context Creation

Team contexts (for example, TEAM_Z) are created as child contexts of the main project context (PROJECT_EG).

The steps to create a team context are the same as those for the project context (see Section 2.2.3), except that the project context is specified as a parent (see Figure 2–11).

Figure 2–11 Environment Manager New Context Dialog Box



As with the project context, once the team context has been created, the next stage is to define the settings. The following steps show how a linker options file (for MMS builds) can be defined:

1. Select MMS Options from the Settings pull-down menu, then select Linker Options. This brings up the MMS Linker Options Files dialog box.
2. Click the Select... button and specify the options file (see Figure 2–12).
3. Click the OK button to return to the MMS Linker Options Files dialog box.
4. Click the Append button to add the options into the list.
5. Click the OK button to set the options file.

Figure 2–12 Environment Manager Select Option File Dialog Box

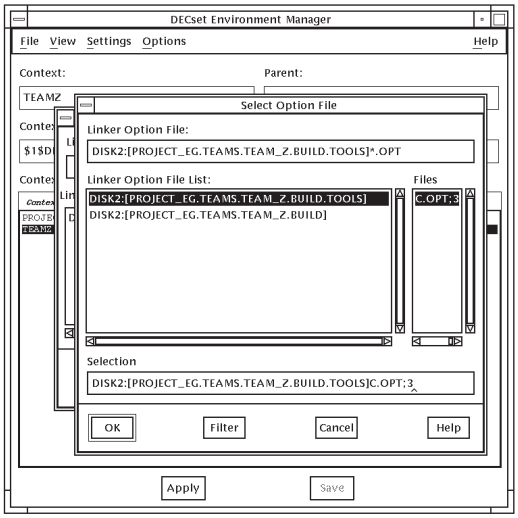


Figure 2–13 contains extracts from the TEAM_Z context file after the libraries and source directories, an MMS rules file, a default description file, a linker options file, and some symbols have been set.

Figure 2–13 Sample Context File for TEAM_Z

```
!
! Context file $1$DIA3:[PROJECT_EG]TEAMZ.DECSET_CONTEXT
! Written by DECset Environment Manager at 31-OCT-1995 17:11:59.71
! Do not edit this file
!

DEFINE/TABLE=LNM$PROCESS SCA$LIBRARY -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SCALIB]
DEFINE/TABLE=LNM$PROCESS DTM$LIB -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING.DTMLIB]
            :
DEFINE/TABLE=LNM$PROCESS CMS$LIB -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SRC_CMSLIB], -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.DOC_CMSLIB]

LEADER == "EDSDS1::SMITH"
     :
QA == "EDSDS1::JONES"

SET_DIRECTORY SOURCE -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE]

CMS SET LIBRARY -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SRC_CMSLIB], -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.DOC_CMSLIB]

SCA SET LIBRARY -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SCALIB]

DTM SET LIBRARY DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING.DTMLIB]

SET_MMS RULES=DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]RULES.MMS

SET_MMS DESCRIPTION=( -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]DESCRIP.MMS)
```

(continued on next page)

Figure 2–13 (Cont.) Sample Context File for TEAM_Z

```
SET_MMS SCA_LIBRARY

SET_LINKER OPTIONS_FILE -
 DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]C.OPT

SET_MMS NOOVERRIDE

SET_MMS CMS
  :
SET_MMS NOLOG
```

2.2.5 Personal Context Creation

A personal or user context (for example, TOWNSEND) is created as a child of the team context. The steps to create it are identical to those for TEAM_Z (see Section 2.2.4), except TEAM_Z is specified as the parent.

Since the steps in the previous examples are similar, sample screens showing the tailoring process for a personal context have not been provided. However, for the purposes of this example, the following definitions are assumed:

- Redefinition of the LSE environment and section files to pick up the local customizations.
- Definition of a local SCA library in a search list with the team's library.
- MMS definitions to build from the source directories, rather than CMS, by default, and specification for compilation settings (Diagnostic and Debug flags).

You can show how a context (combined with its parent contexts) will tailor an environment by selecting View Context Including Inheritance from the View pull-down menu in the Environment Manager.

The equivalent DCL command is DECSET SHOW CONTEXT. Figure 2–14 contains extracts from this command for the TOWNSEND context.

Figure 2–14 DECSET SHOW CONTEXT Command Example

```
$ DECSET SHOW CONTEXT

Context TOWNSEND
------------------------------------------------
Context description: DECUS Presentation Example Individual Context
Name of parent context: TEAMZ
Database file spec: $1$DIA3:[PROJECT_EG]PROJECT.DECSET_CONTEXT_DB
Context file spec: $1$DIA3:[PROJECT_EG.TEAMS.TEAM_Z.USERS.TM1]PT1.DECSET_CONTEXT;
No default directory was specified.
```

(continued on next page)

Figure 2–14 (Cont.) DECSET SHOW CONTEXT Command Example

```
Logical Name Definitions + Deassignments (by table):
    Logical Name Table: LNM$PROCESS
      Definitions:
        DTM$LIB =
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING.DTMLIB]
        CMS$LIB =
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SRC_CMSLIB]
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.DOC_CMSLIB]
        SCA$LIBRARY =
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.USERS.TM1.WORK.SCALIB]
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SCALIB]
        LSE$SOURCE =
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE]
        DOC$ =
            DISK2:[PROJECT_EG.DOCUMENTATION.DOC_REF]
        SRC$ =
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCES]
        TOOLS$ =
            DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]
      Deassignments:
    Logical Name Table: LNM$JOB
      Definitions:
        LSE$SECTION =
            DISK2:[PROJECT_EG.SOURCE]LSE$SECTION.TPU$SECTION
        LSE$ENVIRONMENT =
            DISK2:[PROJECT_EG.SOURCE]LSE$ENVIRONMENT.ENV
      Deassignments:

Symbol definitions:
       :
Symbol deletions:

MMS macro definitions:

Source directories:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE]

CMS libraries:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SRC_CMSLIB]
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.DOC_CMSLIB]

SCA libraries:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.USERS.TM1.WORK.SCALIB]
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SCALIB]

DTM libraries:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.TESTING.DTMLIB]

MMS macro definition files:

MMS rules file:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]RULES.MMS

MMS description files:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]DESCRIP.MMS

MMS changed sources:

MMS SCA library to generate:
    :
Linker options files:
    DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]C.OPT
```

(continued on next page)

Figure 2–14 (Cont.) DECSET SHOW CONTEXT Command Example

```
MMS-related Flags:
Override macros: 0
    :
Enable DEBUG macro (MMS): 0
    :
[End of Listing]
```

2.2.6 Applying and Using a Context

After a context has been created, it must be applied before taking effect. You can apply a context in any of the following ways:

- Click the Apply button on the Environment Manager main window.
- Enter the DECset SET CONTEXT command:

```
$ DECSET SET CONTEXT TEAM_Z -
_$ /DATABASE=DISK2:[PROJECT_EG]PROJECT.DECSET_CONTEXT_DB
```

- Define the DECSET$CONTEXT logical names:

```
$ DEFINE DECSET$CONTEXT TEAM_Z
$ DEFINE DECSET$CONTEXT_DB DISK2:[PROJECT_EG]PROJECT.DECSET_CONTEXT_DB
```

Once a context has been applied and a DECset tool activated, it influences the current environment.

This chapter has focused on using the Environment Manager to tailor the development environments for people working on related projects and across teams. However, the Environment Manager is also useful for an individual user for tailoring and switching between contexts, especially when working in environments with varied development requirements.

3
The MMS Description File Generator

This chapter describes the description file generator for the Module Management System for OpenVMS (MMS). The MMS description file generator currently supports the following programming languages: BASIC, BLISS, C, C++, CDD/Plus, COBOL, Fortran, Pascal, RDB, and SQL.

You can generate a description file from the MMS command-line (MMS/GENERATE) or from within the DECwindows Motif interface. The general process is as follows:

1. You supply the generator with a list of language source files.
2. The MMS description file generator scans these source files.
3. Once the scan is complete, it then writes the following information about each source file to a separate description file:
   - Target for the source
   - Source
   - Related dependencies
   - An optional action line

Detailed information on the description file generator may be found in the following locations:

- *Guide to DIGITAL Module Management System for OpenVMS Systems* (/GENERATE qualifier)
- Online help ($ HELP MMS /GENERATE)

3.1 MMS Builder and Generator

Description file generation is separate and distinct from application building. They are implemented by very different parts of MMS and are subject to different qualifiers and options.

Generally, anything set using the Options pull-down menu in the main MMS window *only* affects application builds, and anything set in the MMS Description File Generator dialog box *only* affects description file generation.

Although, many of the specifications captured in the description file affect an application build, the steps and qualifiers used in each process are independent. This distinction carries to the command line, where the qualifiers on the MMS build command line are separate from the ones used on the MMS generate command line.

3.2 Basic Description File Generation

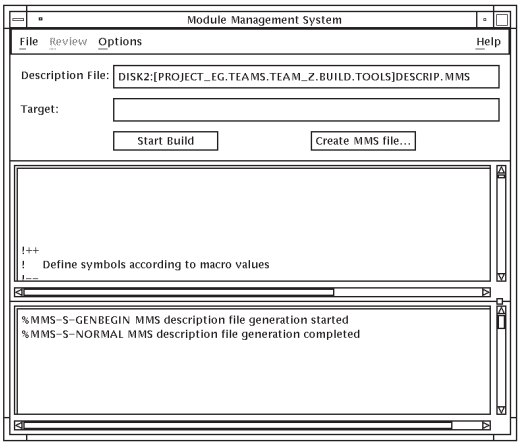
This section shows how a description file can be generated for a C program consisting of a main module (MAIN.C) and some external modules (MODULE*.C) with header files (MODULE*.H). The source files contain interdependencies and references to HP C Run-Time Library functions.

Assuming that all the source files are in the default directory, the MMS/GENERATE command can be used:

```
$ MMS/GENERATE/OPTIONS_FILE=TOOLS$:C.OPT MAIN.C, MODULEA.C, MODULEB.C, -
_$ MODULEC.C, MODULED.C
MMS-S-GENBEGIN MMS description file generation started
MMS-S-NORMAL MMS description file generation completed
```

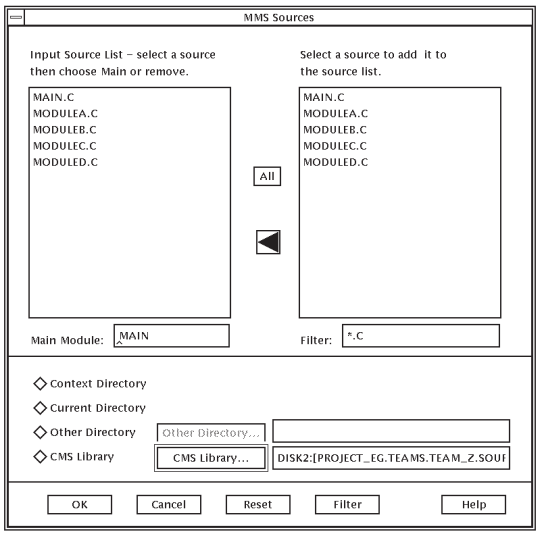
The description file can also be generated from the DECwindows Motif interface. The remainder of this section shows the DECwindows Motif interface being used on the same set of files; however, this time the files are stored in a CMS library.

Figure 3–1 MMS Main Window



1. From the MMS main window, click the Create MMS File... button to bring up the MMS Description File Generator dialog box.
2. Click the Sources... button to bring up the MMS Sources dialog box, select the CMS library, and create the input source list (see Figure 3–2).

Figure 3–2 MMS Sources Dialog Box



3. Exit the MMS Sources dialog box and generate the description file by clicking the Generate button. Figure 3–1 shows the MMS main window following the successful generation of a description file.

Both the command line and DECwindows Motif interface generate the same description file (see Figure 3–3).

Figure 3–3 Automatically Generated Description File

```
!     Define symbols according to macro values ❶
!==

.IFDEF DEBUG
DBG = /DEBUG
DBGOPT = /NOOPTIMIZE/DEBUG
.ELSE
DBG = /NODEBUG
DBGOPT = /OPTIMIZE/NODEBUG
.ENDIF

.IFDEF LIST
    :
.IFDEF DIAG
    :
.IFDEF PCA
    :

!++
!      List of tools used and required symbols ❷
!==

!      !C used
!      !Executables used

!++
!     Missing sources catch-all
!==

.DEFAULT  ❸
   ! No source found for $(MMS$TARGET_NAME)
   ! - Attempting to continue

!++
!   Complete application - default build item ❹
!==

COMPLETE_APPLICATION depends_on -
     MAIN.EXE
        CONTINUE

!++
!   C  ❺
!==

MMS$OLB.OLB(MAIN=MAIN.OBJ) depends_on -
      MAIN.C -
     ,MODULEA.H -
     ,MODULEB.H -
     ,MODULEC.H -
     ,MODULED.H -
     !
       $(CC) $(CFLAGS)  $(LST) $(DBGOPT) $(DIA)  /OBJ=MAIN MAIN.C
       LIBRARY/REPLACE MMS$OLB.OLB MAIN.OBJ
       DELETE MAIN.OBJ;*
```

(continued on next page)

Figure 3–3 (Cont.) Automatically Generated Description File

```
MMS$OLB.OLB(MODULEA=MODULEA.OBJ) depends_on -
      MODULEA.C -
     !
       $(CC) $(CFLAGS)  $(LST) $(DBGOPT) $(DIA)  /OBJ=MODULEA MODULEA.C
       LIBRARY/REPLACE MMS$OLB.OLB MODULEA.OBJ
       DELETE MODULEA.OBJ;*

MMS$OLB.OLB(MODULEB=MODULEB.OBJ) depends_on -
     :
MMS$OLB.OLB(MODULED=MODULED.OBJ) depends_on -
     :

!++
!   Links ❻
!==

MAIN.EXE depends_on -
      MMS$OLB.OLB(MAIN=MAIN.OBJ) -
     ,MMS$OLB.OLB(MODULEA=MODULEA.OBJ) -
     ,MMS$OLB.OLB(MODULEB=MODULEB.OBJ) -
     ,MMS$OLB.OLB(MODULEC=MODULEC.OBJ) -
     ,MMS$OLB.OLB(MODULED=MODULED.OBJ) -
     !
       LINK $(DBG) $(PCAOPT)  /EXE=MAIN.EXE MMS$OLB.OLB/LIBRARY/INCLUDE=(MAIN) -
       ,DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD.TOOLS]C.OPT/OPT- !Link options file
          ! End of Link

!++
!   Files not found.  MMS references to these  ❼
!   files will generate errors.
!==
!++

!++
!   Objects not found.  The following symbols may be undefined.  ❽
!==

!++
!  Create object library if it doesn't already exist ❾
!==

.FIRST
       IF F$SEARCH( "MMS$OLB.OLB" ) .EQS. "" -
  THEN $(LIBR)/CREATE MMS$OLB.OLB

!++
!  End of build cleanup work ❿
!==

.LAST
       CONTINUE
```

All description files that are generated automatically have the same general format, as described below:

❶ Macro symbols for DEBUG, DIAGNOSTICS, LIST, and PCA.

❷ A list of the tools and images used (normally language compilers).

❸ A rule to handle missing source files.

❹ A rule for the complete application, with a dependency on the main image.

❺ Rules for the individual source files. These rules will usually lead to the creation of an object in an object library, followed by a list of dependent files and compilation action lines.

❻ Rules for linking the main image followed by a list of dependencies on the object in the object library and linker rules.

❼ A list of any dependency files not found by the description file generator.

❽ A list of any symbols that appear to be referenced but are not defined.

❾ A first directive creating the object library, if it does not already exist.

❿ A last directive.

Each generated description file may contain a list of files that could not be found and a list of symbols for which no definition can be found. If both missing files and symbols definitionst are listed, check why the generator has been unable to find them. It may be due to a missing logical name or an incorrect definition.

If only missing symbols definitions are listed, the symbols may be defined in object libraries that were not constructed as part of current the build. If this is the case, performing a full build will determine whether the symbol definitions are really missing.

3.3 More Complex Description File Generation

This section illustrates description file generation using more complex examples. Section 3.3.1 shows a Pascal example where the main source files are in one directory and the environment files in another. Section 3.3.2 shows a COBOL example with the source files in CMS and a requirement to place all the output files into type-related directories.

3.3.1 Generating a Description File for Pascal

In this example, the main Pascal source files (TEST22.PAS and SUBTEST*.PAS) reside in DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.PASCAL], while the environment files (TEST_FUNC*.PAS) are in DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.ENV].

The following steps show how to generate a description file using the DECwindows Motif interface:

1. Set default to a search list of directories that allows the description file generator to see all the source files:

```
$ DEFINE SOURCE$DIR DISK2:[PROJECT_EG.TEAMS.TEAM_Z.BUILD], -
_$ DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.PASCAL], -
_$ DISK2:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.ENV]
$ SET DEFAULT SOURCE$DIR
```

2. Before building the image, ensure that any logical names used within the source files are defined:

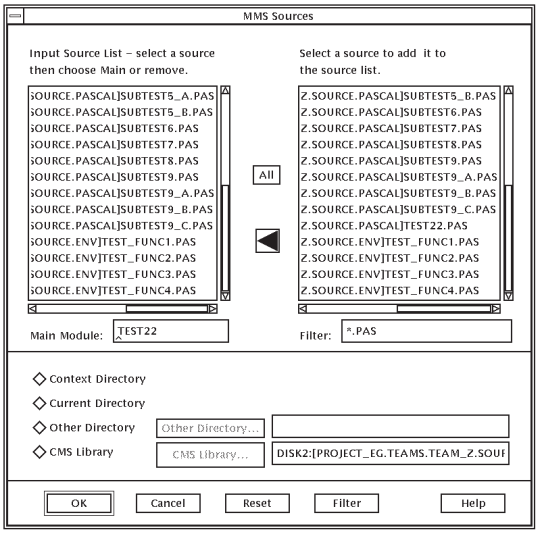
```
$ TYPE TEST_FUNC1.PAS
[ ENVIRONMENT ('ENVDIR:TEST_FUNC1') ]
MODULE TEST_FUNC1;
   :
END.

$ DEFINE ENVDIR DISK$:[DECUS.PASCAL.ENV]
```

3. From the MMS main window, click the Create MMS File... button to bring up the MMS Description File Generator dialog box.

4. Click the Sources... button to bring up the MMS Sources dialog box.
5. Select the .PAS files for the Input Source List. Note that the environment file sources should not be selected, as the description file generator will find them when it scans for dependencies (see Figure 3–4).

Figure 3–4 MMS Sources Dialog Box: Pascal Example



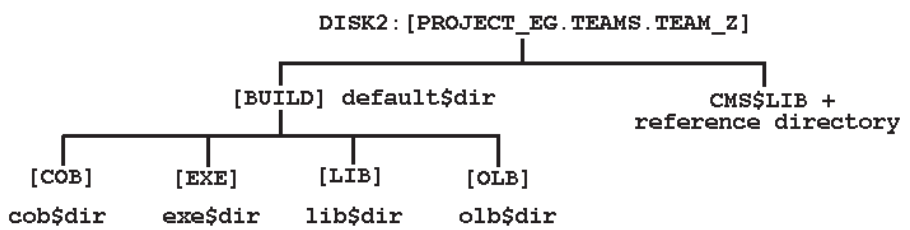
6. After entering the Input Source List and Main Module values, click the OK button to return to the MMS Generate Description File dialog box.
7. Click the Generate button to create the description file and return to the MMS main window.

The description file is now ready for use.

3.3.2 Generating a Description File for COBOL

This example is based around the following:

- The COBOL source files are fetched into one directory (COB).
- The library files are fetched into another directory (LIB).
- The image file is built in an images directory (EXE).
- The object library is built in an object library directory (OLB).
- The description file(s) are built and kept in another directory (BUILD).



Logical name definitions (which could be defined in an Environment Manager context) are required for each of the directories.

3.3.2.1 Creating the Description File from the CMS Library

The steps below show how the description file may be generated from the DECwindows Motif interface.

1. Click the Create MMS File... button to bring up the MMS Description File Generator dialog box.
2. Enter the Object Library (OLB$DIR:TEST.OLB), the description file name (COBOL.MMS), and then click the Sources... button to bring up the MMS Sources dialog box.
3. Enter the CMS library, filter the COBOL files, place all the source files into the Input Source List, and specify the main module (see Figure 3–5).
4. Click the OK button to return to the Generate Description File dialog box.
5. Click the Generate button to return to the MMS main window and generate the description file (see Figure 3–6).

Figure 3–5 MMS Sources Dialog Box: COBOL Example

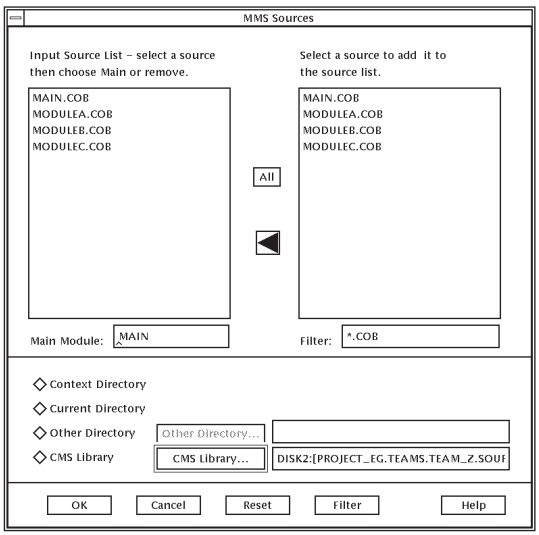
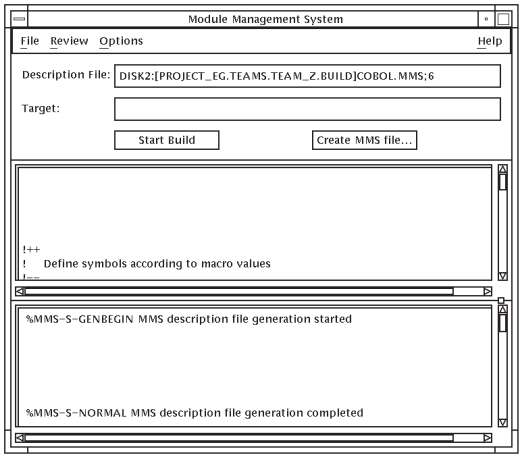


Figure 3–6 MMS Main Window: COBOL Example



3.3.2.2 Creating the Description File from the Command Line

The description file may also be generated from the command line using the CMS reference directory:

```
$ SET DEFAULT $1$DIA3:[PROJECT_EG.TEAMS.TEAM_Z.SOURCE.SRC_REF]
$ !
$ MMS/GENERATE/DESCRIPTION=DEFAULT$DIR:COBOL.MMS/OBJECT=OLB$DIR:TEST.OLB -
_$ MAIN.COB, MODULEA.COB, MODULEB.COB, MODULEC.COB
%MMS-S-GENBEGIN MMS description file generation started
%MMS-S-NORMAL MMS description file generation completed
```

3.3.2.3 Building the Application

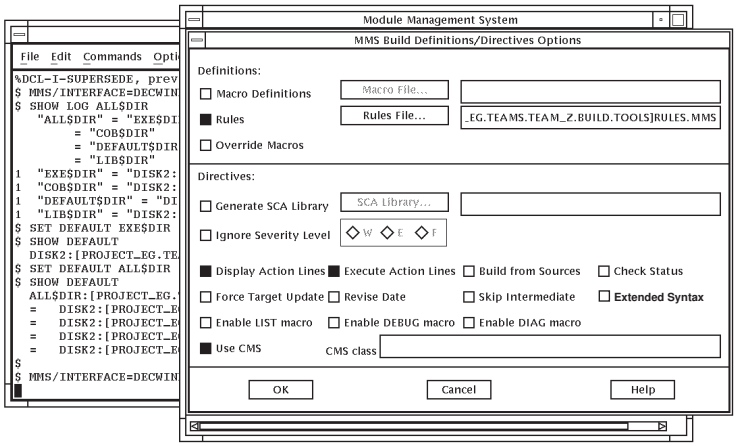
Regardless of how the description file is generated, you must set the default directory to the sources directory (COB$DIR) before building the application.

Like the generation of the description file, you can build the application using either the DECwindows Motif interface or the command line.

The steps below show how to build the application using the description file generated by following operations in the previous sections:

1. Select Open Description file... from the File pull-down menu and set the description file, COBOL.MMS, in the Select MMS description file dialog box.
2. Select Definition/Directive... from the Options pull-down menu and make sure that the Use CMS flag is set in the MMS Build Definitions/Directives Options dialog box. (see Figure 3–7).
3. Once the description and build options have been set, click the Start Build button in the main window (see Figure 3–1).

Figure 3–7 MMS Build Definitions/Directives Options Dialog Box



To build an application from the command line, entering the follow command:

```
$ MMS/DESCRIPTION=DEFAULT$DIR:COBOL.MMS/CMS
```

4
DECset Component Integration

Individually, each DECset tool can significantly increase productivity in specific areas of the OpenVMS application development cycle. However, when used in concert with each other, the DECset tools form an powerful, integrated application development environment with the following characteristics:

- Support for the multiple phases of the software lifecycle.
- Support for applications written in multiple languages.
- Compilers and tools that pass substantial information among themselves to automate tasks previously performed manually.

This chapter describes the integration that exists between DECset and the OpenVMS application development environment. It provides an overview of the data flow between the OpenVMS language compilers and the DECset tools as well as examples of interaction between the tools themselves.

For additional, detailed information about how the DECset tools work with OpenVMS system services and its language compilers to improve application design and maintenance, see the following manuals:

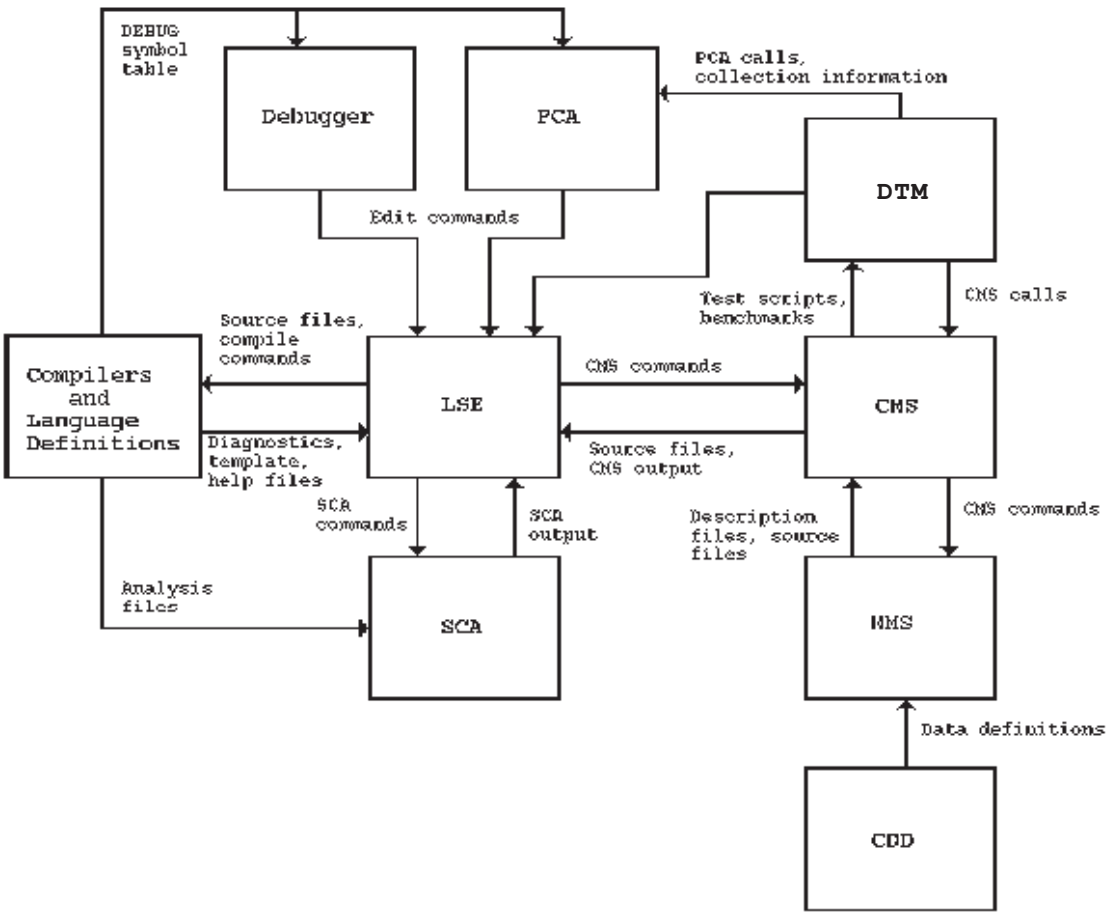
- *DECset Guide to Detailed Program Design for OpenVMS Systems*
- *Using DECset for OpenVMS Systems*

4.1 Integration With the OpenVMS Environment and Language Compilers

The OpenVMS language compilers generate a substantial amount of information for the DECset tools, such as symbol table information for PCA, diagnostic information for LSE, and cross-reference and calling-sequence information for SCA.

Figure 4–1 shows the many connections and information flow among the tools that give the OpenVMS programming environment its high level of integration. The boxes represent tools, and the arrows represent information flow, either by means of files or through direct calls between the tools.

Figure 4–1 DECset Tools Integration



4.2 Integration Among the DECset Tools

The following sections provide examples of how the various DECset tools interact in an application development environment. Note that an Environment Manager context (TEST1) was used to tailor all the DECset tools used in the examples.

4.2.1 CMS

CMS elements may be placed into LSE buffers directly from the DECwindows Motif interface, as follows:

1. Select the element (MB1 DISPLAY_TABLES.ADA).
2. Select Fetch from the File pull-down menu (see Figure 4–2).
3. Click the OK Button on the Fetch dialog box. The element is fetched into an LSE Buffer (see Figure 4–3).

Figure 4–2 CMS: Fetching an Element

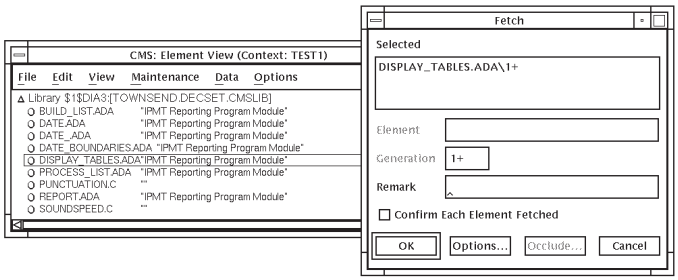
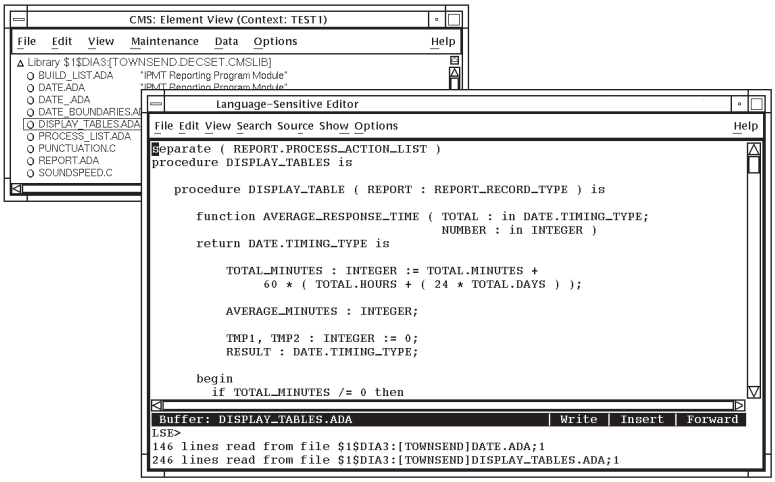


Figure 4–3 CMS & LSE: Element Fetched into an LSE Buffer



This process also works for reserving elements. If an LSE DECwindows session is not active (either expanded or iconified), the element is fetched into the default directory. Once the elements are in the buffer they can be manipulated via LSE commands.

Because CMS can store any file as an element, it is particularly useful as a central repository, not only for source files of code and documentation, but also for a variety of files generated by other DECset tools. CMS can store description files for MMS, test files (prologue, template, epilogue) for the Digital Test Manager, and benchmark files. MMS (see Chapter 3) can also access elements in CMS libraries automatically.

4.2.2 LSE

LSE provides a highly interactive environment for source code development. Within LSE, you can create and edit code, compile and review that code, and correct compile-time errors. You can also invoke LSE from the OpenVMS Debugger to correct source errors.

LSE is integrated with CMS to provide source code management. You can invoke LSE from the Analyzer portion of PCA and from OpenVMS Mail. LSE is also integrated with SCA, as described in the following section.

4.2.3 LSE and SCA

Using LSE and SCA together creates an integrated editing environment. You can browse through all of your code to look for specific declarations of symbols or other information without regard to file location. This integration is illustrated below in an example which continues on from Figure 4–3.

A CMS element has been fetched into an LSE buffer. Now we have a source file, and assuming that analysis data for the source files have been placed into the current SCA library, SCA FIND commands can be used to access other parts of the program:

1. After finding a reference to a record called TABLE, the Goto Declaration command from the Source pull-down menu is used to find the declaration of TABLE (see Figure 4–4).
2. This finds a declaration based on COMPONENT_COUNT_RECORD within the same file. The process is repeated for COMPONENT_COUNT_RECORD and this time the declaration is in another source program stored in the CMS library.
3. LSE then asks if this file should be fetched into a buffer (see Figure 4–5).
4. LSE fetches the file into a buffer (see Figure 4–6).

Figure 4–4 LSE & SCA: Goto Declaration in Same File

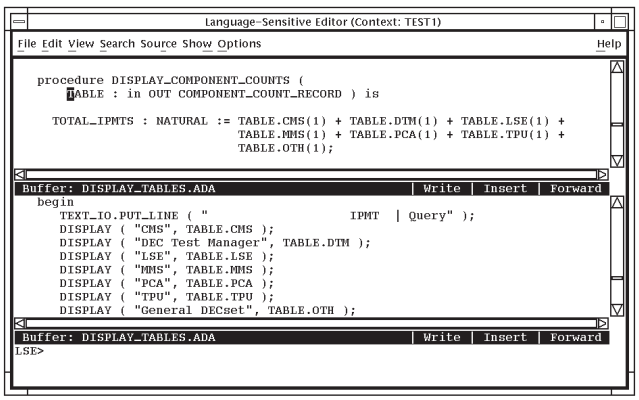


Figure 4–5 LSE & CMS: Goto Declaration in a CMS Element

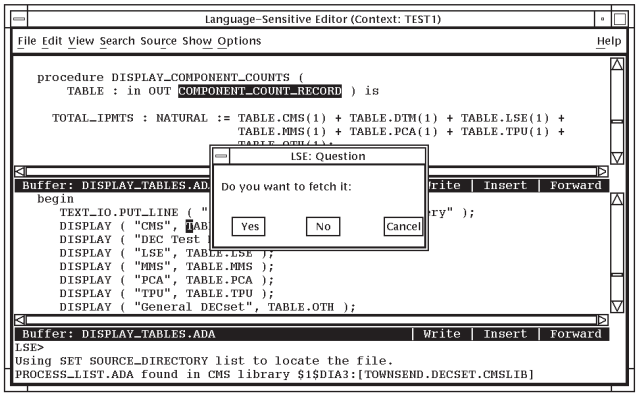
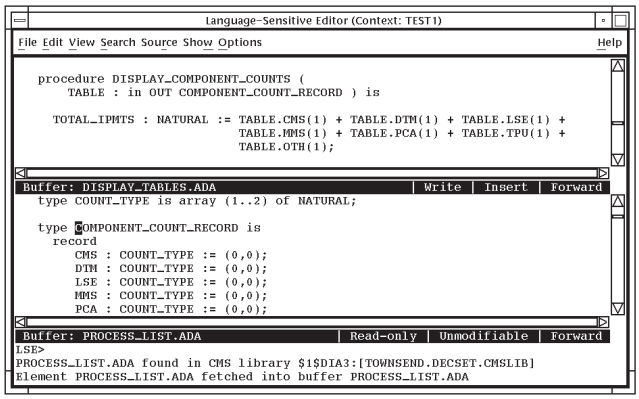


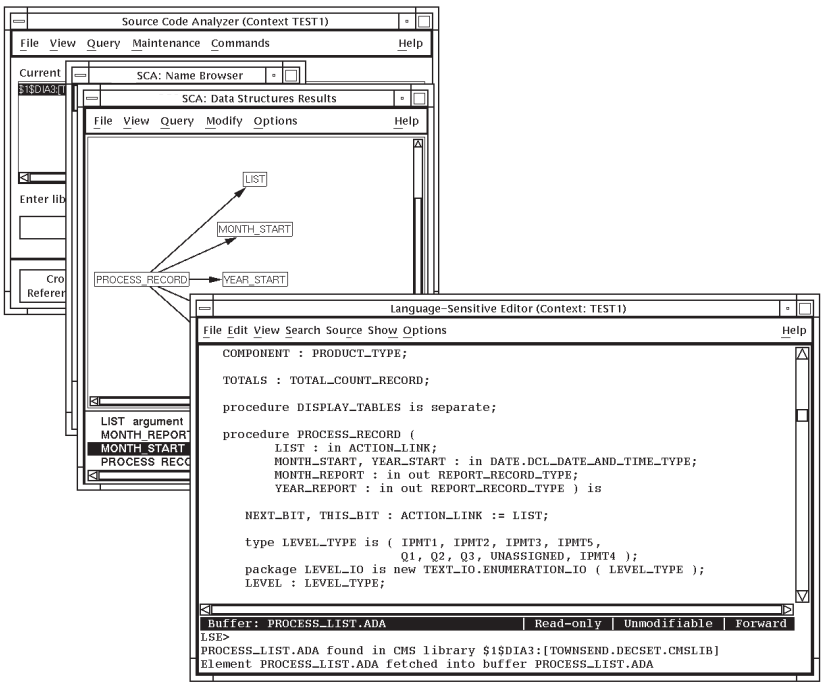
Figure 4–6 LSE & CMS: Element Fetched into an LSE Buffer



If SCA and LSE are running simultaneously in separate windows, LSE can display the source code associated with symbols displayed by SCA. This is illustrated by the example below:

1. Select the Name Browser from the SCA DECwindows interface.
2. Specify a filter (PROCESS*) and then select an object (PROCESS_RECORD) from the list displayed.
3. With the object highlighted, select Type Of from the Query pull-down menu, displaying a diagram of the object (procedure PROCESS_RECORD) in a SCA: Data Structures Results screen.
4. Double-click on the procedure in the Data Structures screen, and the file is loaded into an LSE buffer (see Figure 4–7).

Figure 4–7 SCA & LSE: From Data Structures Results to LSE Buffer



The combination of LSE and SCA also provides the report generation capability of the program design facility. LSE generates and formats the report based on design information provided by SCA.

4.2.4 MMS

MMS can access elements in CMS libraries during builds (see Section 3.3.2.3). All the files used for the build, product documentation, and the MMS description file can be kept in a CMS library. An MMS description file can also build from CMS classes. Additionally, MMS can access records stored in the Oracle Common Data Dictionary (CDD/Plus) or forms stored in libraries for the Forms Management System for OpenVMS (FMS).

MMS provides support for SCA. Analysis data can be generated automatically as part of the MMS build procedure and then stored in an SCA library.

The MMS DECwindows Motif interface is also integrated with LSE, as the example below shows:

1. If an LSE window is open, MMS places the description file into an LSE buffer rather than the Description File Area (see Figure 4–8).
2. When you compile with the diagnostics option, if a build fails due to a compilation failure, MMS (via the Review pull-down menu) allows you to scan the diagnostic files and place the failing source file into an LSE buffer (see Figure 4–9).
3. Once in the buffer, you can use LSE to resolve the problem (see Figure 4–10).

Figure 4–8 MMS & LSE: MMS Description File in LSE

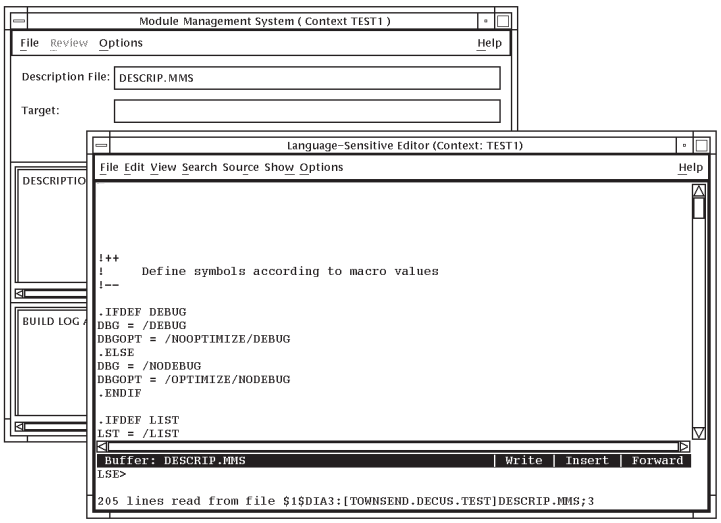


Figure 4–9 MMS & LSE: Compilation Errors During an MMS Build

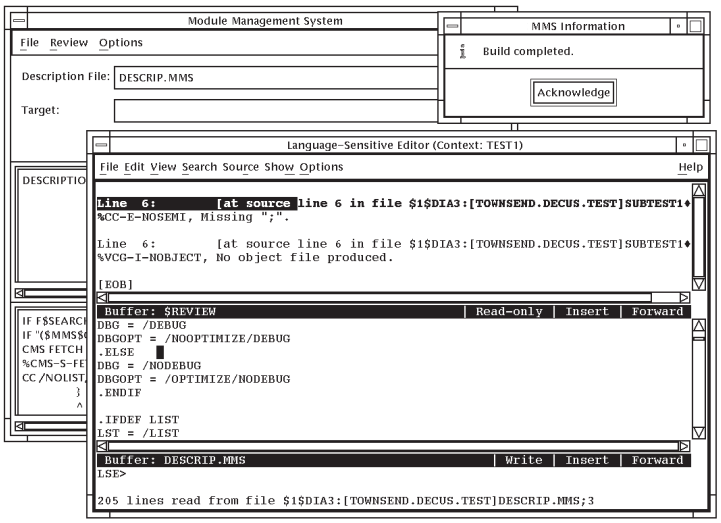
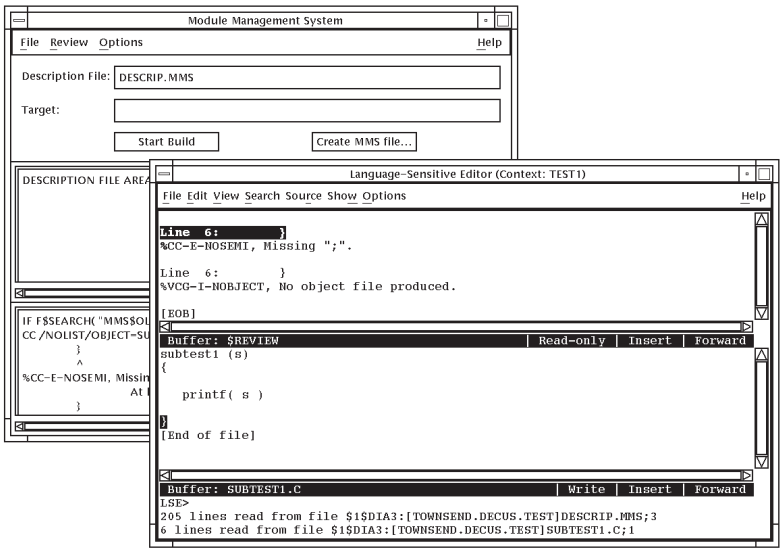


Figure 4–10 MMS & LSE: Correcting Compilation Errors



4.2.5 DTM

You can use the storage and update capabilities of CMS to manage the DTM template, benchmark, test data, prologue, and epilogue files. This is how the libraries were set up in Chapter 2. This feature can be useful in testing multiple versions of a software system in situations where the expected results change from version to version. For example, you can run previous versions of tests and compare their results against the results that were valid for the corresponding maintenance version of a system.

By using DTM with PCA, you can measure the performance or coverage of tests run under DTM. Using an MMS description file to build the application, you can also run those tests automatically.

4.2.6 PCA

When used with DTM, PCA can evaluate the code coverage of your test system. Additionally, you can use the regression tests as performance tests for PCA.

You can also use PCA to analyze programs that are composed of modules written in different languages. Additionally, from PCA you can invoke LSE and have access to all of its features, such as the links to SCA and CMS.

5
CMS and Software Configuration Management

A recurring request from CMS users is for better configuration management support. This chapter attempts to answer these requests by explaining the general philosophy of CMS in regards to configuration management. It also illustrates ways to fulfill various configuration management requirements not directly supported by CMS.

Specifically, this chapter describes the following:

- The Software Configuration Management (SCM) system
- Functions of CMS and its intended role in configuration management
- Useful CMS commands and callable routines for configuration management and how they are used
- Various methods for securing a CMS library and its contents

5.1 Software Configuration Management

This description of a Software Configuration Management (SCM) system is based on the Level 2 SCM process area described in the Software Engineering Institute's (SEI) Capability Maturity Model (CMM). See the *The Capability Maturity Model: Guidelines for Improving the Software Process (Carnegie Mellon University / Software Engineering Institute (1995))* .

The following list of requirements is based on the key SCM activities described in the CMM:

- A configuration library system for a software baseline repository providing support for such things as:
  - Multiple control levels of SCM
  - Storage and retrieval of configuration items
  - Sharing and transfer of configuration items
  - Storage and retrieval of archive versions
  - Ensuring the correct creation of products
  - SCM reports
  - Maintenance of the library structure and contents
- Change requests and problem report tracking, recording and reviewing
- Controlled and managed changes to software and its release
- Check in/out procedures that maintain the integrity of the software baseline

While this list is not exhaustive, it does provide a basis for evaluating an SCM tool, or in this case, CMS.

5.2 The Goals of CMS

CMS was not designed to be a standalone SCM system but to be part of such a system. CMS is a code management tool that provides an efficient and reliable system for storing and tracking all changes to a set of files. To this end, the main goal of CMS is to ensure the integrity of the files within a library. It is this library which forms the base of CMS.

Most of the interaction with the library (from the command line, the DECwindows Motif interface, or the CMS Client interface) is performed using public, callable routines. The routines can be used to write an interface to a CMS library. Section 5.4 briefly describes and provides an example of using the callable routines.

Many of the requirements identified for SCM are directly supported by CMS:

- OpenVMS and CMS ACLs may be used to provide multiple control levels (see Section 5.5).
- The library supports storage and retrieval of configuration items.
- CMS, combined with the OpenVMS environment and the other DECset tools, supports the sharing and transfer of configuration items.
- Storage and retrieval of archive versions is provided through generations and classes.
- Correct creation of products can be ensured by using CMS classes and MMS.
- Maintenance of the library structure and contents is supported via the CMS VERIFY REPAIR and RECOVER procedures.
- Check in/out procedures that maintain the integrity of the software baseline are supported through combinations of these features.

Although CMS meets most of the version control and reliability requirements of an SCM system directly, additional effort is required in the areas of SCM reporting, change requests, and problem report tracking.

5.3 Common CMS Commands for Configuration Management

The following sections describe the CMS commands most often used for configuration management.

5.3.1 Generate a CMS History File (SHOW HISTORY)

A record is generated in a library's history file whenever a CMS command alters the state of the library or is entered with a remark.

The SHOW HISTORY command allows you to review the history file and display a chronological list of the transactions performed on a library. It allows you to select the transactions that are displayed through different parameters and qualifiers on the command.

For example, you can display information on a particular transaction (such as REPLACE) on transactions occurring within specified dates, on a particular object (element, class or group), or on any combination of the following:

```
SHOW HISTORY  [object-expression]

  /BEFORE=date-time and /SINCE=date-time

  /TRANSACTIONS=(keyword,...)

      Displays all transaction records generated by a specific command.
       You can specify the following keywords with this qualifier:

            ACCEPT   FETCH     REMOVE      ALL      INSERT    REPLACE
            CANCEL   MARK      RESERVE     COPY     MODIFY    REVIEW
            CREATE   REJECT    SET         DELETE   REMARK    UNRESERVE
            VERIFY
  /UNUSUAL

     Displays the history records related to unusual occurrences and
     transactions, such as:

         o  A RESERVE command specifying an element that is already
            reserved
         o  A concurrent replacement
         o  A VERIFY/RECOVER command
         o  A VERIFY/REPAIR command
         o  A CONVERT LIBRARY command
         o  A REMARK/UNUSUAL command

  /USER=name

      Displays all history records generated by a specific user.
```

5.3.2 Report on a Single Element (DIFFERENCES and ANNOTATE)

In addition to the SHOW HISTORY command, CMS provides the DIFFERENCES and ANNOTATE commands which generate information about changes made to individual elements. ANNOTATE documents the development of an element by creating a line-by-line file listing the changes made in each specified element generation. DIFFERENCES compares files, element generations, or a file and an element generation.

5.3.3 Track File Development (CREATE or MODIFY ELEMENT/REVIEW)

Development status may be maintained and monitored in CMS if schemes are devised to do so. For example, elements may be placed into release or testing classes when ready. As long as these schemes are followed, showing the contents of these classes will reveal the current status of the elements. Changes to elements or classes may also be tracked through the review attribute or notification access control entries (see Section 5.5.5).

Using the CREATE or MODIFY ELEMENT /REVIEW qualifier, you can specify that newly-created generations of elements are marked for review by placing them on a review pending list. If an attempt is made to reserve or fetch a generation under review, CMS displays a message and prompts for confirmation to continue:

```
CMS> RESERVE test-element "remark"
Generation 1 of element TEST-ELEMENT has a review pending
Proceed?  [Y/N] (N):
```

These messages are issued for all attempts to access the generation until the review status is resolved. Table 5–1 provides a summary of the review commands, which are available from the command line, DECwindows Motif interface, and callable routines:

Table 5–1 CMS Review Commands

Command	Action
ACCEPT GENERATION	Changes the review status of each specified element generation from "pending" to "accepted" and removes it from the review pending list.
CANCEL REVIEW	Changes the review status of each specified element generation from "pending" to "none" and removes it from the review pending list.
MARK GENERATION	Marks each specified element generation for review and adds it to the review pending list.
REJECT GENERATION	Changes the review status of each specified element generation from "pending" to "rejected" and removes it from the review pending list.
REVIEW GENERATION	Associates a review comment with one or more specified element generations.
SHOW REVIEWS_PENDING	Displays a list of element generations that currently have review pending status.

If a review is accepted or canceled, CMS halts the review-related messages and confirmation on subsequent reservation attempts. If the review is rejected, CMS issues a message that the generation was reviewed and rejected. However, the generation is still accessible so that the problems within it may be corrected:

```
CMS> FETCH test-element "remark"
Generation 1 of element TEST-ELEMENT has been rejected
%CMS-S-FETCHED, generation 1 of element TEST-ELEMENT fetched
```

Note

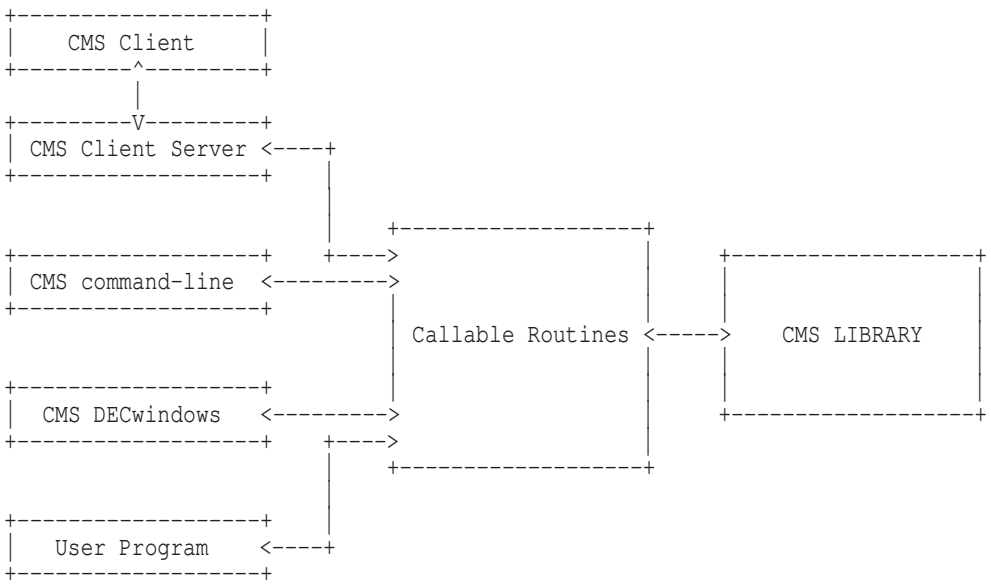
A generation that currently has a review pending or that was previously rejected is marked for review automatically, regardless of the element's review attribute.

5.4 CMS Callable Routines

CMS provides a complete set of routines that may be used to access and interact with CMS libraries from user-written programs. The callable routines represent the kernel that supplies a uniform interface to the CMS front ends, the CMS Client, and any user-written front ends:

Figure 5–1 shows the relationship between the callable routines, the other components of CMS, and a CMS library.

Figure 5–1 CMS Callable Routines Interfaces



The callable routines are described in *DIGITAL Code Management System Callable Routines Reference Manual*. These routines can be called from the majority of the languages supported on OpenVMS, and the documentation reflects this by having examples in different languages.

In general, there is an entry point into CMS for each command at the DCL level, for example:

- CMS$ANNOTATE for CMS ANNOTATE
- CMS$REVIEW_GENERATION for CMS REVIEW GENERATION
- CMS$SHOW_HISTORY for CMS SHOW HISTORY

The routine CMS$CMS allows calling programs to pass a complete CLI to CMS.

Appendix A contains several programs that demonstrate how the CMS callable routines can be used.

Note

The product release notes should also be consulted for descriptions of features added after the documentation was printed and for information about known restrictions and software or documentation errors.

5.5 CMS Library Security

A CMS library is composed of a single directory. This directory, along with the files and subdirectories contained in it, is known as the CMS library directory. This section describes different methods that can be used to control access to a CMS library and its contents.

Caution

To prevent corruption or loss of data, HP strongly recommends that only CMS be used to access a CMS library and the files within it.

5.5.1 CMS Library Access Control Mechanisms

Access to a CMS library can be controlled by using one or more of the following methods:

- OpenVMS file access controls
- CMS file access control
- OpenVMS protected subsystems (OpenVMS Versions 6.2 and later)

Each of these mechanisms is discussed in more detail below. The rest of this section assumes the reader has a basic understanding of OpenVMS file access control. For detailed information on this topic, see the *OpenVMS Guide to System Security*.

5.5.1.1 OpenVMS File Access Controls

The OpenVMS file system provides the lowest level of access control to a CMS library. A user cannot perform CMS operations requiring file access if the appropriate permissions have not been granted by the file system. This also means that any user who has sufficient access privileges to modify a CMS library (for example, to create or reserve and replace elements) can also modify the library files without using CMS.

OpenVMS provides the following mechanisms to control file access:

- **UIC-based protection**

  This is the mechanism most OpenVMS users are familiar with – the READ, WRITE, EDIT, and DELETE (RWED) file protections. UIC-based protection is simple to manage but does not permit detailed access control. UIC-based protection is often inadequate in situations where multiple users with different library security profiles have the ability to create and replace elements in the library. When an element is created or replaced, a new file is created in the CMS library. The file is owned by the user who performed the operation. This can make it difficult to ensure that all users have the same access rights to all elements without the use of ACLs.

- **Access control lists (ACLs)**

  An ACL defines the access rights to an object based on the user's identity (UIC) or the rights identifiers granted to the user's process. The access identifiers can be granted to the user in the system RIGHTSLIST database, be assigned by OpenVMS based on a process characteristic (such as, Interactive, Local, Dialup), or can be assigned to a gatekeeper application for a protected subsystem. In the last case, the user process inherits the rights granted to the application only while the application is executing.

Each ACL consists of one or more access control entries (ACEs). There are several types of ACEs; of note, the Identifier ACE consists of three main components:

1. Identifier—specifies to whom the ACE applies
2. Access Rights—the access to be granted or denied to users holding the specified identifier
3. Options— provide further information on how the ACE is to be applied

An ACL may have multiple ACEs for a given user. If this is the case, the user's access rights are determined by evaluating each Identifier ACE in order until one is found that matches an identifier held by the user. Once a match is found, the access rights specified in that ACE determine the user's access to that object; no further ACEs will be examined. For this reason **the ordering of ACEs in an ACL is very important.**

For more information on OpenVMS ACLs in general, see the *OpenVMS Guide to System Security*. For information on how OpenVMS file protections interact with CMS commands, see the *Guide to DIGITAL Code Management System for OpenVMS Systems*.

An OpenVMS user may possess one or more privileges that override file protection settings: BYPASS, READALL, SYSPRV, or GRPPRV. For example, a user with BYPASS privilege has full access to any file on the system regardless of the file protection. A user with READALL privilege can read all files; all other file access rights are determined by the normal file access controls. For more information on OpenVMS privileges and how they affect file access rights, see the *OpenVMS Guide to System Security*.

5.5.1.2 CMS Access Control

Access to CMS can also be controlled by CMS ACLs. CMS ACLs provide more flexible control of individual CMS commands, CMS library objects (elements, groups, and classes), and objects lists. CMS ACLs refine access granted by OpenVMS file access control mechanisms; they do not override file access restrictions imposed by the OpenVMS file system.

In the absence of CMS ACLs, all users have full access to a library, subject to the file system access controls. If any CMS ACLs are present for a command or object, all access to that command or object is denied unless an access control entry (ACE) specifically granting access is present. This is an important point to remember, as it is easy to inadvertently create conditions where all users are prevented from operating on an object or class of objects.

For example, consider the following situation:

```
CMS> SET ACL/OBJECT_TYPE=COMMAND CREATE_ELEMENT -
_CMS> /ACL=(IDENTIFIER=USER1, ACCESS=EXECUTE+CONTROL)
CMS> SET ACL/OBJECT_TYPE=LIBRARY ELEMENT_LIST -
_CMS> /ACL=(IDENTIFIER=USER2, ACCESS=CREATE)
```

The first CMS command gives USER1 access to execute the CREATE ELEMENT command; all other users are implicitly restricted from using this command because there are no ACEs granting any other user EXECUTE permission. Similarly, the second CMS command gives only USER2 access to add element to the library element list.

Creating an element in a CMS library requires access to both the CREATE ELEMENT command and the library element list. In this library, no users have access to both the command and the element list; therefore, no one will be able to create elements in the library.

CMS ACLs are only checked for operations performed by CMS and the CMS callable routines; they have no effect on OpenVMS utilities or user programs that access the library files by any other means.

Note

Users with BYPASS privilege are granted full access to all CMS commands and objects.

See the *Guide to DIGITAL Code Management System for OpenVMS Systems* for a detailed description of CMS ACLs.

5.5.1.3 OpenVMS Protected Subsystems

Protected subsystems provide a mechanism by which the OpenVMS file system restricts access to certain files related to predefined applications. They allow a CMS library to be configured so that it can only be accessed by CMS (or a suitably privileged user).

In a protected subsystem, file access is denied to all users; access is only allowed via a special subsystem identifier. Rather than being granted to users, the subsystem identifier is assigned to executable images. Only the images that have the appropriate subsystem identifier are able to access the controlled files.

Protected subsystems are available on systems running OpenVMS Versions 6.2 and later. For more information on protected subsystems, (cms_prot_subsys) or the *OpenVMS Guide to System Security*.

5.5.2 Performance Considerations

The use of ACLs provide a great deal of flexibility in configuring the security profile of a CMS library. However, that flexibility comes at some cost. Large access control lists can be expensive in terms of disk storage space and execution speed.

Each ACE occupies disk space in the file header for OpenVMS ACLs and in the CMS library control files for CMS ACLs. When large ACLs are applied to many objects, the cumulative storage requirements can be significant.

Each time an object with an ACL is referenced, each ACE in the list must be compared to the identifiers held by the user until either a match is found or all of the ACEs in the list have been checked. This can be a very time consuming process when the ACLs are very long and/or the checks are performed frequently.

It many cases it may be possible to improve the performance of CMS library access by adhering to following guidelines:

- **Keep ACLs as short as possible.** If there are many users who have the same access rights, use a rights identifier to identify sets of users rather than creating an ACE for each user.
- **Whenever possible, place CMS ACLs on commands rather than on individual objects.** When a command is executed on multiple objects (e.g., FETCH *.*), a command ACL is checked once; the object ACLs must be checked once for each object accessed.

- **Order ACEs so that the most heavily used are closest to the top of the ACL.** This reduces the number of lookups in the most frequent cases. For example, the ACE for a project builder would be before that of a developer, which would be before that of an occasional reader.

  The order of ACEs in an ACL also has security implications that must be considered ahead of the performance implications. In general, ACEs should be specified in the following order:

  - ACEs giving access to critical users belong at the top of the list.
  - ACEs giving specific access to smaller groups belong before ACEs giving access to larger groups.
  - ACEs giving more access rights belong before ACEs giving fewer access rights, unless you mean to selectively deny access.

  More information on ACE ordering can be found in the *OpenVMS Guide to System Security*. The same principles apply to the ordering of ACEs in CMS.

  Notes

  The default behavior of both the OpenVMS SET ACL command and the CMS SET ACL command is to add an ACE to the *beginning* of an existing ACL. If this is not the desired behavior, the position of the ACE can be specified using the /AFTER qualifier (for OpenVMS and CMS) or the /BEFORE qualifier (for OpenVMS only).

  Adding an OpenVMS ACL to existing files within a CMS library will update the file headers. This causes CMS to generate ''File not closed by CMS'' error messages. Once the ACLs have been added, run VERIFY/REPAIR on the library to eliminate the ''not closed by CMS'' messages on the files.

- **Consider using a protected subsystem instead of large OpenVMS ACLs.**
  If your security policy requires large OpenVMS ACLs, consider using a protected subsystem where the user access ACLs are applied to the CMS executable images rather than to each file in the CMS library. The overhead for the ACL check is once per image activation rather than once per file access. To minimize the overhead, multiple CMS commands should be issued from within CMS rather than on separate DCL command line.

5.5.3 Sample CMS Library Configuration

This section shows how to setup a CMS library for a project where different classes of users have different access privileges to a library and its contents. A combination of OpenVMS ACLs and CMS ACLs are used to control access.

5.5.3.1 Access Policy

For this project there are four classes of users with regard to CMS library access:

- Librarian, has complete access to the library and all CMS commands
- Developer, full access except for the DELETE commands
- User, read-only access to the library
- All others, no access

The users in each class are identified by means of OpenVMS rights identifiers. The following rights identifiers are used in this example:

PROJ_LIBRARIAN
PROJ_DEVELOPER
PROJ_USER

The following commands can be used to create these rights identifiers and assign them to users:

```
$ SET DEFAULT SYS$SYSTEM
$ RUN AUTHORIZE
UAF> ADD/IDENTIFIER PROJ_LIBRARIAN
UAF> ADD/IDENTIFIER PROJ_DEVELOPER
UAF> ADD/IDENTIFIER PROJ_USER
UAF> GRANT/IDENTIFIER PROJ_LIBRARIAN librarian
UAF> GRANT/IDENTIFIER PROJ_DEVELOPER devo1
UAF> GRANT/IDENTIFIER PROJ_DEVELOPER devo2
UAF> GRANT/IDENTIFIER PROJ_USER user1
UAF> EXIT
```

5.5.3.2 Library Creation

The CMS library directory is created; then the library itself is created with the CMS CREATE LIBRARY command.

```
$ CREATE/DIRECTORY DISK:[PROJ.CMSLIB]
$ CMS CREATE LIBRARY DISK:[PROJ.CMSLIB]
```

5.5.3.3 Security Policy and Implementation

UIC-based protection

Because OpenVMS ACLs are used to grant or deny access to different classes of users, UIC-based protection is not particularly relevant. In most cases, the ACL overrides the UIC-based protection; however, there are three exceptions to this rule. The UIC-based protection mask is evaluated to determine file access rights if a user in one of the following categories is denied access by the ACL:

- The user has a system group ID
- The user's UIC is the same as the file owner
- The user has file access privileges (BYPASS, READALL, SYSPRV, or GRPPRV)

In the case of a user with BYPASS or READALL privileges, the corresponding file access rights are granted regardless of the file protection mask.

For a more thorough discussion of how file access rights are determined, see the *OpenVMS Guide to System Security*.

Since the UIC file protection can potentially override the access rights specified in an ACL, it is important consider the following when creating the CMS library:

- The CMS library should be owned by either Librarian's UIC or a system UIC.
- If the library is owned by the Librarian, no other user should be assigned the same UIC. (In general, it is not a good idea for multiple users to have the same UIC in any situation.)
- The System field of the UIC protection mask must be changed if you need to restrict library access to users with a System UIC group or SYSPRV privilege. Note that users in either class can modify the file protection or the ACL.
- Any library access control mechanism will be overridden by users with the BYPASS privilege.

OpenVMS ACLs

The following example shows how to set OpenVMS ACLs on library directory and files. For details of the file access permissions required for each of the CMS commands, see the *Guide to DIGITAL Code Management System for OpenVMS Systems*.

- Set the library access ACL:

```
$ SET SECURITY DISK:[PROJ]CMSLIB.DIR -
_$  /ACL=((IDENTIFIER=PROJ_LIBRARIAN, ACCESS=READ+WRITE), -
_$        (IDENTIFIER=PROJ_DEVELOPER, ACCESS=READ+WRITE), -
_$        (IDENTIFIER=PROJ_USER, ACCESS=READ), -
_$        (IDENTIFIER=*, ACCESS=NONE))
```

- Set the library default ACL (gets propagated to all files created in the library):

```
$ SET SECURITY DISK:[PROJ]CMSLIB.DIR -
_$  /ACL=((IDENTIFIER=PROJ_LIBRARIAN, OPTIONS=DEFAULT, -
_$         ACCESS=READ+WRITE+DELETE),-
_$        (IDENTIFIER=PROJ_DEVELOPER, OPTIONS=DEFAULT, -
_$         ACCESS=READ+WRITE+DELETE),-
_$        (IDENTIFIER=PROJ_USER, OPTIONS=DEFAULT, ACCESS=READ), -
_$        (IDENTIFIER=*, OPTIONS=DEFAULT, ACCESS=NONE))
```

Note

The CMS REPLACE command requires DELETE access to element data files. For this reason PROJ_DEVELOPER is given DELETE access permission in the DEFAULT ACL. Access to the CMS DELETE command will be restricted by use of CMS ACLs.

- Apply the default ACL to the CMS library subdirectories (which contain the element data files):

```
$ SET SECURITY DISK:[PROJ.CMSLIB]*.DIR /DEFAULT
```

- Set the ACLs on the CMS library control files:

```
$ SET SECURITY DISK:[PROJ.CMSLIB]*.CMS -
_$  /ACL=((IDENTIFIER=PROJ_LIBRARIAN, ACCESS=READ+WRITE), -
_$        (IDENTIFIER=PROJ=DEVELOPER, ACCESS=READ+WRITE), -
_$        (IDENTIFIER=PROJ_USER, ACCESS=READ), -
_$        (IDENTIFIER=*, ACCESS=NONE))
$ SET SECURITY DISK:[PROJ.CMSLIB]*.HIS -
_$  /ACL=((IDENTIFIER=PROJ_LIBRARIAN, ACCESS=READ+WRITE+DELETE), -
_$        (IDENTIFIER=PROJ=DEVELOPER, ACCESS=READ+WRITE), -
_$        (IDENTIFIER=PROJ_USER, ACCESS=READ), -
_$        (IDENTIFIER=*, ACCESS=NONE))
```

CMS ACLs

CMS ACLs are used to provide more granular restrictions than are available using the OpenVMS file system access control mechanisms. For our project, we need to restrict the CMS DELETE commands to the project librarian.

```
$ CMS SET ACL /OBJECT_TYPE=COMMAND -
_$  DELETE_ELEMENT, DELETE_CLASS, DELETE_GENERATION, MODIFY_ELEMENT -
_$  /ACL=((IDENTIFIER=PROJ_LIBRARIAN, ACCESS=EXECUTE+CONTROL), -
_$        (IDENTIFIER=*, ACCESS=NONE))
```

The security configuration of the project CMS library is complete. See the *Guide to DIGITAL Code Management System for OpenVMS Systems* for detailed information about CMS ACLs.

5.5.4 Implementing a CMS Library as a Protected Subsystem

In a protected subsystem, users do not have access to the subsystem's objects unless they run the application serving as a gatekeeper. For example, for CMS, users do not have access to the library and its files outside of CMS (see the *OpenVMS Guide to System Security*).

The following steps describe how a CMS library can be created as a protected subsystem.

Note

CMS libraries set up as protected subsystems cannot currently be accessed from within LSE or from images not assigned the subsystem identifier.

1. Create the identifier(s) for the subsystem using the subsystem attribute:

```
$ SET DEFAULT SYS$SYSTEM
$ RUN AUTHORIZE
UAF> ADD/IDENTIFIER cms_subsystem/ATTRIBUTES=(RESOURCE,SUBSYSTEM)
%UAF-I-RDBADDMSG, identifier CMS_SUBSYSTEM value %X8001003E added to rights
database
UAF> SHOW/IDENTIFIER cms_subsystem
  Name                            Value           Attributes
  CMS_SUBSYSTEM                   %X8001003E      RESOURCE SUBSYSTEM
```

2. If required, grant the subsystem identifier to anyone who will manage the library, allowing them to assign the subsystem identifier to the images comprising the subsystem and access the library files outside of CMS (if required).

```
UAF>! Grant the librarian access
UAF> GRANT/IDENTIFIER CMS_SUBSYSTEM librarian/ATTRIBUTE=(RESOURCE,SUBSYSTEM)
```

Add the identifier to your rights list and assign the subsystem identifier to the CMS images:

```
$ SET RIGHTS_LIST/ENABLE cms_subsystem/ATTRIBUTE=(RESOURCE,SUBSYSTEM)
$ !
$ SET SECURITY/ACL=(SUBSYSTEM,ID=cms_subsystem,ATTRIBUTES=RESOURCE) -
_$ SYS$SYSTEM:CMS.EXE
$ SET SECURITY/ACL=(SUBSYSTEM,ID=cms_subsystem,ATTRIBUTES=RESOURCE) -
_$ SYS$SYSTEM:CMS$DW.EXE
$ !
$ ! If CMS Client is being used, assign the identifier to the server image
$ SET SECURITY/ACL=(SUBSYSTEM,ID=cms_subsystem,ATTRIBUTES=RESOURCE) -
_$ SYS$SYSTEM:CMS$SERVER.EXE
```

3. Create the directories, assign them the identifier, and create the library (the library creation may be performed from any user account):

```
$ ! Create the directory
$ CREATE/DIR disk$:[protected.secure_lib]
$ !
$ ! Assign the identifiers
$ SET SECURITY/ACL=( -
_$   (ID=cms_subsystem,ACCESS=READ+WRITE+EXECUTE+DELETE), -
_$   (ID=*,ACCESS=NONE)) disk$:[000000]protected.DIR
$ SET SECURITY/ACL=( -
_$   (ID=cms_subsystem,ACCESS=READ+WRITE+EXECUTE+DELETE), -
_$   (ID=*,ACCESS=NONE)) disk$:[protected]secure_lib.DIR
$ !!
$ ! ****** from any (non-privileged account)
$ !!
$ CMS CREATE LIBRARY disk$:[protected.secure_lib]
_Remark: "CMS Library as a Protected Subsystem"
%CMS-S-CREATED, CMS Library DISK$:[PROTECTED.SECURE_LIB] created
%CMS-I-LIBIS, library is DISK$:[PROTECTED.SECURE_LIB]
%CMS-S-LIBSET, library set
$ !
$ CREATE test.ele
this is just a test
$ CMS CREATE ELEMENT test.ele "Testing the library's security"
%CMS-S-CREATED, element DISK:[PROTECTED.SECURE_LIB]TEST.ELE created
$ DIRECTORY DISK:[PROTECTED.SECURE_LIB]
%DIRECT-E-OPENIN, error opening DISK:[PROTECTED.SECURE_LIB]*.*;* as input
-RMS-E-PRV, insufficient privilege or file protection violation
```

5.5.5 Using CMS ACEs for Event Handling

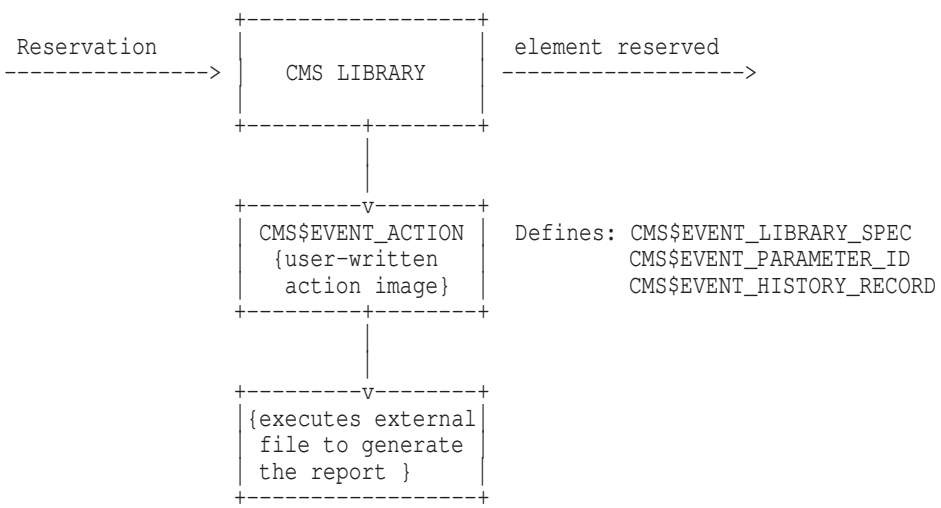
Using CMS ACEs, you can detect events (CMS operations) and specify actions to be taken when these events occur. Event handling can be used for all operations involving one or more of the following objects:

- All elements (element_list), groups (group_list), or classes (class_list)
- Specific elements, groups, or classes
- History, commands, and library attributes

Once an event has been handled, selected information can be extracted to update a database, notify specified accounts, modify a report, and so on. Appendix B contains two sample programs that further illustrate CMS event handling.

Figure 5–2 provides a flow diagram of how the event handler is activated. The rest of this section describes how the event handler is built and used.

Figure 5–2 CMS Event Handling Flow Diagram



The commands below show the event handler being built on an OpenVMS Alpha or OpenVMS I64 system:

```
$ CC/NODEBUG cms_event.c
$ LINK cms_event/SHAREABLE, SYS$INPUT/OPTIONS
SYMBOL_VECTOR=(CMS$EVENT_ACTION=PROCEDURE)      ! Alpha/I64 only
Ctrl/Z
```

On an OpenVMS VAX system, the options file would contain UNIVERSAL=CMS$EVENT_ACTION. For the purposes of this example, assume the image has been built in DISK$:[CMS.ALPHA].

The next step is to define the logical names CMS$HANDLER to point to the location of the command file and SYS$SHARE to point to location of the event handler:

```
$ DEFINE/TABLE=LNM$JOB CMS$HANDLER disk$:[cms]
$ DEFINE/TABLE=LNM$JOB SYS$SHARE disk$:[cms.alpha], SYS$LIBRARY
```

Then assign the ACL:

```
$ CMS SET LIBRARY disk$:[cms.cmslib]
$ CMS SET ACL/OBJECT_TYPE=ELEMENT t.t -
_$ /ACL=(ACTION=CMS_EVENT,PARAMETER="my string value",ACCESS=RESERVE+REPLACE)
_Remark: "Adding the ACL"
%CMS-S-MODACL, modified access control list for element DISK$:[CMS.CMSLIB]T.T
$ !
$ CMS SHOW ACL/OBJECT_TYPE=ELEMENT T.T
ACLs in CMS Library DISK$:[CMS.CMSLIB]

T.T  (ACTION=CMS_EVENT,PARAMETER="MY STRING VALUE",IDENTIFIER=*,
      ACCESS=REPLACE+RESERVE)
CMS> EXIT
```

What happens now when T.T is reserved is illustrated below:

```
$ CMS RESERVE t.t "this is my remark"

An interesting event has occurred:
DISK$:[CMS.CMSLIB]
3-AUG-1995 13:36:52.68 TOWNSEND RESERVE T.T(2) this is my remark
MY STRING VALUE

%CMS-S-RESERVED, generation 2 of element DISK$:[CMS.CMSLIB]T.T reserved
```

6
DECwindows Motif Testing Using DTM

Testing DECwindows Motif applications using the Digital Test Manager for OpenVMS (DTM) contains many pitfalls. This chapter identifies some common problems and gives advice on how best to avoid them.

The objective of DTM is to build a test suite that does not require manual intervention, runs efficiently, and reliably produces the same output. Section 6.1 gives general advice on test organization. Section 6.2 and Section 6.3 discuss ways of reproducing reliable test results and optimizing performance.

When a single system is used to run DTM and test the application, it is difficult to prevent DTM output from interfering with the tests. If possible, DTM should be run from a display other than the test display, as described in Section 6.4. If this is not possible, there are ways to reduce the impact of DTM output. Section 6.5 describes how to configure DTM to run on a single system when the option of running the application on two displays is unavailable.

Section 6.6 and Section 6.7 describe how to start the application to be tested and discuss the use of the Play/Record User Interface.

6.1 Test Organization

Use the following guidelines when designing a DTM test suite:

- **Identify a standard screen configuration.**
  This will be used for the starting and ending screen state for all tests. Determining a standard configuration includes identifying the:
  - Set of visible windows
  - Position of each window
  - Window currently in focus

  This screen configuration will need to be setup before the start of the test suite, so it must be reproducible manually. After completion, the tests should tidy up the screen to return it to the original configuration.
- **Keep tests small.**
  Try to analyze only one function per test. Although this could result in a large suite of many detailed tests, it also makes it less likely that a particular test will be affected by a change in the application. More importantly, when a test fails, finding the cause is quicker and easier when the test is only comprised of a few functions.
- **Consider how application changes may impact test results and operation.**
  Application changes may not only require benchmarks to be updated. Functional or design changes can also interferes with the operation of the test. This can result in the test not only failing, but leaving windows on the screen which cause later tests to fail. Keep test design in mind when

updating an application. For example, when making changes to a menu interface, add new commands to the end of menus, where possible.

- **Keep a record of the purpose, method and actions of each test.**
  This is particularly important if tests need to be rerecorded for some reason, and can help with understanding test failures. The information can be in a separate file or included as comments in the test session file.

  Test session files are simple text files that can be edited with a text editor. For some changes to the application, it may be possible to edit the file rather than rerecording the test. For example, if a button changes position, the coordinates of a button press and release could be changed.

See the *Guide to DIGITAL Test Manager for OpenVMS Systems* for more information on test organization.

6.2 Creating Reproducible Tests

Creating DECwindows Motif tests that succeed reliably is difficult due to the many variables involved. This section presents some important points to consider when creating tests and provides suggestions on increasing successful test reproduction.

6.2.1 Configuring System Options

Most users will configure system options to suit their own preferences. However, in some cases, this can cause DTM DECwindows Motif tests to fail if another user runs the tests, or if the user alters options between recording and running the tests. Similar considerations apply to configuring any options in the applications to be tested or otherwise displayed during testing, such as DECterm options.

The best arrangement is to establish a separate, generic user account for DTM DECwindows Motif testing that uses the default options. This account can then be used to record and run all tests. If DTM is run from a second display device, the other display running the DTM test should be using a DTM account with generic options. Otherwise you may need to configure the account and set security options.

Note, however, there are a few exceptions to using the default options. You should make the following minimal changes to optimize the display for testing purposes:

- **Disable cursor blinking.**
  Select Cursor Blink Disable on the KeyBoard Options dialog box (accessed from the Options menu on the Session Manager). If the DECterm application is used, also disable the cursor blink using the Options, Display screen. Similarly disable cursor blink in any other applications to be used.

- **Minimize all windows.**
  If the test does not require interaction with icons, from the Workspace: Icons Options screen, select Display the Icon Box. During testing, minimize all windows which are not part of the test, including the Icon Box window itself.

- **Increase double-click timeout value.**
  Set the Mouse Double Click Timeout value to the highest allowed value. For further information on problems associated with double clicks, see Section 6.2.2.

6.2.2 Preventing Problems with Multi-Click Operations

There is a feature of DTM DECwindows Motif testing associated with double clicks of the mouse. This can result in what was originally a double click being processed as two single clicks when the test is run.

The problem occurs when the first click causes a program action that updates the screen. In some cases, the time taken to switch from DTM to the application, perform the action, update the screen and switch back to DTM exceeds the Mouse Double Click Timeout period. The problem is therefore more likely to occur when DTM is running on the same system as the application under test.

Similar problems can also occur with other multi-click operations, for example triple-clicking to highlight a line of text.

To prevent this problem from occurring, perform a single-click first, wait for the action and screen update to occur, and then perform the double-click.

6.2.3 Hiding Copyright Notices

DECwindows Motif applications normally display a copyright notice in the window title on startup, which is replaced by a normal title on certain actions or after a certain period of time. This change of title can lead to synchronization delays and failed screen comparisons.

Unless the copyright notice is the subject of the test, it is best to trigger the normal message at the start up the test. This can usually be done by clicking on a menu item, then by clicking on the desktop outside the menu to cancel the menu action.

6.3 Improving Test Performance

Much of the time spent running tests is actually a reflection of the time spent by users deciding which action to perform next. This time can be reduced by editing the session file and modifying the value in the SetDelay record. This value indicates the percentage of the original time to wait. For example, a value of 100 means play all tests at the original speed, and a value of 20 means reduce the think time to one-fifth of the original time.

The actions affected by the SetDelay record are key presses and releases; mouse button presses and releases; mouse movements and screen saves. A value of 5 often works well, which represents a reduction to one-twentieth of the original delay. However, you may need to use a higher value if text synchronization is insufficient for the test to operate correctly. Or, you may need to use a lower value for greater speed. If necessary, the value can be changed dynamically during the test by adding further SetDelay records.

On systems running DTM Version 3.9 or later, you can further enhance performance by eliminating all unnecessary mouse movements when a test is run. This option is controlled on the command line using the /POINTER_ MOTION qualifier or from the DECwindows Motif interface using the Pointer Motion button on the Record window. Note that this option is not compatible with applications that respond to mouse movements without button presses.

6.4 Using an Alternate Display for Testing

The best way of using DTM for DECwindows Motif testing is to employ a second workstation or a PC running eXcursion to control the testing. This prevents the DTM windows from interfering with the testing of the application.

The method for testing on an alternate display varies between the DTM commands and between the character cell and DECwindows Motif interfaces to DTM.

For RECORD and PLAY commands, you can directly specify the display to be used. When using the DTM character cell interface, specify the display using the /DISPLAY qualifier. When using the DECwindows Motif interface to DTM, use the Display field in the Record/Play window. Note that the string identifying the display should include the value ::0, for example, WS01::0.

For RUN commands, you can use the DECW$DISPLAY logical name to define the required display. This works well for the character cell interface. However, if the DECwindows Motif interface to DTM is used, defining this logical name also causes the DTM windows to be displayed on the same display, which nullifies the advantage of using a second display.

For SUBMIT commands, you can define the DECW$DISPLAY logical name in either the LOGIN.COM file or the collection or test prologue files.

A better solution for the RUN and SUBMIT commands, which works for both the character cell and DECwindows interfaces, is to define a global, logical DTM variable called DECW$DISPLAY. For example:

```
$  DTM CREATE VARIABLE/LOGICAL/GLOBAL DECW$DISPLAY WS01::0
```

This causes the logical name DECW$DISPLAY to be defined when collections are run. The default value can be overridden when a collection is created.

6.5 Using DTM on the Display Under Test

When only a single workstation is available for both the application under test and to run DTM, output from the DTM windows can affect the tests. However, there are ways to reduce the impact on the tests. The method depends, to some extent, on which interface is used—either the command line or the DECwindows Motif interface.

DTM displays informational and other messages when tests are recorded, played, and run. With the command line interface, these messages are displayed in the DECterm window in which the DTM command was entered.

With the DECwindows Motif interface, the messages are displayed in a separate window. You can choose as to whether or not to have these windows visible while the tests are running. If the windows are visible, they will appear in screen saves and will need to be masked in order for DTM to provide successful comparisons. The messages displayed in these windows during recording and any text which is redrawn when the window is exposed after being hidden, will become synchronization text for the test. Since the messages output during play or run commands will usually not match those during recording, the test will wait for the timeout period for each synchronization point, which may slow the test down considerably.

One resolution to this problem is to update the session file to remove the synchronization text that does not occur on playback. This can be done automatically by using the PLAY/AUTOSYNC command, or by checking the Auto synchronize box on the Play window. While the test is playing, the F9/F key sequence can be used to avoid waiting for the timeout period.

As an alternative, testing can be conducted with the message windows minimized. This has the disadvantage that if errors are reported, the messages are not immediately visible. In addition, it may not be apparent when tests have completed. However, these are probably outweighed by the advantage of not needing to mask the windows.

The simplest way of minimizing the message windows is to include the click on the Minimize button in the test. On systems running DTM Version 3.9 or later, the window position is fixed and under user control, as described in the *Digital Test Manager for OpenVMS Release Notes*. For systems running earlier versions, the position varied according to the tiling algorithm built into DECterm, so this method would not work.

An alternative is to pause the test, minimize the windows, and resume the test. This can be done using the key sequences F9/A to pause, and F9/B to resume.

6.6 Starting the Application Under Test

There are several ways to start the application to be tested. When DTM is running on a node different from the one on which the application windows are displayed, these methods result in the application being run on either the same node as DTM or on the system being used as the display. The application can always be made to run on the display system by submitting the test collection to a queue on the system, even if the DTM command is entered on another node.

One method is to start the application via the Session Manager Applications menu. This causes the application to run on the display system. It is also possible to start the application from a DECterm window that is running on the system.

The other methods result in the application running on the same node as the collection, or DTM, in the case of RECORD and PLAY commands.

The application can also be started by a DCL command associated with the test. This can be set with the /COMMAND qualifier to the DTM CREATE TEST or DTM MODIFY TEST commands, or the corresponding DECwindows Motif interface windows, accessed from the Maintenance menu. Note that this command is not executed when the test is recorded or played using the character cell interface and must be entered again using the /COMMAND qualifier. This is also true in the DECwindows Motif interface, although if the test is selected before choosing Record or Play from the Testing menu, the Command field will be filled in with the command automatically.

Finally, the application can be started by a DCL command in the test prologue file. However, the prologue is only executed when the test is run as part of a collection, so the application must be started externally or by using the /COMMAND qualifier for the RECORD and PLAY commands.

6.7 Using the Play/Record User Interface

Use of the Play/Record User Interface, also referred to as the Record Tool Window, is not recommended. Pointer movements, button presses and output associated with these windows will almost certainly interfere with the smooth running of tests. The windows will also appear in the screen saves during record and playback and will need masking to prevent unsuccessful screen comparisons. This applies whether a single system is used or when a second display is used for DTM.

A

CMS Callable Routine Examples

The following sections contain a series of sample C programs that demonstrate how CMS callable routines can be used.

A.1 Reserving and Replacing Elements

Example A–1 uses the CMS callable routines to reserve and replace elements in a CMS library.

Example A–1 CMS Callable Routines: RESERVE and REPLACE

```
/*
**  FACILITY:
**
**      CMS_EXAMPLE1.C
**
**  ABSTRACT:
**
**      Example program illustrating how CMS callable routines
**      can be used to reserve and replace elements in a library.
**
**      3 routines are used:  CMS$SET_LIBRARY
**                            CMS$REPLACE
**                          &  CMS$FETCH      - there's no CMS$RESERVE
**
*/
#include <descrip.h>
#include <lib$routines.h>
#include <ssdef.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int cms$set_library ();
int cms$fetch ();
int cms$replace ();

main()
{
typedef struct dsc$descriptor_d DESCRIPTOR;
#define DESC_BLD(name)      DESCRIPTOR name = {0,DSC$K_DTYPE_T,DSC$K_CLASS_D,0}
#define DESC_FIL(name,str)                                  \
        name.dsc$w_length = strlen(str);                     \
        name.dsc$a_pointer = str
```

(continued on next page)

Example A–1 (Cont.) CMS Callable Routines: RESERVE and REPLACE

```
/*
** Allocating space for the library data block (LDB).  The LDB is a user-
** allocated structure, CMS uses to maintain information about the library.
*/
int    library_data_block [50];
int    status;
int    reserve;
char   action = ' ';
char   element_list[80];
char   remark[80];

DESC_BLD(remark_desc);
DESC_BLD(element_desc);
DESC_BLD(library_desc);
DESC_FIL(library_desc,"LIB$DIR");

    /* Set to the CMS library */
    status = cms$set_library (&library_data_block, &library_desc);
    if (! (status & SS$_NORMAL))
        {
        lib$signal(status);
        return status;
        }

    do {
      /* Reserve or Replace ? */
      printf("\nG. Get a File\n");
      printf("R. Replace a File\n");
      printf("Q. Quit\n\n");

      do {
           printf("Make your selection : ");
           gets(&action);
           action = toupper(action);
      } while (action != 'G' && action != 'R' && action != 'Q');

      /* Action menu */
      switch(action) {
          case 'G':
             /* Reserve */
             printf("Enter File name(s):");
             gets(element_list);
             /* Remark */
             do {
               printf("Why [80]? ");
               gets(remark);
             } while (strlen(remark) <= 1);
```

(continued on next page)

Example A–1 (Cont.) CMS Callable Routines: RESERVE and REPLACE

```
            /*
            ** CMS$FETCH ( LIBRARY_DATA_BLOCK,  by reference
            **             ELEMENT_EXPRESSION,  by descriptor
            **             [REMARK],            by descriptor
            **             [GENERATION_EXRESSION], always get latest
            **             [MERGE_GENERATION_EXRESSION], not used
            **             [RESERVE],      1 = reserve, (by reference)
            **                ... )
            */
            DESC_FIL(remark_desc, remark);
            DESC_FIL(element_desc, element_list);
            reserve = 1;
            status = cms$fetch (&library_data_block,
                                &element_desc,
                                &remark_desc,0,0,
                                &reserve);
            if (! (status & SS$_NORMAL))
            {
              lib$signal(status);
              return status;
            }
            break;

         case 'R':
            /* Replace */
            printf("Enter File name(s):");
            gets(element_list);
            /* Remark */
            do {
              printf("Why [80]? ");
              gets(remark);
            } while (strlen(remark) <= 1);

            /*
            ** CMS$REPLACE ( LIBRARY_DATA_BLOCK,  by reference
            **               ELEMENT_EXPRESSION,  by descriptor
            **               [REMARK],            by descriptor
            **                  ... )
            */
            DESC_FIL(remark_desc, remark);
            DESC_FIL(element_desc, element_list);
            status = cms$replace (&library_data_block,
                                  &element_desc,
                                  &remark_desc);
            if (! (status & SS$_NORMAL))
            {
              lib$signal(status);
              return status;
            }
            break;

         default:
            /* Quit */
            printf("\nGood Bye");
      }
    } while (action != 'Q');

    return SS$_NORMAL;
}
```

A.2 Showing and Formatting Reservation Information

Example A–2 illustrates how the CMS callable routines can be used to customize the output of a SHOW RESERVATION command.

Example A–2 CMS Callable Routines: SHOW RESERVATION

```
/*
**  FACILITY:
**
**      CMS_EXAMPLE2.C
**
**  ABSTRACT:
**
**      Program to illustrate how the CMS callable routines can be
**      used to create a customized version of the SHOW RESERVATIONS
**      command.
**
**      2 routines are used:   CMS$SET_LIBRARY
**                             CMS$SHOW_RESERVATIONS
**
**      The include file CMS$ROUTINES.H, which defines the CMS callable
**      routine entry points and the CMS error message symbols, is created
**      from the file SYS$SYSROOT:[SYSHLP.EXAMPLES.CMS]CMS$ROUTINES.SDL.
**      The command to create the file is $SDL/LANG=CC CMS$ROUTINES.SDL.
**      The SDL processor can be obtained from the OpenVMS Freeware CD.
**
**  IMPLICIT INPUTS:
**
**      This program assumes the current CMS library has been set.
**
*/
#include <ctype.h>
#include <descrip.h>
#include <lib$routines.h>

#include <ssdef.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cms$routines.h"

#define LDB_SIZE        50

typedef struct dsc$descriptor_s DESCRIPTOR;

/*
**  strCreateFromStringID
**
**  Extract a character string from a CMS string identifier.
**
*/

char* strCreateFromStringID( DESCRIPTOR* desc )
{
  char* string;    /* string copied from desc */
```

(continued on next page)

Example A–2 (Cont.) CMS Callable Routines: SHOW RESERVATION

```
  string = malloc( desc->dsc$w_length + 1 );
  if (string == NULL) {
    printf("Fatal storage allocation failure.\n");
  }
  else {
    strncpy( string, desc->dsc$a_pointer, desc->dsc$w_length );
    string[desc->dsc$w_length] = '\0';
  }

  return string;
}

/*
**  Output Routine
**
**  This routine is called by CMS$SHOW_RESERVATIONS once for each reservation
**  in the CMS library.  The information about the reservation is passed to
**  this routine in the argument list.
**
**  Details of the parameter passing mechanism may be found in the "DIGITAL
**  Code Management System Callable Routines Reference Manual".
**
*/

static int output_routine( int* new_element,
      int* library_data_block,
      int* user_param,
      DESCRIPTOR** element_id,
      DESCRIPTOR** generation_id,
      int* time,
      DESCRIPTOR** user_id,
      DESCRIPTOR** remark_id,
      int* concurrent,
      DESCRIPTOR** merge_generation_id,
      int* nonotes,
      int* nohistory,
      int* access,
      int* reservation_id )

{
  char* element_name;                   /* element name string */
  char* generation;                     /* generation string */
  char* username;                       /* username string */
  char* remark;                         /* remark string */

  /* Extract character strings from CMS string identifiers */
  element_name = strCreateFromStringID( *element_id );
  generation = strCreateFromStringID( *generation_id );
  username = strCreateFromStringID( *user_id );
  remark = strCreateFromStringID( *remark_id );

  /*
   * Generate the output string.  The format is:
   *
   *  element-name  generation  reservation-id  username  remark
  */
```

(continued on next page)

Example A–2 (Cont.) CMS Callable Routines: SHOW RESERVATION

```
  printf( "%s  %s  %d  %s  \"%s\"\n",
   element_name,
   generation,
   *reservation_id,
   username,
   remark );

  /* Free the memory allocated for character strings */
  free( element_name );
  free( generation );
  free( username );
  free( remark );

  return( CMS$_NORMAL );       /* Successful completion */
}

int main( int argc,   /* arg count from command line */
   char* argv[] )  /* arg pointers */
{

 int   library_data_block[LDB_SIZE]; /* LDB structure */
 int   status;        /* completion status from CMS routines */

 $DESCRIPTOR( element_desc, "*.*" ); /* default element name for SHOW */
 $DESCRIPTOR( library_desc, "CMS$LIB" ); /* CMS library name */


 /* Check for element name(s) specified on the command line. */
 if (argc > 1) {   /* use supplied value, if present */
   element_desc.dsc$a_pointer = argv[1];
   element_desc.dsc$w_length = strlen(argv[1]);
 }

 /*
  * The LDB must be initialized before using the CMS callable routines.
  * This is done by calling CMS$SET_LIBRARY.  For the purposes of this
  * example, we assume the library has already been set by a previous
  * CMS SET LIBRARY command, so the logical name CMS$LIB translates to
  * the library name.
  *
  */
 status = cms$set_library( &library_data_block, &library_desc );

 if (! (status & SS$_NORMAL)) {
     switch( status )
       {
       case CMS$_NOREF:                  /* library access error */
  printf("Unable to access CMS library\n");
  return status;

       default:                          /* unexpected error */
  printf("Unexpected error returned from CMS$SET_LIBRARY\n");
  lib$signal( status );
  return status;
       }
 }

 /* Get the reservation information for the library */
 status = cms$show_reservations( &library_data_block,
     output_routine, 0,
     &element_desc,
     0,0,0,0 );
```

(continued on next page)

Example A–2 (Cont.) CMS Callable Routines: SHOW RESERVATION

```
 if (! (status & SS$_NORMAL)) {
     switch( status )
       {
       case CMS$_NORES:               /* no reservations found */
  printf("There are no reservations in the library\n");
  break;

       default:                       /* unexpected error */
  printf("Unexpected error returned from CMS$SHOW_RESERVATIONS\n");
  lib$signal( status );
  return status;
       }
 }

 return SS$_NORMAL;
}
```

A.3 Showing and Filtering Element Information

Example A–3 demonstrates how to implement a version of the SHOW ELEMENT command with customized element selection criteria. It also shows how data can be passed from the user program to the CMS callback routine using the **user_arg** parameter.

Example A–3 CMS Callable Routines: SHOW ELEMENT

```
/*
**  FACILITY:
**
**      CMS_EXAMPLE3.C
**
**  ABSTRACT:
**
**      Program to illustrate how the CMS callable routines can be
**      used to create a customized version of the SHOW ELEMENT
**      command.  The program also demonstrates how to pass information
**      from the user program to the output callback routine using the
**      user_param argument to the CMS callable routines.
**
**      This program displays information about elements in a CMS library
**      which match user specified criteria.  The criteria available are
**      whether the REFERENCE_COPY attribute is set or not and whether or
**      not a notes string is specified.
**
**      The program must be invoked as a foreign command (i.e., via a DCL
**      symbol) to pass parameters on the command line.  To do this, define
**      a symbol such as the following:
**
**              $ SHOWELE == "$dev:[dir]CMS_EXAMPLE3"
**
**      The program would then be invoked as follows:
**
**              $ SHOWELE [element-expression [attribute [attribute]]]
**
**  INPUTS:
**
**      (1) element expression - specifies the elements or groups to
```

(continued on next page)

Example A–3 (Cont.) CMS Callable Routines: SHOW ELEMENT

```
**          include in the search.  The default is *.*.
**
**      (2)-(n) element slection criteria - keywords representing the element
**              characteristics which must be present for an element to be
**              listed.  Valid values are REFERENCE (CMS reference copy
**              attribute is set), NOREFERENCE (reference copy attribute is
**              not set), NOTES (a notes string is defined), and NONOTES.
**
**              All specified attributes must be set for an element to
**              be displayed.  If no attributes are specified, all elements
**              matching the element expression will be displayed.
**
**              The selection attributes may not be abbreviated.  An attribute
**              and its negated form (e.g., NOTES and NONOTES) may not be
**              specified together.
**
**  IMPLICIT INPUTS:
**
**      Logical name CMS$LIB translates to the location of the CMS library.
**
**
**      The include file CMS$ROUTINES.H, which defines the CMS callable
**      routine entry points and the CMS error message symbols, is created
**      from the file SYS$SYSROOT:[SYSHLP.EXAMPLES.CMS]CMS$ROUTINES.SDL.
**      The command to create the file is $SDL/LANG=CC CMS$ROUTINES.SDL.
**      The SDL processor can be obtained from the OpenVMS Freeware CD.
**
**
*/
#include <ctype.h>
#include <descrip.h>
#include <lib$routines.h>
#include <ssdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cms$routines.h"

#define LDB_SIZE        50

/* element selection attributes */
struct element_select {
  unsigned REF     : 1;
  unsigned NOREF   : 1;
  unsigned NOTES   : 1;
  unsigned NONOTES : 1;
  unsigned fill    : 28;
};
typedef struct element_select ELEMENT_SELECT;
#define ELEMENT_SELECT_INIT {0,0,0,0,0}

typedef struct dsc$descriptor_s DESCRIPTOR;

/*
**  strCreateFromStringID
**
**  Extract a character string from a CMS string identifier.
**
*/

char* strCreateFromStringID( DESCRIPTOR* desc )
{
  char* string;    /* string copied from desc */
```

(continued on next page)

Example A–3 (Cont.) CMS Callable Routines: SHOW ELEMENT

```
  string = malloc( desc->dsc$w_length + 1 );
  if (string == NULL) {
    printf("Fatal storage allocation failure.\n");
  }
  else {
    strncpy( string, desc->dsc$a_pointer, desc->dsc$w_length );
    string[desc->dsc$w_length] = '\0';
  }

  return string;
}

/*
**  Output Routine
**
**  This routine is called by CMS$SHOW_ELEMENT once for each requested
**  element in the CMS library.  The information about the element is
**  passed to this routine in the argument list.
**
**  The user selection options are passed via the user_param parameter.  All
**  attributes must be present for an element to be listed.
**
**  Details of the parameter passing mechanism may be found in the "DIGITAL
**  Code Management System Callable Routines Reference Manual".
**
*/

static int output_routine( int* first_call,
      int* library_data_block,
      ELEMENT_SELECT* user_param,
      DESCRIPTOR** element_id,
      DESCRIPTOR** remark_id,
      DESCRIPTOR** history_string_id,
      DESCRIPTOR** notes_string_id,
      int* position,
      int* concurrent,
      int* reference_copy,
      DESCRIPTOR* group_list_id,
      int* review )
{
  int   notes_len;   /* length of notes string */
  char  reference_string[18];          /* string to output for reference
        state */
  char* element_name;                  /* element name string */
  char* notes;                         /* notes string */
  char* remark;                        /* remark string */

  /*
   * Check the attributes of this element which correspond to the
   * selection options.  Logically, it's simpler to check whether
   * the element should not be displayed.
   */
  notes_len = (**notes_string_id).dsc$w_length;
```

(continued on next page)

Example A–3 (Cont.) CMS Callable Routines: SHOW ELEMENT

```
  if (! (((*user_param).REF && (*reference_copy == 0)) ||
  ((*user_param).NOREF && (*reference_copy == 1)) ||
  ((*user_param).NOTES && (notes_len == 0)) ||
  ((*user_param).NONOTES && (notes_len != 0))))
    {
      /* There are no conflicting criteria, so display the element */
      if (*reference_copy == 1)
 strcpy( reference_string, "/REFERENCE_COPY" );
      else
 strcpy( reference_string, "/NOREFERENCE_COPY" );

      /* Extract character strings from CMS string identifiers */
      element_name = strCreateFromStringID( *element_id );
      notes = strCreateFromStringID( *notes_string_id );
      remark = strCreateFromStringID( *remark_id );

      /*
       * Generate the output string.  The format is:
       *
       *  element-name  reference_string  notes-string  remark
       */
      printf( "%s  %s  \"%s\"  \"%s\"\n",
       element_name,
       reference_string,
       notes,
       remark );

      /* Free the memory allocated for character strings */
      free( element_name );
      free( notes );
      free( remark );
    }

  return( CMS$_NORMAL );       /* Successful completion */
}

/*
**  MAIN() - main program entry point.
**
*/
int main( int argc,   /* arg count from command line */
   char* argv[] )  /* arg pointers */
{
 int   library_data_block[LDB_SIZE]; /* LDB structure */
 int   status;        /* completion status from CMS routines */
 ELEMENT_SELECT options = ELEMENT_SELECT_INIT;
                                          /* element selection characteristics */
 int   arg_index;

 $DESCRIPTOR( element_desc, "*.*" ); /* default element name for SHOW */
 $DESCRIPTOR( library_desc, "CMS$LIB" ); /* CMS library name */

 /* Check for arguments specified on the command line. */
 switch (argc) {
 case 1:    /* no command line args */
   /* Nothing to be done in this case.  The default element expression is set
      in the string descriptor initialization. */
   break;
```

(continued on next page)

Example A–3 (Cont.) CMS Callable Routines: SHOW ELEMENT

```
default:    /* command line args are present */
  /* get the specified element expression */
  element_desc.dsc$a_pointer = argv[1];
  element_desc.dsc$w_length = strlen(argv[1]);

  /*
   * Any remaining command line arguments are expected to be element selection
   * criteria.  Scan the remaining args and record the criteria present.
   *
   * The character strings presented to the program in argv[] have been
   * converted to lowercase unless they were quoted on the command line.
   * Because it is unlikely a user would quote these strings we check
   * only for the lowercase form of the strings.
   *
   * N.B.  strcmp() returns 0 if strings match.
   */
  for (arg_index = 2; arg_index < argc; ++arg_index) {
    if (! strcmp( argv[arg_index], "reference" ))
      options.REF = TRUE;
    else if (! strcmp( argv[arg_index], "noreference" ))
      options.NOREF = TRUE;
    else if (! strcmp( argv[arg_index], "notes" ))
      options.NOTES = TRUE;
    else if (! strcmp( argv[arg_index], "nonotes" ))
      options.NONOTES = TRUE;
    else
      printf("Invalid option \'%s\' ignored.\n", argv[arg_index]);
  }
}

/* Check for conflicting options */
if ((options.REF && options.NOREF) || (options.NOTES && options.NONOTES)) {
  printf("Conflicting options!\n");
  return SS$_NORMAL;
}

/*
 * Command line processing is complete.
 *
 * The LDB must be initialized before using the CMS callable routines.
 * This is done by calling CMS$SET_LIBRARY.  The location of the CMS
 * library is assumed to be pointed to by the logical name CMS$LIB.
 * This logical name is defined when the CMS SET LIBRARY command is
 * issued.
 *
 * It is not necessary to report any error status from the CMS callable
 * routines as these routine cause the appropriate error messages to be
 * printed before returning here.
 *
 */
status = cms$set_library( &library_data_block, &library_desc );

if (! (status & SS$_NORMAL)) {
    switch( status )
      {
      case CMS$_NOREF:                  /* library access error */
 printf("Unable to access CMS library\n");
 return SS$_NORMAL;
```

(continued on next page)

Example A–3 (Cont.) CMS Callable Routines: SHOW ELEMENT

```
      default:                          /* unexpected error */
 printf("Unexpected error returned from CMS$SET_LIBRARY\n");
 return SS$_NORMAL;
      }
}

/* Get the reservation information for the library */
status = cms$show_element( &library_data_block,
      output_routine,
      &options,
      &element_desc,
      0,0 );

if (! (status & SS$_NORMAL)) {
  printf("Unexpected error returned from CMS$SHOW_RESERVATIONS\n");
  return SS$_NORMAL;
}

return SS$_NORMAL;
}
```

B

CMS Event Handling Example

The following examples illustrate how all the reservations and replacements of an element (T.T) can be written to a file.

Example B–1 CMS_EVENT.C

```
/*
**  FACILITY:
**
**      CMS_EVENT.C
**
**  ABSTRACT:
**
**      Event handler for CMS ACTION ACE mechanism - sets 3 global symbols:
**
**          CMS$$EVENT_LIBRARY_SPEC   - library_specification_id;
**               = CMS Library Specification
**          CMS$$EVENT_PARAMETER_ID   - ace_parameter_id;
**               = the PARAMETER clause (any string value specified on the ACE)
**          CMS$$EVENT_HISTORY_RECORD - history_record_id;
**               = The history line
**
**      then calls CMS$HANDLER:CMS_ACTION.COM.
*/
#include <descrip.h>
#include <lib$routines.h>
#include <ssdef.h>
#include <stdio.h>
#include <string.h>

#define BUF_SIZE               2048
#define DEL                    127
#define LDB_SIZE               50
#define LIB$K_CLI_GLOBAL_SYM   2

typedef struct dsc$descriptor_s DESCRIPTOR;

#define DESC_BLD(name) DESCRIPTOR name = {0,DSC$K_DTYPE_T,DSC$K_CLASS_D,0}
#define DESC_FIL(name,str)                                  \
        name.dsc$w_length = strlen(str);                     \
        name.dsc$a_pointer = str
```

(continued on next page)

Example B–1 (Cont.) CMS_EVENT.C

```
/*
** CMS$EVENT_ACTION should follow the rules for callback routines.  The
** routine calling format and arguments are described in CMS Guide, section
**   8.1.3 "Using Your Own Event Handler"
*/
CMS$EVENT_ACTION(
    library_data_block,        /* LDB for the current library */
    user_param,                /* user_arg value from callable CMS routine */
    library_specification_id,  /* string id for CMS library directory spec */
    ace_parameter_id,          /* string id from PARAMETER clause of ACE */
    history_record_id)         /* string id from history record */

/*
** Allocating space for the library data block (LDB).  The LDB is a user-
** allocated structure, CMS uses to maintain information about the library.
*/
int         library_data_block [LDB_SIZE];
int         user_param;
DESCRIPTOR  **library_specification_id;
DESCRIPTOR  **ace_parameter_id;
DESCRIPTOR  **history_record_id;

{
int         i;
int         status;
char        hist_rec [BUF_SIZE];
DESCRIPTOR  hist_id;

    $DESCRIPTOR(command,"@CMS$HANDLER:CMS_ACTION.COM");
    $DESCRIPTOR(library_symb,"CMS$$EVENT_LIBRARY_SPEC");
    $DESCRIPTOR(history_symb,"CMS$$EVENT_HISTORY_RECORD");
    $DESCRIPTOR(param_symb,"CMS$$EVENT_PARAMETER_ID");

    DESC_BLD(history_desc);

    hist_id = **history_record_id;

    /* Get history record specification */
    strncpy ( hist_rec, hist_id.dsc$a_pointer, hist_id.dsc$w_length);

    /* Change embedded 'DEL' characters to spaces */
    for (i=hist_id.dsc$w_length; i>=0; i--)
        {
        if (hist_rec[i] == DEL)
            {
             hist_rec[i] = ' ';
            }
        }

    DESC_FIL(history_desc, hist_rec);

    /* Set the symbols */
    status = lib$set_symbol(&param_symb, *ace_parameter_id,
        &LIB$K_CLI_GLOBAL_SYM);
    if (! (status & SS$_NORMAL))
        {
        lib$signal(status);
        return status;
        }
```

(continued on next page)

Example B–1 (Cont.) CMS_EVENT.C

```
    status = lib$set_symbol(&library_symb, *library_specification_id,
        &LIB$K_CLI_GLOBAL_SYM);
    if (! (status & SS$_NORMAL))
        {
        lib$signal(status);
        return status;
        }

    status = lib$set_symbol(&history_symb, &history_desc,
        &LIB$K_CLI_GLOBAL_SYM);
    if (! (status & SS$_NORMAL))
        {
        lib$signal(status);
        return status;
        }

    /* Call the command procedure */
    status = lib$spawn (&command);
    if (! (status & SS$_NORMAL))
        {
        lib$signal(status);
        return status;
        }

    /* Delete the symbols */
    lib$delete_symbol(&library_symb,&LIB$K_CLI_GLOBAL_SYM);
    lib$delete_symbol(&history_symb,&LIB$K_CLI_GLOBAL_SYM);
    lib$delete_symbol(&param_symb,&LIB$K_CLI_GLOBAL_SYM);

    return SS$_NORMAL;
}
```

Example B–2 CMS_ACTION.COM

```
$ VF = F$VERIFY(0)
$!
$! CMS_ACTION.COM
$!
$! Command procedure invoked by a CMS action routine.  Captures the library
$! specification, history record, and ACE parameter in a file.
$!
$! Inputs : 3 symbols: CMS$$EVENT_LIBRARY_SPEC
$!                     CMS$$EVENT_HISTORY_RECORD
$!                   & CMS$$EVENT_PARARMETER_ID (string)
$!
$! Output File : EVENT_SEND.LIS
$!
$ DELETE := DELETE
$ OPEN   := OPEN
$ WRITE  := WRITE
$ CLOSE  := CLOSE
$!
$ OPEN/WRITE tmp event_send.lis
$ WRITE tmp ""
$ WRITE tmp "An interesting event has occurred:"
$ WRITE tmp ""
$!
$ WRITE tmp cms$$event_library_spec
$ WRITE tmp ""
$!
$ IF F$TYPE (cms$$event_history_record) .NES. ""
$ THEN
$    WRITE tmp cms$$event_history_record
$ ELSE
$    ! No history - get name from the process's username
$    who = F$EDIT (F$GETJPI(""USERNAME"),"TRIM")
$    WRITE tmp "(No history record for action by ''who' at ''F$TIME()')"
$ ENDIF
$!
$ WRITE tmp cms$$event_parameter_id
$ WRITE tmp ""
$!
$ CLOSE tmp
$ TYPE event_send.lis
$ EXIT
```