HP DECset for OpenVMS

Using HP DECset for OpenVMS Systems

Order Number: AA–PQ9ND–TE

July 2005

This guide describes how to use the Software Engineering Tools (HP DECset) with other OpenVMS facilities to create an effective software development environment on OpenVMS systems.

Revision/Update Information:	This is a revised manual.
Operating System Version:	OpenVMS I64 Version 8.2
	OpenVMS Alpha Version 7.3–2 or 8.2
	OpenVMS VAX Version 7.3
Windowing System Version:	DECwindows Motif for OpenVMS I64 Version 1.5
	DECwindows Motif for OpenVMS Alpha Version 1.3–1 or 1.5
	DECwindows Motif for OpenVMS VAX Version 1.2–6
Software Version:	HP DECset Version 12.7 for OpenVMS

Hewlett-Packard Company
Palo Alto, California

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Tables

Preface

This guide has four objectives:

- To describe the OpenVMS programming environment, with special attention to using the HP software engineering tools (HP DECset) to extend programmer productivity in this environment
- To show HP DECset in the HP DECwindows Motif environment
- To provide helpful hints to the project leader who is managing a project using HP DECset, with special attention to organizing project files, libraries, and directories
- To show a case study of HP DECset tools being used to accomplish specific tasks that make up the major steps in the software life cycle

Intended Audience

This guide is intended for programmers, software engineers, and project managers using one or more of the HP DECset tools. The users should be familiar with the OpenVMS operating system, OpenVMS program development facilities, and OpenVMS utilities.

Document Structure

This guide consists of two parts. The topics covered in each part are as follows:

- Part I, HP DECset and Managing the Project Environment, consists of four chapters that describe HP DECset and show how you can use it to set up and manage a software project. Specifically, Part I covers the following topics:
  - The six tools that make up the HP DECset package.
  - A description of the HP DECset environment manger and how to use this interface with other HP DECset tools.
  - HP DECset in the OpenVMS programming environment with a focus on integration features.

  - Additional OpenVMS tools that you can use to extend the programming environment.
  - Suggestions on how to create a project directory area. Special emphasis is given on organizing subdirectories for the project, including CMS, HP DIGITAL Test Manager, and SCA libraries.
  - Topics of interest to a project leader who is using HP DECset.
- Part II, HP DECset Case Study, consists of eight chapters that provide specific procedures and examples for using the HP DECset tools at each major stage of the software life cycle. Part II describes the steps that a hypothetical project team takes to plan, implement, build, and test a particular application, showing how this team uses the HP DECset tools during the entire project. All examples, where possible, show each tool in the HP DECwindows environment.

At the end of this book is a glossary of HP DECset terms.

References to Other Products

Some older products that HP DECset components previously worked with might no longer be available or supported by HP. Any reference in this manual to such products does not imply actual support, or that recent interoperability testing has been conducted with these products.

Note

These references serve only to provide examples to those who continue to use these products with HP DECset.

Refer to the Software Product Description for a current list of the products that the HP DECset components are warranted to interact with and support.

Associated Documents

The following documents might be helpful when using HP DECset:

- OpenVMS documentation set
- *OpenVMS DECwindows User's Guide*
- *HP DECSet for OpenVMS Language-Sensitive Editor*
- *Guide to DIGITAL Source Code Analyzer for OpenVMS Systems*
- *HP DECSet for OpenVMS Code Management System*

- *HP DECset for OpenVMS Module Management System*
- *HP DECset for OpenVMS Guide to DIGITAL Test Manager*
- *Guide to DIGITAL Performance and Coverage Analyzer for OpenVMS Systems*
- *HP DECSet for OpenVMS Installation Guide*
- *Guide to Detailed Program Design for OpenVMS Systems*
- *CMS Client for Windows User's Guide*—Component of the product "HP DECset Clients for CMS and MMS"

Conventions

Table 1 lists the conventions used in this guide.

Table 1 Conventions Used in this Guide

Convention	Description
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
$\boxed{\text{Return}}$	In interactive examples, a label enclosed in a box indicates that you press a key on the terminal, for example, $\boxed{\text{Return}}$.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
KP*n*	The phrase KP*n* indicates that you must press the key labeled with the number or character *n* on the numeric keypad, for example, KP3 or KP-.
file-spec, ...	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates additional parameters, options, or values you can enter.
. . .	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being described.

(continued on next page)

Table 1 (Cont.) Conventions Used in this Guide

Convention	Description
( )	In format descriptions, if you choose more than one option, parentheses indicate that you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term, and HP DECwindows menu items in description lists.
`monospace boldface text`	Boldface, monospace text represents user input in interactive examples in this guide.
UPPERCASE	Uppercase text indicates the name of a command, routine, macro, directory, or file.
italic text	Italic text represents book titles, parameters, arguments, variables, and information that can vary in system messages (for example, Internal error *number*).
lowercase	Lowercase in examples indicates that you are to substitute a word or value of your choice.
mouse	The term **mouse** refers to any pointing device, such as a mouse, puck, or stylus.
MB1,MB2,MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button.

Part I

HP DECset and Managing the Project Environment

Part I of this guide gives an overview of HP DECset and some suggested guidelines for organizing and managing a project using HP DECset tools.

1
HP DECset Overview

This chapter presents an overview of the HP software engineering tools (HP DECset) and how they fit into the OpenVMS programming environment to give you a cohesive, tightly integrated software development environment. At the end of this chapter is a summary of other OpenVMS tools and utilities that you can use to further build upon the OpenVMS environment.

1.1 HP DECset Product Set

HP DECset is a group of the following layered products:

- HP Code Management System (CMS)
- HP Language-Sensitive Editor (LSE)
- HP Source Code Analyzer (SCA)
- HP Module Management System (MMS)
- HP DIGITAL Test Manager
- HP Performance and Coverage Analyzer (PCA)

HP DECset also includes the HP DECset Environment Manager, which provides a single mechanism for tailoring the execution environ ment across the HP DECset tools. In addition to the tools, HP DECset provides the program design facility, a set of features in LSE, SCA, and the compilers that aid in the detailed program design phase of software development. HP DECset Clients for CMS and MMS—a separately orderable product—allows access from a PC to CMS libraries residing on OpenVMS systems, and allows users to run MMS on OpenVMS systems from a PC.

Individually, each tool is a useful product that can enhance programmer productivity. Used together, these tools combine with the program design facility, the OpenVMS programming languages, system services, and utilities to make an integrated environment with the following characteristics:

- Support for the multiple phases of the software life cycle

- Support for applications written in multiple languages
- Compilers and tools that pass substantial information among themselves to automate tasks previously performed manually

The following sections provide a brief overview of each of the six products that make up HP DECset.

1.1.1 HP Code Management System

The HP Code Management System (CMS) provides an efficient method for storing project files and tracking all changes to those files. Code management is especially important on large projects with long lifespans and with several versions of the software. CMS is available in HP DECwindows, character-cell, and a PC interface using the HP DECset Clients for CMS and MMS product.

CMS stores any kind of valid RMS format file, including files created by an editor, a compiler, or a linker. You can use CMS to efficiently store documents, plans, specifications, status reports, object files, executable images, sixel files, or other records. (CMS cannot store directory files.) It is a tool that all team members can use—managers, system analysts, technical writers, and programmers.

1.1.1.1 Features of CMS

You can use CMS to do the following:

- Keep track of files at every stage of development by showing who made changes, when, and why.
- Monitor changes in files to avoid conflict.
- Allow multiple team members to work concurrently on the same file without losing the changes made by any team member, while reporting any conflicts.
- Conserve disk space as it stores consecutive versions of your files in a space-efficient manner.
- Maintain a history of library activity.
- Store files created by other software development tools, such as HP DIGITAL Test Manager, by means of the CMS callable interface. Using the callable interface, other tools can integrate directly with CMS.
- Initiate an action when a specific CMS event occurs; for example, to send mail notification to project members when a specific CMS object is accessed.
- Mark an element generation for review to indicate its contents should be reviewed by other users.

- Automatically copy the latest generation of a library element into a reference copy area.

1.1.1.2 CMS Libraries

CMS keeps your files in project **libraries**, which are OpenVMS directories. These directories store your project's files, or **elements**, as well as history information. As a project evolves, CMS tracks changes to the library by storing only the changes made to a file with each reservation and replacement. Not only does this reduce the amount of disk space used for storing multiple versions of files, but it allows CMS to reconstruct any previous version of a file and to identify the changes made between any two versions, or **generations**.

In addition to storing successive changes, CMS maintains a record of who is currently working on a library element and a historical record of library access. Team members can retrieve information about library transactions and contents. A project leader can restrict access to the library or individual elements by using security features such as access control lists (ACLs), user identification codes (UICs), and rights identifiers. See Chapter 5 for examples of how to use these features.

The CMS library provides a record of the following:

- Transactions that created specific element generations
- Transactions related to the evolution of a specific element
- The entire transaction history of the library; that is, all actions which create, delete, or modify the library or its contents

1.1.1.3 Groups and Classes

CMS can create both functional and time-phased collections of elements.

Groups are functional collections of elements in the CMS library that are combined for easy handling. For instance, you can make a group of all the project documents. With one command, you can reserve all of the documents at once from the library. Groups enable you to easily manipulate large numbers of related elements.

Classes are time-phased collections of elements that represent a current or past state of the application. A class contains one generation of each element that makes up the application. At HP, a common use of this feature is to specify a base level or version of the software system. This base level or version represents a major stage in the system, such as a field test version or a version ready for customer release.

1.1.2 HP Language-Sensitive Editor

The HP Language-Sensitive Editor (LSE) is a multilanguage, programmable editor designed to help develop and maintain source code. LSE is layered on top of the HP Text Processing Utility (HP DECTPU), and is available with the EVE and EDT interfaces. LSE provides language-specific templates for each language it supports. These templates help both the novice and experienced programmer build syntactically correct programs faster and with fewer errors.

LSE provides the following features:

- Syntax support for each of the following languages and products:

  HP Ada
  VAX ACMS
  HP BASIC
  VAX BLISS-32
  HP C
  HP C++
  Oracle CDD/Repository
  HP COBOL
  HP VAX COBOL
  HP DATATRIEVE
  HP DIBOL
  HP DECdocument
  HP Fortran
  LSE
  VAX MACRO
  HP DEC Pascal
  PLSE
  VAX PL/I
  VAX Rdb/ELN
  HP DECwindows UIL

- Language-specific source code templates to quickly and efficiently enter source code.
- Compiling, reviewing, and correcting compile-time errors within a single editing session.
- Interactive editing capabilities during a debugging session.
- Support for the program design facility, enabling you to perform the following tasks:
  - Enter structured text, or pseudocode, in source files.

  - View top-level design information of source files; this is known as code elision, holophrasting, or outlining.
  - Capture design information in comments when you replace the pseudocode with actual code.
- Ability to modify existing language environments, or define a new language environment.
- Ability to customize your editing environment to your programming style or preference.
- Ability to extend your editing environment to handle highly specialized editing needs through HP DECTPU.
- Integrated access to the cross-referencing features of SCA.
- Support for a package facility to define your own subroutine call templates. LSE packages allow you to specify a subroutine and its calling sequence once, and have **tokens** and **placeholders** for the subroutine and its parameters available for use by multiple LSE language environments.
- Source code templates for calls to OpenVMS system services and the OpenVMS Run-Time Library.
- Support for user-written diagnostic files, enhancing support for user-modified or non-supported compilers.

1.1.3 HP Source Code Analyzer

The HP Source Code Analyzer (SCA) is a multilanguage, multimodule, interactive cross-reference and static analysis tool. It can help you to understand the complexities of a large software project by enabling you to make inquiries about the symbols used in the project's code. With SCA, you can move through all your project's sources, quickly locating the definitions of any identifier or any references made to that identifier.

SCA provides the following features:

- Navigation capabilities that let you locate and view the components of your source code. SCA accomplishes this by storing compiler-generated information about a set of source files in an SCA library. SCA then enables you to perform queries about your source code in several ways:
  - By using a name browser to quickly locate all items that match a search string
  - By specifying a cross-reference query to find how and where program symbols are used

  - By specifying a call graph query to graphically display call relationships between routines
  - By specifying a data structure query to graphically display the structure of data types in your code, or find symbols of a given type

  After you have a query result, you can use the go-to-source feature to navigate to locations of interest in your source code.

- Support for the following languages:

  HP Ada
  HP BASIC
  VAX BLISS-32
  HP C
  HP C++
  HP COBOL
  HP VAX COBOL
  HP Fortran
  VAX MACRO
  HP DEC Pascal
  VAXELN Pascal
  VAX PL/I

- A graphical user interface that makes it easy to specify complex queries without learning a specialized query language. It is easy to navigate between call graph queries, data structure queries, and cross-reference queries. The context-sensitive help facility enables you to display help information on all parts of the user interface.
- Library creation and maintenance features in which SCA merges analysis data (.ANA) files generated by supporting compilers into SCA libraries to create a picture of your entire source code. These files contain a collection of information relating to all the program symbols, modules, and files contained in your source files. Once you open an SCA library for a particular software project, you can use the SCA navigational and static analysis features. You can also open a personal library, containing information on only those modules that you are working on, and use this library with the main library describing the rest of the system.
- A sample library that you can open from the SCA Main window. The library is in SCA$ROOT:[EXAMPLE].
- Static analysis capabilities that let you check for consistent use of program symbols. This capability is provided by the INSPECT command:
  - The INSPECT command provides different types of consistency checking; for example, symbols implicitly declared, unused symbols,

symbols written but never read, symbols read but never written, or consistent argument types among routine calls.

- The results produced by the INSPECT command are reported in the form of a query with diagnostic error messages.
- To extend its ability to produce specific results, INSPECT allows you to tailor checks.

- Support for the program design facility, allowing you to perform the following tasks:
  - Extract design information into formal documentation.
  - Locate design information through the use of tagged comments.
  - Cross-index modules using keyword tags.

1.1.4 HP Module Management System

The HP Module Management System (MMS) automates and simplifies the building of software applications, whether they are simple programs of only one or two files or complex programs consisting of many source files, message files, and documentation. MMS can optimize the build process by rebuilding only those components of a system that have changed since the system was last built. In this way, MMS eliminates the steps of recompiling and linking modules that have not changed. Once set up, MMS can build both small and large systems with one command.

MMS provides the following features:

- Increases the speed of building a system because it builds only the parts that need building
- Increases the accuracy of the build because MMS consistently reproduces the same system each time you build it

When you initially use MMS to build your application, you perform two steps:

1. Create a **description** file.
2. Invoke MMS to carry out the build.

The description file is an ASCII text file that contains rules describing the relationships among the components of your application and the MMS commands to build the application. Once you create a description file, you can use it every time you invoke MMS to build your system. In addition, this description file keeps all the application’s structural information in one place, giving a clear representation of the application, while at the same time making the build procedure available to all team members.

1.1.5 HP DIGITAL Test Manager

The HP DIGITAL Test Manager organizes software tests and automates the way you run tests and evaluate test results. HP DIGITAL Test Manager automates regression testing, a procedure in which you run established software tests and compare the current test results with previously established benchmark results. These benchmark results, retained from previous testing or created otherwise, must be duplicated if the software is functioning properly. If the current results do not agree with the benchmark results, the modified software might contain errors. If this is the case, the software has regressed from previously established behavior. HP DIGITAL Test Manager enables you to quickly discover any regressive developments in your software. HP DIGITAL Test Manager provides testing support for multiple testing environments, including the noninteractive environment (command file), interactive terminal environment, and HP DECwindows environment.

HP DIGITAL Test Manager provides the following features:

- Creates descriptions of software tests.
- Captures terminal and HP DECwindows sessions to be used as test scripts.
- Groups these test descriptions into meaningful combinations for later runs.
- Tests interactive applications in batch mode. This includes applications that normally need to interact with a terminal or workstation; for example, an editor or a menu system.
- Executes specific tests, groups of tests, and combinations of test groups, either interactively or in batch mode.
- Sets up the test environment so tests are executed under controlled conditions.
- Compares the results of each executed test with its benchmark test results to determine differences.
- Lets you examine test result files interactively.
- Generates summary reports of test set runs.

1.1.6 HP Performance and Coverage Analyzer

The HP Performance and Coverage Analyzer (PCA) helps you analyze the run-time behavior of your application. PCA serves two functions, as follows:

- Pinpoints execution bottlenecks, then enables you to determine the cause of them.
- Analyzes test coverage by measuring what parts of an application are executed by a given set of test data. Using this information, you can create tests that thoroughly exercise your application.

PCA consists of two facilities: the Collector and Analyzer. When you link the Collector with your application, the Collector gathers and deposits data into a file during execution. After execution, you can invoke the Analyzer to interactively analyze the data stored in that file.

The following sections describe the features of PCA in three categories: Collector features, Analyzer features, and additional features.

1.1.6.1 Collector Features

The Collector gathers the following kinds of data:

- Elapsed time information—Provides a good overview of where your program consumes the most elapsed time
- Process time information—Tells where your program spends the most process, or CPU, time
- Page fault data—Helps you to determine what sections of the program cause the most page faults
- System services data—Tells you which sections of the program call system services
- Input/Output data—Details all OpenVMS Record Management Services (RMS) calls in your program, helping you to understand your program's input and output behavior
- Exact execution counts—Tells you the exact number of times your program executes specified locations, thereby helping you to find the inefficient algorithms (for example, the $O(n^2)$ algorithms)
- Test coverage data—Shows you which sections of code are or are not executed when you run tests of your program
- Tasking data—Shows all context switches in HP Ada multitasking applications

In addition, PCA allows you to selectively state which of these measurements should also include the current set of return addresses on the stack (except for page faults and tasking data). This allows you to determine relationships among the called subroutines.

1.1.6.2 Analyzer Features

The Analyzer reads the performance data file written by the Collector and uses the data to produce the following types of symbolic reductions that help you to evaluate your program's performance or coverage:

- Histograms and tables—The Analyzer enables you to produce performance histograms that plot the distribution of resource usage over your program or over other data domains. If you prefer, the Analyzer will produce tables that present the same information in the form of actual data counts instead of scaled histogram bars.
- Annotated source file listings—The Analyzer displays high-level language program source code next to the requested performance or coverage data on a line-by-line basis.
- Call trees—The Analyzer enables you to perform specific call stack analysis, from which you can get a call tree plot, displaying the dynamic call stack relationship of program units by name. This permits you to pinpoint precisely the set of subroutine calls that is consuming most of the time. This is useful for programs that utilize commonly called, time-consuming subroutines.

1.1.6.3 Additional PCA Features

Additional features include the following:

- Traversing—Once the Analyzer has tabulated the performance and coverage data by means of histograms or tables, PCA enables you to traverse through the data by using traverse commands. Using traverse commands, you can sift through your performance data, directing the Analyzer from one area of high activity to the next.
- Screen mode—If you are viewing your performance data on a terminal that supports multiple windows, the Analyzer enables you to display different types of data in separate windows.
- Multiple data kinds—The Analyzer enables you to display different categories of performance data in the same histogram or table. For example, you can display program counter (PC) sampling data, page fault addresses, and I/O service calls in the histogram. This enables you to correlate each bottleneck with its cause (for example, I/O service call, page fault, CPU consumption, and so on).

- Acceptable noncoverage—If you have portions of code that you do not expect to be tested—for example, internal error paths or difficult-to-test sections—you can indicate to the Analyzer that those portions are acceptably noncovered. On iterative test runs of your code, the Analyzer keeps track of those portions so you can ignore them in future coverage analysis.
- Filtering—The Analyzer enables you to filter performance or coverage data before it creates histograms or tables. This feature is useful when you want only a certain subset of that data to be analyzed; for example, data associated with a particular event.

1.1.7 Program Design Facility

HP DECset provides for incremental program design and development by means of the program design facility, allowing you to write structured text, or **pseudocode**, in source files. You use this pseudocode to describe the design of the code before you write the actual program statements. You can then process source files containing pseudocode using any of the following supported OpenVMS compilers:

HP Ada
HP BASIC
VAX BLISS-32
HP C
HP COBOL
HP VAX COBOL
HP Fortran
HP DEC Pascal
VAX PL/I

Unlike other HP DECset tools, the program design facility is not an individual tool that you can run independently. Rather, you access it through supported language compilers and the following HP DECset tools:

- LSE—To enter the pseudocode into your source programs to express design information, and to interactively display designs at various levels of detail
- Compilers—To compile the design information and create analysis data files
- SCA—To provide static analysis and cross-reference information on the source files containing pseudocode, and to generate a variety of reports, including online help

In addition, the program design facility enables you to capture the design information in comments, and provides various reports such as design cross-reference, help files, and LSE package reports.

1.2 HP DECset and the OpenVMS Programming Environment

The OpenVMS programming environment supports many programming languages and a large set of utilities and services. By combining a set of programmer productivity tools with this variety of languages and utilities, you extend your productivity beyond the usual range of operating system services and program development utilities and languages. HP DECset tools also help you plan, write, compile, debug, test, and build applications and systems by addressing multiple phases of the software development life cycle.

The remaining sections of this chapter describe the following primary areas in which HP DECset is integrated with the OpenVMS programming environment:

- The stages of the software life cycle in which the HP DECset tools are used
- The programming languages supported in the OpenVMS programming environment
- The integration features of the individual HP DECset tools: both with the compilers and among the tools themselves

1.2.1 Using HP DECset to Manage the Software Life Cycle

The following sections list and describe the typical stages of the software life cycle and show which stages of the software life cycle each HP DECset tool is specifically designed to manage.

The software life cycle typically includes the following five phases:

1. Requirements and specifications
2. Design
3. Implementation
4. Testing
5. Maintenance

This section describes each of these steps in detail.

Phase 1: Requirements and Specifications

This first phase in the software development life cycle defines the project in two distinct steps: the requirements step and the specifications step. During the requirements step, a project team of technical and management people identify business opportunities, product objectives, and technical options. They might also perform an analysis of the costs versus benefits for the application. During the specifications step that follows, the project team formulates detailed specifications that define what the system will do and how it will be used.

Defining the product requirements depends heavily on feedback from outside the team—from customers, internal users, and marketing representatives. Other groups within an organization (a central quality group and engineering departments) also provide information. The requirements define project goals: that is, what is to be built. They do not define how to build it. Documents typically produced during the requirements step include a requirements document and a business plan.

During the specifications step, the project team maps out technical approaches for building the new application. Often, the team produces prototypes to help solve difficult technical problems. These prototypes can be tested, providing references for future development work. These prototypes, in turn, help ensure that the team understands the risks involved with implementing the product. Prototyping can also help develop usability requirements, if the team subjects the prototypes to human engineering testing. By comparing the specifications to the requirements, the project team members can show that if the system is built as specified, it will meet its requirements. At the end of this stage, the project team has defined the application, produced a preliminary functional specification, and must decide whether to go forward with the project.

Phase 2: Design

During the design phase, the project team determines in more detail what they must build and how to build it. The first step is to write the final specifications for development and documentation.

Based on the functional specifications (and, optionally, usability requirements), the project team completes top-level design for all forms, data structures, program modules, file formats, and human interfaces. Once complete, the design technically defines the project.

To document the design, the team generates a design specification and test plan to serve as a basis for acceptance by strategic and technical planners. It is important to have the design standards and policies documented in this specification. The team estimates the resources needed to implement a particular piece of functionality. As project designs evolve, the team can modify the design specification to include changes. This design document makes it possible to keep the design plans in one location, which are accessible to all programmers. Once the design is complete, it is possible to develop a schedule as well, which is included in the design document and applies to each deliverable item in the project.

As part of the design process, the team should also refine its procedural strategy to aid in completing the project. By the end of this phase, the team should have established storage areas (libraries and a basic directory structure) and implementation standards.

Phase 3: Implementation

During the implementation phase, the team builds and modifies source code modules, then compiles, links, and executes the resulting images. Often the team implements the system in a series of stages or base levels. Each base level adds more of the required functionality. Along with the software, the team must generate user documentation, which remains up-to-date with the ongoing changes in the software's features.

At the end of this phase, the project team arranges for master copies of the user documentation and the software to be handed over to a production group that copies and distributes the product to customers. A final task is to archive copies of the product.

Phase 4: Testing

During the testing phase (which usually runs parallel with the implementation phase), the project team tests the software to make sure that it conforms to the initial requirements. The team then refines the application's code to optimize its performance. To aid in this refinement stage, the application might be made available to selected customer sites. The project team stays in close contact with these test sites to ensure that any problems are corrected in the version of the software or user documentation shipped to customers. In the final stages of this phase, the source code and documentation are frozen, and the team prepares final copies of the documentation and distribution media.

Phase 5: Maintenance

After the product has been shipped, a process of maintenance and evolution begins. If errors exist in the software or documentation, the maintenance team makes the necessary changes. Enhancements might be planned. At the same time, suggestions for new requirements arrive from customers. This phase becomes an information-gathering activity that can begin the early phases for the next version of the software.

1.2.2 Programming Languages

Programming languages are central to the OpenVMS software development environment. Applications and systems programmers have a diversified range of higher-level programming languages at their disposal, including the following:

HP Ada
HP BASIC
VAX BLISS-32
HP C
HP C++
Oracle CDD/Repository
HP COBOL
HP VAX COBOL
HP DATATRIEVE
VAX DIBOL
HP Fortran
VAX MACRO
HP DEC Pascal
VAX PL/I

With this range of supported languages, you are free to choose the language best suited to your needs. Additionally, because most languages support the OpenVMS Calling Standard, you can combine modules written in different languages into the same program.

1.2.3 Integration Features of the HP DECset Tools

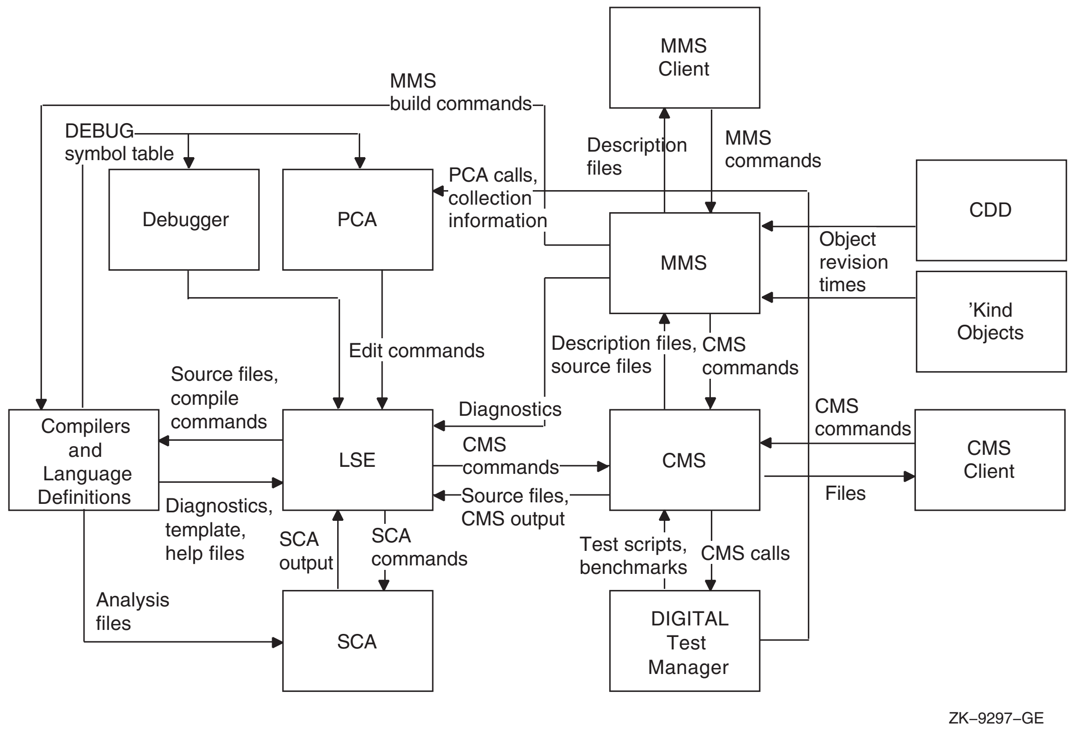
Integration features of the HP DECset tools exist on three levels:

- With the language compilers
- With the multilanguage programming environment
- Among the tools themselves

The primary integration is in the information flow among the tools and compilers. The compilers generate a substantial amount of information for tools, such as symbol table information for PCA, diagnostic information for LSE, and cross-reference and calling-sequence information for SCA. The compilers are the sole sources of semantic program information, and they make that information available in suitable forms to all tools that need it.

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

Figure 1–1 DECset Tools Integration



1.2.3.1 CMS Integration

You can access CMS elements directly from within the HP Language-Sensitive Editor (LSE). LSE commands then let you manipulate CMS elements from the LSE command line.

Because CMS can store any file as an element, it is particularly useful as a central repository, not only for source files for code and documentation, but also for a variety of files generated by other OpenVMS tools. CMS can store description files for the HP Module Management System (MMS), and test files (prologue, template, epilogue) for the HP DIGITAL Test Manager as well as results description files (benchmarks). These tools can also access elements in CMS libraries automatically. Also, as developers modify the files, CMS can track the changes. CMS has a callable interface and can be customized to assist in collecting project metrics.

1.2.3.2 LSE Integration

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.

LSE can move among languages in different files. LSE determines the language you are using by the file type, thereby providing the proper interface for the appropriate compiler and the language-specific templates and online help.

LSE is integrated with CMS to provide source code management. From LSE, you can enter commands that direct how you want LSE to obtain a file from CMS. For example, you can obtain a read-only copy of a generation with the OPEN FILE command, which instructs CMS to perform a Fetch operation. You can also use the RESERVE command to reserve a generation of a CMS element and have that generation placed in your default directory. See Section 6.2.2 for more details and an example of LSE integration with CMS.

LSE is integrated with the HP Source Code Analyzer (see Section 1.1.3 for more details). You can also invoke LSE from the Analyzer portion of the HP Performance and Coverage Analyzer (see Section 1.1.6 for more details), from OpenVMS Mail. Chapter 7 provides additional examples of LSE's integration with other OpenVMS tools.

1.2.3.3 LSE/SCA Integration

Using LSE and SCA together creates an extremely powerful, integrated editing environment. Instead of relying on memory, cross-reference listings, or guess work to locate items, you can access your entire system quickly from LSE. You can browse through all of your code to look for specific declarations of symbols or other pertinent information without regard to file location, saving you considerable time.

LSE and SCA are part of the multilanguage environment on the OpenVMS operating system. Your source code can be written in more than one language; LSE always provides you with the right language support for the file you are editing. Similarly, SCA lets you navigate through your entire project regardless of the languages used in each module.

If you have SCA and LSE running simultaneously in separate windows, LSE can display the source code associated with symbols displayed by SCA. Because these capabilities are available in LSE and SCA, you save considerable time when finding particular symbols, retrieving the related files from a reference copy area or a CMS library, and then editing the files.

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

1.2.3.4 MMS Integration

You can access elements in CMS libraries during your build. All files used for the build, documentation, and the MMS description file can be kept in a CMS library. All these files can be updated, including the description file, so MMS works with the latest sources your team has produced. Alternatively, your MMS description file can build from CMS classes that represent previous versions of your system. Additionally, MMS can access records stored in the Common Data Dictionary (Oracle CDD/Repository), or forms stored in libraries for HP DEC or VAX Forms Management System (FMS) or HP DECforms.

MMS also provides support for SCA, in that analysis data can be automatically generated as part of the MMS build procedure for storage in an SCA library.

1.2.3.5 HP DIGITAL Test Manager Integration

You can use the storage and update capabilities of CMS for HP DIGITAL Test Manager's templates, benchmark files, test data files, prologue files, and epilogue files. When stored in a CMS library, these files can be retrieved by HP DIGITAL Test Manager to run specified versions of tests. 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. CMS enables you to easily store and access the tests that correspond to older versions of the system still being maintained.

By using HP DIGITAL Test Manager with the HP Performance and Coverage Analyzer, you can measure the performance or coverage of tests run under DIGITAL Test Manager's control. Using an MMS description file to build your application, you can also execute your tests automatically. Examples of these types of integration are detailed in Chapter 9 and Chapter 11.

1.2.3.6 PCA Integration

When used with HP DIGITAL Test Manager, PCA can evaluate 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 the LSE features, such as links to SCA and CMS.

1.3 Additional OpenVMS Tools and Utilities

In addition to the HP DECset tools, the OpenVMS environment and other layered products are also important components of an integrated software development environment.

The OpenVMS Mail Utility (MAIL) aids communication within your development group and among your group and other groups by providing a means to send mail electronically. This feature of the OpenVMS environment should be part of any development team's effort to maximize the quality and timeliness of their project.

Related software includes optional capabilities for information management, data communications and networking, program migration, cross development, and many other capabilities. You can incorporate features of related software products into your application programs. Descriptions of some of these products are as follows:

- **The Common Data Dictionary (Oracle CDD/Repository)**—The Common Data Dictionary (Oracle CDD/Repository) can act as the central repository for data descriptions and definitions used by various languages (including fourth-generation languages), databases, and tools.

  Oracle CDD/Repository provides an efficient way to help manage and control definitions across the modules that make up an application. Early planning for its use can simplify management tasks.
- **DEC Notes**—DEC Notes is a computer conferencing system that lets you conduct online conferences or meetings. Using DEC Notes, you can communicate conveniently and economically. A development team can take advantage of DEC Notes to discuss and exchange information on product design and development issues, particularly with people geographically separated from the primary team.
- **DECplan**—DECplan is a tool that automates project management activities throughout the software development life cycle. You can use DECplan to plan, control, and estimate software projects. Using DECplan, you can record and modify project information, track project costs and resources, and produce project schedules, charts, and reports quickly.
- **DEC Visual User Interface Tool**—DEC Visual User Interface Tool (VUIT) is an interactive design tool (IDT) for building and testing Motif user interfaces. VUIT addresses the problems that contribute to the high expense of producing Motif applications. VUIT provides continuous visual feedback as user interface construction progresses, immediately reflecting the effect of design decisions. Programmers can position and label components, build forms, create icons, and apply colors and fonts

in real time. The ability to create or modify a user interface by directly manipulating its graphical elements drastically cuts development time for the graphical user interface.

With VUIT, programmers can produce a working prototype in a fraction of the time it would take to write such a program by hand. Its simulation and code generation capabilities enable programmers to profitably involve end users in the design of the interface early in the product lifecycle, resulting in greater satisfaction with the finished application.

- **DECdesign**—DECdesign is a tool to automate and facilitate systems analysis and design methods. DECdesign completes and validates your work by expanding implicit designs into explicit objects. For example, DECdesign automatically creates and names objects, places entries into data dictionaries, and links related objects together whenever your work implies the need for these actions. DECdesign builds on industry-wide trends and takes advantage of the HP clustered hardware environment and strategic directions.
- **DECdocument**—DECdocument is a system for producing technical documentation and is designed to fully automate the creation of typeset-quality documentation from generically coded input files. Using DECdocument, a project team can write and maintain files for a document, produce typeset-quality output for a wide range of output devices, and correct errors and incorporate changes into source files using a text editor. DECdocument supports a wide variety of **document types**, ranging from interoffice memos to software reference manuals, which ensure consistency in format, typeset design, and organization. DECdocument also provides a document type that meets the federal government's requirements for DOD-STD-2167 and templates for each of the Data Item Descriptors (DIDs) specified in that standard.

- **HP DATATRIEVE**—HP DATATRIEVE is a tool for managing and manipulating data either interactively at a terminal or from an application program. With a set of English-like commands and statements, you can interactively retrieve, store, modify, and report on data. For applications programmers, HP DATATRIEVE can save coding and debugging time and source space by handling the following actions:
  - Finding and opening data files
  - Performing input and output operations
  - Formatting data
  - Converting data types
  - Handling error and end-of-file conditions

2 DECset Environment Manager

This chapter describes the DECset Environment Manager, its graphical user interface, and command-line interface.

2.1 Environment Manager Basic Concepts

The DECset Environment Manager provides a single mechanism for tailoring the execution environment for a set of DECset tools. The tailoring factor is known simply as a **context**.

Using the DECwindows user interface, a context can be applied to all tools displaying on a particular DECwindows display. Using the command line interface, a context can be applied to a particular process and all subprocesses subsequently created by that process.

2.1.1 Context

A context is defined as a named set of values that tailors the DECset tools to a specific software development activity (for example, fixing bugs or implementing enhancements). The supported attributes for which these values are supplied are: logical name definitions; symbol definitions; MMS macro definitions; MMS build options; linker options file specifications; library settings; a source directory specification; output directory specifications; and a default directory specification.

Each context has a unique name. The context name is limited to a length of 16 characters. There can be only one context in a particular context database (see Section 2.1.4) with that name.

At any given time, the named context most recently applied to a tool is known as the tool's **current context**. If no context has been applied, the tool has no current context.

A tool's execution environment is not fully defined by the values associated with a named context. Rather, a tool's execution environment is constructed from the following sources, in order:

1. Parent process
2. X Windows resource file (if applicable)
3. Named context (if applicable)
4. Command-line qualifiers used to invoke the tool

A specification retrieved from a higher-numbered source always supersedes any specification of the same object retrieved from a lower-numbered source.

2.1.2 Context Hierarchy

A **project context** is associated with a development project. A project context is typically used by all users working on the particular project.

A user-specific context can be based on a project context, thereby tailoring the project context to a particular user. In this case, the project context becomes a **parent** context.

A context based upon another context is said to be a **child** context. Each child context **inherits** all the information associated with its parent context. If the same attribute in both a child and its parent have different values, the child's value overrides that of the parent.

Any context can be based upon another context, which (in turn) can itself be based upon yet another context. A child context inherits information from its entire "chain" of parent contexts. The term **context object** refers to a context that includes all of this inherited information.

2.1.3 Context File

A **context file** is used to store information specific to a particular named context. The context file is specified by an OpenVMS file specification, which is stored in the context database entry (see Section 2.1.4) associated with the named context. The default context file type is .DECSET_CONTEXT.

The context file is a sequential ASCII text file. In a context file, a continuation line must be terminated by a hyphen character. Trailing comments are not allowed on a continuation line. The context file specification must exist in the format: DISK:[DIRECTORY.SUBDIRECTORY]*.DECSET_CONTEXT.

If you specify a node name in the File menu New Context... Context File field—even if the node name is part of a local cluster—the node specification is invalid at the time of context file retrieval from the context database.

Example 2–1 shows the contents of the context file PLAN9.DECSET_CONTEXT.

Example 2–1 Context File PLAN9.DECSET_CONTEXT

```
!
! Context file DISK12:[JONES]XXX01.DECSET_CONTEXT
! Written by DECset Environment Manager at  7-SEP-1998 15:03:24.82
!

SET DEFAULT DISK12:[JONES]

DEFINE/TABLE=LNM$GROUP_TABLE LOGICAL -
 EQUIV

SET_MMS RULES

SET_MMS DESCRIPTION

SET_MMS NOOVERRIDE

SET_MMS NOCMS

SET_MMS VERIFY

SET_MMS ACTION

SET_MMS NOIGNORE

SET_MMS NOFORCE

SET_MMS NOCHECK_STATUS

SET_MMS NOREVISE_DATE

SET_MMS NOSKIP_INTERMEDIATE

SET_MMS NOLOG
```

2.1.4 Context Database

A context is defined in a **context database**. This database is stored as a sequential ASCII text file. Each record of this file has the following format:

```
  context-name  context-file-spec  [parent-context]  ["context-description"]
```

A context database entry must include a complete file specification. For example:

```
DISK12:[USER.DIRECTORY]context_database_file.DECSET_CONTEXT_DB
```

Example 2–2 shows the contents of context database file PLAN1.DECSET_CONTEXT_DB.

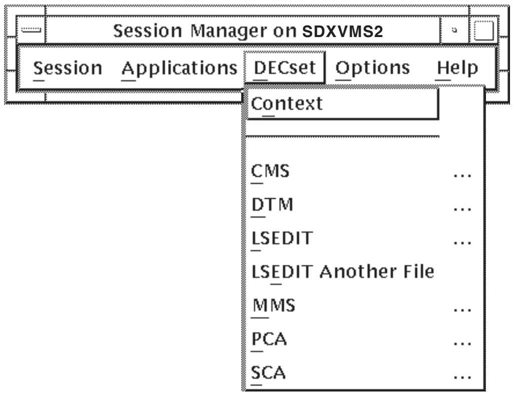
Example 2–2 Context Database File PLAN1.DECSET_CONTEXT_DB

```
PLAN1 DISK12:[JONES]PLAN1.DECSET_CONTEXT "Top Level Product for LSE"
PLAN4 DISK12:[JONES]PLAN4.DECSET_CONTEXT PLAN1 "Debug Context for LSE"
XXX01 DISK12:[JONES]XXX01.DECSET_CONTEXT PLAN1 "Experiment 01 for LSE"
PLAN9 DISK12:[JONES]PLAN9.DECSET_CONTEXT PLAN4 "Modified Debug Context for LSE"
```

2.2 DECset Tool Selection

The DECset tool selection pull-down menu appears in the FileView and Session Manager menu bars under the label DECset. Figure 2–1 illustrates the DECset tool selection pull-down menu.

Figure 2–1 DECset Tool Selection Pull-down Menu



By default, when you start a DECset tool, the tool runs as a subprocess of the session manager’s process. The DECset tool shares its quotas and environment with the Session Manager. Click on the ellipsis next to the DECset tool to bring up a dialog box that allows you to specify whether the process is to be started as a detached process.

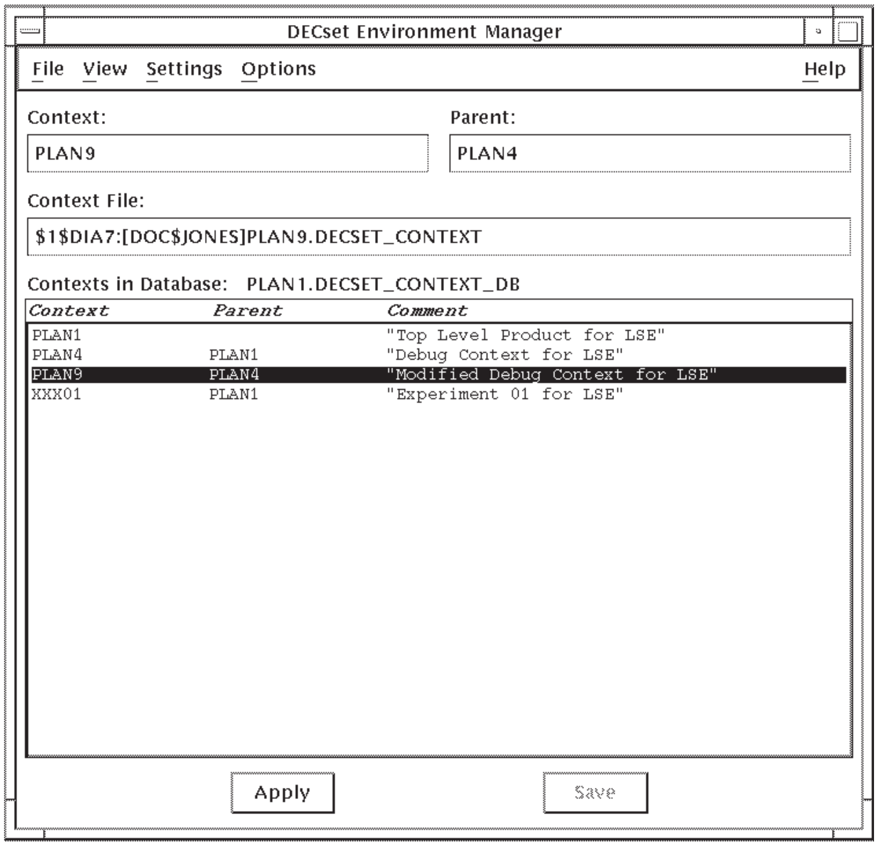
When a DECset tool is started up within the DECwindows interface, the tool invokes the Environment Manager (if the Environment Manager is installed) as part of tool initialization. If the Environment Manager is not already running, it is started up and initially appears as an icon on the DECwindows display; the tool registration then proceeds. The Environment Manager processes are named “DECSET$EM_xxxx”, where *xxxx* refers to the hexadecimal representation of your process identifier. Multiple versions of each file can exist.

The base priority of an Environment Manager process is initially set to the current value of the DEFPRI system parameter on the host system. When the Environment Manager is applying a context, the base priority is raised to 10. After the context apply operation has completed, the initial base priority setting is restored.

2.3 DECset Context Window

The DECset Context window is the main window of the Environment Manager’s graphical user interface. Select the Context option from the DECset tool selection pull-down menu (see Section 2.2) to bring up the DECset Context window, shown in Figure 2–2.

Figure 2–2 DECset Context Window



Context: PLAN9

Parent: PLAN4

Context File: 1DIA7:[DOC$JONES]PLAN9.DECSET_CONTEXT

Contexts in Database: PLAN1.DECSET_CONTEXT_DB

```
Context            Parent            Comment
PLAN1                                "Top Level Product for LSE"
PLAN4              PLAN1             "Debug Context for LSE"
PLAN9              PLAN4             "Modified Debug Context for LSE"
XXX01              PLAN1             "Experiment 01 for LSE"
```

The DECset context window consists of the following working areas:

- Menu bar
- Context specification area
- Dialog buttons area

By default, the DECset Environment Manager tries to load a context database. The default context database is specified by the translation of the DECSET$CONTEXT_DB logical name. If this logical name is undefined, SYS$LOGIN:DECSET$KNOWN_CONTEXTS.DECSET_CONTEXT_DB is used as the default database, if that file exists.

The default context is specified by the translation of the DECSET$CONTEXT logical name; if this logical name is undefined, SAMPLE is used as the default context if a database entry exists for the SAMPLE context.

2.3.1 Menu Bar

The Menu bar contains some standard pull-down menus: File, View, Settings, Options and Help. The menus are described in the following sections.

2.3.1.1 File Menu

The menu items in this pull-down are used to control the contents of the DECset Context, Context File, and Context Database fields. The menu items are as follows:

- **Apply Context**—Used to apply the context shown in the DECset Context field. This menu item has the same function as the Apply button.
- **Delete**—Used to delete the context entry, context file, or context database.

  Click on Context Entry to delete the context entry shown in the Context field from the Context Database Entries field, and to delete the selected context entry from the context database. The DECset Environment Manager checks to ensure that the context to be deleted is not a parent of another context.

  Click on Context File to delete the context file shown in the Context File field. The DECset Environment Manager updates the window to reflect the deletion.

  Click on Database to delete the currently-opened context database.
- **Modify Database Entry**—Used to modify any of the fields in a selected database entry.

  The DECset Environment Manager displays a dialog box with fields filled in with information from the selected context.
- **New**—Used to create a context or database.
- **Open Database**—Used to select a context database file to open
- **Print**—Used to print the context, context file, context including attributes from the parents, or context database.
  - Click on Context to print the information associated with the current selected context, without its inherited information but including any modifications made to the context that may not have been saved to the associated context file.

  - Click on Context File to print the information associated with the current selected context in a scrollable window. This context file might not be the same as the result of Print Context depending on whether you have made modifications to the current selected context and have saved the modifications to the associated context file.
  - Click on Context Including Inheritance to print the information associated with the context that would be the result of applying the selected context and its inherited context.
  - Click on Database to print the contents of the context database.
- **Save Context File**—Used to save the contents of the selected context file. The Save Context File menu item performs the same operation as the Save button.
- **Exit**—Used to exit the DECset Context window.

2.3.1.2 View Menu

The menu items in this pull-down are used to review the contents of the context, context file, and context including inheritance.

- Click on Context to display in a scrollable window the information associated with the current selected context, without its inherited information but including any modifications made to the context that may not have been saved to the associated context file.
- Click on Context File to bring up the context file associated with the current selected context in a scrollable window. This context file might not be the same as the result of View Context, depending on whether you have made modifications to the current selected context and have saved the modifications to the associated context file.
- Click on Context Including Inheritance to display the current context in a scrollable window. This is the context that would be the result of applying the selected context with its inherited information.

2.3.1.3 Options Menu

The menu items in this pull-down are used to apply, save, or confirm context entry information. The menu items are as follows:

- **Apply Context on Save**—Used to automatically apply the selected context (the context shown in the Context field). The Apply Context on Save toggle button, when turned on, applies the context object after saving the context. By default, the Apply Context on Save toggle button is turned off.

- **Save Context on Apply**—Used to automatically save the context shown in the Context File field. The Save Context on Apply toggle button, when turned on, saves the context file prior to applying the context. A context must be saved before it can be applied. By default, the Save Context on Apply toggle button is turned off.
- **Confirm Apply**—Used to bring up a Yes/No confirmation message box as part of the apply operation. By default, the Confirm Apply toggle button is turned on.
- **Confirm Save**—Used to bring up a Yes/No confirmation message box on the Save Context File operation. By default, the Confirm Save toggle button item is turned on.
- **Confirm Delete Context Entry**—Used to bring up a Yes/No confirmation message box on the Delete Context Entry operation. By default, the Confirm Delete Context Entry toggle button is turned on.
- **Confirm Delete Context File**—Used to bring up a Yes/No confirmation message box on the Delete Context File operation. By default, the Confirm Delete Context File toggle button is turned on.
- **Confirm Delete Database**—Used to bring up a Yes/No confirmation message box on the Delete Database operation. By default, the Confirm Delete Database toggle button is turned on.
- **Save Options**—Used to write the current options settings to a user-profile file. If this file exists, the settings in the user-profile file override the default settings.

2.3.1.4 Help Menu

This menu provides access to help screens. It contains the items usually present in DECset tool Help pull-down menus, including the following:

- **On Context**—Used to bring up information about an object in a window or dialog box
- **On Window**—Used to display an overview
- **On Help**—Used to bring up information on how to use the help system
- **On Version**—Used to display copyright and version information

2.3.1.5 Settings Menu

The Settings pull-down menu specifies or modifies the definitions and settings that define a particular context. The Settings menu is disabled until values are supplied in both the Context and Context File fields.

The Settings pull-down menu items include the following:

- **CMS Library**—Directory search list to specify a CMS library
- **Default Directory**—Used to specify a default directory associated with the current context
- **DTM Library**—Used to specify a directory as the HP DIGITAL Test Manager library
- **Logical Names**—Used to specify logical name definitions that are specific to the current context
- **MMS Generate Function**—The MMS Generate function provides you the capability to define several qualifiers to be used during the creation of a descriptor file.
- **MMS Options**—Used to specify the following MMS build command options:
  - **Linker Object Libraries**—Directory search list to specify the object library to be included as input to the linking operation
  - **Linker Options**—Directory search list to specify user-written options files to be included in the LINK command in the MMS description file
  - **Input/Output**—Used to set the input and output qualifiers that could go on the MMS command line
  - **Definition/Directive**—Used to specify definitions and directives for the MMS build command
  - **MMS Macros**—Used to direct MMS to add to or to override the macro definitions in the MMS description file
  - **Executable Files**—Used to specify a directory into which executable files will be written
  - **Listing Files**—Used to specify that compilations use the /LIST qualifier and optionally write the listing files to the specified directory
  - **Object Files**—Used to specify a directory into which object files are to be written
  - **Object Library**—Used to specify into which library the object files from the compilations are to be inserted

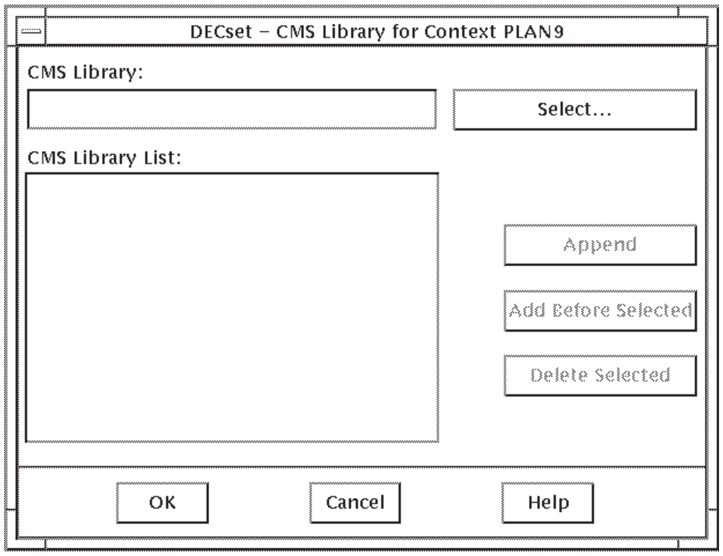
- **SCA Library**—Directory search list to specify an SCA library
- **Source Directory**—Directory search list to specify a directory from which source files will be retrieved
- **Symbols**—Used to add DCL symbol definitions to the specified context

The following sections describe these menu items in detail.

CMS Library

The CMS Library... menu item brings up a dialog box as shown in Figure 2–3.

Figure 2–3 DECset Context CMS Library Dialog



Enter a directory specification into the Directory field and click on either the Append or Add Before Selected button to add the directory in the appropriate place in the Directory List field. The Directory List field contains a search list of directory specifications, one directory specification per line.

If CMS library information exists for the current context, that information is shown in the CMS Library List field when the dialog box is displayed. Delete items from the directory list by selecting the appropriate items in the Directory List field and clicking on the Delete Selected button. The OK button accepts the directory list and dismisses the dialog box. The Cancel button dismisses the dialog box without changing the CMS library.

Click on the Select... button to specify the directory.

Default Directory

The Default Directory... menu item displays a file selection window that enables you to specify a default directory associated with the current context. If the default directory specification is available for the currently selected context, the input fields of the file selection window are filled in with the appropriate file specification.

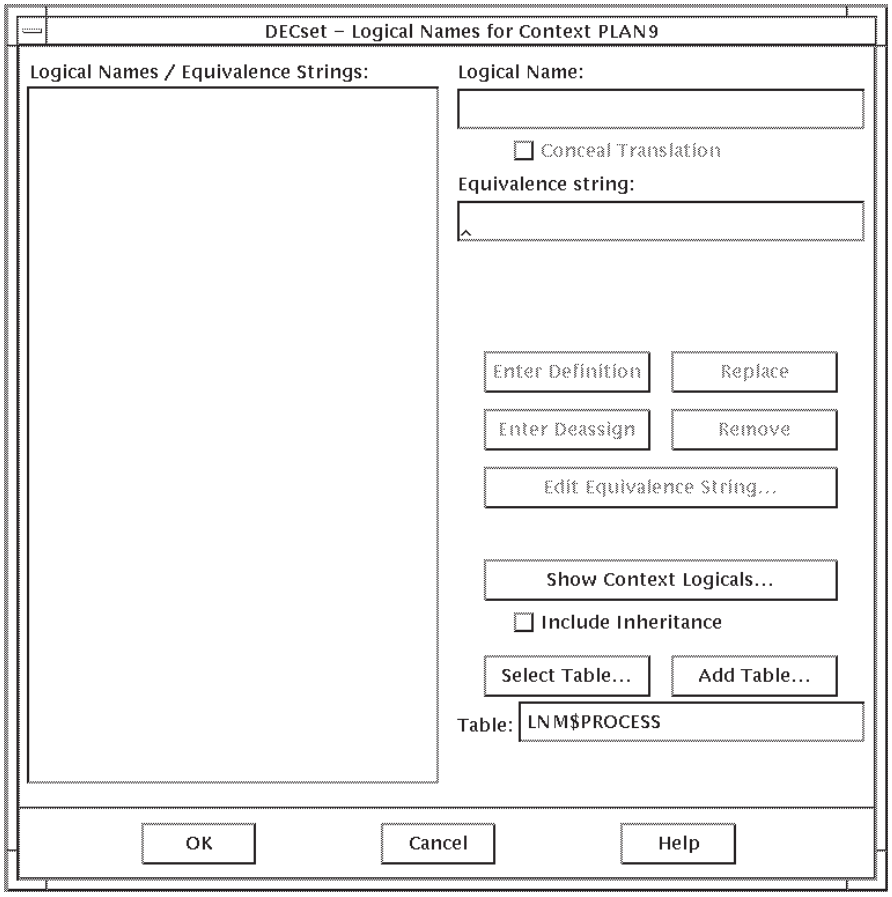
DTM Library

The DTM Library... menu item brings up a file selection window that enables you to specify a HP DIGITAL Test Manager library (directory specification) to be associated with the current context. If the currently selected context already includes a HP DIGITAL Test Manager library specification, the input fields of the file selection window are filled in with the appropriate file specification.

Logical Names

The Logical Names... menu item displays the dialog box shown in Figure 2–4.

Figure 2–4 DECset Context Logical Names Dialog



The logical name definitions retrieved from the context file specified in the Context File field of the main window are displayed in the Logical Names/Equivalences field of this dialog box.

The dialog box contains the following buttons :

- **Conceal Translation**—Used to specify that the translation(s) of the logical name are "concealed".
- **Enter Definition**—Used to add the logical name and corresponding equivalence strings specified in the Logical Name and Equivalences fields to the Logical Names/Equivalence Strings list shown in the dialog box.

  Use this feature to supersede a definition from a parent context.

- **Enter Deassign**—Used to enter a deassign logical name specified in the logical name field to the Logical Names/Equivalence Strings list. The contents of the Equivalence string field are ignored.

  Use this feature in a context to supersede a definition from a parent context.

  The DECset Environment Manager does not allow duplicate definitions of a logical name in the Logical Names/Equivalence Strings field.
- **Remove**—Used to remove the selected logical name definition or the DEASSIGN from the Logical Names/Equivalence Strings list.
- **Show Context Logicals**—Used to display the context logicals in memory.

  When you create or edit a logical name, if logical has no inheritance, the Environment Manager does not display logical names when you press the Include inheritance dialog button.
- **Add Table**—Used to specify an option menu containing the options system, group, job, the default setting, process, or other user-defined tables.

  The value indicates the logical name table into which the logical name is inserted or from which the logical name is deassigned.
- **OK**—Used to dismiss the dialog box and write changes made to the current context.

  The current changes are not written to the context file until after you either apply or save these changes. Changes are written to the current context in memory only.
- **Cancel**—Used to dismiss the dialog box, discarding current changes made.
- **Help**—Used to bring up a help window containing text describing the dialog box.

The Logical Names/Equivalence Strings field places equivalence strings on separate lines to distinguish between logicals defined. For example:

```
DEFINE LIONS BAR
DEFINE TIGERS A,B,C
DEASSIGN BEARS
```

The Environment Manager does not accept logical name specifications that include quotation marks.

When you enter a logical name definition or deassignment, the definition is entered into the list box and the corresponding logical name table as specified in the Add Table dialog button. If you switch logical name tables, you work in that tables' list box and name table. It is only when you press OK that you effectively enter these modifications into the currently selected context. If you press Cancel, the context is not modified.

It is the context logicals that are displayed by the Show Context Logicals dialog button, not the list box or the logical name tables. Further, it is the context in memory that is modified when you press OK to one of these types of dialog buttons. It is only when you press the Save button that the context on disk is modified. Again, you may Cancel by refusing to save the changes.

MMS Generate Function

The MMS Generate function provides you the capability to define several qualifiers to be used during the creation of a descriptor file. There are two toggle buttons that can be set and two text input fields that can contain additional qualifiers for compile and link commands.

- **Scan_Include**—Setting this button causes entries to be placed in the descriptor file to scan C/C++ include files for any dependencies.
- **Built-In Rules Apply**—Setting this button causes the Generate function to use the built-in rules when creating the descriptor file. When this button is set, the Additional Compiler text field is disabled.
- **Additional Compiler Switches**—Use this text field to provi de additional qualifiers that are to be placed on the compiler command when crea ting a descriptor file. The entry requires that the slash character (/) on each qualifier also to be included.

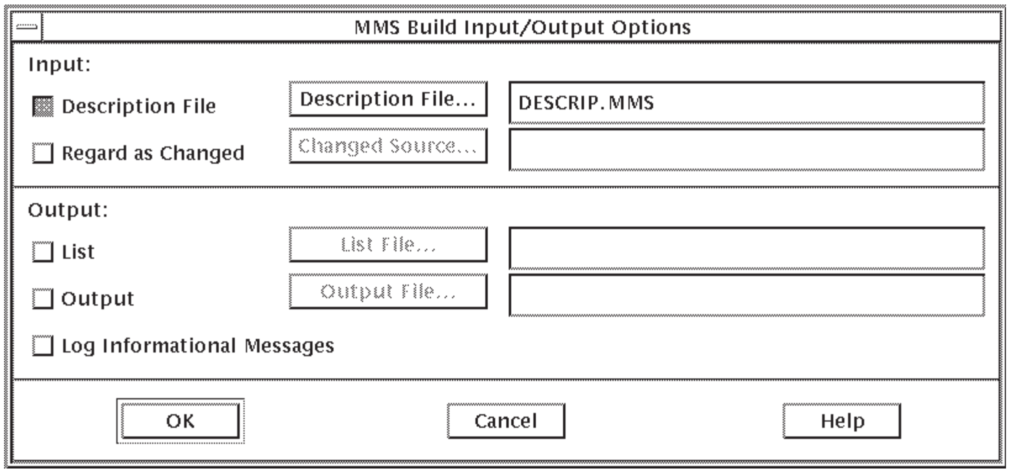
  ```
  /ABC/DEF
  ```
- **Additional Linker Switches**—Use this text field to provide additional qualifiers that are to be placed on the Link command when creating a descriptor file. The entry requires that the slash character (/) on each qualifier also to be included.

  ```
  /ABC/DEF
  ```

MMS Options

The Input/Output... menu item from the MMS Options... menu displays the dialog box shown in Figure 2–5.

Figure 2–5 MMS Options Input and Output Dialog



Input:

▨ Description File — DESCRIP.MMS

☐ Regard as Changed

Output:

☐ List

☐ Output

☐ Log Informational Messages

Use this dialog box to set options for the MMS command. These options correspond to MMS command-line qualifiers. If the currently selected context already includes any of these MMS option settings, this dialog box reflects that information; otherwise, the default settings are represented as shown in Figure 2–5.

The following sections describe the dialog box in detail.

Input Section Click on the buttons and use the text fields following the Input: label to specify input files for any subsequent build. The buttons are as follows:

- **Description File**—Click on this toggle button to specify an MMS description file for any subsequent build. Click on the **Description File...** push button and select a file from the dialog box, or enter a file specification into the text field. If you use the file selection box, this field is filled in with the selected file.
- **Regard as Changed**—Click on this toggle button to direct MMS to treat only the specified sources as having been changed, regardless of their actual modification times. No date checking is performed at all; that is, MMS simply rebuilds any targets that depend on one or more of the specified sources.

  Click on the **Changed Source...** push button and select a file from the dialog box, or enter a file specification into the text field. If more than one source is indicated, use a comma-separated list. If the file selection box is used, this field is filled in with the selected file.

Output Section Click on the buttons and use the text fields following the Output: label to specify output files for any subsequent build. The buttons are as follows:

- **List**—Click on this toggle button to specify an MMS list file for any subsequent build. Click on the **List File...** push button and select a file from the dialog box, or enter a file specification into the text field. If you use the file selection box, this field is filled in with the selected file.
- **Output**—Click on this toggle button to specify an MMS output file for any subsequent build. Click on the **Output File...** push button and select a file from the dialog box, or enter a file specification into the text field. If you use the file selection box, this field is filled in with the selected file.
- **Log Informational Messages**—Click on this toggle button to instruct MMS to log informational messages, in addition to warning and error messages, to the List or Output file.

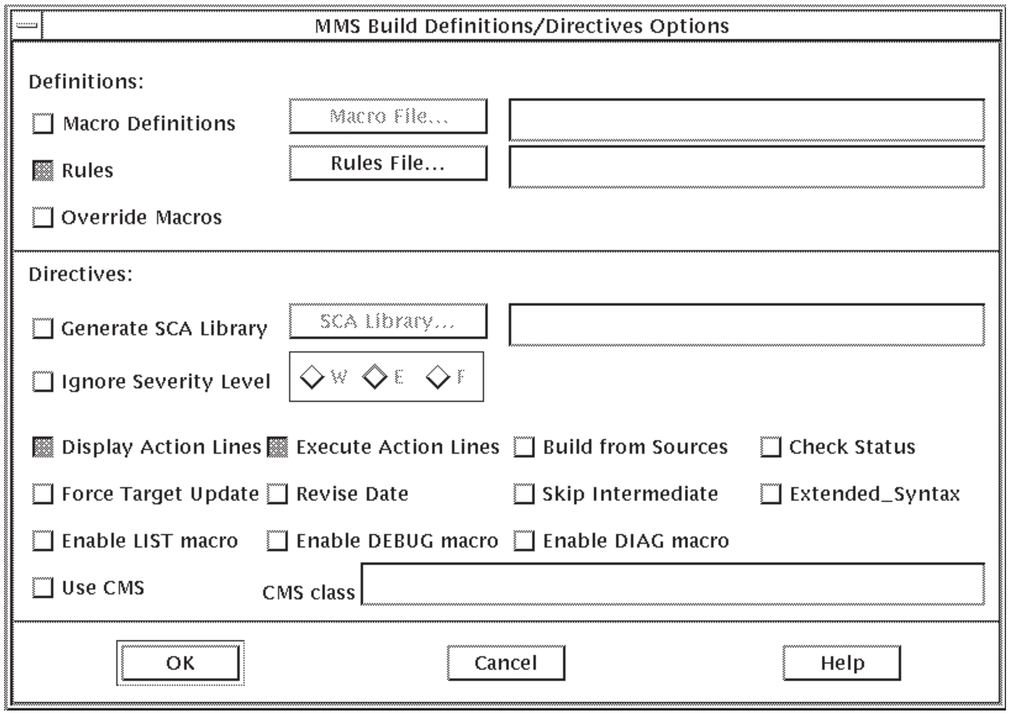
Navigation Buttons Click on the navigation buttons to save your selections, cancel your selections, or get online Help. The buttons are as follows:

- **OK**—Click on the OK button to indicate that the options for the MMS command are complete and you want to exit out of the dialog box.
- **Cancel**—Click on the Cancel button to quit the dialog box without completing options and discard changes for the MMS command.
- **Help**—Click on the Help button to bring up the textual explanation of the dialog box in a window.

MMS Definition/Directive

The Definition/Directive... menu item from the MMS Options... menu displays the dialog box shown in Figure 2–6.

Figure 2–6 MMS Options Definitions and Directives Dialog


MMS Build Definitions/Directives Options

Definitions:

- ☐ Macro Definitions — Macro File...
- ▨ Rules — Rules File...
- ☐ Override Macros

Directives:

- ☐ Generate SCA Library — SCA Library...
- ☐ Ignore Severity Level — ◇ W ◇ E ◇ F
- ▨ Display Action Lines
- ▨ Execute Action Lines
- ☐ Build from Sources
- ☐ Check Status
- ☐ Force Target Update
- ☐ Revise Date
- ☐ Skip Intermediate
- ☐ Extended_Syntax
- ☐ Enable LIST macro
- ☐ Enable DEBUG macro
- ☐ Enable DIAG macro
- ☐ Use CMS — CMS class

Use this dialog box to set options for the MMS command. These options correspond to MMS command-line qualifiers. If the currently selected context already includes any of these MMS settings, this dialog box reflects that information; otherwise, the default settings are represented as shown in Figure 2–6.

Definitions Section Click on the buttons and use the text fields following the Definitions: label to specify definitions for any subsequent build. The buttons are as follows:

- **Macro Definitions**—Click on this toggle button to direct MMS to add to or override the macro definitions in the description file. Click on the **Macro File...** push button and select a file from the dialog box, or enter a file specification into the text field. If you use the file selection box, this field is filled in with the selected file.

- **Rules**—Click on this toggle button to direct MMS to apply user-defined built-in rules and a suffixes precedence list when it builds a system. Click on the **Rules File...** push button and select a file from the dialog box, or enter a file specification into the text field. If more than one source is indicated, use a comma-separated list. If you use the file selection box, this field is filled in with the selected file.
- **Override Macros**—Click on this toggle button to control the order in which MMS applies definitions when it processes macros.

Directives Section The buttons and text fields following the Directives label are implemented as toggle buttons, although some require additional information. (For details on these toggle buttons, see the *Guide to HP Module Management System for OpenVMS Systems*.) If an additional file specification needs to be specified, a file selection window is used.

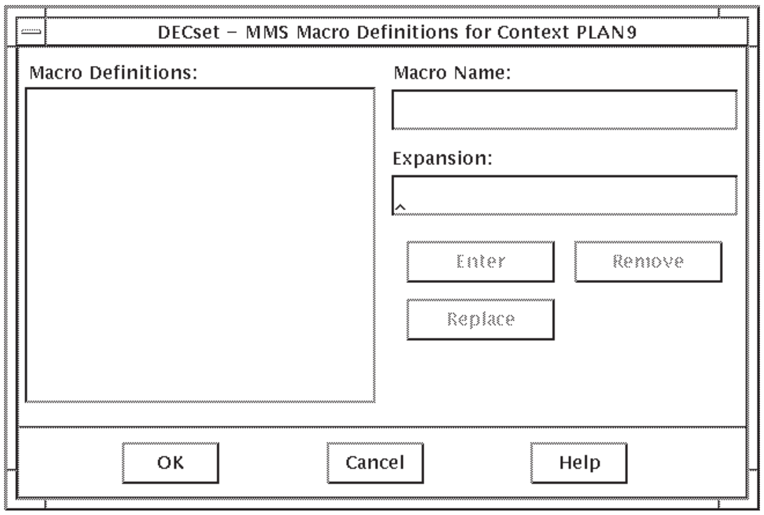
Navigation Buttons Click on the navigation buttons to save your selections, cancel your selections, or get online Help. The buttons are as follows:

- **OK**—Click on the OK button to indicate that the options for the MMS command are complete and you want to exit out of the dialog box.
- **Cancel**—Click on the Cancel button to quit the dialog box without completing options and discard changes for the MMS command.
- **Help**—Click on the Help button to bring up the textual explanation of the dialog box in a window.

MMS Macros

The MMS Macros... menu item from the MMS Options... menu displays the dialog box shown in Figure 2–7.

Figure 2–7 MMS Macro Definitions Dialog



The macro definitions retrieved from the context file specified in the Context File field of the main window are displayed in the Macro Definitions field of this dialog box.

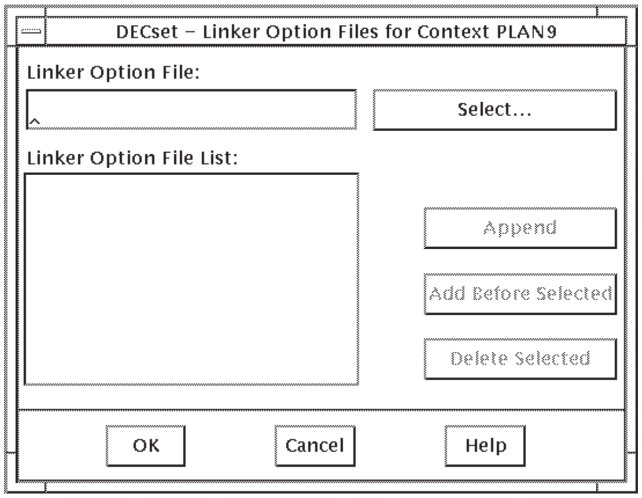
The dialog box contains the following buttons:

- **Enter**—Used to add a new macro definition to the Macro Definitions list shown in the dialog box.
- **Remove**—Used to remove the selected macro definition from the Macro Definitions list.
- **OK**—Used to dismiss the dialog box and write changes made to the current context.
- **Cancel**—Used to dismiss the dialog box, discarding current changes made.
- **Help**—Used to display the help window containing text describing the dialog box.

MMS Linker Options

The MMS Linker Options... menu item from the MMS Options... menu displays the dialog box shown in Figure 2–8.

Figure 2–8 MMS Linker Options Files Dialog



Click on the Select... button to bring up a file selection window that allows you to choose an options file.

Enter a linker options file specification into the Options File field and click on either of the Append or Add Before Selected buttons to add the options file specification in the appropriate place in the Options File List field.

The Options File List field contains an ordered list of options file specifications, one file specification per line. Delete items from the options file list by selecting the appropriate item(s) in the Options File List field and clicking on the Delete Selected button. The OK button accepts the options file list and dismisses the dialog box. The Cancel button dismisses the dialog box without changing the current options file list.

Press Select... to specify the options file.

SCA Library

The SCA Library... menu item displays a dialog box similar to the one shown in Figure 2–3. The only difference for the three menu choices is the text of the title bar indicating that an SCA library is being set.

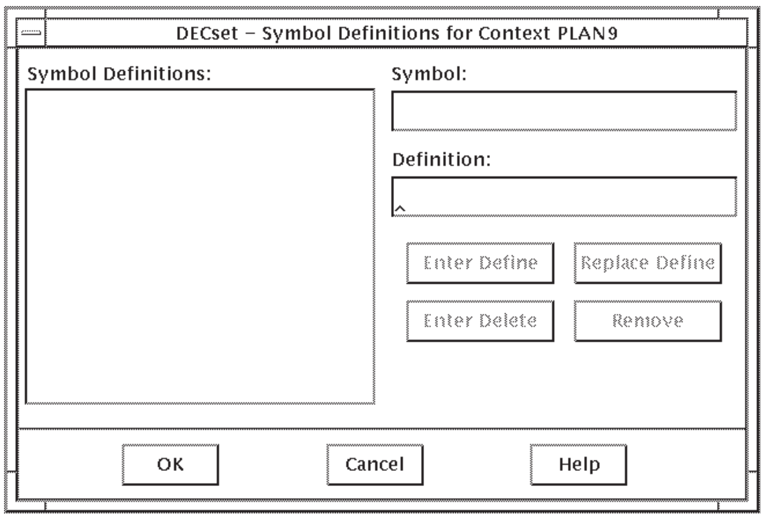
Source Directory

The Source Directory... menu item displays a dialog box similar to the one shown in Figure 2–3. The only difference for the three menu choices is the text of the title bar indicating that a source directory is being set.

Symbols

The Symbols... menu item displays the dialog box shown in Figure 2–9.

Figure 2–9 DECset Context Symbol Definitions Dialog



The symbol definitions retrieved from the context file specified in the Context File field of the main window are displayed in the Symbol Definitions field of this dialog box.

Use this dialog box to add DCL symbol definitions to the specified context.

The Environment Manager stores symbols only as global symbols. Both single and double equal sign ("=" or "= =") syntax is allowed for symbol definitions in a context file; however, either syntax is interpreted as a global symbol definition.

The dialog box contains the following buttons:

- **Enter Define**—Used to add a new symbol definition to the Symbol Definitions list shown in the dialog box.

  Use this in a context to supersede a definition from a parent context.
- **Remove**—Used to remove the selected symbol definition or the DELETE SYMBOL entry from the Symbol Definitions list.

- **Enter Delete**—Used to add a DELETE SYMBOL entry to the Symbol Definitions list shown in the dialog box.

  Use this feature in a context to supersede a definition from a parent context.
- **OK**—Used to dismiss the dialog box and write changes made to the current context.
- **Cancel**—Used to dismiss the dialog box, discarding current changes made.
- **Help**—Used to display a help window containing text describing the dialog box.

2.3.2 Context Specification Area

The context specification area is used to display information about contexts, and consists of the following fields:

- Context
- Parent
- Context file
- Context database entries

The context specification area consists of read-only fields. Use the menu bar functions to make changes to any of these fields.

Context Field

You can display a context name in the Context field by single-clicking on a line in the Context Database Entries field, or by selecting the New menu item from the File menu.

Parent Field

In Figure 2–2, PLAN4 shown in the Context Database Entries field is the parent of PLAN9. PLAN1 is the parent of PLAN4. This example illustrates an ancestral chain of contexts. The DECset Environment Manager does not allow a circular ancestral chain. If a circular chain is detected, the DECset Environment Manager displays an error message dialog box, and aborts the operation on that context.

Context File Field

If you select an existing context file from the Context Database Entries field, the name of the context and the associated information is displayed in the read-only Context, Parent, and Context File fields.

For a new context, or one for which you are modifying the associated context file specification, you can use the Modify Database Entry... menu item from the File menu to bring up a file selection window.

If any part of the file specification is not supplied (such as the disk name), the corresponding component of your current, default directory specification is used. If the name or the file type is not specified, the name of the currently specified context is used as the file name and .DECSET_CONTEXT is used as the file type.

Context Database Entries Field

The context names (and some associated information) are retrieved from the context database listed in the Context Database Entries field of the DECset context window.

2.3.3 Dialog Buttons Area

The dialog buttons area is used to apply or save information associated with the currently-selected context. The buttons are as follows:

- **Apply**—Causes the selected context object to be applied to any DECset tools displaying on the same DECwindows display as is the Context window. You may also double-click on the context in the Context Database Entries field to apply the context. In addition to applying this context, any command line qualifiers that are in effect are then re-applied to the tool.

  A named context must be saved before a context apply operation is initiated.

- **Save**—Saves the context file shown in the Context File field.

  This action saves the current definitions and settings (possibly modified via the Settings menu) to the context file indicated in the Context File field of the main window. If the value in the Context File field has changed since the last Save operation for the current context, the DECset Environment Manager writes the new file specification to the context database. For a new context, the Save menu item ensures that the context is known by adding the specified context name and associated context file specification to the context database. The DECset Environment Manager displays a confirmation dialog box for final approval before saving new information in an existing context database entry or an existing context file. Saving a context affects other DECset tools displaying on the DECwindows display only when that context is subsequently applied.

Note

If you place your context files in a separate directory, you might want to put a version limit on the directory to keep a set number of versions of any context file or database. Use the SET FILE/VERSION_LIMIT or the SET DIRECTORY/VERSION_LIMIT commands.

For example, you might choose to limit only the number of versions of the database and purge context files by setting a version limit on the database file.

2.4 Command-Line Interface

Use the DECSET SET CONTEXT and DECSET SHOW CONTEXT commands to set or show the current context or the current context database for your process. The following sections describe these commands in detail.

2.4.1 DECSET SET Command

The SET command line has one of the following forms, depending upon whether a context or a context database is being set:

```
DECSET SET CONTEXT context-name [/DATABASE=database-file]
```

```
DECSET SET DATABASE database-file
```

DCL wildcards cannot be used in either the context-name or database-file argument.

The /DATABASE qualifier allows you to set a context that is defined in the specified database. If the /DATABASE qualifier is missing, the value of the DECSET$CONTEXT_DB logical name is used. If this logical name is undefined, your SYS$LOGIN directory is searched for a file named DECSET$KNOWN_CONTEXTS.DECSET_CONTEXT_DB. If a database cannot be located, an error message is displayed.

The DECSET SET CONTEXT command automatically defines the logical names DECSET$CONTEXT and DECSET$CONTEXT_DB. The DECSET SET DATABASE command automatically defines the logical name DECSET$CONTEXT_DB.

2.4.2 DECSET SHOW Command

The SHOW command line has one of the following forms, depending on whether information about a context or a context database is being shown:

```
DECSET SHOW CONTEXT [context-name] [/DATABASE=database-file]

DECSET SHOW DATABASE
```

DCL wildcards cannot be used in either the context-name or database-file argument.

If *context-name* is not specified in the DECSET SHOW CONTEXT command, the value of the DECSET$CONTEXT logical name is used (if that logical name is defined). If the context name cannot be determined, an error message is displayed.

The /DATABASE qualifier enables you to show information about a context that is defined in the specified database. If the /DATABASE qualifier is missing, the value of the DECSET$CONTEXT_DB logical name is used. If this logical name is undefined, your SYS$LOGIN directory is searched for a file named DECSET$KNOWN_CONTEXTS.DECSET_CONTEXT_DB. If a database cannot be located, an error message is displayed.

2.5 Disabling Environment Manager

You can stop the Environment Manager from running when you activate a DECset component by defining the logical DECSET$DISABLE_ENVMGR. The value of the logical is irrelevant. When invoking a component using /INTERFACE=DECWINDOWS, the logical can be set in one of following three ways:

- Use the DCL command to define the logical before running the component DEFINE DECSET$DISABLE_ENVMGR " "
- Use the DCL command DECSET SET CONTEXT context-name /DATABASE=database- file-name if the database and the context file are already defined using the Environment Manager.This causes the logical defined in the context file to be set for the current process.
- Set the logical while in the Session Manager, before invoking a DECset component. Use the session pull-down menu and select the Logical Names. The "Logical Names..." function is an optional menu item and if not currently available it can be added to the Session Manager menu bar. For more information on adding the "Logical Names..." item to the Session Manager, refer to the DECwindows documentation. When running the Common Desktop Environment (CDE), the command line interface works the same as the first two items in the preceding list. For more

information, refer to the CDE documentation for creating applications using the Application Manager.

3
Organizing Files, Libraries, and Directories

This chapter describes ways to set up the files, libraries, and directories for a software development project. The intended audience of this chapter is the software development project leader who might be responsible for setting up any or all of the following:

- User accounts
- Project directories
- File protection schemes
- CMS, HP DIGITAL Test Manager, and SCA libraries
- Build directories
- CMS library reference copy areas

The descriptions in this chapter are not steadfast rules that must be followed in order to use the DECset and other OpenVMS tools; however, you might find them useful for creating and maintaining a well-organized directory structure for your project.

3.1 Setting Up User Accounts and Directory Protection

Before setting up user accounts for project team members, you must determine what project information and resources each team member needs. Not every member needs access to every directory and component of the project. Restricting team members' access to files and resources to only the ones they need reduces the chances of errors related to revising or building versions of software. Additionally, you want to prevent unauthorized users from gaining access to confidential files and resources.

OpenVMS provides several tools to authorize and control the use of system resources by individual users. The *OpenVMS System Manager's Handbook* describes these tools and describes how and when to use them. The *OpenVMS DCL Concepts Manual* gives a detailed explanation of the following:

- Protection procedures
- User identification codes (UICs)
- Access control lists (ACLs)

To begin, set up a basis for system accounting, file protection, and interprocess communication:

1. To protect your data, establish a group of user accounts by specifying a common UIC group number for all project members; or set up rights identifiers to provide varied and selective access, if necessary.
2. Create a list of rights identifiers for the members of your project team. Enter those identifiers in the system rights list. For example, you might want to make developers holders of the PROJECT_SOURCE identifier, and make technical writers holders of the PROJECT_SOURCE_READ identifier. See Chapter 6 for examples of using these rights identifiers.
3. Set up UIC protection masks (use the DCL SET PROTECTION command) to further restrict outside access (system, owner, group, and world) to member directories, certain project and library directories, and selected files. You do so by defining categories of access within each group (read, write, execute, and delete).
4. Use ACLs to restrict access to specific elements by defining protection rules (UIC or identifier-based protection masks) for a file or a directory. By using ACLs, you can provide access to specific elements for users having other UIC numbers.

See Chapter 5 for an example of setting up protection for directories and libraries.

3.2 Setting Up a Project Directory Structure

This section describes a suggested project directory structure, presented in three parts:

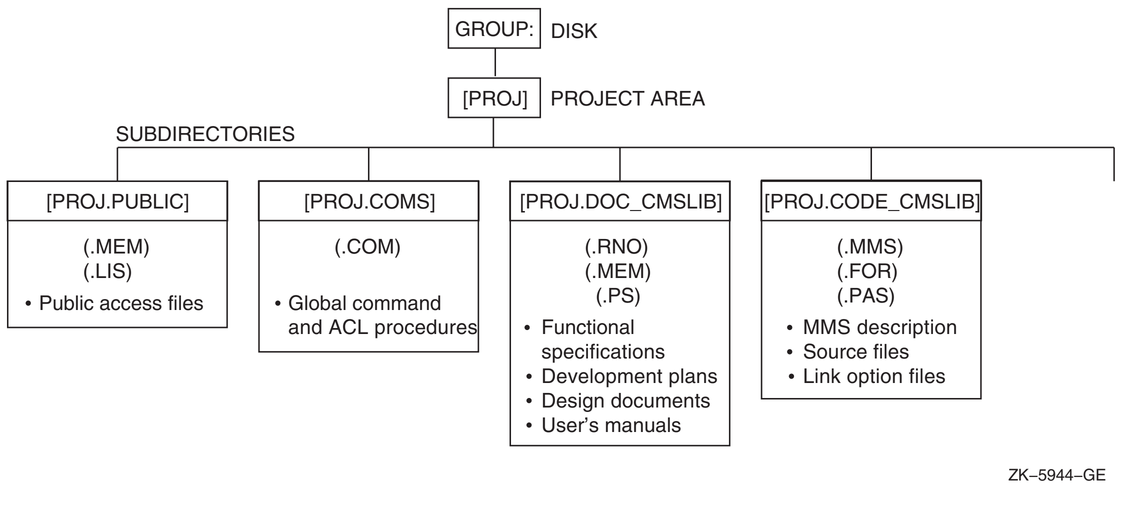
1. The initial project directory structure—Contains public access files, command files, logical name definitions, and CMS libraries for source code and project documentation

2. The build directory—Contains the following subdirectories:
   - Object libraries
   - Source files fetched from CMS
   - SCA library system
   - CMS library reference copy area
3. HP DIGITAL Test Manager storage area directory

3.2.1 Setting Up the Initial Project Directory Structure

Figure 3–1 shows the initial storage areas of an effective directory structure. It contains the directories and libraries that you must set up in the early stages of your project. You will add more directories and libraries to this hierarchy as your project develops, as described in later sections of this chapter.

Figure 3–1 Initial Storage Areas for a Typical Project



The first storage areas you need include the following:

- A subdirectory for public access—[PROJ.PUBLIC]

  This subdirectory contains information for users and interested parties (for example, printable documentation files). This directory could also be kept outside the project area.
- A subdirectory for global command procedures—[PROJ.COMS]

This subdirectory can include command files that contain logical name definitions and files with command procedures developed during your project cycle.

- A CMS library for documentation—[PROJ.DOC_CMSLIB]

  This CMS library can contain files that document the application; for example, those produced for HP Standard Runoff (DSR) (.RNO files) or DECdocument (.SDML files). You can keep the most recent printable files (.MEM or .PS files, for example) in the public subdirectory, [PROJ.PUBLIC], or in your CMS library. If disk space is a concern, do not store files in a CMS library that can easily be reproduced; for example, .MEM or .PS files. In addition, you might want to store your document sources in more than one CMS library; for example, use one CMS library for requirements documentation and another CMS library for specifications documentation.

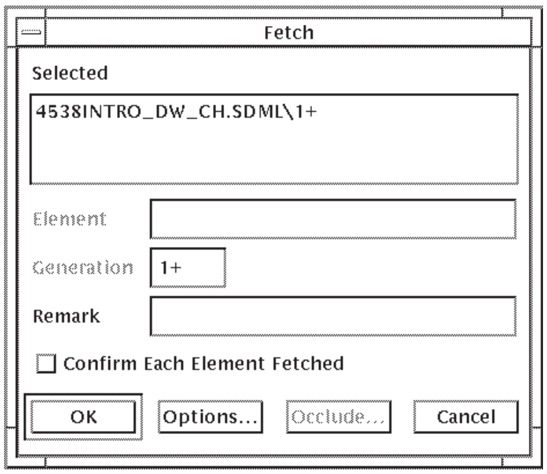
- A CMS library for source code—[PROJ.CODE_CMSLIB]

  This CMS library contains your source files. It is most useful for dynamic storage; that is, storage of objects that frequently change. The code library can include program module source files (.BAS, .PAS, .FOR, and others) and precompiler source files (.RCO, .RBA, and others).

  Initially, the CMS code library can store your team's prototype sources. This library is also a good location for the MMS description file, which details and invokes your build procedures. CMS stores the various versions of this file as your team periodically updates it. You can also store link option files here to be fetched by MMS for link procedures.

An alternative to setting up separate CMS libraries for the documentation sources is to place them in the CMS code library. This way, you can fetch or reserve both the documentation and the code source files with one generic file designation. Figure 3–2 shows the CMS Fetch dialog box, using DECwindows on a workstation, in which all project files are chosen for a fetch operation.

Figure 3–2 CMS Fetch Dialog Box


Fetch

Selected

4538INTRO_DW_CH.SDML\1+

Element

Generation: 1+

Remark

☐ Confirm Each Element Fetched

Chapter 6 provides an example of setting up CMS libraries. See the *Guide to HP Code Management System for OpenVMS Systems* for detailed explanations of CMS libraries and commands.

3.2.1.1 Planning CMS Libraries

In planning a CMS project library, you need to consider the following questions:

- What project tasks require a CMS library?
- Will the library be used mainly for static storage or more for dynamic storage; that is, tracking files that change frequently?
- Will you be using the Distributed File Service (DFS)?
- Do you need a librarian to manage the library?
- What access rights do you need, if any, and for which files or libraries?
- How often will you build your application?

- What will the naming conventions be for CMS library elements, groups, and classes?
- What, if any, will be the protocols to use the system?

3.2.1.2 Considerations for Large Projects

For large projects, use a list of CMS libraries, or virtual libraries, to store your project files instead of using only one library. Typically, as the size of a library increases, so does the number of users attempting to gain access to it. Although CMS does not have a restriction on the number of elements you can have in a library, the problems of having a large library with numerous users include the following:

- Access—CMS provides only single-thread entry; that is, only one door for operations that modify the library's contents (for example, Reserve, Replace, and Insert operations).

  Developers involved with a large project might interfere with one another if they attempt frequent library operations that change the library's contents. This type of bottleneck is particularly noticeable during major batch operations that change the contents of the library. Users can reduce problems during these operations by adjusting their work habits; for example, they should avoid wildcard operations during busy periods, or use only FETCH operations during build procedures.

- Redundancy—Large applications do not necessarily have their functional development occurring in parallel. Some pieces of functionality are complete early in the project life cycle and remain relatively stable. To some extent, build procedures repetitively access these stable modules, if only to check their update status.

  To avoid this problem, design your build procedure to be modular. Rather than rebuilding stable targets, set up your build procedure to group stable components and unstable components into separate targets. Then build only the unstable targets.

Multiple CMS libraries keyed to each logical or functional subsection (or **facility**) of the application help to avoid the performance problems of many elements. However, multiple CMS libraries create problems of their own. For example, CMS does not have a cross-library referencing feature. This means that CMS features such as groups and classes no longer easily encompass the entire application.

To offset this, you can use the Search List facility of CMS to help with managing multiple libraries. As you subdivide libraries to improve performance, though, there is a tradeoff between performance and ease of library management.

For operations that are common, and need to be faster than is possible with a command procedure, you might want to write programs that use CMS callable routines to perform the operations. Refer to the *Guide to HP Code Management System for OpenVMS Systems* for more information.

The directory structure for a large project can still be based on Figure 3–1. However, each facility of the application will have its own [PROJ] area as part of the structure. The entire application will come together at major application milestones or base levels.

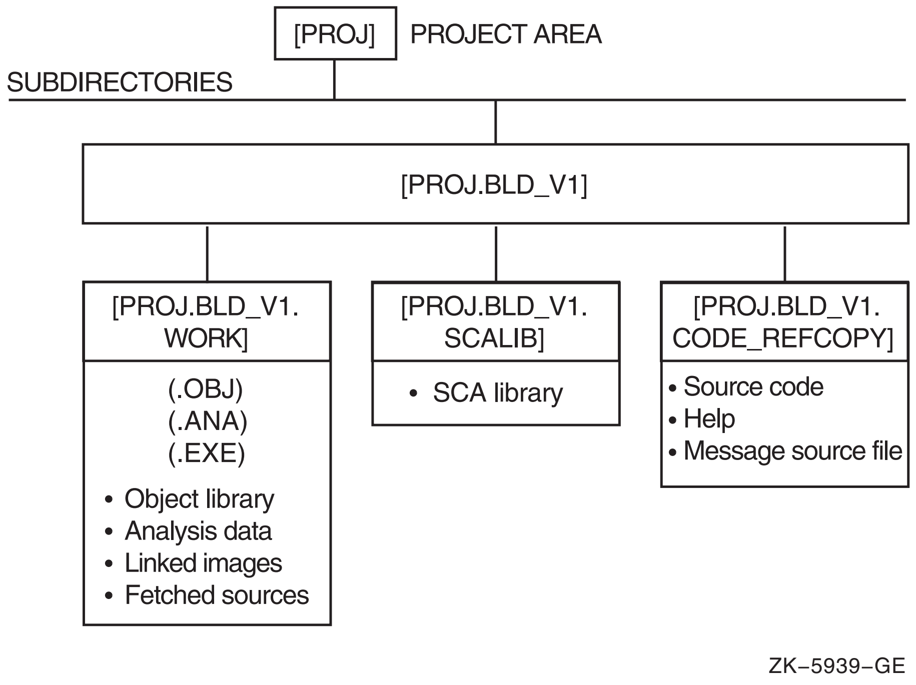
Note

Although CMS can store nontext files, such as binary sources, you might choose not to store files that can be produced from other stored files (for example, object files that can be produced from source files). This saves disk space.

3.2.2 Setting Up the Build Directory

This section describes the second area of your project directory structure—the build directory. You can set up the build areas as parallel subdirectories that correspond to the initial prototype build directory. Figure 3–3 shows a build directory structure using this practice.

Figure 3–3 Build Directory Structure



In Figure 3–3, [PROJ.BLD_V1] is a directory area devoted to the periodic building of your application, where *1* represents the version number of the software. (As your project develops, you can create subdirectories to store successive versions: [PROJ.BLD_V2], [PROJ.BLD_V3], and so on.) This directory area contains the following subdirectories:

- [PROJ.BLD_V1.WORK]—Area for carrying out your project builds. This area is described in more detail in Section 3.2.2.1.
- [PROJ.BLD_V1.SCALIB]—Area containing your project-wide SCA library system. This area is described in more detail in Section 3.2.2.2.
- [PROJ.BLD_V1.CODE_REFCOPY]—The CMS library reference copy area. This area is described in more detail in Section 3.2.2.3.

The end of this section describes an MMS description file you can use to automate the building process itself.

3.2.2.1 Work Area for Carrying Out Your Builds

During builds, the [PROJ.BLD_V1.WORK] directory stores the source files fetched from CMS and the .EXE, .OBJ, and .ANA files produced during compilation and linking.

You might want to group all the .OBJ files into an object library (.OLB to simplify your linking procedures). You can link against the object library rather than all the individual object files. See the *OpenVMS Librarian Utility Manual* and the *OpenVMS Linker Utility Manual* for more information about OpenVMS object libraries and how the linker uses libraries.

You need to determine how often you want to make major builds, and which builds you want to keep. You can establish procedures where you rebuild or update only at predetermined times. Alternatively, you can update the shared build directory using CMS sources as soon as all your developers replace the necessary CMS elements.

The advantage to keeping your build versions intact is that it allows your team easy access to the files as they existed when that build was carried out. However, because of the disk space needed to store all the contents of a build, you probably need to limit the number of build versions that you actually keep.

You might find it useful to keep one major build that you frequently return to for support and maintenance purposes. You do not need to keep intermediate builds because of the class feature of CMS, which allows you to freeze the sources for any build. By fetching the source files in a given class, you can recreate a build from any development stage using the original source files in CMS.

3.2.2.2 SCA Library System

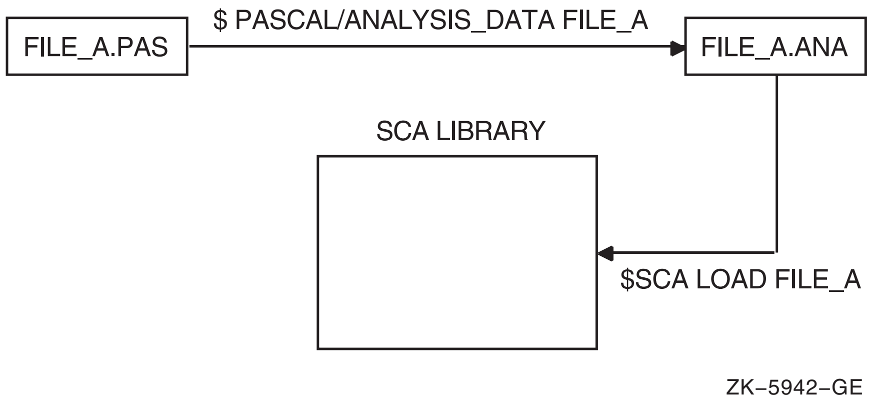
You can make your project-wide SCA library a part of the build tree hierarchy (for example, [PROJ.BLD_V1.SCALIB] as shown in Figure 3–3). To provide an index of detailed source information for your project, you can arrange your SCA library system to consist of the following:

- A comprehensive project-wide SCA library that reflects the development of the entire software application
- Individual SCA libraries that correspond to local sources used by team members during individual development tasks

During the compilation stage, compilers produce .ANA files containing modules of analysis data. These .ANA files need to be loaded into an SCA library. SCA then accesses the library for this information when carrying out any requested queries.

Figure 3–4 shows how to generate the analysis data from your source files. In this case, a Pascal file, when compiled with the /ANALYSIS_DATA switch, produces a corresponding .ANA file. Using the SCA LOAD command, you can place the analysis data into the SCA library's database.

Figure 3–4 Filling an SCA Library



Project-Wide Development

By using the /SCA_LIBRARY switch on the MMS command line that initiates a project-wide build procedure, MMS automatically compiles your files with the /ANALYSIS_DATA switch and loads the .ANA files into the SCA project-wide library. This produces a project-wide SCA library that reflects the latest developmental work as maintained by the CMS library.

Chapter 9 contains an example of an MMS description file, which shows how to set up your description file to take advantage of this feature.

Incremental Development

Your team will benefit from an SCA library structure in which team members independently manage the libraries that are small and change frequently. This is important for extremely large projects; individual developers benefit from updating only their local SCA libraries on a regular basis, and not the project-wide SCA library. It will be to your advantage to create functionally distinct project-wide libraries to speed access to these libraries.

Developers can start each work day by reserving files from the CMS library to their local work area where they can continue development. After compiling the files, they can use their local SCA library to store the newly created analysis data.

When using SCA to query the application, developers can take advantage of SCA's ability to treat its multiple physical libraries as a single virtual library (combination of physical libraries), shown by the following SET LIBRARY command:

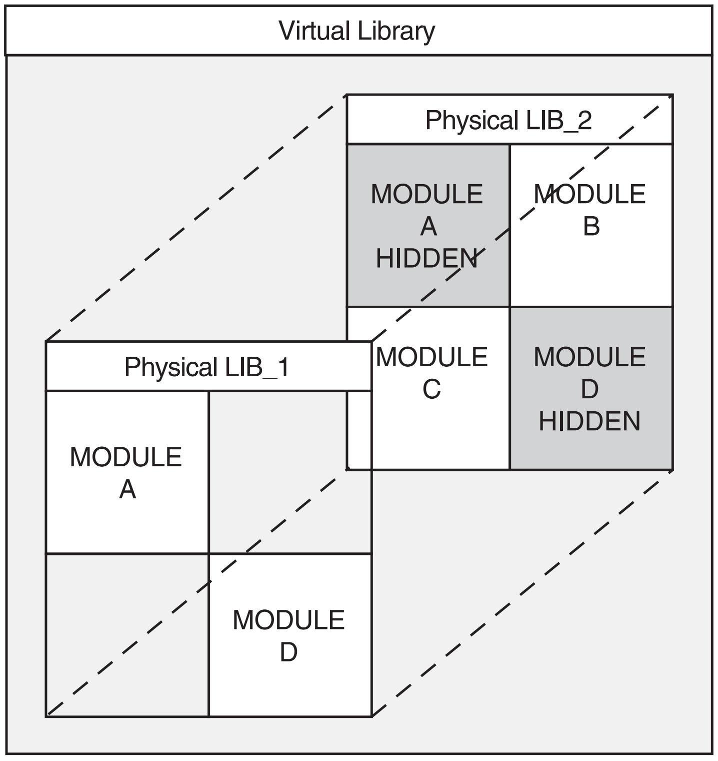
```
$ SCA SET LIBRARY LIB_1,LIB_2
```

This example sets up a library search list (similar to a OpenVMS directory search list) for SCA to follow. For instance, the first SCA library on the list can be a developer's small local library. The last library on the list is the project-wide SCA library. You can designate more than two libraries on the search list.

Every module of the first library in the search list is included for viewing in the virtual library. Each module in each subsequent library on the list is then compared with the modules previously selected. If a module is uniquely named, it is included for viewing in the virtual library; otherwise, it is excluded (because it has already been defined).

Figure 3–5 shows how a virtual library concatenates the physical libraries on a library list. As a result of the previous SET LIBRARY command, you create a virtual library containing four modules: A and D from Physical LIB_1 together with C and B from Physical LIB_2.

Figure 3–5 Physical and Virtual SCA Libraries



ZK−5946−GE

When making queries, you have access to the information in any of these libraries. Because SCA looks into the libraries in the order specified in your library list, you can efficiently access small portions of your application, unless the sources' analysis data can be found only in the libraries further down on the list. When SCA joins the physical libraries into a virtual library, only modules found in the first libraries will be accessed by SCA even if they are duplicated in libraries further down on the library list. Only duplicate modules *with the same name* are hidden. Consequently, when you modify source modules, the library list will ensure that only the local SCA library that corresponds to these source modules will be updated. This lets you use the virtual library without concern for the fact that the project-wide library has not been updated.

A benefit of this procedure is that it prevents unintended changes to the project-wide SCA library. If you do not specify which library you want modified, SCA modifies the first physical library in the list. In Figure 3–5, if Module C is modified, SCA will copy that module into Physical LIB_1 and incorporate the modifications; Module C in Physical LIB_2 will remain unchanged. When you update an SCA library with the LOAD command, only you have access to the library until the load process is finished. If many people are using the same library, and that library is subject to frequent updating, access will be restricted. In general, loading is a slow procedure (comparable in time to a compilation) and should be done in batch mode for a project-wide library. In contrast, local library updating is more manageable. An SCA library supports queries by multiple users.

As team members complete programming tasks, they need to update the original source files and the project-wide SCA library. By replacing elements in CMS, team members update the original sources. The next run of the project-wide build procedure generates updated SCA information in the form of .ANA files, which in turn can replace the outdated source analysis data in the project-wide SCA library. These procedures produce a project-wide library that reflects periodic modifications. By using local libraries, programmers efficiently use SCA on immediate development tasks. Finally, when the team reaches predefined milestones and begins a major application build, they recreate the entire project-wide library. In this way, the team ties current build sources and development status to specific milestones. In all cases, the team creates or modifies build procedures to automatically and selectively update SCA libraries at appropriate times; for example, when compiling local source files or during a major application build.

For more details on SCA, see the *Guide to HP Source Code Analyzer for OpenVMS Systems*.

3.2.2.3 Reference Copy Area

A reference copy area is a directory you create in which CMS maintains a copy of the latest generation of the main line of descent of each element. You can use a reference copy area as an alternative storage area (for example, [PROJ.BLD_V1.CODE_REFCOPY] in Figure 3–3). Rather than MMS fetching files directly from CMS, MMS accesses the files in a reference copy area and builds up from all the sources located there.

One advantage to building from a reference copy area is the increase in speed during the initial part of the build procedure. You instruct CMS to duplicate every updated source in the reference copy area. CMS then functions as a reliable backup with a detailed history of all transactions. A CMS library only updates one reference copy area per library at a time. If you want to keep more than one reference copy area intact as your project moves from version to version, you can either direct CMS to use a new reference copy area or use multiple CMS libraries.

Whether you use the reference copy area for builds depends on the work procedures of your project. You also need clear copies of your sources for PCA, and for LSE when used with SCA. However, these clear copy sources can be drawn from a project work area as well as from the reference copy area.

The main disadvantage of doing builds from the reference copy area is the lack of historical tracking. However, you can use the CMS VERIFY command to check for discrepancies between elements in the CMS library and corresponding files in the reference copy area. If you choose to use a reference copy area for building, you must restrict access to this directory so people outside your group cannot copy your sources, and members of your group can copy read-only versions of the files. Your developers must reserve the files from CMS if they want to edit a file. This generates a history of the file transaction.

Another disadvantage of building from the reference copy area is that you cannot recreate earlier versions of your application: the reference copy area maintains only the current versions of your sources. A final limitation of this feature is that it works only with the main line of element descent in the CMS library. You cannot use this feature for variant source development. Even with these limitations, the reference copy area provides a useful intermediary between the CMS library and individual work areas.

There are advantages to carrying out build procedures that rely on the CMS library rather than a reference copy area. MMS can automatically carry out a build based on a comparison of sources in a local directory and in the CMS library. This is useful when a developer wants to see the effect of modified modules on the entire application before replacing the modules to the CMS library. In this case, MMS can rebuild the application using the new modules only. This is the build procedure followed in the scenario described in Chapter 9.

3.2.2.4 MMS Description File

MMS helps you with the build process by automatically accessing the source files you store in a CMS library. It follows the sequence of dependencies among your source files, includes all the necessary modules in the build, and ensures that your build uses the current sources. You can use MMS to build not only your code applications, but your documents as well.

You describe the structure of your application to MMS in a description file. Because you will update it as your project evolves, the description file should be stored in your CMS source library. It can be incorporated into a CMS class as part of a list of your files at each successive base level. When you need to rebuild a class (for instance, an older base level), the sources are stored in that class along with the MMS description file.

Your use of MMS will affect what you decide about your build procedures and the frequency of your builds. MMS allows reliable incremental building; that is, builds that incorporate only the changes made to modules since the last build. Because of this feature, you might want to carry out nightly incremental builds, leaving builds that use all your sources and regenerate all intermediate files only for milestones during your project's development. The advantage in building incrementally is that you avoid the additional time needed to build from all the sources when only a relatively small percentage of code actually changed since the last build.

Chapter 9 contains an example of setting up an MMS description file. See the *Guide to HP Module Management System for OpenVMS Systems* for a complete description of MMS.

Considerations for Large Projects

For large projects, the functionally defined individual facilities can build incrementally, as described previously in this section. Coordinating full application builds involves the following steps:

1. Determine stages of development (milestones or base levels) at which updated facilities contribute their sources.
2. Create classes within each facility that represent the current sources for that facility.
3. Fetch from those classes to a storage area that has limited access.
4. Use a master build procedure to assemble the sources into the executable application.

The way a project implements these steps varies depending on the demands of the project. For example, the person who is the configuration manager can layer another CMS library on top of the facility libraries to monitor the fetched classes. This library will not be an active development library, but a storage facility with the feature of detailed history for transactions. New classes at future stages of development will be added to this library, and the application builds will proceed from there.

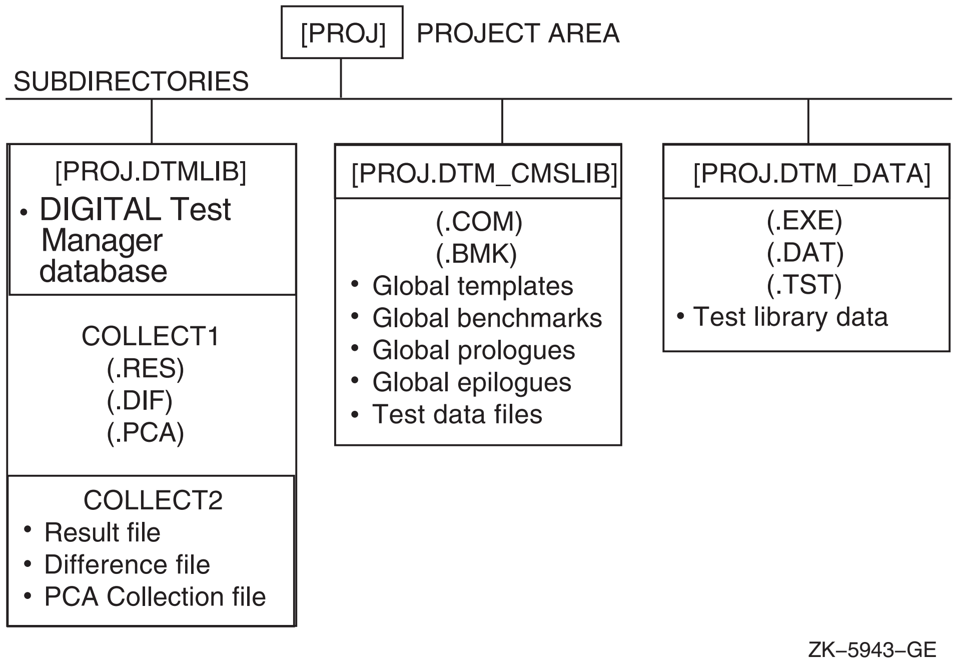
3.2.3 Setting Up the HP DIGITAL Test Manager Storage Area Directory

As part of your directory organization, you must set up test directories for the HP DIGITAL Test Manager storage area. Setting up test directories early in your project provides the following advantages:

- Your organizational directory structure is almost complete and you can easily add components as your project grows.
- Your programmers can begin writing tests early in the project life cycle; this is particularly useful on large-scale projects.
- Your team can produce tests for undeveloped code; these tests will make it less likely that some design functionality will be lost as the project develops.
- Your developers have a mechanism for intermediate testing before the code has reached refined development.

As shown in Figure 3–6, your HP DIGITAL Test Manager directory organization might require three storage areas.

Figure 3–6 Directory Structure of HP DIGITAL Test Manager Libraries



The three areas are as follows:

- A HP DIGITAL Test Manager library for test results

  This library is a separate subdirectory (in Figure 3–6, it is designated as [PROJ.DTMLIB]). This library stores HP DIGITAL Test Manager test description information. This test information, located in a HP DIGITAL Test Manager database, provides HP DIGITAL Test Manager with the following:

  – The name of the test
  – The name and location of the file that executes the test (template)
  – The name and location of any procedure that runs before or after the test (prologue and epilogue)
  – The name and location of the file (benchmark) against which to measure the test results
  – The values for any symbols or logical names that HP DIGITAL Test Manager sets up for each test or collection of tests while executing (HP DIGITAL Test Manager variables)

  - Any associated filter procedures

  The HP DIGITAL Test Manager library also stores the results of your tests; these results can also include PCA test coverage or performance results if HP DIGITAL Test Manager is used to control PCA data gathering. In this case, because HP DIGITAL Test Manager controls the data collecting, it automatically places the PCA results in the HP DIGITAL Test Manager library. After you use HP DIGITAL Test Manager and PCA results, if problems occur you can purge the result files as part of your development process. Note that the collections shown in the HP DIGITAL Test Manager library ([PROJ.DTMLIB]) in Figure 3–6 are generated automatically by the HP DIGITAL Test Manager.

  When you collect data using PCA alone rather than through HP DIGITAL Test Manager, the PCA data file can reside temporarily in your work area or default directory. After analyzing the data collected by PCA, you can discard this file.

- A CMS library for HP DIGITAL Test Manager test files

  You can store templates, benchmarks, test data files, prologue files, and epilogue files in this library. (In Figure 3–6, it has the designation [PROJ.DTM_CMSLIB].) As you complete your test cycles, you can store validated test output as benchmarks for future tests. CMS organizes and tracks these files, ensuring the integrity of past and future test performances.

  As an alternative, you can create separate subdirectories for your HP DIGITAL Test Manager test files rather than using a CMS library. You can also create multiple subdirectories for this purpose that correspond to important stages in the product's development. The advantage will be some gain in speed; however, you will use more disk space and not have any of the history tracking features of CMS.

- A HP DIGITAL Test Manager subdirectory for test data

  Whether you need this particular subdirectory depends on your application. You might choose to use this directory to store input files needed for generating output files for tests. For example, a linker project might regularly, as part of its test procedure, read an .OBJ input file to generate an .EXE output file. Although CMS can store these non-ASCII files, it might be an organizational aid to maintain this subdirectory for testing purposes. In Figure 3–6, [PROJ.DTM_DATA] stores these types of data files.

4 Managing a Project in the OpenVMS Environment

This chapter contains procedures for the project leader of a software project in the OpenVMS software development environment. The topics included in this chapter are as follows:

- Project standards—Ways to establish and maintain standards in the following areas:
  - Application design
  - Code
  - Testing
  - Software performance
- Using logical names to establish a context for the project
- Communication management and reporting mechanisms
- Project and individual build procedures
- Customizing the LSE environment
- Setting up tests
- Analyzing performance and coverage
- Field test management
- Final steps in a project

The procedures presented in this chapter are suggestions only. They are not rigid guidelines that must be followed to use the OpenVMS programming environment correctly. You might want to modify some of these procedures to suit your specific needs.

4.1 Project Standards

As project leader, you establish standards for your team, such as design, coding, and testing standards. When everyone adheres to these standards early in the project's development cycle, you avoid confusion and conflict later. Standards also provide a consistent framework for team members joining the project midway through the development cycle, and enable them to learn about the project faster.

4.1.1 Design Standards

Part of establishing design standards means getting team members to agree and adhere to the format of the application's designs. The designs should consider not only the specifics of how to implement functional areas, but also the context of your application. Additional considerations include evaluating dependencies outside of the project and deciding what languages you will use to implement a particular functionality.

To make development decisions easy in the future, record design features you decide to keep and alternatives that you discard.

4.1.2 Coding Standards

Set up coding standards, such as consistent naming conventions for modules and routines, during the early stages of your project. Consistent naming conventions provide the following advantages:

- Faster identification of code elements by developers
- Easier access to files, directories, and so on by means of wildcard characters
- Faster learning for new members of your team
- Faster work with the OpenVMS Debugger
- Easier maintenance of software in the future

You might want to set up periodic code review meetings. You can use these meetings to inspect sections of code to see that they meet the team's standards, and that the code works efficiently.

The following examples show naming conventions for modules, routines, and variables in an application. (See the *Guide to Creating OpenVMS Modular Procedures* for more detailed information on coding standards.) The following table lists the names used in the examples.

Name	Meaning
PROJ	Facility prefix for the routine
DB	Indicates a database routine
GT	Indicates a global text declaration
k	Indicates a constant variable
g	Indicates a global variable

- Routines: PROJ_DB_CREATE_OBJECT

  To make it easier to locate a set of related routines, group your routines into facilities. Providing related routines with a common facility prefix organizes the routines. The remainder of the name indicates the type of function of the routine.
- File name: PROJ_DB.PAS

  The DB designation shows which type of routines are in this file.
- Modules: PROJ_DB

  Module names are identical to file names, except module names do not have extensions.
- Variables:
  - *proj_gt_username*
  - *proj_k_maxfile_count*

    The *k* designates a constant, whereas the rest of the name describes the function; in this case, a value for the maximum number of files to be used. Unlike the last example, this is a local constant because it does not have a *g* immediately before the *k*.

4.1.3 Using LSE Templates

LSE provides templates that you can use to enforce consistency in your coding and commenting conventions. You can modify constructs, menus, and descriptions within existing definitions. In addition, you can create your own templates for a language if LSE does not support that language directly. See Section 4.6.1 for procedures to create templates.

Another example of coding standards is the use of coding formats. Built into LSE coding constructs is a consistent indenting format for each of its supported languages. By using LSE in your project, you will ensure a consistent format that makes your code much easier to read. LSE allows you to customize your own language templates so they adhere to site-specific coding standards. This benefits not only your team during development, but also new programmers who might maintain or update your team's work. See Chapter 7 for an example of using LSE to help set up coding standards on your project.

You can create LSE templates for your documentation; for instance, design or specification formats. These templates can be specific to your project or tailored to the needs of your company or primary customer. LSE allows you to customize the language templates provided with the LSE kit. Each individual developer can make and save modifications. These templates can be available to each developer, or they can become a permanent part of the environment files provided with your system.

LSE also provides the CHECK LANGUAGE command to analyze the definitions associated with a language. When you use this command on a language, LSE detects and reports on the following:

- Undefined tokens
- Undefined or unreferenced placeholders
- Package routines with the same name as placeholders
- Package parameters with the same name as tokens
- Parameters defined with the same name in multiple packages
- Routines defined with the same name in multiple packages
- Invalid topic strings

If LSE detects any of these conditions, they are reported in an editing window and can be sent to an output file. For more details, refer to the *Guide to HP Language-Sensitive Editor for OpenVMS Systems*.

4.1.4 Testing Standards

Realistic testing goals include testing that starts with the building of tests during the specifications stage, or at least early in the development process, and runs through to the end of the cycle. By testing as your project grows, you will not omit a particular piece of code from testing as future work layers new functionality over it. In addition, you will discover errors while the code is still fresh in people's minds; waiting to test can result in developers having to relearn code.

Your tests should cover every user feature that is part of your application, along with all user error conditions. Additionally, you need to ensure that all interactions between your application and other software applications perform as described in your specifications. This type of testing (one that tests against predicted external view) is referred to as **black box testing**, and is most effective in advanced stages of development.

White box, or **unit testing**, is an internal approach to code testing. By creating driver programs to exercise all the decision branches of a piece of code, you can create tests that approach 100 percent coverage of asynchronous applications. This type of exhaustive testing, in combination with black box testing, meets the needs of those projects that demand extensive testing.

A team using these types of testing benefits from a feature of HP DIGITAL Test Manager: when used in combination with PCA, it produces an annotated listing of all code lines exercised by the test set. This shows how much of the code is being exercised by the tests.

4.1.5 Performance Standards

After writing a certain amount of code, programmers will be able to run a simple set of tests to compare against performance benchmarks. PCA enables you to improve overall performance by helping you find performance bottlenecks in your code.

4.2 Using Logical Names

Logical names serve two functions:

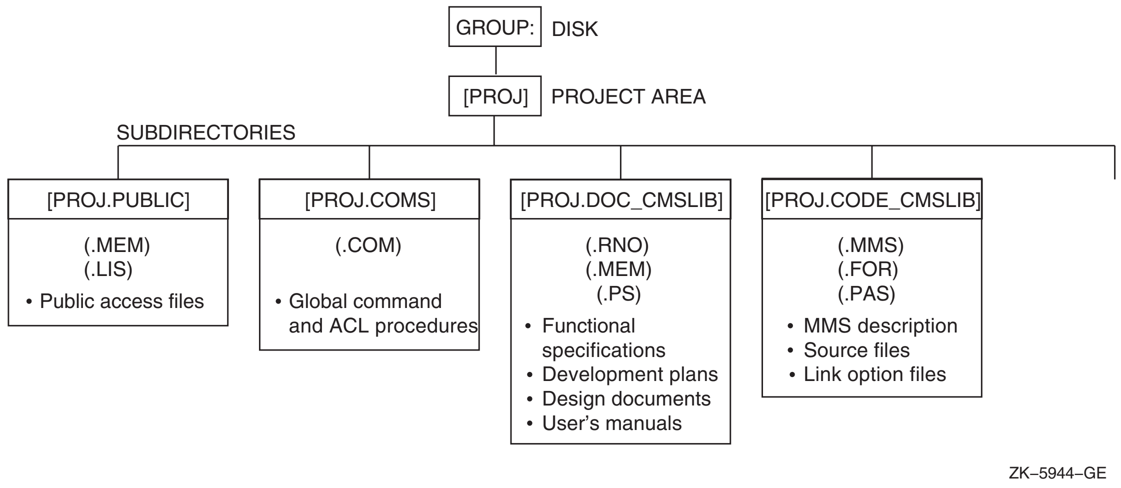
- To define commonly used files, directories, and devices to have short, meaningful logical names. Such names are easier to remember and type than full specifications.
- To safeguard the project against changes in the computing environment. For example, if the project moves to a new disk, or to another OpenVMS system, all you need to do is change the definitions for the logical names. No changes need to be made to the application itself. Using logical names can save last-minute changes when installing the application.

As part of the initial setup of your project, you want to implement the context for the project from a command file. Store the file itself in an area such as [PROJ.COMS] as shown in Figure 4–1.

You include logical name definitions in this command file. You can define names that your team uses frequently in this common command procedure that can be invoked by the individual login procedures of your team members, or define these logicals in the system startup file with the /SYSTEM switch so they only are defined once. See Section 5.4 for an example of this procedure.

As another alternative, you can define your logical names in the group logical name table. The group table contains logical names available to all users with your UIC group number. See the *OpenVMS DCL Concepts Manual* for more information on logical name tables.

Figure 4–1 Initial Storage Areas for a Typical Project



The following table lists examples of naming conventions for your directories.

Example	Directory
PROJ_PUBLIC	Public access area
PROJ_CMS_CODE	CMS library for source code
PROJ_BLD	Current version of build area
PROJ_DTM	HP DIGITAL Test Manager library

You can also use logical names to save time as your project evolves. You are likely to create multiple versions of your build areas along with corresponding build subdirectories. You can have your command procedure update your logical name for the build subdirectory by redefining it to the most recent build version.

4.3 Communications Management and Status Reporting

This section describes communication management both within and outside your project, and describes the following three OpenVMS tools that are designed for electronic correspondence, computer conferencing, and automated project status reporting:

- OpenVMS Mail Utility (MAIL)
- DEC Notes
- DECplan

Note

Documentation reviews are described in Section 4.3.4.

4.3.1 Using MAIL

In addition to face-to-face contacts and telephone conversations, you can use MAIL. By forwarding documents and memos to everyone in the team, you enhance team awareness and cohesiveness. Using MAIL, you can track the status of a project. Usually team leaders require monthly reports from the team members. With MAIL, team leaders can consolidate these reports and forward them to managers who are tracking the project's schedule.

You can create a mail subdirectory for your project to hold project-related messages. In the directory hierarchy, this would be a subdirectory of your top-level directory: [PROJ.MAIL].

4.3.2 Using DEC Notes

If your project team has many members, and especially if the members are not in the same location, you might want to use DEC Notes, a conferencing tool.

A goal of any project during development is to gain feedback from its users. You can speed this process by using DEC Notes to create a conference for your project that people outside of your group can access. You can create multiple conferences with different titles and functions, and with varying degrees of user access.

DEC Notes conferences can be used by anyone on a network and can promote communication to a larger audience than MAIL. DEC Notes keeps a record of all entries and replies. At a later date, you can use some of this online information to document changes to the software. In this way, DEC Notes is a useful tool for providing **traceability** for group decisions on project design, problem reporting and correcting, and so on. Some ways to use DEC Notes conferences to facilitate your project's goals include the following:

- Problem Forum Conference

  Users outside your group can list errors or problems that they have found. Someone from your group can suggest remedies, or if necessary, make changes to the software. Other users who read the conference can also suggest remedies or contribute their own opinions about the problem. Keywords can be used to open, close, assign, and prioritize problem reports.

- Wish List Conference

  This type of conference limits the entries to features that users would like to have as part of the software functionality.

- Design Conference

  This type of conference can be set up by the project team to discuss design issues for a project.

- Restricted Conference

  You can set up a conference accessible to a limited group of users. For instance, if your group is large or its members are not near one another, you can create a private conference for this group only. The conference can also be open to a limited number of people who are not part of your group, yet have a particular interest or a supervisory role in the project.

  This type of conference can do more than distribute information or solicit feedback; it can be used to speed your review procedures. For instance, you can set up a private conference for requirements and another for specifications. Your reviewers can read the information online and respond within the conference. In addition, your group can extract comments from the conference and incorporate these comments into documentation.

Your group can combine your use of MAIL and DEC Notes with traditional means of communication to create an effective and customized communication environment.

4.3.3 Using DECplan

DECplan allows you to perform integrated time and project management activities throughout the software development life cycle. For time management, you can use DECplan to manage your personal calendar, record your activities, and coordinate your commitments to others. For project management, you can define work, assign resources, schedule projects, control costs, and display progress through charts and reports.

DECplan is designed to handle task assignment negotiation, scheduling, and individual status updating and is an important communications tool. The *Guide to DECplan* has more details on using DECplan.

If you have DECplan installed on your system, you will find it useful for monitoring project progress and for project management in general. DECplan automates project management activities throughout the software development life cycle.

You use DECplan to plan, monitor, and control projects. Using DECplan, you define tasks and their dependencies, estimate effort, assign resources, schedule the project, track progress and cost, and display status through charts and reports.

You display information stored in a project planner through a **view**. Each view is a separate window allowing you to display, edit, and print information. Views enable you to see planner information displayed simultaneously in a variety of formats. For example, you can display your daily calendar while viewing a list or table of assigned tasks as an aid to daily planning. DECplan views are summarized in the following list:

- Calendar views for time management
  - Day View—Your daily calendar
  - Month View—A one-month calendar
  - Year View—A one-year calendar
- Table views—Customized spreadsheet-like displays of information about meetings, tasks, resources, and other items
- Work breakdown structure (WBS)—Graphical breakdown of work to be performed
- Precedence network (PN)—Graphical representation of tasks, milestones, and dependencies
- Gantt chart—Bar chart showing work to be performed in relation to time
- Resource loading chart—Chart showing the percentage of time a person or group is expected to work on a task or set of tasks
- Rate charts—Set of charts showing comparisons of actual and expected effort, cost, or earned value
- Report Layout Editor—Tool for modifying standard report formats and creating customized reports

These views, and the operations available from these views, support all aspects of project management. The following list describes what views and operations you would use on a typical project:

1. Create a project planner from the initial DECplan display. This planner stores information about your project.
2. Break down work into distinct tasks using the work breakdown structure (WBS). The WBS is a tree structure that helps you organize, define, and display project tasks. The WBS also allows information such as effort and cost to be rolled up for reporting purposes.
3. Use a table view of tasks to enter predicted effort and other information for tasks.
4. Define dependencies for tasks and milestones using the precedence network (PN). The PN shows the order in which tasks must be completed and milestones must occur.
5. Schedule the project using the critical path method (CPM) from any view. This schedule shows you the scope of the work and can help in making staffing decisions.
6. Assign resources to tasks from any view. Each resource can have its own personal planner. You can also define local resources within the project planner.
7. Schedule the project using the constrained resources method from any view. This schedule takes into account assignments to tasks and information about resources. Resources are not scheduled to do more work than they are available for, providing a more realistic schedule than a CPM schedule.
8. Negotiate or mandate assignments to resources from any view. This causes task assignments to appear in the applicable resource's planners.
9. Specify cost rates, define cost accounting periods, and calculate projected costs for the project from any view.
10. Track progress with time charges. Each resource creates time charges against specific assignments as they perform work and posts these charges to the project planner.
11. Display project status with charts and reports. You can use standard charts and reports or create your own through customization.
12. Make adjustments to the project based on changes in resource availability, scope of work, or external dependencies and reschedule as required.

See the *Guide to DECplan* for details on using DECplan.

4.3.4 Documentation Reviews

Because accurate and timely user documentation is critical to a product's success, the team must build in adequate review procedures. The review procedures depend upon your team's size. Smaller teams can rely on informal means of review, although formal reviews are still necessary. Larger projects might need defined responsibilities and more careful oversight.

All the members of your team should review the complete user documentation. This overall review can combine with having individual developers responsible for specific sections of the documentation.

Formal reviews supplement the informal and ongoing review process. The result is documentation that is technically accurate and complete when the product is shipped to customers.

4.4 Extending the Library Structure

As your project moves into the implementation stage, you might need to re-evaluate the status of your directory structure. For instance, you might find that a single CMS code library is inadequate. It is recommended that you start with one CMS library for source code and break it up if you find that you have library conflict problems.

You might want to rearrange your directory structure to reflect ongoing development needs. You might find that by adding new storage areas, you can better access specific groups of files.

4.5 Establishing Project and Personal Build Procedures

Section 3.2.2 described some of the considerations for determining your team's build procedures. You will need to finalize a number of decisions before you begin to implement your code:

- Whether to use incremental builds or build from the sources; that is, whether to use a reference copy area or allow MMS to fetch from CMS directly
- How often to create builds
- How thoroughly to automate your build procedure

4.5.1 Individual Build Procedures

Individual storage areas are typically less complicated than the overall project structure. For example, individual developers probably do not need a personal HP DIGITAL Test Manager library. Many developers rely on the project CMS libraries to provide a storage area, although some developers set up personal CMS libraries to track and organize their own work. Developers have a private work area and possibly a separate build area; they might also need a local SCA library.

You will want to establish procedures for how members of your team will work on particular files. Your team members can follow these steps:

1. Identify a problem (for example, a routine that needs to be modified).
2. Use SCA and its project-wide library to locate the routine and source file.
3. Use LSE commands with SCA to bring a read-only copy of that file into an LSE buffer (from the project work area or a reference copy area—both set to read-only access).
4. Determine whether any changes need to be made in the file.
5. Use the LSE RESERVE function to reserve a generation of the corresponding element from the CMS code library.
6. Use LSE to modify the file.
7. Use SCA to see if any other code is depending upon the code you have changed. If so, you can reserve that element directly from CMS and make the needed modifications.
8. Compile the modified files using the LSE COMPILE REVIEW function.
9. Review the compilation errors with LSE until the file compiles correctly.
10. Link the image.
11. Carry out any test procedures using HP DIGITAL Test Manager.
12. Replace the modified file into the CMS code library.

4.5.2 Project Build Procedures

Different members of your team will make varying degrees of progress on their work as they modify files. Because modified modules affect the work of other team members, you benefit by keeping your entire application up to date. Otherwise, you risk accumulating discrepancies between modified files and outdated files.

You can automate the build process with MMS, along with automatically running your HP DIGITAL Test Manager test system. By analyzing your tests with PCA, you can have information supplied about how thoroughly your tests exercise your application. You can also use your tests to analyze and track performance problems in your application. By setting up a HP DIGITAL Test Manager epilogue, you can have a full status report on the test collection forwarded to everyone on the team. This automates repetitive tasks while enhancing overall team communication. For examples of using these tools together, see Chapter 11.

4.5.3 Access to SCA Libraries

When individual developers work on a small number of modules, they can work within a virtual library consisting of their local SCA library and the project-wide library. The SCA New Library... function creates a local library specifically for the modules under development. These modules are then loaded into the local SCA library ([USER.BASELEVEL_1.LIB1] in Figure 4–4). (If there are substantial changes to code or to routine calling sequences, you might want to use the COMPILE $/ANALYSIS_DATA command in LSE instead of the Compile Review function.) When you create a new library, you can specify that the new library be placed before the existing project-wide SCA library ([PROJ.BLD_V1.SCALIB] in Figure 4–4) on the library search list. All of these actions take place within the framework of the individual developer's work area.

Figure 4–2 shows the New Library dialog box, in which you specify the name of the new library and where it is placed on the library search list.

Figure 4–2 SCA New Library Dialog Box

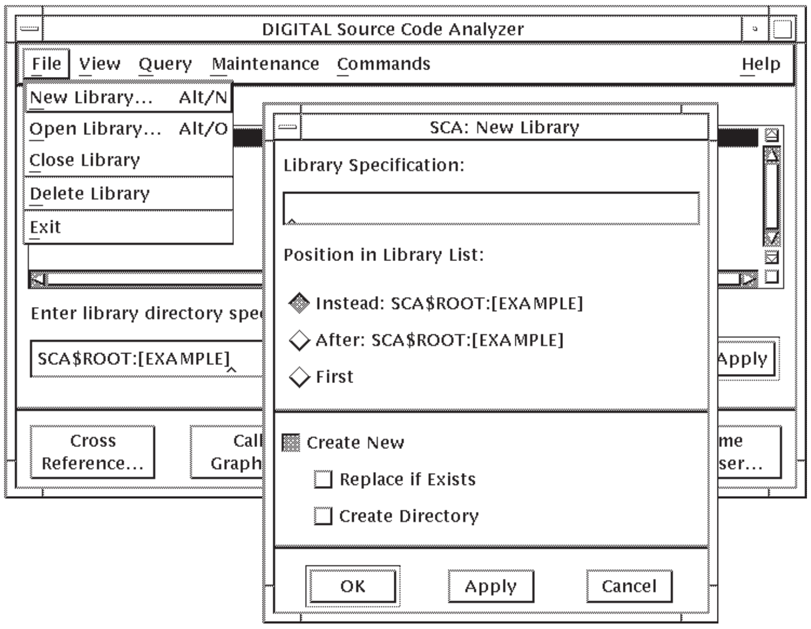
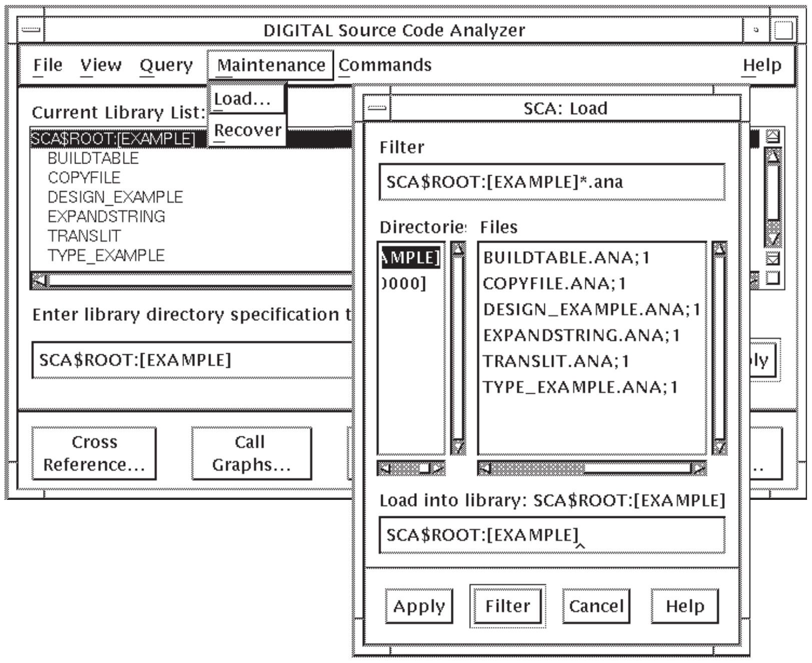


Figure 4–3 shows the Load dialog box, in which you specify the names of the analysis data files to be loaded into an SCA library.

Figure 4–3 SCA Load Dialog Box

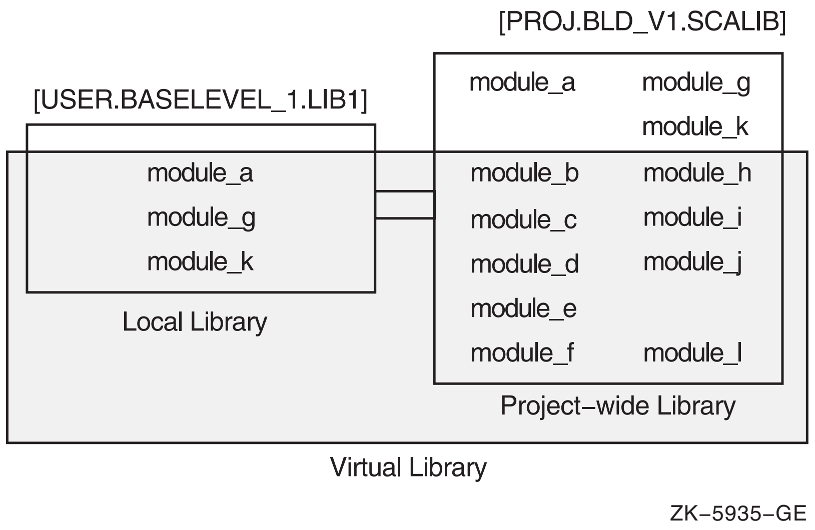


The following example shows these commands:

```
$ SCA
sca> CREATE LIBRARY DISK1:[USER.BASELEVEL_1.LIB1]
sca> LOAD DISK1:[USER.BASELEVEL_1.SOURCE]MODULE_A.ANA,MODULE_G.ANA,MODULE_K.ANA
sca> SET LIBRARY/AFTER DISK1:[PROJ.BLD_V1.SCALIB]
```

Figure 4–4 shows the relationship between the two physical libraries.

Figure 4–4 Working SCA Libraries for Developers



4.5.4 Access to Source Files

Together, LSE, SCA, and CMS provide ways to manage access to files. The goals are as follows:

- Easily display source code corresponding to variables and routines
- Readily access modifiable files when you are ready to make changes
- Prevent inadvertent changes to sources

All CMS transactions generate history information, and keep the sources intact over their development cycle.

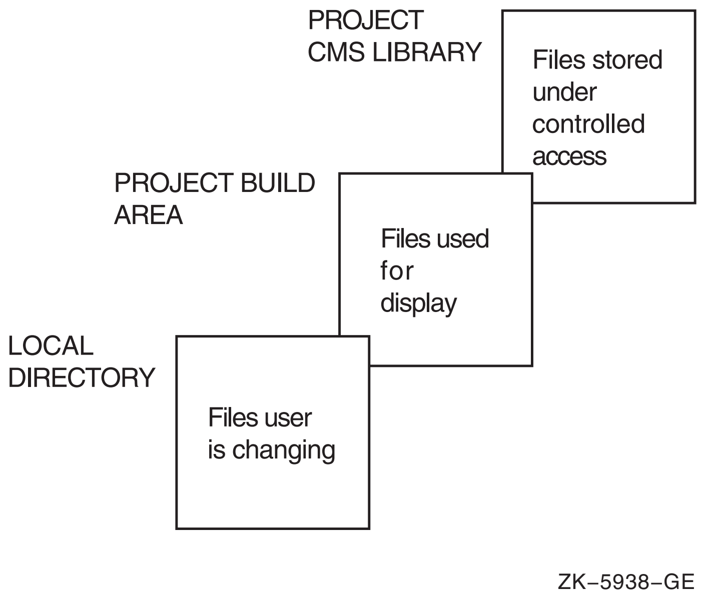
LSE can help manage source code by accessing your storage areas, including your CMS library. The LSE file-access commands let you create a search list of directories. Typically, this search list might consist of a local work area, a project build area, and, optionally, a CMS library. The following commands demonstrate how you do this:

```
LSE Command> SET DIRECTORY SOURCE [], PROJ_BUILD, CMS$LIB
LSE Command> SET DIRECTORY READONLY PROJ_BUILD
```

The first command designates the order in which you want LSE to look for a source: your local directory first, the project build area second, and the CMS library last. The second command ensures that any sources drawn from the project build area will be read only. Section 5.4 shows how to carry out a similar procedure using logicals in a LOGIN.COM file. Figure 4–5 represents the directories that correspond to the search list.

CMS objects can have access control lists (ACLs) attached to them. This situation should be considered when reserving and replacing elements and executing the commands themselves. You might want to use CMS ACLs to control access to library elements rather than using the LSE SET DIRECTORY/READ_ONLY command. While this command provides a mechanism for preventing the accidental modification of files in a directory or CMS library, it does not provide the level of protection that ACLs can. By attaching OpenVMS ACLs to directories and CMS ACLs to CMS libraries, you protect objects that should not be modified (except by selected people).

Figure 4–5 Source Code Management



CMS enables team members to modify a file concurrently. When team members return the modifications, they can use CMS to merge them, informing the most recent user of any conflicting code that needs to be resolved.

Team members should use the "Remark" feature of CMS each time they do a transaction. Each CMS transaction is documented, providing a history that gives the project leader a way to monitor ongoing changes to files and also the progress of programmers' work. If problems turn up later, the CMS history information shows who made changes, when, and why. By providing informational remarks, team members can speed future work on the application, particularly during the maintenance stage. This information can also contain information on metrics, such as how many modules have been added or changed, how often, and so on.

4.6 Customizing the LSE Environment

This section provides guidelines for customizing your LSE environment in the following three areas:

- Creating an environment file
- Redefining tokens, placeholders, and language definitions
- Saving modified definitions

Note

The LSE commands shown in this manual conform to the LSE portable command language syntax. You can also use the original VMSLSE command language if you prefer. See the *HP Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual* for more information.

4.6.1 Environment File for LSE

The LSE environment file is a mechanism for providing language-specific templates for members of a project team. A team can use an environment file to implement coding and commenting conventions as well as design standards. Using environment files gives you a way to apply programming conventions and standards consistently. Additionally, the environment file is in binary format, so it does not need to be compiled each time you invoke LSE.

LSE enables you to customize the environment file to redefine tokens, placeholders, or language definitions.

4.6.2 Redefining Tokens, Placeholders, and Language Definitions

You can add or delete constructs, reformat menus, or edit descriptions within existing definitions. To redefine templates for the current editing session only, do the following:

1. Enter the NEW BUFFER command followed by a new buffer name to set up an empty buffer.
2. Enter the EXTRACT TOKEN, EXTRACT PLACEHOLDER, or EXTRACT LANGUAGE command, followed by the name of the token, placeholder, or language element you want modified.
3. Edit the definition of the selected token, placeholder, or language.
4. Enter the EXECUTE BUFFER LSE command to execute the new definition.

Note

If you want to save these modifications for future use, enter the SAVE ENVIRONMENT command described in Section 4.6.3.

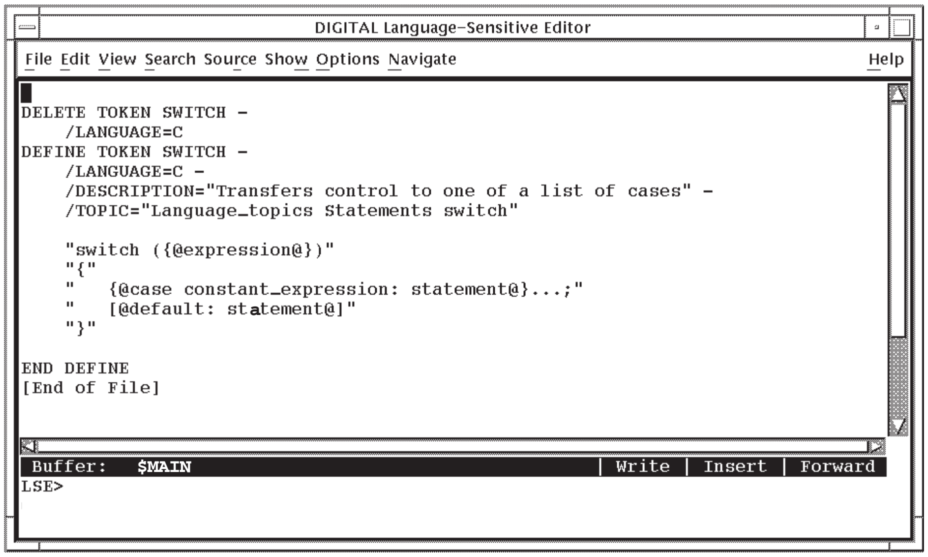
For example, to add a BREAK construct to the default definition of a C SWITCH statement, enter the EXTRACT command to begin modifying the default construct. After creating an empty buffer, enter the following command:

```
LSE Command> EXTRACT TOKEN SWITCH C
```

Figure 4–6 shows the results of entering the EXTRACT command.

Edit the token by adding a BREAK statement. Once you have the definition the way you want it, press Ctrl/Z to get the LSE Command> prompt. Enter the EXECUTE BUFFER LSE command to execute the new definition. Now, each time you use the SWITCH token in C during the current editing session, LSE provides the new definition.

Figure 4–6 Extracting a Token


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

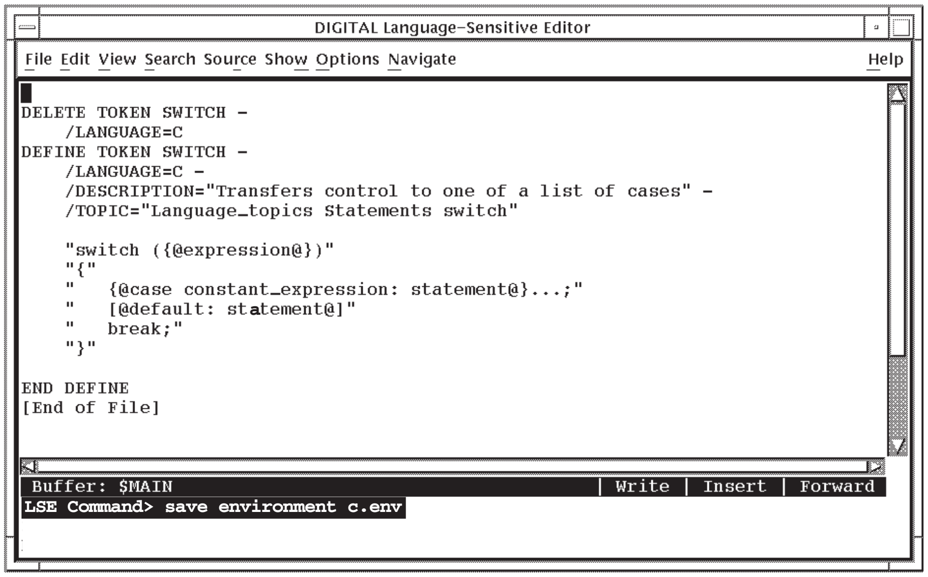
```
DELETE TOKEN SWITCH -
    /LANGUAGE=C
DEFINE TOKEN SWITCH -
    /LANGUAGE=C -
    /DESCRIPTION="Transfers control to one of a list of cases" -
    /TOPIC="Language_topics Statements switch"

    "switch ({@expression@})"
    "{"
    "   {@case constant_expression: statement@}...;"
    "   [@default: statement@]"
    "}"

END DEFINE
[End of File]
 Buffer:   $MAIN                                       | Write | Insert | Forward
LSE>
```

4.6.3 Saving Modified Definitions

You can save this modified SWITCH definition so it is available in subsequent work sessions. You need to create an environment file that can store the SWITCH definition and any other language-specific definitions you might make. It is good practice to save the definition source in a text file as well.

To create an environment file, enter the SAVE ENVIRONMENT command followed by the file name when you are still in the editing session. Figure 4–7 shows this step.

Figure 4–7 Creating an Environment File


DIGITAL Language-Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
DELETE TOKEN SWITCH -
    /LANGUAGE=C
DEFINE TOKEN SWITCH -
    /LANGUAGE=C -
    /DESCRIPTION="Transfers control to one of a list of cases" -
    /TOPIC="Language_topics Statements switch"

    "switch ({@expression@})"
    "{"
    "   {@case constant_expression: statement@}...;"
    "   [@default: statement@]"
    "   break;"
    "}"

END DEFINE
[End of File]
 Buffer: $MAIN                                  | Write | Insert | Forward
LSE Command> save environment c.env
```

To use the definitions in subsequent sessions, enter the /ENVIRONMENT switch on the LSE command line, as follows:

```
$ LSEDIT/ENVIRONMENT=DEVICE:[DIRECTORY]FILENAME.ENV
```

The environment file provides coding standards, which ensures consistency among all the team's developers. To make the environment file available to the entire team, store it in a project directory. Each developer's LOGIN.COM file can define a logical name that LSE translates to obtain the file specification for the environment file to be used when they invoke LSE.

If everyone on a project defines the logical name LSE$ENVIRONMENT in their LOGIN.COM files, they can use the stored definitions without specifying the /ENVIRONMENT switch every time they invoke LSE. To do this, they add the following command to their LOGIN.COM files:

```
$ DEFINE LSE$ENVIRONMENT TRN_DISK:[TRN.COMS]C.ENV
```

4.7 Setting Up Tests

Testing should be an integral part of all your developers' work in the design and implementation stages of your project. During the design stage, your development team should be designing tests and a testing strategy in the same way they are designing a strategy for coding.

As you move into the implementation stage and begin writing code, your team should be writing tests in parallel with writing code. Because code develops in an iterative fashion, programmers need to be sure they run tests on the latest versions of a file. Otherwise, new code might introduce errors that are not picked up, and the code will regress. All the modified code should have tests stored in the CMS project library for HP DIGITAL Test Manager. (See Section 3.2.3 for information about HP DIGITAL Test Manager libraries.)

To maximize the benefits from your tests, you need to decide how your team will organize its tests. The HP DIGITAL Test Manager provides several features for this task; however, the organization itself is left to the test users.

The first structure for organizing a test system is the test description. A test description identifies a test to the HP DIGITAL Test Manager. It consists of fields whose contents point to files needed to run the test. The core of each test description is the test template file. The test template file is a DCL command procedure or a recorded HP DIGITAL Test Manager session file that exercises your software. Each test must have a template file.

Prologues and epilogues are command files associated with tests and collections. The HP DIGITAL Test Manager provides two types of prologue and epilogue files: those associated with collections and those with individual tests. Collection prologue files run before, and collection epilogue files run after, the collection. Similarly, test prologue files run before, and test epilogue files after, the test template file. You can use the prologue file to establish any special environment the test requires. The epilogue file can be used to perform clean-up operations, or to filter the result file of run-dependent data.

For noninteractive and interactive terminal tests, the HP DIGITAL Test Manager can perform certain filtering operations, such as time-stamp information, automatically. For DECwindows tests, the HP DIGITAL Test Manager provides selective masking during the comparison, allowing for specified run-dependent information to be ignored.

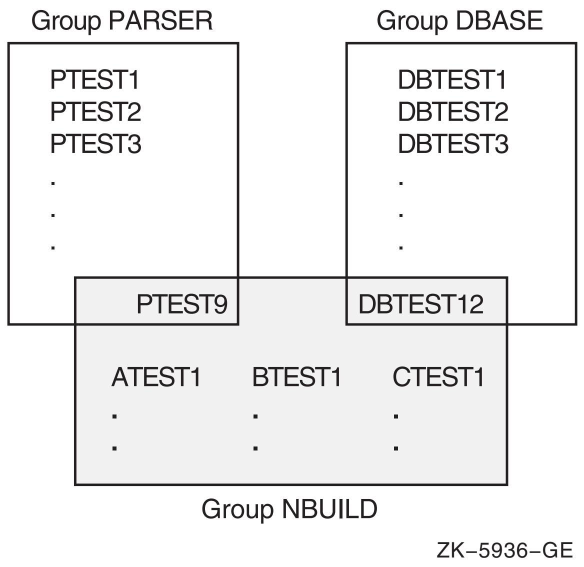
A variable in the HP DIGITAL Test Manager is either a DCL symbol or logical name. The HP DIGITAL Test Manager stores variables and uses them when executing tests. You can use variables in templates, prologues, and epilogues. Variables provide a convenient way for you to tailor a single template, prologue, or epilogue file so you can use it with many tests. For example, by using a variable in place of a particular test name in a template file, you can use that same template file to run many tests. You must define variables to the HP DIGITAL Test Manager in a separate step, and also include them in each test description that uses them. Chapter 10 contains an example of using the HP DIGITAL Test Manager variables.

You can organize your test descriptions in the HP DIGITAL Test Manager library by placing them in groups. For example, if you have several tests that share a similar characteristic or function, such as testing a parser, you can create a group called PARSER and place those test descriptions in the group. You can group the test descriptions by developer, or group them by both developer and function.

If you are doing incremental nightly builds, you might want to run a subset of all the system's tests. This subset should exercise all the basic functionality of the application. Additionally, individual team members might choose to run subsets of tests as they work. Running subsets can save considerable time during the building and testing process. However, note that one of the purposes of regression testing is to catch errors in seemingly unrelated areas that were introduced by the changes. If possible, you should run an entire test set, even for nightly builds. The sooner you find a problem, the less expensive and easier it is to fix.

Another use of test groups might involve forwarding tests to a quality review process in which a representative sample of tests rather than a full test system is used. Figure 4–8 shows the concept of grouping test descriptions.

Figure 4–8 Grouping Test Descriptions



Grouping test descriptions simplifies the process of creating test collections—the mechanism by which you initiate the running of tests. The key is to create efficient test description subsets as groups to encourage your programmers to use the tests. If you automate the test process, you can be sure that all new code has been adequately tested. Chapter 9 explains how to use automated testing procedures.

4.8 Analyzing Performance and Coverage

As your project moves into advanced stages of development, you need to check that your application and its component parts perform efficiently. PCA can identify those parts of your application that use excessive processing time, and indicate when you have reached an optimum for the current design.

You can use PCA to analyze the coverage of your tests, ensuring that your application has been thoroughly tested. PCA also lets you exclude certain portions of your code from testing (if those portions are difficult to test or are for internal error tracking) by allowing you to specify those portions as acceptably noncovered. PCA keeps track of those portions from test to test, and can automatically note if there have been any changes in those portions between iterations.

4.9 Tracking Reports During Field Testing

During field test stages, the team needs an effective means of getting and tracking feedback from its test sites. A team can use one or more of the following techniques to ensure that it quickly receives and responds to information from its test sites:

- Surveys
- Telephone contacts
- A quality assurance report (QAR) system to report, assign, and track problems

A QAR system helps track feedback from field test sites. It organizes the feedback from the test sites and your team's responses. The following information describes how to use DEC Notes to set up a functional QAR system.

Using DEC Notes

For detailed information on using DEC Notes, refer to the *Guide to DEC Notes*.

You can use DEC Notes to organize feedback from the test sites. The feedback mechanism can be written comment cards. In this case, members of the team enter the comments into the QAR system. They also enter their responses to the test site comments, as well as additional information for the team itself; for example, the status of the response, or any changes made to eliminate program errors. Finally, the response is sent to the test site by mail.

In addition to written comment cards, users might be able to dial in to accounts to directly send their comments to the QAR system. In this case, your team's responses will also be entered directly into the QAR system. This method will require users to refer back to the online QAR system for a response to their comment. By having an online QAR system, test site users can look in the QAR database for solutions to their problems and avoid entering duplicate problem reports.

DEC Notes organizes and speeds up the process by organizing input and replies. Further, it takes advantage of **keywords**, a feature of DEC Notes that allows you to group notes that concern a particular subject and do not have other attributes (such as title, author, or time of entry) in common. This feature is similar to CMS groups.

A QAR system using DEC Notes will have the following characteristics:

- A conference devoted to problems, comments, suggestions, and so on.
- Restricted access (optional) to the conference to team members and possibly field test users by means of a **membership list** (a feature of DEC Notes).
- Team members with CREATE_KEYWORDS or **moderator** privileges along with responsibilities for responding to specific field test queries. A moderator has certain privileges not available to other users.
- A team member assigned to the role of QAR system monitor (weekly, monthly, or permanently), whose tasks might include collecting responses from field test sites and adding them as topics in the conference. This person also adds appropriate keywords.

When setting up the conference, you can use the first two topics to explain the purpose and format of the notes. You can supply a template file in the second introductory note that users can extract and then use, ensuring that information is supplied in a consistent format.

The topic note contains the text of the problem report. For easy cross-referencing, the note can be titled as follows:

{problem summary}

For example:

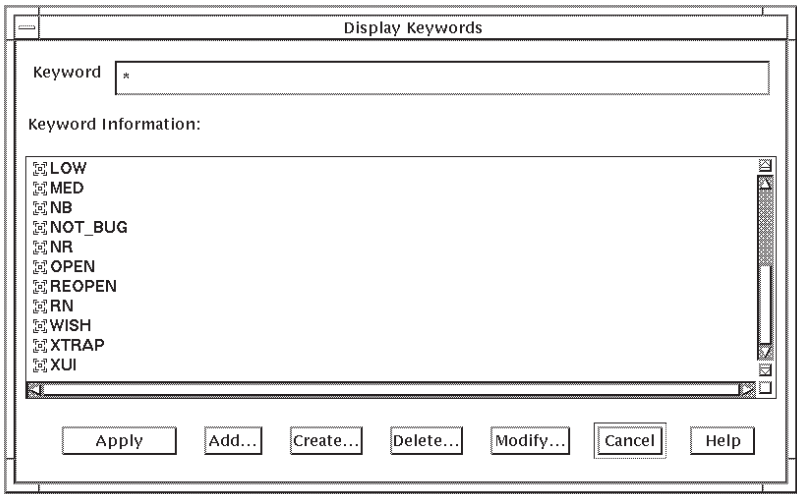
```
Generates Bad DEBUG SYMBOL TABLE Records
```

The system monitor sends the problem report to the maintainer assigned the task of responding to the report. The response will be a standard DEC Notes **reply**. This reply will then be sent to the field test user, or, if the user has access to the QAR conference, the user will be able to read the reply directly.

The keyword feature permits the maintainers of the QAR system to quickly access relevant notes. Keywords are first created for a conference by anyone with moderator privileges using the DEC Notes CREATE KEYWORD function. As stated previously, the system monitor adds the appropriate keywords (using the ADD KEYWORD function) to the notes. Maintainers can then retrieve all the notes to which a particular keyword has been added by using the DEC Notes functions: DIRECTORY KEYWORD, SAVE KEYWORD, or PRINT KEYWORD.

Figure 4–9 shows the DEC Notes DISPLAY KEYWORDS dialog box scrolled to the last window. You might also type a partial string and wildcard in the Keyword field at the top. For example, if you typed X* and pressed Return, the window would display only the two keywords XTRAP and XUI.

Figure 4–9 DEC Notes Display Keywords Dialog Box



LOW
MED
NB
NOT_BUG
NR
OPEN
REOPEN
RN
WISH
XTRAP
XUI

The following table lists some categories and examples of potential keywords. Your project can use this feature to tailor its own cross-referencing system.

Category	Keyword
Status	Open Closed Answered
Answer Type	Fault Documentation Suggestion User_error
Maintainer	Jones Lewis Peters
Version	V1.0 (Released version) T1.1 (Field test version)
Component	Callable Database User_interface

4.10 Final Project Steps

The final steps in a project are usually not the end of work on the software application. Maintenance and support can account for a high percentage of a project's cost. Software is continuously being maintained and enhanced. Maintenance is an integral part of the application; in addition, future releases might follow. The procedures described for software development will also help to effectively maintain the software. The following steps of a product release will contribute to this goal.

4.10.1 Prepare Final Build

The procedure for the final build is the same as that for earlier builds that marked major milestones. After completion of the build, the sources, both code and documentation, should be frozen using the class feature of CMS. Included with this class should be the MMS description file that builds this version of the software.

To make it easier to rebuild the final version, a master summary file stores pertinent build information in the same CMS library. Using this master file to provide information will ease future work, including:

- Names of related classes in different CMS libraries
- List of product dependencies—what versions of related software were used to build this application

The amount of information you keep depends on how much disk space you have. For instance, keeping your HP DIGITAL Test Manager tests and results will benefit the people who maintain or upgrade the project. SCA library information is useful, but it can be rebuilt if necessary from the sources in CMS during compilation.

4.10.2 Permanent Storage

The sources for the code and documentation, along with any other information (for example, online Help files), should be stored in a CMS library to prevent loss of the master files.

Part II

HP DECset Case Study

Part II of this guide gives examples and procedures for using the HP DECset tools on a hypothetical software development project called the Transliteration project. All HP DECset tools are shown in the HP DECwindows Motif environment. (For information on using a particular tool with its DCL interface, refer to the documentation set for that tool.) Part II contains the following chapters:

- Example Project Setup describes the basic project directory structure of the Transliteration project.
- Using CMS to Maintain Project Source Files gives examples using CMS.
- Writing, Compiling, and Debugging Source Code provides examples of using the debugger, SCA, and LSE to debug, modify, and compile code.
- Designing Programs with the Program Design Facility provides an overview of designing programs using the program design features of LSE and SCA, and shows an example of designing a module of code.
- Building the Application with MMS shows and explains the MMS description file used to build the TRANSLIT application.
- Setting Up a Test System with HP DIGITAL Test Manager describes using the HP DIGITAL Test Manager to set up a test system.
- Using HP DIGITAL Test Manager with PCA describes how you can use HP DIGITAL Test Manager with PCA to optimize your testing environment.
- Maintaining the Application describes how you can use HP DECset tools to manage the maintenance and support stages of software development.

5 Example Project Setup

This chapter describes the demonstration project, Transliteration, which is based on the examples in Part II, referred to as the Transliteration project. The objective of this project is to create the Transliteration application, known as TRANSLIT. Part II of this guide shows how the Transliteration project team organizes its code libraries and builds and tests TRANSLIT, using the DECset tools, the OpenVMS Debugger, a HP processor running the OpenVMS operating system, and multiple languages. These examples present techniques that will help you make the best use of the DECset tools.

Although most of the examples relate to a single application being developed by a relatively small team to support a larger project, many examples are applicable to the problems associated with larger projects.

This chapter describes the following procedures carried out by the Transliteration project team:

- Setting up project directories
- Creating CMS libraries
- Creating SCA libraries
- Creating HP DIGITAL Test Manager libraries
- Setting up an OpenVMS ACL-based, file access security mechanism
- Setting up LOGIN.COM files and other command files

5.1 Setting Up Directories

The Transliteration application manipulates and substitutes text strings within a file. The project team, referred to throughout this chapter as the Transliteration team, will produce documentation for the designs, specifications, and the application itself. The application requires several source code modules to be built periodically. To ensure that these tasks proceed efficiently, the Transliteration team planned a directory structure to support the tools and the project's storage and communication needs.

5.1 Setting Up Directories

The Transliteration team needs a project directory structure that can meet the following objectives:

- To manage the specific needs of the project
- To organize and facilitate access to the files of the application for the team members
- To provide public directories to allow those outside of the project to have access to nonrestricted files
- To create and maintain build directories for both the project and individual developers

The following table lists the directories and libraries created by the Transliteration team. (The table lists local directories for only one developer; in reality, there would be local directories for each team member.) The directory structure provides adequate storage for all the required libraries. It also organizes the project's files into functional units. A build area also stores the end products of any build procedures.

Directory	Logical Name	Function
General Directories		
[TRN.PUBLIC]	TRN_PUBLIC	Stores information (for example, printable documentation) for public users
[TRN.COMS]	TRN_COMS	Stores project command procedures
[TRN.DOC_CMSLIB]	TRN_CMS_DOC	Stores documentation sources
[TRN.CODE_CMSLIB]	TRN_CMS_CODE	Stores source files
[TRN.DTM_DATA_CMSLIB]	TRN_DTM_DATA_ CMS	Stores input files necessary to generate output files for tests

Directory	Logical Name	Function
HP DIGITAL Test Manager Directories		
[TRN.DTMLIB]	TRN_DTM	Stores test descriptions and results, including PCA results
[TRN.DTM_CMSLIB]	TRN_CMS_DTM	Stores templates, benchmarks, test data files, prologue, and epilogue files
[TRN.DTM_DATA]	TRN_DTM_DATA	Serves as a directory for files and test data not stored in a CMS library
Build Tree		
[TRN.BLD_V1]	undefined	Root directory for Version 1 build subdirectories
[TRN.BLD_V1.WORK]	TRN_BLD	Stores sources and products of build: source, .OBJ, and .EXE files
[TRN.BLD_V1.SCALIB]	TRN_SCA	Stores SCA modules for project-wide access
[TRN.BLD_V1.CODE_REFCOPY]	TRN_REF	Stores most recent clear copy source files
Local Directories		
[JONES.WORK]	MY_AREA	Local area work directory
[JONES.SCALIB]	MY_SCALIB	Stores local SCA analysis data

5.2 Creating Directories and Libraries

One of the first tasks in a project is to set up the directories and corresponding libraries for the tools. The Transliteration team needs libraries for CMS, SCA, and HP DIGITAL Test Manager.

Two steps are necessary to create a project library:

1. Create a directory for the library.
2. Create the library.

The Transliteration team sets protection by the DCL procedures for directory protection. The following example shows how to set up a CMS library for documentation while restricting access to the library's directory:

```
$ CREATE/DIRECTORY/OWNER=[TRN,JONES]/PROTECTION=(S:RWE,O:RWE,G:RWE,W:RE) -
_$ TRN_DISK:[TRN.DOC_CMSLIB]
$ CMS CREATE LIBRARY TRN_DISK:[TRN.DOC_CMSLIB]
_Remark: CMS Library for Project Translit documentation
```

Creating a CMS library for source code follows a similar procedure:

```
$ CREATE/DIRECTORY/OWNER=[TRN,JONES]/PROTECTION=(S:RWE,O:RWE,G:RWE,W:RE) -
_$ TRN_DISK:[TRN.CODE_CMSLIB]
$ CMS CREATE LIBRARY/REFERENCE_COPY=TRN_DISK:[TRN.BLD_V1.CODE_REFCOPY] -
_$ TRN_DISK:[TRN.CODE_CMSLIB]
_Remark: CMS Library for Project Translit Code
```

The CMS CREATE LIBRARY command uses a /REFERENCE_COPY switch to designate the directory where all reference copies will reside. Every time a developer creates a main-line element generation (by using a CMS CREATE ELEMENT or CMS REPLACE command), CMS puts a copy of the new generation into the reference copy area, while deleting the previous version of the element from your area. (Section 3.2.2.3 explains the use of a reference copy area.)

The team creates an SCA library with the following commands:

```
$ CREATE/DIRECTORY/OWNER=[TRN,JONES]/PROTECTION=(S:RWE,O:RWE,G:RE,W:RE) -
_$ TRN_DISK:[TRN.BLD_V1.SCALIB]
$ SCA CREATE LIBRARY TRN_DISK:[TRN.BLD_V1.SCALIB]
```

The team creates the HP DIGITAL Test Manager library in the same way. In all these examples, the CREATE LIBRARY command performs an implicit SET LIBRARY command, so the developer can proceed to use CMS or SCA commands with the respective libraries. Subsequently, when accessing existing libraries, developers will need to first use the SET LIBRARY command before any specific CMS, SCA, or HP DIGITAL Test Manager commands that access the library, as in the following example:

```
$ CMS SET LIBRARY TRN_CMS_DOC
```

5.3 Setting Up an OpenVMS ACL-Based Security Mechanism

After creating a library, the team might want to further restrict access to information within the library. By using access control lists (ACLs) together with the standard UIC-based protection, the team can refine library protection throughout its directory. The following example shows how to create and add to ACLs:

```
$ SET DIRECTORY/ACL=(IDENTIFIER=[TRN],ACCESS=R+W+E+C) -
_$ [TRN.DOC_CMSLIB]
$ SET DIRECTORY/ACL=(IDENTIFIER=[TRN],OPTIONS=DEFAULT,ACCESS=R+W+E+D+C) -
_$ [TRN.DOC_CMSLIB]
$ SET DIRECTORY/ACL=(IDENTIFIER=[NETWORK],ACCESS=NONE) -
_$ [TRN.DOC_CMSLIB]
$ DIR/SECURITY

Directory TRN_DISK:[TRN]

DOC_CMSLIB.DIR;1     [PROJECT_SOURCE]           (RWE,RWE,RE,E)
          (IDENTIFIER=NETWORK,ACCESS=NONE)
          (IDENTIFIER=[PROJECT_SOURCE],OPTIONS=DEFAULT,ACCESS=READ+WRITE+
          EXECUTE+CONTROL)
          (IDENTIFIER=[PROJECT_SOURCE],ACCESS=READ+WRITE+EXECUTE+CONTROL)

Total of 1 file.
```

The first command establishes access to the newly created directory file itself (note the lack of DELETE access). The second command sets the default for the CMS library: all users associated with the identifier TRN are allowed READ, WRITE, EXECUTE, DELETE, and CONTROL access to the contents of this library. This default access control entry (ACE) applies to all files added to this library. The third command adds an ACE that prevents network access. The last command gives a directory listing of the newly created directory along with the ACL associated with it.

If you have already created a CMS library and placed files in that library without specifying a default ACE, you can change the access to the files in that library with the following commands:

```
$ SET DIRECTORY/ACL=(IDENTIFIER=[TRN],OPTIONS=DEFAULT,ACCESS=R+W+E+D+C) -
_$ [TRN.DOC_CMSLIB]
$ SET FILE/ACL/DEFAULT [TRN.DOC_CMSLIB...]*.*
```

The first command sets the default for the CMS library. The second command applies the entire default ACL of the parent directory to all the files in that directory as if they were newly created. See the *OpenVMS DCL Concepts Manual* for a detailed explanation of protection procedures, user identification codes (UICs), and ACLs. The *OpenVMS Access Control List Editor Manual* contains information on using the ACL Editor, an OpenVMS utility used to create and maintain ACLs. For more information on CMS library security, see the *Guide to HP Code Management System for OpenVMS Systems*.

5.4 Setting Defaults with a LOGIN.COM File

Another early task of the team is to establish work conditions and default values, or a work context, by means of LOGIN command files. A developer's LOGIN.COM file can automatically perform the following tasks at system startup:

- Access a previously created LSE environment file.
- Define logical names that are both for the entire project and for each individual.
- Set CMS, HP DIGITAL Test Manager, and SCA libraries.
- Set CMS search lists if using the project work areas as well as individual work areas.
- Set up search lists if using the project's work areas as well as individual work areas.
- Set a source list to be used by LSE when accessing sources.

Note

You can improve the performance of your LOGIN.COM file if you include the /NOVERIFY option with the SET LIBRARY commands for CMS and HP DIGITAL Test Manager.

Logical names can be defined and made available for the project. On the Transliteration project, they are defined in a command file stored in the [TRN.COMS] directory, which individual LOGIN.COM files then execute.

Example 5–1 shows a LOGIN.COM file from one of the Transliteration project's developers.

This file makes available the environment file, C.ENV, and executes the logical name command file, TRN_LOGICALS.COM. It also sets this developer's CMS, HP DIGITAL Test Manager, and SCA libraries, and establishes a library search list for SCA that accesses the local SCA library first, followed by the project-wide SCA library.

The file also defines two LSE logical names to help in source management. The first defines a logical name that causes LSE to draw files from locations in a specific order (the user's default directory, project build area, and CMS code library). The second LSE logical name causes files that are read by LSE from this directory to be placed in nonmodifiable, read-only buffers. This prevents changes to files when developers are using LSE and SCA to move through different modules. As a result of this command, developers must reserve a module from CMS before making any changes, thus providing a history of their activity.

Next, the LOGIN.COM file defines a logical name for the debug initialization file (which is often tailored individually for each developer), and sets up LSE as the editor to be invoked from MAIL.

Finally, the file defines logical names for the PCA Collector and Analyzer initialization files. The file specification for the Collector is assigned to the logical name PCAC$INIT.

Example 5–1 Sample LOGIN.COM File

```
$ DEFINE LSE$ENVIRONMENT        TRN_DISK:[TRN.COMS]C.ENV
$ @TRN_DISK:[TRN.COMS]TRN_LOGICALS.COM
$ CMS SET LIBRARY               TRN_CMS_CODE /NOVERIFY
$ DTM SET LIBRARY               TRN_DTM /NOVERIFY
$ SCA SET LIBRARY               MY_SCALIB, TRN_SCALIB
$ DEFINE LSE$SOURCE             [], TRN_BLD, CMS$LIB
$ DEFINE LSE$READ_ONLY_DIRECTORY TRN_BLD
$ DEFINE DBG$INIT               TRN_DISK:[TRN.COMS]MYDEBUGINIT.COM
$ DEFINE MAIL$EDIT              CALLABLE_LSE
$ DEFINE PCAC$INIT              TRN_DISK:[TRN.COMS]MYINITFILE.PCAC
$ DEFINE PCAA$INIT              TRN_DISK:[TRN.COMS]MYINITFILE.PCAA
   .
   .
   .
```

The Collector initialization file enables you to collect the same performance or coverage data in several separate collection runs, or in a batch run.

The file specification for the Analyzer initialization file is assigned to the logical name PCAA$INIT.

The logical name definitions file (TRN_LOGICALS.COM), accessed by the LOGIN.COM file, defines the various directories and libraries for the project. These logical names can be used in place of the full directory or library specifications. Example 5–2 shows the contents of this file.

Example 5–2 Project Logical Definitions File

```
$ DEFINE TRN_PUBLIC             TRN_DISK:[TRN.PUBLIC]
$ DEFINE TRN_COMS               TRN_DISK:[TRN.COMS]
$ DEFINE TRN_CMS_CODE           TRN_DISK:[TRN.CODE_CMSLIB]
$ DEFINE TRN_DTM                TRN_DISK:[TRN.DTMLIB]
$ DEFINE TRN_CMS_DOC            TRN_DISK:[TRN.DOC_CMSLIB]
$ DEFINE TRN_CMS_DTM            TRN_DISK:[TRN.DTM_CMSLIB]
$ DEFINE TRN_DTM_DATA_CMS       TRN_DISK:[TRN.DTM_DATA_CMS]
$ DEFINE TRN_DTM_DATA           TRN_DISK:[TRN.DTM_DATA]
$ DEFINE TRN_BLD                TRN_DISK:[TRN.BLD_V1.WORK]
$ DEFINE TRN_REF                TRN_DISK:[TRN.BLD_V1.CODE_REFCOPY]
$ DEFINE TRN_SCA                TRN_DISK:[TRN.BLD_V1.SCALIB]
$ DEFINE MY_AREA                TRN_DISK:[JONES.WORK]
$ DEFINE MY_SCALIB              TRN_DISK:[JONES.SCALIB]
```

6 Using CMS to Maintain Project Source Files

This chapter provides examples of the following procedures:

- Using the CMS Access Control List (ACL) feature to set up a security mechanism for CMS objects
- Storing files in a CMS library
- Modifying elements in a CMS library
- Gaining concurrent access to elements in a CMS library
- Creating classes in a CMS library
- Retrieving contents of a class and preparing for a build

6.1 Using CMS Access Control Lists

OpenVMS user identification code (UIC) and ACL-based mechanisms do not meet all the needs of a larger project. To refine the protection required in CMS, CMS provides its own ACL mechanism. As OpenVMS ACLs protect access to OpenVMS files and directories, CMS ACLs protect access to the following CMS objects:

- Commands
- Elements
- Element lists
- Classes
- Class lists
- Groups
- Group lists

- History
- Library attributes

CMS ACLs are not designed to provide tighter security than OpenVMS ACLs; however, they can give you greater control over your CMS libraries by enabling you to control access to all CMS objects, not just files and directories.

CMS ACLs are not designed to take the place of OpenVMS ACLs; it is recommended that you use them together. In this section, OpenVMS ACLs are used in combination with CMS ACLs for a well-tuned access control mechanism. When considering access control for large projects, first create a list of goals you want to achieve. Attaching ACLs to CMS objects costs in disk space, performance, and ease of management, so plan the use of ACLs carefully.

This section gives examples of creating ACLs for both CMS library elements and commands. In some cases, it might not be necessary to have ACLs on both elements and commands; how you design your protection mechanism depends on your project needs. For the purposes of showing examples of CMS ACLs, this section first shows a protection scheme geared toward protecting library elements. Later, some examples are given showing the use of ACLs on CMS commands.

6.1.1 Example of a CMS ACL Scheme

In the example used in this section, the Transliteration project team chose the following goals for the project's CMS security mechanism:

1. To establish four groups of users in the system rights list, each having their own specific types of access to CMS objects. They hold one of the following identifiers:
   - CONFIG_MANAGER—Held only by the person with the responsibility for building the entire system. This person has complete access to all CMS objects.
   - PROJECT_SOURCE_READ—Held by any user requiring read-only access to files under the PROJECT product root (for example, a technical writer).
   - PROJECT_SOURCE—Held by any user requiring read and write access to files under the PROJECT product root (for example, a developer).

- PROJECT_RED—Held by all users when the project enters its RED period, in which users are prevented from executing the Create Element and Replace functions. This RED period helps the project leader prepare the PROJECT product root for a system build.

2. To grant CONTROL access to the library elements to the holder of CONFIG_MANAGER. At least one person should have CONTROL access to any ACLs that are created because CONTROL is the only access type that permits you to modify an ACL (other than BYPASS privilege, which is described later).
3. To specify a default access control entry (ACE) for elements added to the library. This is needed because unless you specifically restrict access to a CMS object with a CMS ACL, no restrictions will be enabled by default for that object. Once you attach an ACL to a CMS object, access to that object is permitted only to the holder of the identifiers specified in the ACL. For example, one simple method to secure all elements in a CMS library is to attach an ACL to the element list. Any user not matched to the identifier in that ACL will not be permitted access to that list. Execution of ACL commands from this point on is a process of granting access as opposed to restricting it.
4. To grant holders of the PROJECT_SOURCE_READ identifier ACCEPT, ANNOTATE, FETCH, and REVIEW access to the library elements.
5. To copy the previous ACLs to existing elements that have no ACLs.
6. To deny everyone access to the Create Element and Replace functions when the project enters the RED period. This involves placing ACLs on commands as well as library elements.
7. To use a CMS event notification ACE to cause the holders of the CONFIG_MANAGER identifier to receive MAIL notification of any Replace function executed while the project is in its RED period.

A brief description of using the BYPASS privilege is included in Section 6.1.5.

Note

When creating CMS ACLs, prepare the sequence of ACEs carefully. As with OpenVMS ACLs, access is determined when CMS finds the first match between the user attempting to gain access and the identifiers in the ACEs, regardless of subsequent ACEs.

6.1.1.1 Specifying ACLs in the CMS ACL Dialog Box

In the CMS DECwindows interface, CMS provides a dialog box for specifying ACLs. In this dialog box, CMS provides fields for specifying the following:

- The object name
- The object type
- A remark string
- The ACE itself

In this dialog box, the Copy ACL From Object button enables you to specify the name of an object whose ACL you want to copy to the current object. You can also use the Add ACEs buttons to specify the order of the current ACE in an existing ACL.

The following sections provide specific information you enter into the Set ACL dialog box to create your security scheme.

6.1.1.2 Granting Control Access to Library Elements

As mentioned previously, at least one person should have CONTROL access in any ACLs that are created because CONTROL is the only access type that permits you to modify an ACL. The holder of CONFIG_MANAGER is given the following access types in the library element list:

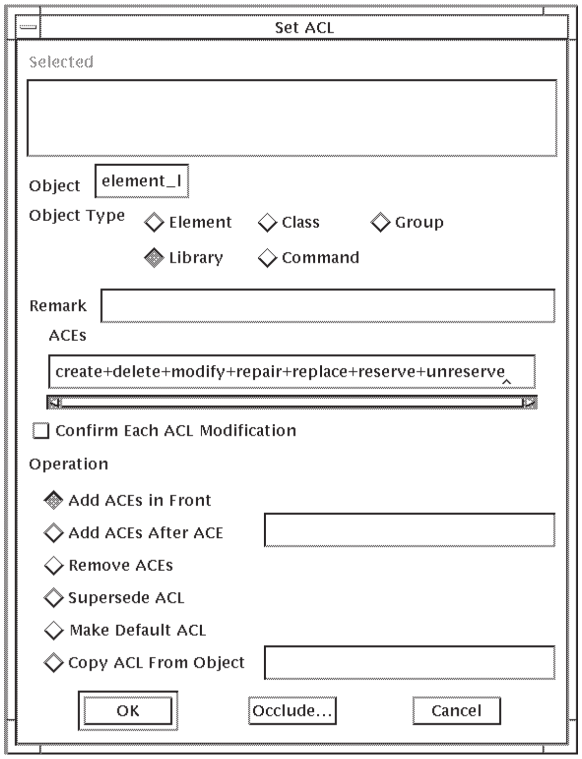
CONTROL
CREATE
DELETE
MODIFY
REPAIR
REPLACE
RESERVE
UNRESERVE

Specifying CONTROL access in the library element list ACE, along with OPTIONS=DEFAULT, gives CONFIG_MANAGER the ability to modify the ACLs on any element added to the library, as in the following ACL:

```
(IDENTIFIER=CONFIG_MANAGER,OPTIONS=DEFAULT,ACCESS=CONTROL+CREATE-
+DELETE+MODIFY+REPAIR+REPLACE+RESERVE+UNRESERVE)
```

To assign this ACL to the library's element list, specify ELEMENT_LIST in the Object field of the Set ACL dialog box, and click on Library, as shown in Figure 6–1.

Figure 6–1 Specifying an ACL for an Element List



Set ACL

Selected

Object: element_l

Object Type: Element, Class, Group, [✓ Library], Command

Remark

ACEs: create+delete+modify+repair+replace+reserve+unreserve

☐ Confirm Each ACL Modification

Operation

- [✓ Add ACEs in Front]
- Add ACEs After ACE
- Remove ACEs
- Supersede ACL
- Make Default ACL
- Copy ACL From Object

6.1.1.3 Specifying a Default ACE for Library Elements of project_source

When you specify OPTIONS=DEFAULT in an identifier ACE for a library element list, all newly created elements in the library will inherit this ACE by default. In this sample ACL, holders of PROJECT_SOURCE are granted the following access types to the library element list:

CREATE	REPLACE
DELETE	RESERVE
MODIFY	UNRESERVE
REPAIR	

The following ACL sets these access types, and is attached to the library element list:

```
(IDENTIFIER=PROJECT_SOURCE,OPTIONS=DEFAULT,ACCESS=CREATE-
+DELETE+MODIFY+REPAIR+REPLACE+RESERVE+UNRESERVE)
```

6.1.1.4 Granting Access for Holders of project_source_read

The next goal of the Transliteration project team is to grant holders of PROJECT_SOURCE_READ the following access types to the library elements:

ACCEPT
ANNOTATE
FETCH
REVIEW

The following ACL, attached to the element list, grants these access types:

```
(IDENTIFIER=PROJECT_SOURCE_READ, ACCESS=ACCEPT+ANNOTATE+FETCH+
REVIEW)
```

To make this ACL the default for any new elements added to the CMS library, click on Make Default ACL in the Set ACL dialog box.

6.1.1.5 Copying CMS ACLs to Existing Objects

The ACLs shown in the previous examples affect only those elements created after the ACL commands are entered. To have those ACLs apply to elements that already exist, enter the following ACL command:

```
$ CMS SET ACL/DEFAULT/OBJECT=ELEMENT *.* "Retrofit with ACLs -rmy"
```

6.1.2 Placing ACLs on CMS Commands

The previous examples dealt with ACLs placed on library elements. The example in this section shows how to place ACLs on CMS commands. One major advantage to be gained by placing an ACL on a command, rather than on each element, is that it reduces the number of ACLs that CMS checks when a command is entered. (CMS needs to check only the ACL on the command, rather than each ACL on each element.) This results in improved performance and simplified maintenance.

One of the goals of the Transliteration project team is to establish a period during which the libraries must be stable in order to prepare for a system build. During this period, team members should be prohibited from being able to create or replace elements in the libraries. This period is referred to as the RED period. One way to prevent activity that interferes with the stability of the libraries is to deny access to the Replace and Create Element commands during that RED period. The following ACL on those commands restricts access to them:

```
(IDENTIFIER=PROJECT_SOURCE+PROJECT_RED,ACCESS=NONE)
```

6.1.3 Using Event Notification ACEs

An event notification ACE is used to send MAIL notification when a specified event occurs. The Transliteration project team needs to create an Event Notification ACE that sends mail to CONFIG_MANAGER when anyone attempts to execute the Replace function when the Transliteration project is in its RED period. The following ACL placed on REPLACE accomplishes this:

```
(NOTIFY=CONFIG_MANAGER,IDENTIFIER=[*]+PROJECT_RED,ACCESS=EXECUTE)
```

CMS enables you to write your own event handling program to take a specific action when a specific event occurs. See the *Guide to HP Code Management System for OpenVMS Systems* for more details.

6.1.4 Access During Normal Development

During normal development, holders of either the PROJECT_SOURCE_READ or PROJECT_SOURCE identifiers are the only users that require access to the PROJECT product root. The system builder, who holds the CONFIG_MANAGER identifier, also has access to the PROJECT product root during this period, but generally does not have to access the product root at other times.

6.1.5 Access when Preparing for a System Build

When creating a build class, the PROJECT product root must be locked for read-only access. This RED period is needed so the system builder has a stable product root when determining which sources to include in the next system build.

The lock mechanism results from adding the PROJECT_RED identifier to the system rights list. The following command carries out this procedure:

```
$ SET RIGHTS_LIST/ENABLE/SYSTEM PROJECT_RED
```

Because the system is a holder of the identifier, every process on the system immediately becomes a holder of the identifier. This action affects only the users who are the holders of the PROJECT_SOURCE identifier. Because the PROJECT_SOURCE+PROJECT_RED identifier precedes the PROJECT_SOURCE identifier in the product root ACL, any holder of both the PROJECT_SOURCE and the PROJECT_RED identifiers will have read-only access. All other users are unaffected.

After resolving which sources to include in the next system build, the configuration manager unlocks the product root, allowing normal development work to resume. To unlock the product root, the configuration manager removes the PROJECT_RED identifier from the system rights list, and all processes on the system lose the identifier.

As these examples demonstrate, protection procedures enable you to restrict access to the library as a whole, or parts of the library, or to CMS commands, yielding greater control during builds. Additionally, holding BYPASS or the existence of an ACE granting BYPASS allows a user to replace or unreserve elements for another user. Holding a process BYPASS privilege circumvents CMS access checking, allowing your system manager or a user with system privileges to correct your ACLs if you have inadvertently locked yourself out and are unable to correct them yourself.

You need to maintain consistent team-working procedures to continue to control library access. For more information on CMS security features, see the *Guide to HP Code Management System for OpenVMS Systems*.

6.2 Maintaining CMS Source Libraries

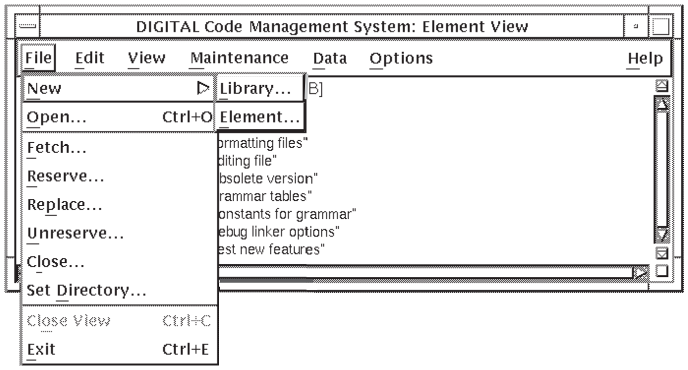
CMS provides tracking, history, and source management information that helps a team maintain its source files, design documents, and functional specifications.

6.2.1 Storing Files in a CMS Library

One of the first tasks of the Transliteration project is to produce specification and design documents. As they are produced, the team stores them in the CMS library for documentation (see Section 5.2 for the procedures that set up this CMS library).

To place these documents into the library, invoke CMS using the /INTERFACE=DECWINDOWS switch, and choose the New Element... item in the File menu, as shown in Figure 6–2.

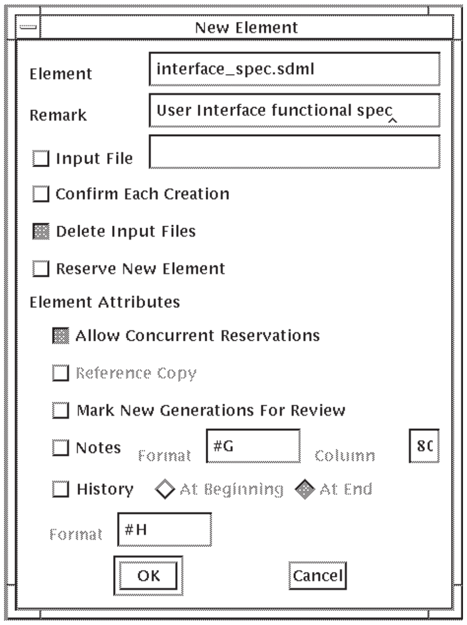
Figure 6–2 Creating an Element



```
B]
ormatting files"
diting file"
bsolete version"
rammar tables"
onstants for grammar"
ebug linker options"
est new features"
```

When you choose New Element... from the File menu, CMS displays the New Element dialog box, as shown in Figure 6–3.

Figure 6–3 New Element Dialog Box



New Element

Element: interface_spec.sdml

Remark: User Interface functional spec

[ ] Input File

[ ] Confirm Each Creation

[■] Delete Input Files

[ ] Reserve New Element

Element Attributes

[■] Allow Concurrent Reservations

[ ] Reference Copy

[ ] Mark New Generations For Review

[ ] Notes Format: #G Column: 80

[ ] History ◇ At Beginning ◆ At End

Format: #H

Enter INTERFACE_SPEC.SDML in the Element field, and “User interface functional spec” in the Remark field, as shown in Figure 6–3.

Once inserted into the CMS library, INTERFACE_SPEC.SDML becomes an **element**. When you place a file into a CMS library with the Create Element function, that file becomes the first generation of the element. As you modify the file, CMS creates subsequent generations of the element.

Although CMS can store both source and formatted files, you might want to save disk space by storing only the source version (which can be recompiled at any time). For example, you might save only the .SDML files but not the .PS files for reports generated in DECdocument.

6.2.2 Modifying Elements

Needing to modify the element SPEC.RNO, one of the developers, Mary, retrieves it from the library with the CMS Reserve function, edits the file, and returns it to the library with the CMS Replace function. All of these steps can be done from within LSE, as in the following example:

```
LSE Command> RESERVE SPEC.RNO
_Remark: Modify tasks system -- expand discussion of capabilities
generation 1 of element SPEC.RNO reserved
LSE Command>
```

After editing the file, she returns it to the CMS library, as follows:

1. Clicks on the File menu in the LSE main window.
2. Chooses Replace Element.

The CMS library now contains two generations of the element. Figure 6–4 shows the contents of the CMS library.

Figure 6–4 CMS Library with Two Generations

CMS LIBRARY
SPEC.RNO;1
SPEC.RNO;2
ZK–5949–GE

6.2.3 Concurrent Access

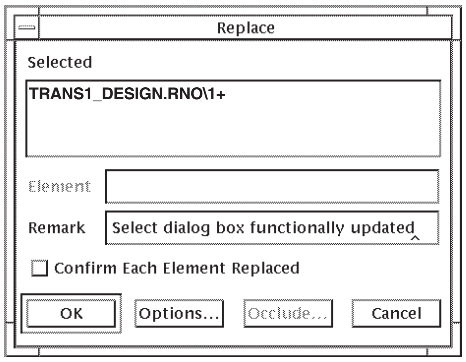
CMS allows more than one user to access a CMS element at the same time. CMS notifies you when you try to access an element when another user is working on that element. For example, Mary has reserved a copy of a design specification. When a second developer, John, tries to reserve the same element, he receives the following message:

```
Element TRANS_DESIGN.RNO currently reserved by:
    (1)   Mary       1      28-FEB-1998 08:37:28 "Reserved for update"
Proceed?  [Y/N] (N): Y
%CMS-S-RESERVED, generation 1 of element TRN_DISK:[TRN.CODE_CMSLIB]
TRANS_DESIGN.RNO reserved
```

John reserved the element after being notified that Mary had previously done so. Meanwhile, Mary replaces her generation. Later that day, she continues her work, developing another generation of the design specification.

Unlike Mary, who used the LSE CMS subsystem, John is using CMS directly. When John replaces his generation, he indicates that his version is a variant in the Replace dialog box, shown in Figure 6–5. John clicks on Options... to get a dialog box in which he can specify that this particular generation is a variant.

Figure 6–5 Replace Dialog Box



CMS displays a message in the DECterm or FileView window that invokes CMS, indicating that the element is already reserved, and prompts if he wants to continue or cancel the operation, as follows:

```
Element TRANS_DESIGN.RNO currently reserved by:
    (1)   Mary       1      28-FEB-1998 08:37:28 "Reserved for update"
Proceed?  [Y/N] (N): Y
%CMS-S-GENCREATED, generation 2A1 of element TRN_DISK:[TRN.CODE_CMSLIB]
TRANS_DESIGN.RNO created
```

John can use CMS to incorporate his changes into subsequent generations of this element while ensuring that his changes are consistent with Mary's changes. To do this when he reserves the generation, he clicks on the Options... button at the bottom of the Reserve dialog box. Another dialog box is displayed, in which John can specify the merge information in the appropriate field.

Before John replaces the merged element, he verifies that the merged element is as expected. If the element is a code module, he compiles, links, and executes the code before replacing it. John then replaces the element by selecting Replace in the Data menu.

In this case, no conflicts occurred between the two versions. Had there been any, CMS would have flagged these and the reserved element would have needed editing to resolve these conflicts.

For more information on merging variants of CMS elements, see the *Guide to HP Code Management System for OpenVMS Systems*.

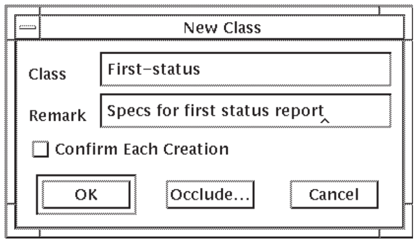
6.2.4 Creating Classes

The project leader now needs to write a status report based on the most up-to-date versions of the files. By creating a **class**, the most up-to-date files can be retrieved easily from the CMS library.

To create a class and begin inserting generations into that class, do the following:

1. Choose New from the Maintenance menu in the CMS main window. (The option Class... is available in a submenu, which you select.) The New Class dialog box appears.
2. Specify the class name and an associated remark string, as shown in Figure 6–6.

Figure 6–6 New Class Dialog Box


New Class

Class: First-status

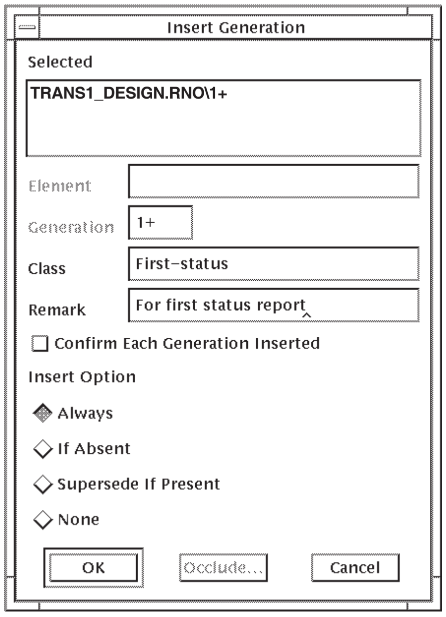
Remark: Specs for first status report

☐ Confirm Each Creation

3. Choose Insert Generation... from the Maintenance menu. (The Generation option is available in a submenu, which you select.)

4. Specify the class First-status and a remark string in the Insert Generation dialog box, as shown in Figure 6–7.

Figure 6–7 Insert Generation Dialog Box



Insert Generation

Selected

TRANS1_DESIGN.RNO\1+

Element

Generation 1+

Class First–status

Remark For first status report

☐ Confirm Each Generation Inserted

Insert Option

◆ Always

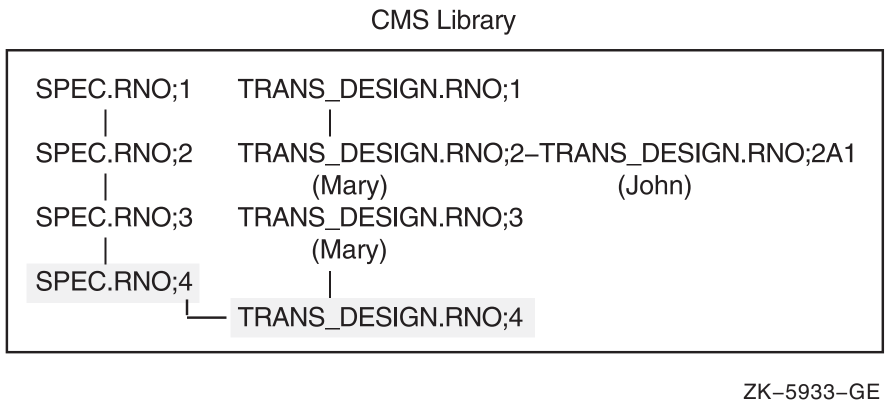
◇ If Absent

◇ Supersede If Present

◇ None

Figure 6–8 shows the contents of the CMS class First-status after the files have been inserted.

Figure 6–8 Classes in a CMS Library



6.2.5 Retrieving Class Contents and Preparing a Build

You can use MMS to automatically fetch specific generations of elements from a CMS library and invoke RUNOFF (and other actions) on those elements. MMS automates the building of documents and software systems (see Chapter 9 for an example of using MMS to build an entire software application). This section describes using MMS to fetch and build a software system.

The first step in using MMS is to create a **description file**. A description file describes the relationships among the modules that make up the system. The description file is made up of **dependency rules** consisting of three parts:

- Target—A file to be updated by MMS
- Source—A file used by MMS to update a target
- Action line—The part of the dependency rule that tells MMS how to use the sources to update the target

The format for arranging these three units in a dependency rule is as follows:

```
TARGET : SOURCE
   ACTION LINE
```

The Transliteration developers created the following description file to build a complete set of specifications:

```
CMSFLAGS = /GENERATION=FIRST_STATUS

ALL_DOCS : SPEC.MEM TRANS_DESIGN.MEM
   PRINT SPEC.MEM,TRANS_DESIGN.MEM

SPEC.MEM : SPEC.RNO
   RUNOFF SPEC

TRANS_DESIGN.MEM : TRANS_DESIGN.RNO
   RUNOFF TRANS_DESIGN
```

This description file also uses a **macro**, CMSFLAGS. A macro is a name that represents a character string. You can use macros already provided with MMS, or you can define your own. You can define a macro at the beginning of a description file, as shown in this example, or on the DCL command line that invokes MMS. Having defined your macro, you can use the macro name in the description file in place of the character string it represents.

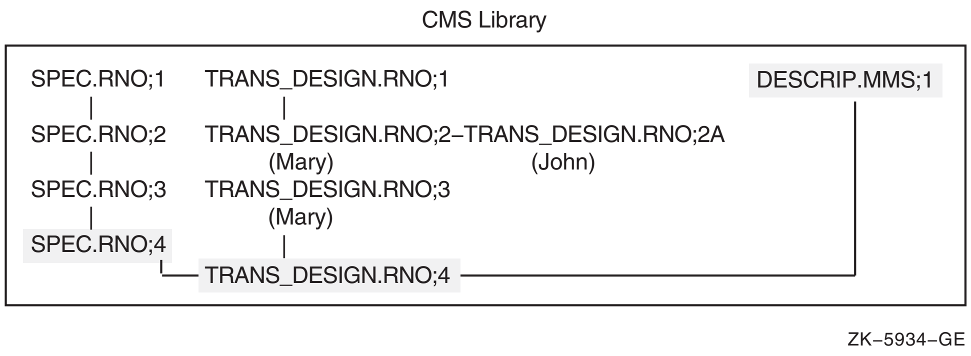
CMSFLAGS is a default macro provided by MMS, which designates that the CMS FETCH command on an MMS action line should fetch the most recent generation of an element on the main line of descent. This macro can be redefined to indicate a specific element generation, or (as in our example) an element generation that belongs to a particular class.

The documents are built using the following command to invoke MMS:

```
$ MMS/CMS
```

Without the /CMS switch, MMS will not access the CMS library for elements unless a rule specifically directs it to do so. The description file itself is placed in the CMS library, as part of the class First-status. In this way, if it becomes necessary to rebuild these specifications, the team will be able to find the description file quickly. Figure 6–9 shows the final relationships in the CMS library.

Figure 6–9 Description File as Part of a CMS Class



7 Writing, Compiling, and Debugging Source Code

The Transliteration team has accumulated a body of code consisting of multiple modules. The source files are stored in a source code library. As an aid to analyzing their code, the team has set up a project-wide SCA library. Individual developers might also use local SCA libraries for small tasks, as needed. The team has developed a standard makefile, which defines the procedure for creating new versions of TRANSLIT.

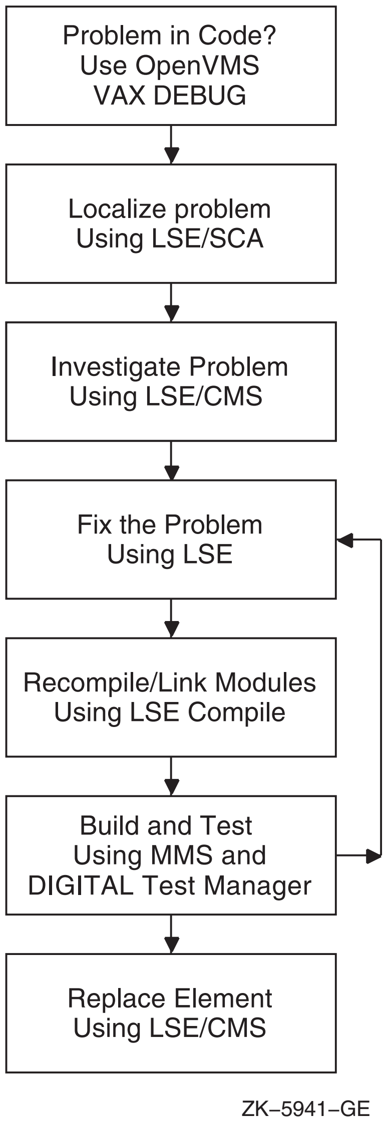
A recent enhancement to TRANSLIT provides a way to describe characters that should not be replaced by preceding them with a hyphen (-). For example, "-A-Z" should cause TRANSLIT to replace all characters except the uppercase letters A to Z.

Preliminary testing has shown that the new enhancement works as intended. The developer has returned the changed modules to the source code library. Subsequent testing shows that the code has regressed. In the process of enhancing the code, some of the original functionality was lost.

The new version of TRANSLIT translates all lowercase letters to uppercase letters, but deletes all other characters. Now when TRANSLIT is invoked, the characters contained in TEST.DAT are incorrectly translated to "ACEGIK" instead of "ABCDEFGHIJK".

This chapter shows how the Transliteration team finds and fixes this problem. Figure 7–1 shows the main steps the developers follow, as well as the DECset tools that support those steps.

Figure 7–1 Steps to Correct translit Bug


Problem in Code? Use OpenVMS VAX DEBUG
Localize problem Using LSE/SCA
Investigate Problem Using LSE/CMS
Fix the Problem Using LSE
Recompile/Link Modules Using LSE Compile
Build and Test Using MMS and DIGITAL Test Manager
Replace Element Using LSE/CMS
ZK–5941–GE

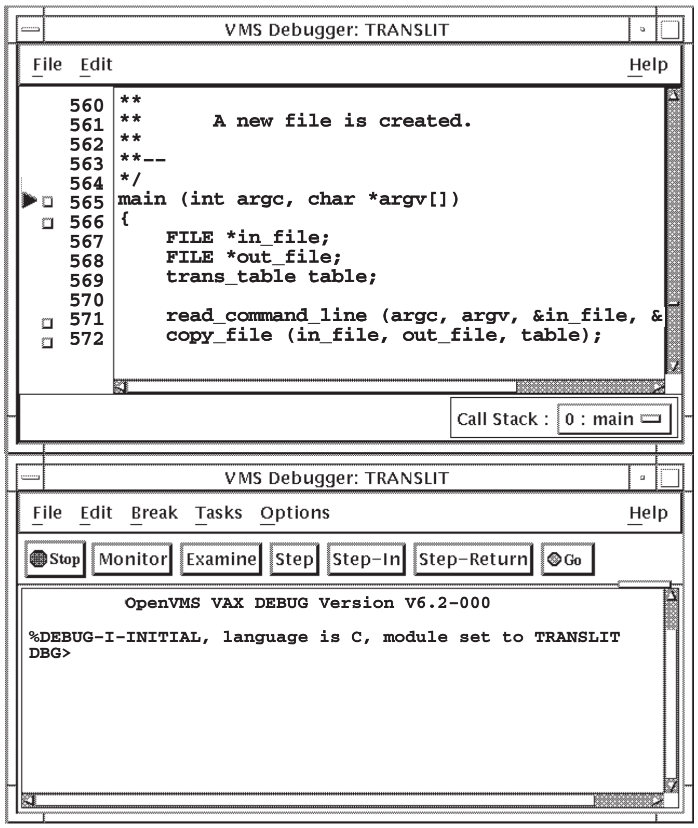
7.1 Finding the Problem

Mary has fetched the most current source files into her working directory and built a version of TRANSLIT to run with the debugger. She can begin to look for the problem by stepping through the code.

To invoke the debugger, she runs a debugger-linked executable image of TRANSLIT from a DECterm.

The debugger displays two windows, as shown in Figure 7–2. The top window, or Source View, contains the source code for the program Mary runs. A pointer is positioned at the first executable line of code. Execution pauses at this point and the debugger waits for Mary to choose an operation. The lower window, or Control View, contains a control panel. In this window, Mary can click on buttons or enter debugging commands at the prompt to control the program's execution.

Figure 7–2 Invoking the Debugger



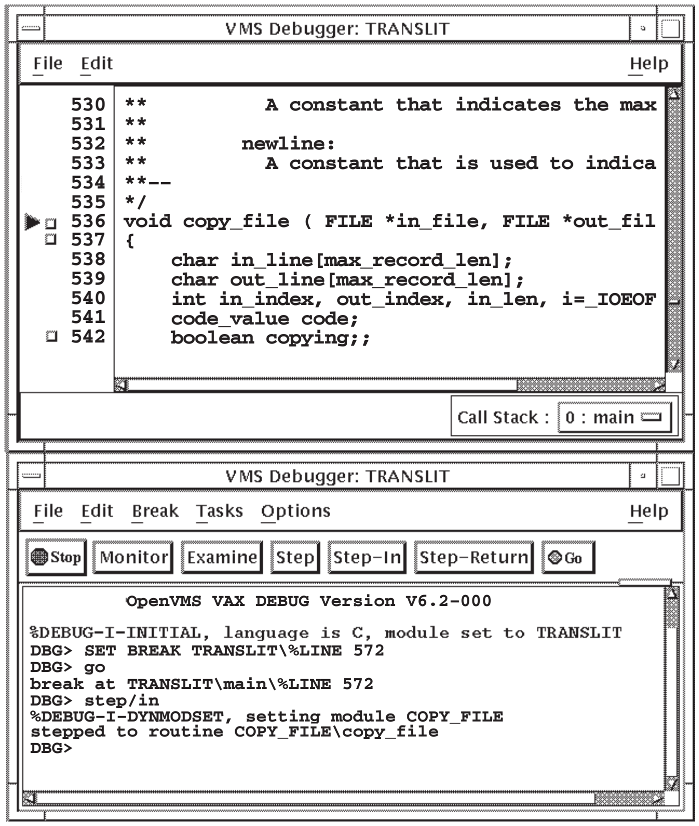
```
560 **
561 **      A new file is created.
562 **
563 **--
564 */
565 main (int argc, char *argv[])
566 {
567     FILE *in_file;
568     FILE *out_file;
569     trans_table table;
570
571     read_command_line (argc, argv, &in_file, &
572     copy_file (in_file, out_file, table);
```

```
         OpenVMS VAX DEBUG Version V6.2-000

%DEBUG-I-INITIAL, language is C, module set to TRANSLIT
DBG>
```

Mary wants to see why the characters are not being written to the output file. She suspects the error occurs during execution of the COPY_FILE function. She sets a breakpoint at that function by clicking on the box to the left of the call to COPY_FILE, at line 572. When she clicks on Go in the Control View, the debugger executes the program up to that breakpoint, then pauses and waits for her to choose another operation.

Mary wants to execute COPY_FILE line by line so she can watch the characters move from the input file to the output file. She clicks on the Step-In button. The debugger executes the function call and pauses at the first executable line in the function, as shown in Figure 7–3.

Figure 7–3 Stepping into copy_file


VMS Debugger: TRANSLIT
File Edit Help
Call Stack : 0 : main
VMS Debugger: TRANSLIT
File Edit Break Tasks Options Help
Stop Monitor Examine Step Step-In Step-Return Go

```
**          A constant that indicates the max
**
**        newline:
**          A constant that is used to indica
**--
*/
void copy_file ( FILE *in_file, FILE *out_fil
{
    char in_line[max_record_len];
    char out_line[max_record_len];
    int in_index, out_index, in_len, i=_IOEOF
    code_value code;
    boolean copying;;
```

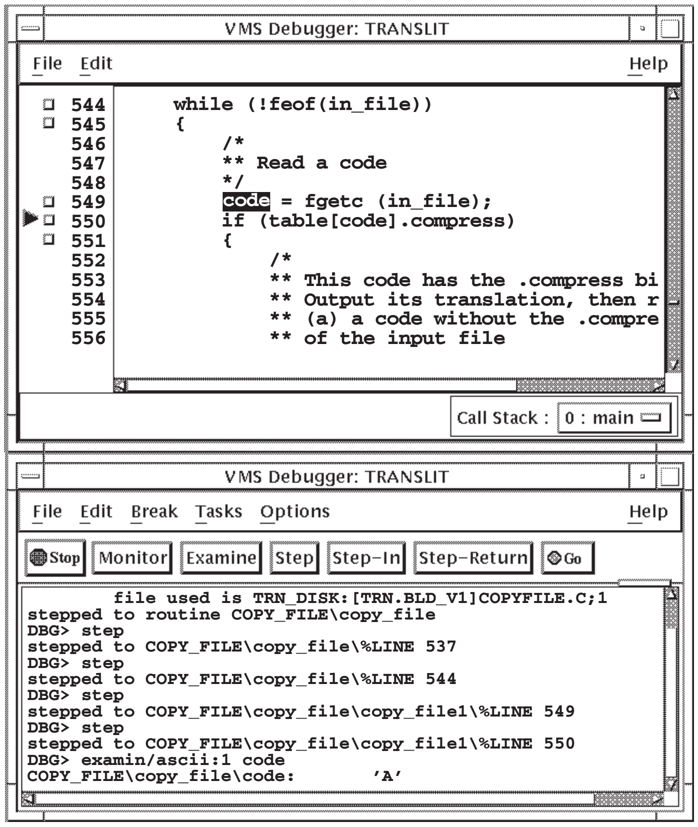
```
          OpenVMS VAX DEBUG Version V6.2-000

%DEBUG-I-INITIAL, language is C, module set to TRANSLIT
DBG> SET BREAK TRANSLIT\%LINE 572
DBG> go
break at TRANSLIT\main\%LINE 572
DBG> step/in
%DEBUG-I-DYNMODSET, setting module COPY_FILE
stepped to routine COPY_FILE\copy_file
DBG>
```

Mary then clicks on Step in the Control View to execute the function one line at a time. She stops when she reaches the first IF statement so she can examine the values of variables.

She double clicks on the variable *code* to select it. Then she clicks on the Examine button to display its ASCII code. To select the variable *table[code]*, she presses MB1 and drags the mouse cursor to the end of the name. Then she clicks on the Examine button to see the values of the structure fields.

To translate the ASCII codes to their printed characters, Mary enters the EXAMINE/ASCII:1 command in the Control View. She sees that the first character is being translated correctly from “a” to “A”. Figure 7–4 shows the result of Mary’s investigation up to this point.

Figure 7–4 Printing the Values of Variables


VMS Debugger: TRANSLIT
File Edit Help
Call Stack : 0 : main
VMS Debugger: TRANSLIT
File Edit Break Tasks Options Help
Stop Monitor Examine Step Step-In Step-Return Go

```
    while (!feof(in_file))
    {
        /*
        ** Read a code
        */
        code = fgetc (in_file);
        if (table[code].compress)
        {
            /*
            ** This code has the .compress bi
            ** Output its translation, then r
            ** (a) a code without the .compre
            ** of the input file
```

```
        file used is TRN_DISK:[TRN.BLD_V1]COPYFILE.C;1
stepped to routine COPY_FILE\copy_file
DBG> step
stepped to COPY_FILE\copy_file\%LINE 537
DBG> step
stepped to COPY_FILE\copy_file\%LINE 544
DBG> step
stepped to COPY_FILE\copy_file\copy_file1\%LINE 549
DBG> step
stepped to COPY_FILE\copy_file\copy_file1\%LINE 550
DBG> examin/ascii:1 code
COPY_FILE\copy_file\code:       'A'
```

Mary clicks on Step twice and sees that the character is written to the file with FPUTC. She clicks on Step two more times to return to the beginning of the loop, where she can repeat these steps to examine the next character in the input file.

The second time through the loop, Mary notices that the function skips over the statement that writes the character to the output file. She examines the variable *code* and finds that it has a value of 66. She verifies that this corresponds to the letter B. Mary then examines the variable *table[code]*. This tells her that TABLE[CODE].TRANS_VALUE has a value of 258. Because this value is greater than 255 (the upper limit for ASCII codes), the function does not write the character to the file.

Mary suspects the problem is in the BUILD_TABLE function, which defines how characters are translated. She decides to use LSE and SCA together to see how the *trans_value* variable is used in the various modules that make up TRANSLIT.

She enters LSE directly from the debugger by performing the following steps:

1. Enters the EDIT command in the bottom window of the debugger. This causes character-cell LSE to be invoked in the DECterm in which the debugger was invoked.
2. Clicks on the DECterm now containing LSE.

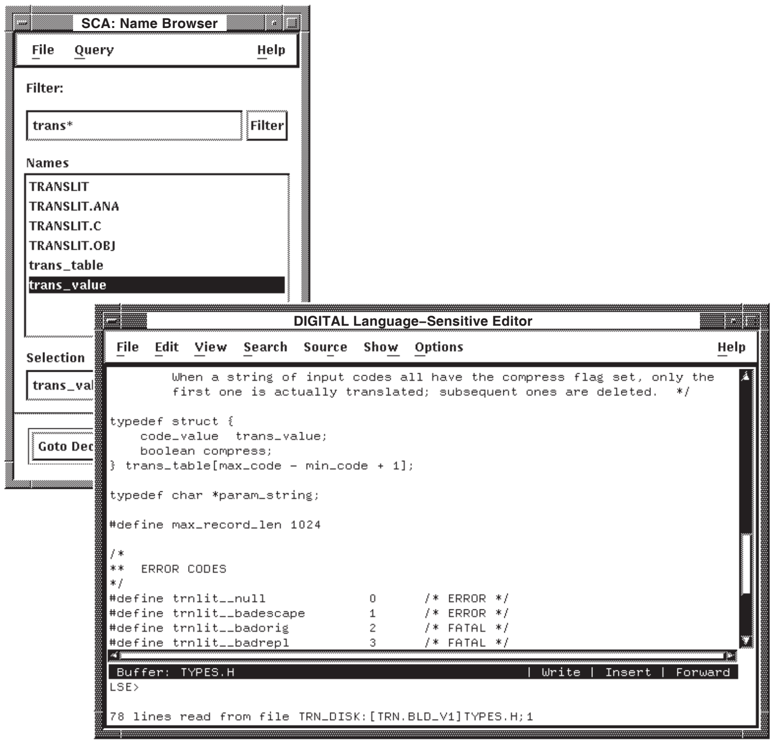
7.2 Localizing the Problem

With LSE and SCA both invoked, Mary has access to all the information stored in the project-wide SCA library and quick access to the sources. Mary uses the name browser in SCA to find the declaration for *trans_value* using the following procedure:

1. In the SCA Main window, she clicks on Name Browser... The Name Browser dialog box is displayed.
2. In the Filter: field, she enters *trans**, and clicks on Filter. All the occurrences matching that search string are displayed in the Names list box.
3. In the Names list box, she double clicks on *trans_value*.

SCA finds the declaration of *trans_value* in the TYPES.H file. LSE brings a read-only version of TYPES.H into a buffer and positions the cursor at the declaration. Figure 7–5 shows the display as it appears in the editing window.

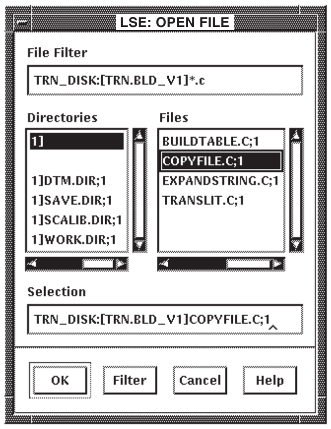
Figure 7–5 Finding the Declaration of a Variable



```
        When a string of input codes all have the compress flag set, only the
        first one is actually translated; subsequent ones are deleted.  */

typedef struct {
    code_value  trans_value;
    boolean compress;
} trans_table[max_code - min_code + 1];

typedef char *param_string;

#define max_record_len 1024

/*
**  ERROR CODES
*/
#define trnlit__null             0      /* ERROR */
#define trnlit__badescape        1      /* ERROR */
#define trnlit__badorig          2      /* FATAL */
#define trnlit__badrepl          3      /* FATAL */
 Buffer:  TYPES.H                                  | Write | Insert | Forward
LSE>

78 lines read from file TRN_DISK:[TRN.BLD_V1]TYPES.H;1
```

She opens the COPYFILE.C source file by choosing Open... from the File menu. She selects the file name from the directory list, as shown in Figure 7–6, and clicks on OK. The editor places a copy of the file in an editing buffer.

Figure 7–6 Opening a File



File Filter: TRN_DISK:[TRN.BLD_V1]*.c

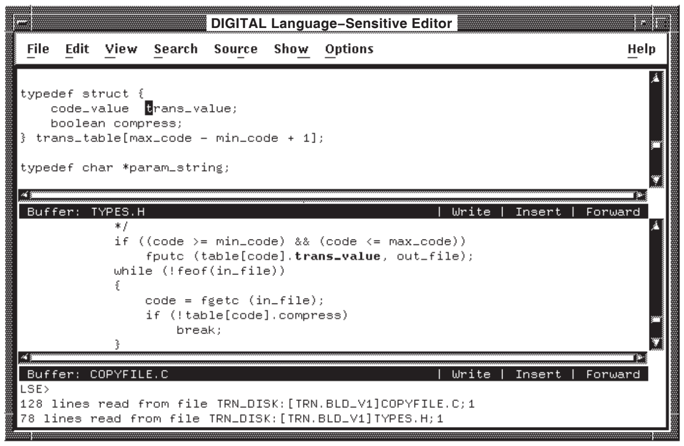
Directories	Files
1]	BUILDTABLE.C;1
	COPYFILE.C;1
1]DTM.DIR;1	EXPANDSTRING.C;1
1]SAVE.DIR;1	TRANSLIT.C;1
1]SCALIB.DIR;1	
1]WORK.DIR;1	

Selection: TRN_DISK:[TRN.BLD_V1]COPYFILE.C;1

Mary now wants to find a reference to *trans_value*. She chooses Search... from the Search menu and enters *trans_value* as the string to search for. LSE positions the cursor at the first reference to *trans_value*. This is the starting point for Mary's inquiry.

Mary double clicks on *trans_value* to select the name, then chooses Goto Declaration from the Source menu. The result displays as shown in Figure 7–7.

Figure 7–7 Finding the Declaration of a Variable


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
typedef struct {
    code_value  trans_value;
    boolean compress;
} trans_table[max_code - min_code + 1];

typedef char *param_string;
```

```
Buffer: TYPES.H                                   | Write | Insert | Forward
            */
            if ((code >= min_code) && (code <= max_code))
                fputc (table[code].trans_value, out_file);
            while (!feof(in_file))
            {
                code = fgetc (in_file);
                if (!table[code].compress)
                    break;
            }
Buffer: COPYFILE.C                                | Write | Insert | Forward
LSE>
128 lines read from file TRN_DISK:[TRN.BLD_V1]COPYFILE.C;1
78 lines read from file TRN_DISK:[TRN.BLD_V1]TYPES.H;1
```

Mary sees that *trans_value* is of type *trans_value*. She double clicks on *trans_value*, then displays its declaration by choosing Goto Declaration from the Source menu. She finds that *trans_value* is a value between MIN_CODE and MAX_CODE.

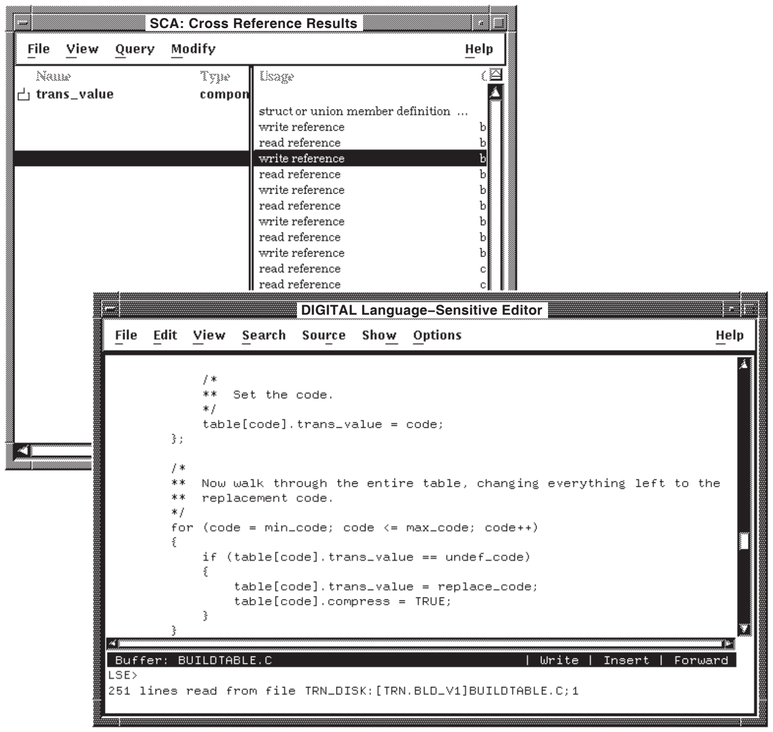
Continuing to gather more information, Mary then displays the declaration of MIN_CODE. This section of code shows that UNDEF_CODE is defined to be 258, the value she found in TABLE[CODE].*trans_value* when using the debugger.

Mary suspects that no translation was defined for uppercase characters. To find out if this is true, Mary wants to find all the places where *trans_value* is assigned a value. In the SCA window, she does the following:

1. Clicks on Cross Reference... This causes the Cross Reference dialog box to be displayed.
2. She enters *trans_value* in the Name field, and clicks on Apply. This causes the Cross Reference Results window to be displayed.
3. In the Cross Reference Results window, she double clicks on the icon to the left of *trans_value*, and SCA expands the entry to display all the locations where *trans_value* is used in the source code.

Mary finds out that *trans_value* is written in five places. She can see the source where each of the write references is indicated by double clicking on each write reference entry. Figure 7–8 shows Mary's Cross Reference Results window with the LSE window displaying the source associated with one particular write reference.

Figure 7–8 Finding All Occurrences of a Variable



Name	Type	Usage
trans_value	compon	
		struct or union member definition ...
		write reference b
		read reference b
		write reference b
		read reference b
		write reference b
		read reference b
		write reference b
		read reference b
		write reference b
		read reference c
		read reference c

```
            /*
            **  Set the code.
            */
            table[code].trans_value = code;
        };

        /*
        **  Now walk through the entire table, changing everything left to the
        **  replacement code.
        */
        for (code = min_code; code <= max_code; code++)
        {
            if (table[code].trans_value == undef_code)
            {
                table[code].trans_value = replace_code;
                table[code].compress = TRUE;
            }
        }
 Buffer: BUILDTABLE.C                            | Write | Insert | Forward
LSE>
251 lines read from file TRN_DISK:[TRN.BLD_V1]BUILDTABLE.C;1
```

Mary studies a number of these references. She decides she can fix the problem by adding a FOR loop that sets the values in the translation table to handle un-translated characters. The loop had been part of the original code. Mary speculates that while making the most recent enhancement, the developer used the original loop as a guide. In the process, the developer might have failed to reinsert the original loop back into the code.

7.3 Fixing the Problem

To change the code, Mary needs to obtain a modifiable source file from the source code library. She reserves the file BUILDTABLE.C from her CMS library directly from LSE by doing the following:

1. Clicking on the File menu.
2. Choosing Reserve.

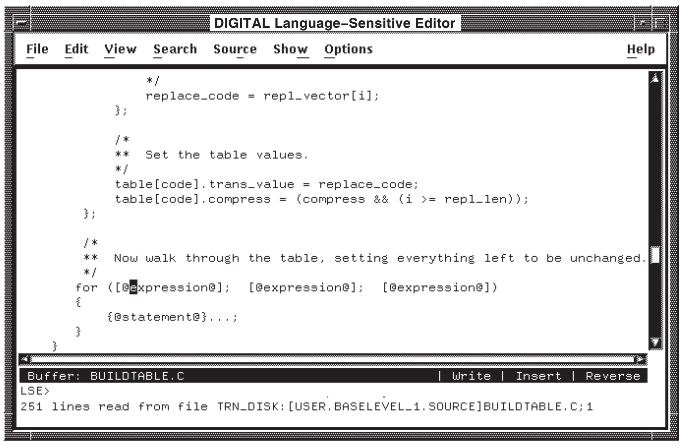
LSE reserves the source file and places it in an editing buffer. Mary can now use the language-sensitive features of LSE to modify the code.

To correct the error, Mary needs to add a FOR loop. This loop will set the values in the translation table for characters that are not translated. Mary modifies the code as follows:

1. She positions her cursor near the end of the file before the closing braces of the ELSE statement.

2. She types the word FOR then presses Ctrl/E. LSE recognizes FOR as a token and expands it as shown in Figure 7–9. The cursor is positioned at the first expression of the loop initialization portion of the statement.

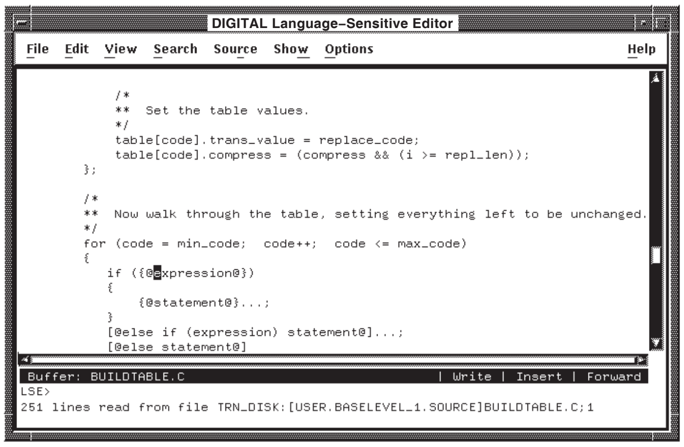
Figure 7–9 Expanding the Token of a for Statement


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
                */
                replace_code = repl_vector[i];
            };

            /*
            **  Set the table values.
            */
            table[code].trans_value = replace_code;
            table[code].compress = (compress && (i >= repl_len));
        };

        /*
        **  Now walk through the table, setting everything left to be unchanged.
        */
       for ([@expression@];  [@expression@];  [@expression@])
       {
           {@statement@}...;
       }
   }
 Buffer: BUILDTABLE.C                                   | Write | Insert | Reverse
LSE>
251 lines read from file TRN_DISK:[USER.BASELEVEL_1.SOURCE]BUILDTABLE.C;1
```

3. Mary fills in the initialization expressions by using Ctrl/N to move her cursor from one placeholder to the next. At the STATEMENT placeholder, Mary types IF and presses Ctrl/E to expand that token, as shown in Figure 7–10.

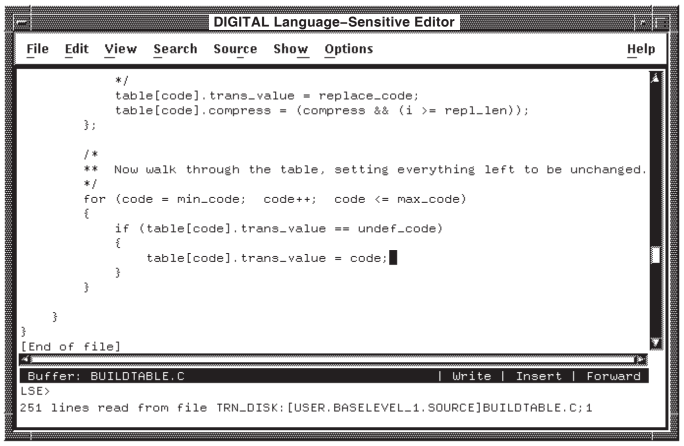
Figure 7–10 Expanding the Token of an if Statement


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
            /*
            **  Set the table values.
            */
            table[code].trans_value = replace_code;
            table[code].compress = (compress && (i >= repl_len));
        };

        /*
        **  Now walk through the table, setting everything left to be unchanged.
        */
        for (code = min_code;  code++;  code <= max_code)
        {
           if ({@expression@})
           {
               {@statement@}...;
           }
           [@else if (expression) statement@]...;
           [@else statement@]
 Buffer: BUILDTABLE.C                                        | Write | Insert | Forward
LSE>
251 lines read from file TRN_DISK:[USER.BASELEVEL_1.SOURCE]BUILDTABLE.C;1
```

4. She fills in the expression and statements and deletes the extra placeholders. The completed loop is shown in Figure 7–11.

Figure 7–11 Completing Changes to the Source Code


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
            */
            table[code].trans_value = replace_code;
            table[code].compress = (compress && (i >= repl_len));
        };

        /*
        **  Now walk through the table, setting everything left to be unchanged.
        */
        for (code = min_code;  code++;  code <= max_code)
        {
            if (table[code].trans_value == undef_code)
            {
                table[code].trans_value = code;
            }
        }

    }
}
[End of file]
 Buffer: BUILDTABLE.C                                | Write | Insert | Forward
LSE>
251 lines read from file TRN_DISK:[USER.BASELEVEL_1.SOURCE]BUILDTABLE.C;1
```

Having modified the file, Mary now wants to see if her changes are syntactically correct. She chooses Compile from the Source menu.

7.4 Compiling and Reviewing Source Code

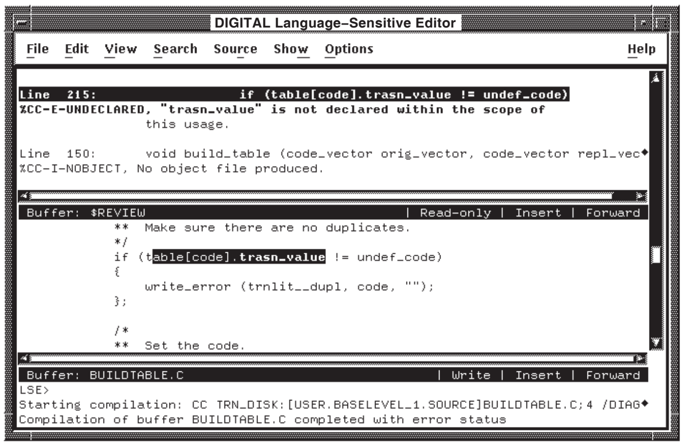
Mary can now perform the following functions within LSE:

- Compile the source code.
- Review any error conditions in a diagnostics file.
- Move automatically from an error message in the diagnostics file to the line in the source code that generated it.
- Edit the line in the source module that generated the error.

The example shown in this section shows how to compile and review a module in TRANSLIT.

To compile and review the code in the current buffer, Mary chooses the Compile Review item in the Source menu. LSE displays a message in the message area at the bottom of the window when the compilation is complete. LSE then automatically begins reviewing the compilation errors generated by the Compile Review function. LSE splits the screen into two windows, displaying the compilation errors in the top window and the source code in the bottom window, as shown in Figure 7–12.

Figure 7–12 LSE Review Buffer


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
Line  215:                 if (table[code].trasn_value != undef_code)
%CC-E-UNDECLARED, "trasn_value" is not declared within the scope of
                this usage.

Line  150:      void build_table (code_vector orig_vector, code_vector repl_vec◆
%CC-I-NOBJECT, No object file produced.

 Buffer: $REVIEW                              | Read-only | Insert | Forward
            **  Make sure there are no duplicates.
            */
            if (table[code].trasn_value != undef_code)
            {
                write_error (trnlit__dupl, code, "");
            };

            /*
            **  Set the code.
 Buffer: BUILDTABLE.C                              | Write | Insert | Forward
LSE>
Starting compilation: CC TRN_DISK:[USER.BASELEVEL_1.SOURCE]BUILDTABLE.C;4 /DIAG◆
Compilation of buffer BUILDTABLE.C completed with error status
```

Once LSE displays both windows, Mary can position the pointer at an error message in the top window and double click on it to move the editing cursor to the line in the source code (in the bottom window) that generated that error. She performs the following steps:

1. Clicks MB2 in the $REVIEW buffer. LSE displays the Review buffer pop-up menu.
2. Chooses End Review.

When no errors are reported, she decides to build and test the program as part of the full system. Mary returns the BUILDTABLE.C file to the source code library. She enters the following REPLACE command at the LSE prompt:

```
LSE Command> REPLACE BUILDTABLE.C
```

This information becomes part of the history for that file.

7.5 Building the Program

Mary needs to compile and link the full application, including the modified module, to make sure the application executes as intended.

Mary is now ready to return the file to CMS. She can directly replace the element, now verified as correct, by performing the following steps:

1. Clicking on the File menu.
2. Choosing Replace.

Mary exits from SCA and LSE.

Working from her local build area ([JONES.WORK]), Mary initiates the MMS build procedure with the following DCL command:

```
$ MMS/CMS WORK_BUILD
```

When the build is complete, the sources have been compiled and linked; the project's HP DIGITAL Test Manager test collection has also been run automatically. Mary then uses the HP DIGITAL Test Manager to review the test collection, and sees that the TRANSLIT utility now produces results that match the benchmark. Her code changes have been successful.

8 Designing Programs with the Program Design Facility

This chapter presents an overview of the program design facility and an example of how the Transliteration project team uses it to design and build the COPY_FILE module.

8.1 Overview

DECset lets you use the program design facility for incremental program design and development by enabling you to specify design information in your source files in the form of structured text, or **pseudocode**. Using a compiler that supports the program design facility, you can then compile these modules and generate SCA analysis data files from them. With SCA, you can perform cross-referencing and static analysis on the design information on these modules, and subsequently extract design information and produce it in a variety of design reports. The following OpenVMS compilers support the program design facility:

- HP Ada
- HP BASIC
- VAX BLISS-32
- HP C
- HP C++
- HP COBOL
- HP VAX COBOL
- HP FORTRAN
- DEC Pascal
- VAX PL/I

The program design facility adds the following capabilities to your DECset programming environment:

- Pseudocode support
- Design refinement: developing real code from pseudocode while preserving the design information in the form of comments
- Tagged comments: allowing you to categorize the design information contained in comments
- Support for compiling files containing pseudocode
- SCA cross referencing and static analysis of design information
- Generation of design reports
- Code elision: obtaining collapsed or expanded views of your modules and design reports (also known as outlining or holophrasting)

The following sections provide details on each of these capabilities.

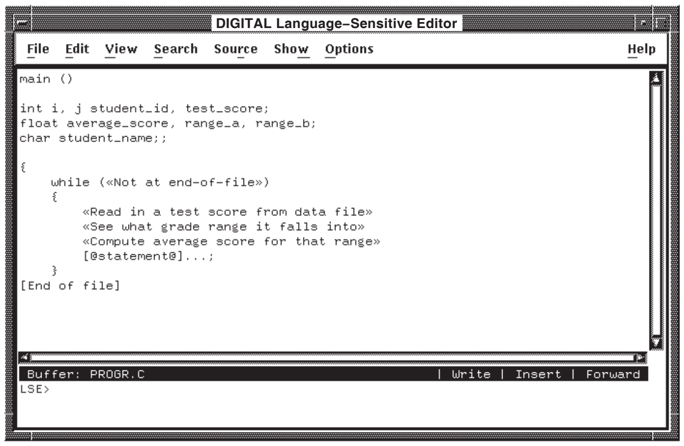
8.1.1 Entering Pseudocode

LSE provides the ENTER PSEUDOCODE command to let you create pseudocode placeholders, in which you can specify design information. When you use the ENTER PSEUDOCODE command, LSE displays the **pseudocode placeholder**.

The pseudocode placeholder appears as a pair of double angle bracket characters («»). When LSE displays this placeholder, you can begin typing in natural language statements, which make up your pseudocode. In this way, the compiler language you are using becomes your program design language.

Figure 8–1 shows a routine containing pseudocode.

Figure 8–1 Sample Routine Containing Pseudocode


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
main ()

int i, j student_id, test_score;
float average_score, range_a, range_b;
char student_name;;

{
    while («Not at end-of-file»)
    {
        «Read in a test score from data file»
        «See what grade range it falls into»
        «Compute average score for that range»
        [@statement@]...;
    }
[End of file]
```

```
 Buffer: PROGR.C                                   | Write | Insert | Forward
LSE>
```

8.1.2 Refining the Design

LSE provides the ENTER COMMENT command to convert pseudocode placeholders to program comments. This mechanism helps the development process by preserving the original design of a program as you replace pseudocode with real code. You can convert pseudocode into block or line comments.

Using the keypad, you can press PF1-L (the ENTER COMMENT LINE command) or PF1-B (the ENTER COMMENT BLOCK command). When you press PF1-L, LSE does the following:

- Moves the pseudocode placeholder text into a comment in the right margin
- Inserts a {TBS} placeholder in the original position of the pseudocode placeholder
- Places the cursor in the {TBS} placeholder, which allows you to enter a line of code

When you press PF1-B, LSE does the following:

- Sets up a comment block

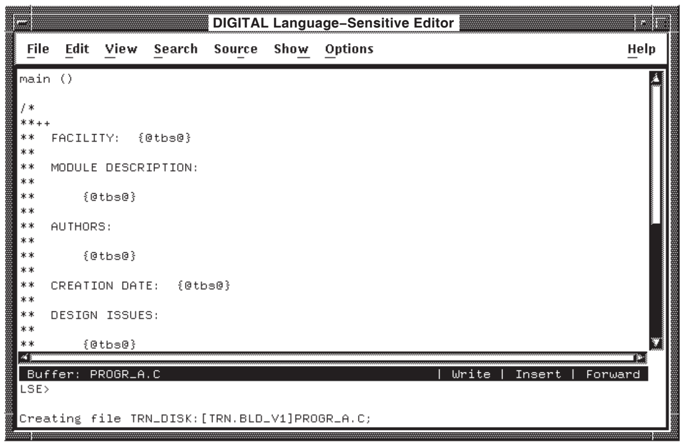
- Places the pseudocode placeholder text within any necessary comment delimiters inside the block
- Creates a {TBS} placeholder after the comment block and positions the cursor in it

8.1.3 Using Tagged Comments

Another feature of the program design facility is support for **tagged comments**. Tagged comments let you extract design information from comments in an organized fashion. This feature is especially useful for generating reports. When you generate a report, you can extract the design information associated with tags and it appears in the report.

LSE provides language templates for compilers supporting the program design facility. These templates contain tags for specific categories of information commonly appearing in comments. For example in C, when you expand the initial {@COMPILATION-UNIT@} placeholder, one of the placeholders LSE displays is [@MODULE LEVEL COMMENTS@]. When you expand this placeholder, LSE displays the comment block shown in Figure 8–2.

Figure 8–2 Tags in a C Header Comment


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
main ()

/*
**++
**  FACILITY:  {@tbs@}
**
**  MODULE DESCRIPTION:
**
**      {@tbs@}
**
**  AUTHORS:
**
**      {@tbs@}
**
**  CREATION DATE:  {@tbs@}
**
**  DESIGN ISSUES:
**
**      {@tbs@}
```

```
 Buffer: PROGR_A.C                               | Write | Insert | Forward
LSE>

Creating file TRN_DISK:[TRN.BLD_V1]PROGR_A.C;
```

In Figure 8–2, note the following tags:

- FACILITY:
- MODULE DESCRIPTION:
- AUTHORS:
- CREATION DATE:
- DESIGN ISSUES:

When SCA generates a report for a module containing tagged comments, it supplies the text associated with these tags. The program design facility provides three categories of tags:

- Text
- Keyword
- Structured

Text tags contain ordinary text. The tags shown in Figure 8–2 are text tags. Keyword tags contain a list of zero or more keywords and can be used to flag sections of code. You can define a tag to accept keywords from a predefined list. The keywords in the predefined list are defined with the LSE DEFINE KEYWORD command.

Structured tags add another layer of structure to a tag. The value of a structured tag consists of a sequence of one or more subtags. For example, the FORMAL PARAMETERS tag consists of a sequence in which each parameter name is a subtag, and the text description of the parameter becomes the value of the subtag.

Using tags in your source files requires that you follow special syntax guidelines so the compilers will recognize the tags and process them correctly. The *Guide to Detailed Program Design for OpenVMS Systems* has complete details on specifying tagged comments in your source files. You can also add your own tags to represent various kinds of design information. LSE provides the DEFINE TAG and DEFINE KEYWORDS commands to create your own tags and keywords, and the SAVE ENVIRONMENT command to save your new tag and keyword definitions.

8.1.4 Compiler Support

After a partial or complete design exists, you can process files containing design information by using a supported compiler. Use the /DESIGN qualifier to tell the compiler to process design information. The /DESIGN qualifier accepts the following two keywords:

- [NO]COMMENT

  Causes the compiler to process comment tags associated with keywords

- [NO]PLACEHOLDERS

  Causes the compiler to recognize pseudocode placeholders as valid program syntax

The default keywords for the /DESIGN qualifier are COMMENT and PLACEHOLDERS.

8.1.5 Loading Design Information into an SCA Library

You can use the SCA LOAD command to load the analysis data files containing design information, as described in Section 8.1.4. Because SCA does not discriminate between an analysis data file containing design information and one containing real code, you can place your designs and implementations in the same library.

Once your design files are loaded into an SCA library, you can perform SCA static analysis and cross-referencing procedures as you would on files containing pure code.

8.1.6 Generating Design Reports

In addition to getting information directly from SCA queries, you can also produce a variety of reports from the SCA database based on your design. Typically, reports cover all or a designated part of your SCA database and present information in a structured, organized way. To generate reports, you must have both LSE and SCA available on your system.

Design reports can present information about your program in ways that you can use to better understand the program. The program design facility provides four types of reports:

- **HELP**—An OpenVMS Help file generated from your design or code
- **PACKAGE**—An LSE package definition
- **INTERNALS**—A general report that formats your entire design in an organized manner

- **2167A_DESIGN**—A report that produces a document that meets the requirements of the US Defense Department's DOD-STD-2167A Software Design Document

An example of an internals report is given at the end of this chapter. For more information on the other reports, see the *Guide to Detailed Program Design for OpenVMS Systems*.

8.1.7 Code Elision

LSE lets you obtain overviews of your code or designs at various levels of detail with the EXPAND and COLLAPSE commands. This is sometimes referred to as code elision, holophrasting, or outlining. You can generate overviews of your source files either interactively in LSE, or in reports. Section 8.2.4 gives an example of generating reports.

When you collapse the lines in your source code, LSE displays **overview lines**. Each overview line corresponds to one or more lines of code. When you collapse lines, you can hide the details of your programs. LSE displays overview lines as pseudocode placeholders.

You can also expand collapsed code to reveal the individual lines previously hidden from view. For examples and more information on code elision, see the *Guide to Detailed Program Design for OpenVMS Systems*.

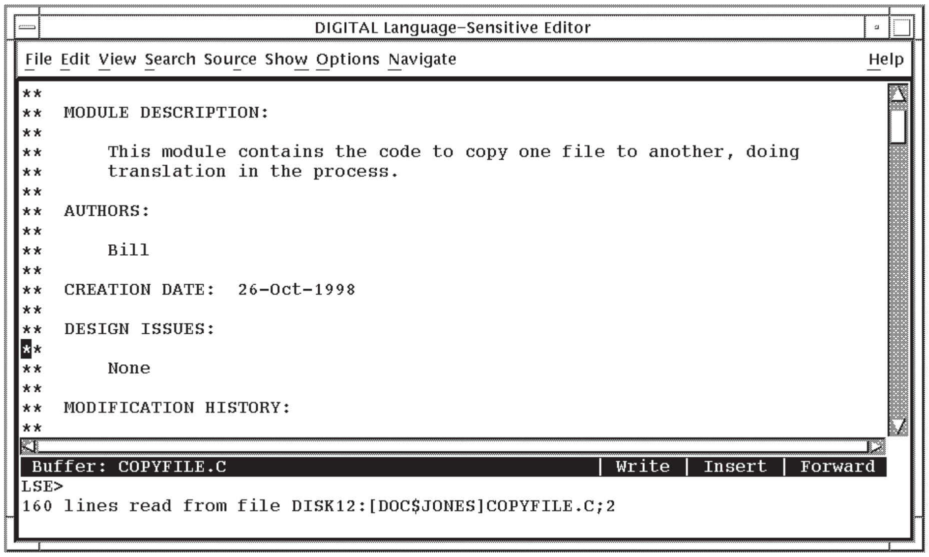
8.2 Designing the COPY_FILE Module

This section shows the steps that the Transliteration project team takes to design and build the COPY_FILE module of TRANSLIT. Bill, one of the Transliteration team developers, has the task of using the program design facility to provide the detailed level design in this module.

8.2.1 Beginning the Design for COPY_FILE

To start the design process, Bill begins filling in header comment information, providing details for the MODULE DESCRIPTION, AUTHORS, CREATION DATE tags, as shown in Figure 8–3.

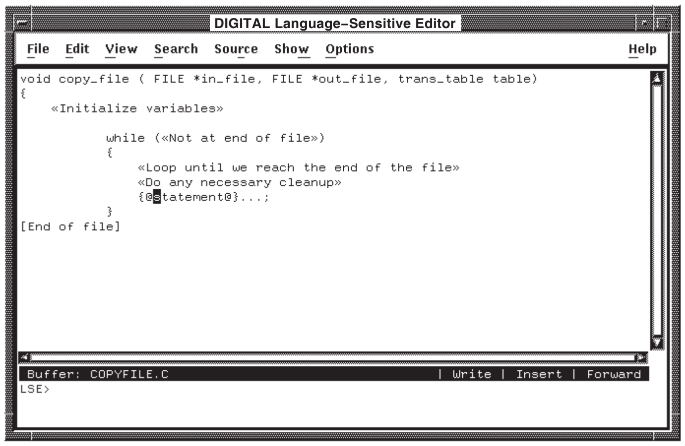
Figure 8–3 Providing Header Comments for COPY_FILE


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
**
**  MODULE DESCRIPTION:
**
**      This module contains the code to copy one file to another, doing
**      translation in the process.
**
**  AUTHORS:
**
**      Bill
**
**  CREATION DATE:  26-Oct-1998
**
**  DESIGN ISSUES:
**
**      None
**
**  MODIFICATION HISTORY:
**
 Buffer: COPYFILE.C                                   | Write | Insert | Forward
LSE>
160 lines read from file DISK12:[DOC$JONES]COPYFILE.C;2
```

Bill then writes a top-level outline of the procedure, as shown in Figure 8–4.

Figure 8–4 COPY_FILE Top-Level Outline


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
void copy_file ( FILE *in_file, FILE *out_file, trans_table table)
{
    «Initialize variables»

           while («Not at end of file»)
           {
               «Loop until we reach the end of the file»
               «Do any necessary cleanup»
               {@statement@}...;
           }
[End of file]
```

```
 Buffer: COPYFILE.C                                  | Write | Insert | Forward
LSE>
```

The design information is enclosed with double angle brackets («»), which indicate pseudocode. When you press PF1-spacebar (the ENTER PSEUDOCODE command), LSE displays the double angle brackets as a placeholder, positioning the cursor on it.

Next, Bill expands the loop. He places the cursor on the following line and types PF1-B (the ENTER COMMENT BLOCK command):

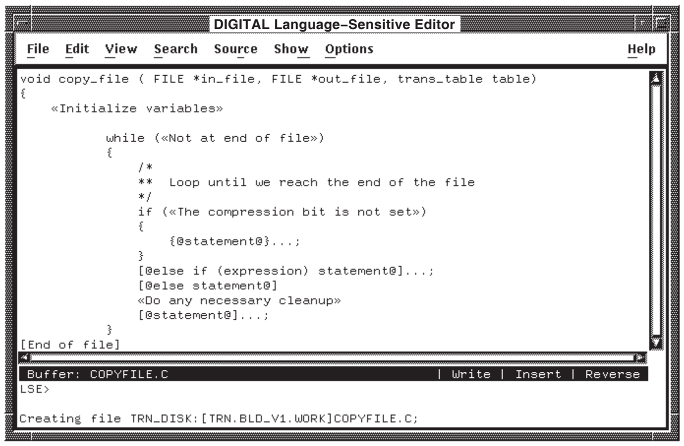
```
«Loop until we reach the end of the file»
```

The ENTER COMMENT BLOCK command does the following:

- Converts a single line of pseudocode into a block-format comment
- Creates a %[TBS]% placeholder and positions the cursor in it

Bill creates a WHILE loop, shown in Figure 8–5, and completes the loop with pseudocode placeholders.

Figure 8–5 Creating a while Loop with Pseudocode Placeholders

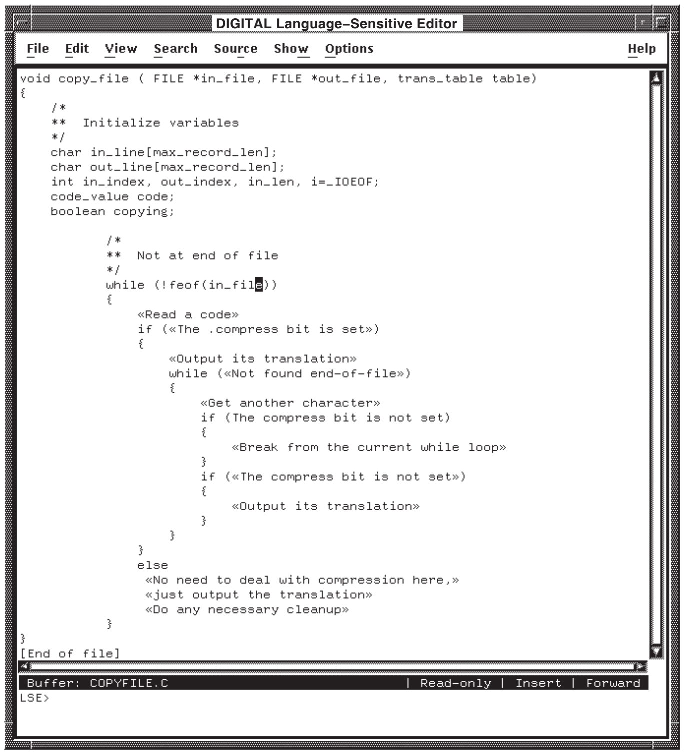

DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
void copy_file ( FILE *in_file, FILE *out_file, trans_table table)
{
    «Initialize variables»

           while («Not at end of file»)
           {
               /*
               **  Loop until we reach the end of the file
               */
               if («The compression bit is not set»)
               {
                   {@statement@}...;
               }
               [@else if (expression) statement@]...;
               [@else statement@]
               «Do any necessary cleanup»
               [@statement@]...;
           }
[End of file]
 Buffer: COPYFILE.C                                  | Write | Insert | Reverse
LSE>

Creating file TRN_DISK:[TRN.BLD_V1.WORK]COPYFILE.C;
```

When Bill is ready to provide actual code where he has a pseudocode placeholder, he positions the cursor on that placeholder and presses PF1-B (the ENTER COMMENT BLOCK command).

Figure 8–6 shows Bill's work up to this point.

Figure 8–6 Replacing Pseudocode with Actual Code


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Help

```
void copy_file ( FILE *in_file, FILE *out_file, trans_table table)
{
    /*
    **  Initialize variables
    */
    char in_line[max_record_len];
    char out_line[max_record_len];
    int in_index, out_index, in_len, i=_IOEOF;
    code_value code;
    boolean copying;

           /*
           **  Not at end of file
           */
           while (!feof(in_file))
           {
               «Read a code»
               if («The .compress bit is set»)
               {
                   «Output its translation»
                   while («Not found end-of-file»)
                   {
                       «Get another character»
                       if (The compress bit is not set)
                       {
                           «Break from the current while loop»
                       }
                       if («The compress bit is not set»)
                       {
                           «Output its translation»
                       }
                   }
               }
               else
                «No need to deal with compression here,»
                «just output the translation»
                «Do any necessary cleanup»
           }
}
[End of file]
 Buffer: COPYFILE.C                                  | Read-only | Insert | Forward
LSE>
```

8.2.2 Compiling COPY_FILE

Bill is now ready to do an initial compile of COPY_FILE. He uses the /DESIGN qualifier, so the C compiler will recognize the pseudocode placeholders and that the /ANALYSIS_DATA qualifier will generate an SCA analysis data file containing the design information. The compile line is as follows:

```
$ CC COPYFILE/NOOBJECT/DESIGN/ANALYSIS_DATA
```

8.2.3 Loading COPY_FILE Design Information into an SCA Library

Bill enters the following command to load the analysis data file created in Section 8.2.2 into his SCA library:

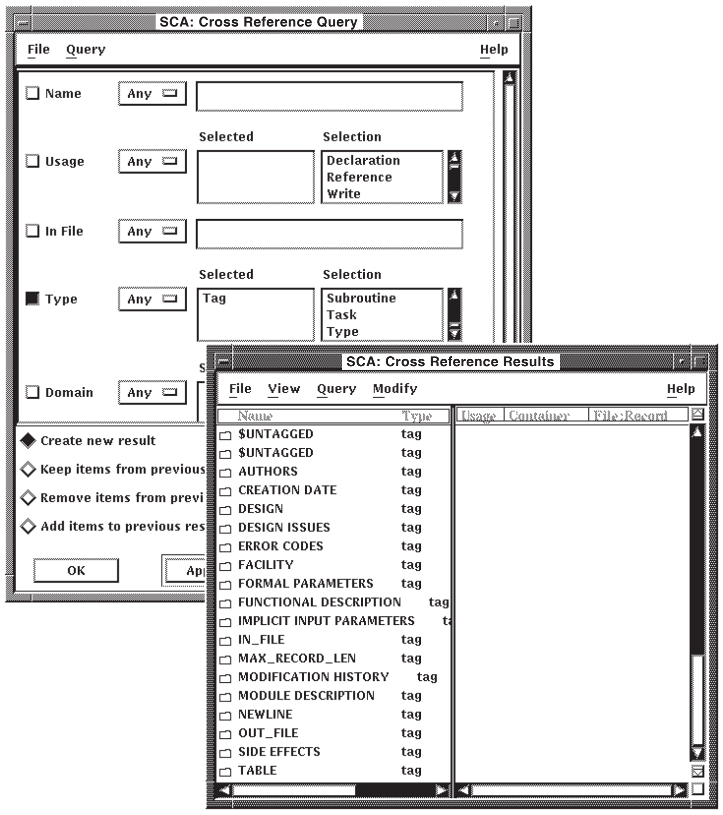
```
$ SCA LOAD COPY_FILE/LIBRARY=MYSCALIB
```

Bill can then perform SCA query operations on his design; for example, checking the usage of tags. He uses the following procedure:

1. In the SCA main window, he clicks on Cross Reference...
2. In the list box to the right of the Type button, he selects Tag.
3. He clicks on Apply.

Figure 8–7 shows the output created by Bill's cross-referencing procedure.

Figure 8–7 SCA Cross-Reference Output



☐ Name — Any

☐ Usage — Any — Selected: (none) — Selection: Declaration, Reference, Write

☐ In File — Any

■ Type — Any — Selected: Tag — Selection: Subroutine, Task, Type

☐ Domain — Any

◆ Create new result
◇ Keep items from previous
◇ Remove items from previ
◇ Add items to previous res

Name	Type
$UNTAGGED	tag
$UNTAGGED	tag
AUTHORS	tag
CREATION DATE	tag
DESIGN	tag
DESIGN ISSUES	tag
ERROR CODES	tag
FACILITY	tag
FORMAL PARAMETERS	tag
FUNCTIONAL DESCRIPTION	tag
IMPLICIT INPUT PARAMETERS	t
IN_FILE	tag
MAX_RECORD_LEN	tag
MODIFICATION HISTORY	tag
MODULE DESCRIPTION	tag
NEWLINE	tag
OUT_FILE	tag
SIDE EFFECTS	tag
TABLE	tag

Usage	Container	File:Record

As shown in Figure 8–7, SCA displays a buffer listing all the tags in COPY_FILE.

8.2.4 Generating an Internals Report of COPY_FILE

Bill's next task is to generate an INTERNALS report of the COPY_FILE design to deliver to the other members of the Transliteration project team. The INTERNALS report is a comprehensive module-by-module, routine-by-routine report that extracts information from tagged comments to describe various aspects of COPY_FILE.

Bill enters the following command to generate an INTERNALS report while in SCA. This command generates a DECdocument source file, which he can subsequently process in DECdocument. The command is as follows:

```
SCA> SET REPORT/INTERNALS/TARGET=DOCUMENT
```

The report created by the last command is shown at the end of this chapter. Note the following characteristics of the code body shown in this report:

- The program statements in the code body are presented in a top-down, hierarchical order—not in the sequential order in which they appear in the source file. Each block is labeled with a numbered callout in the left margin.
- The program design facility inserts overview placeholders to hide details at the uppermost levels (shown as a placeholder with an ellipsis appended to the end). For each overview placeholder that appears in the body, there is a cross-reference callout in the right margin. This callout cross-references the block in which the overview placeholder is expanded.

In DECdocument output, callouts appear as white-on-black numbers. In Runoff output, callouts appear as boldface numbers.

COPY_FILE (void function)

Copy the input file to the output file, with character translation.

Format

COPY_FILE   in_file, out_file, table

Arguments

in_file

Type: pointer to FILE
The input file

out_file

Type: pointer to FILE
The output file

table

Type: trans_table
The translation table

Functional Description

Copy the input file to the output file, with character translation.

Side Effects

Data is read from the input file, and written to the output file.

Design

Use a simple while loop to move through the input file. Each iteration of the while loop represents the processing of a single character or the end-of-line character.

Implicit Input Parameters

max_record_len—A constant that indicates the maximum record length

newline—A constant that is used to indicate a new-line character

COPYFILE
COPY_FILE (void function)

Body

```
1     «while (!feof(in_file)) ...» 2
  }
2 while (!feof(in_file))
  {
      «** Read a code ...» 3
  }
3 /*
  ** Read a code
  */
  code = fgetc (in_file);
  «if (table[code].compress) ...» 4
4 if (table[code].compress)
  {
      «** This code has the .compress bit set. ...» 5
  }
  else
      «** No need to deal with compression here -- just output the ...» 9
5 /*
  ** This code has the .compress bit set.
  ** Output its translation, then read until we find either
  ** (a) a code without the .compress bit set, or (b) the end
  ** of the input file
  */
  «if ((code >= min_code) && (code <= max_code)) ...» 6
  «while (!feof(in_file)) ...» 7
  «if (!table[code].compress) ...» 8
6 if ((code >= min_code) && (code <= max_code))
      fputc (table[code].trans_value, out_file);
7 while (!feof(in_file))
  {
      code = fgetc (in_file);
      if (!table[code].compress)
          break;
  }
8 if (!table[code].compress)
      /*
      ** We found a code without the .compress bit set.
      ** Output its translation.
      */
      if ((code >= min_code) && (code <= max_code))
          fputc (table[code].trans_value, out_file);
9 /*
  ** No need to deal with compression here -- just output the
  ** translation.
  */
  if ((code >= min_code) && (code <= max_code))
  fputc (table[code].trans_value, out_file);
```

Note

The callouts shown in this example are actually a part of the report. Final expanded items are marked with the callout figures seen on the left side. A right-side callout shows pseudocode to be expanded.

9 Building the Application with MMS

This chapter describes the MMS description file used by the Transliteration project team to build the following Transliteration applications:

- Previous versions of their software
- A new stage, or **base level**, of the project's development

This chapter is divided into the following four sections:

- Section 9.1 describes the description file in general and lists what must be in place for it to execute properly.
- Section 9.2 describes the section of the description file devoted to MMS flags and macros.
- Section 9.3 describes the section of the description file that sets up MMS rules.
- Section 9.4 describes the section of the description file that sets up the MMS targets.

9.1 Building from CMS Classes

Building previous versions depends in part on how conscientiously the team has used CMS to create **classes** or stages in the application's development. With the necessary class in place, the team can use MMS to build a previous version from the sources in the CMS library.

The same MMS description file can build the ongoing development work as well as previous versions of the software. The description file can also initiate test procedures, with or without PCA coverage analysis, during the build. Additionally, the build procedure can automatically update the SCA library.

To carry out these different tasks during the build, the following must be in place:

- CMS, HP DIGITAL Test Manager, and SCA libraries
- The logicals for the default libraries
- Tests for the application
- A PCA Collector initialization file
- A HP DIGITAL Test Manager test collection prologue file
- A class in the CMS code library for a previous version; a class in the CMS library for HP DIGITAL Test Manager that uses the same name, which incorporates the tests that correspond to the source class.

The MMS description file shown in Example 9–1 carries out builds for both new and previous development. Only the MMS command entry changes. The description file also can run test collections on the build and optionally perform coverage analysis on the test collection. Finally, it can update the SCA library (usually done if the build constitutes a new version of the system).

Developers can initiate the build from within the group project area, [TRN.BLD_V1.WORK], or from their own local work areas, for example, [JONES.WORK]. Building from the group project area is appropriate during a project build using validated sources. This compilation will update the project-wide SCA library ([TRN.BLD_V1.SCALIB]).

A build from the local area is appropriate when a developer wants to see the effects of changes made to modules under development. In this case, MMS will do a time comparison between the modules in the developer's local directory and those in CMS. MMS carries out the build using any modules in the local directory that are more recent than those in the CMS code library. The resulting files are stored in the developer's local work area. In this case, the local SCA library ([JONES.SCALIB]) is updated based on this compilation.

Example 9–1 shows the complete MMS description file. This is followed by sections of the file with explanatory text.

Example 9–1 A Build Procedure Using an MMS Description File

```
!++
!                      BUILD Procedure
!--

!==================
! FLAGS and MACROS
!==================


! Create default flags and macros
sca    = sca
dtm    = dtm
delete = delete
dtmlibrary     = [.dtm]    ! Default DTM library
scalibrary     = sca$library          ! Default SCA library
pca            = false
pca_init_file  = trn_coms:myinitfile.pcac ! Default PCAC Init file

dtmcollection  = build_test   ! Default collection name
dtmtests       = *    ! Use all tests
dtmcomment     = "Test of build"  ! Default collection remark

trace          = trace
list           = list
debug          = debug
optimize       = nooptimize
analysis_data  = analysis_data
cflags       = /$(debug) /nooptimize /$(analysis_data)
cldflags       = /$(list)
msgflags       = /$(list)

dtmflags       = /submit=(notify,noprint,keep)/class=(template:"$(mms$cms_gen)", -
   benchmark:"$(mms$cms_gen)")/var=(use_pca=$(pca),use_pca_init_file=$(pca_init_file), -
   translit="''f$parse("translit.exe")'" )
linkflags      =/map=$(mms$target_name)/exe=$(mms$target_name)/$(trace)
!Modules in build
!copyfile.obj
translit_modules= buildtable.obj, copyfile.obj, -
    expandstring.obj, translit.obj

!=============
! RULES
!=============

! Modify the C rule to decide if analysis data files should be produced.

.c.obj  ! a rule for compiling C files
if use_sca .eq. 0 THEN $(cc) $(cflags)/noanalysis_data $(mms$source)
if use_sca .eq. 1 then $(cc) $(cflags) $(mms$source)
if use_sca .eq. 1 then $(sca) load $(mms$target_name)
if use_sca .eq. 1 then $(delete) $(mms$target_name).ana;
```

(continued on next page)

Example 9–1 (Cont.) A Build Procedure Using an MMS Description File

```
!===============
! TARGETS
!===============

.first   ! Set a flag to determine how objects are to be compiled
use_sca = 0
if "$(mmstargets)" .eqs. "work_build" .or. "$(mmstargets)" .eqs. "" -
      then use_sca = 1
pca_link = ""
if "$(pca)" .nes. "false" then pca_link = "/debug"

! The normal development build (default if no other target specified)

work_build : translit.exe, -       ! Build the translit executable image
                  test_set  ! Run the regression tests
                   ! Work build completed at this point.

! To build a particular version -- use same as work build, but different target
! will allow .FIRST directive to cause C not to generate .ANA files.

version_build : work_build
                ! Version build completed at this point

!The translit executable

translit.exe : $(translit_modules)   ! put modules into an olb
  $(link) $(linkflags) $(translit_modules) 'pca_link'
  if "$(debug)" .eqs. "debug" then $(link) $(linkflags)/debug/exe= -
        $(mms$target_name).debug $(translit_modules)

! The test set

test_set :  ! Always run a test set
 $(dtm) set library $(dtmlibrary)
 $(dtm) create collection $(dtmcollection) $(dtmtests) -
  $(dtmflags) $(dtmcomment)
 ! test set created at this point

! Source code dependencies to create the needed .OBJ file which
! is source for the target TRANSLIT.EXE.

translit.obj     : translit.c, openfiles.h, types.h
buildtable.obj   : types.h
copyfile.obj     : types.h
expandstring.obj : types.h
openfile.obj  : types.h
```

In the next three examples, the single description file in Example 9–1 has been broken into sections, each followed by explanatory text: Example 9–2, Flags and Macros; Example 9–3, Rules; and Example 9–4, Targets.

9.2 MMS Flags and Macros

This section shows and describes the flags and macros section of the MMS description file shown in this chapter. Example 9–2 is followed by a detailed explanation of each part of the example.

Example 9–2 Flags and Macros Section of the Description File

```
  !++
  !                    BUILD Procedure
  !--

  !==================
  ! FLAGS and MACROS
  !==================


  ! Create default flags and macros
1 sca    = sca
  dtm    = dtm
  delete = delete
  dtmlibrary     = [.dtm]    ! Default DTM library
  scalibrary     = sca$library         ! Default SCA library
  pca            = false
  pca_init_file  = trn_coms:myinitfile.pcac ! Default PCAC Init file

  dtmcollection  = build_test   ! Default collection name
  dtmtests       = *    ! Use all tests
  dtmcomment     = "Test of build"  ! Default collection remark

  trace          = trace
  list           = list
  debug          = debug
  optimize       = nooptimize
2 analysis_data  = analysis_data
  cflags       = /$(debug) /nooptimize /$(analysis_data)
  cldflags = /$(list)
  msgflags = /$(list)

3 dtmflags       = /submit=(notify,noprint,keep)/class=(template:"$(mms$cms_gen)", -
     benchmark:"$(mms$cms_gen)")/var=(use_pca=$(pca),use_pca_init_file=$(pca_init_file), -
     translit="’’f$parse("translit.exe")’" )
  linkflags      =/map=$(mms$target_name)/exe=$(mms$target_name)/$(trace)
  !Modules in build
  !copyfile.obj
  translit_modules= buildtable.obj, copyfile.obj, -
      expandstring.obj, translit.obj
```

Key to Example 9–2

1 This section defines macros, flags, and default libraries and names. These defaults will make the description file more generic; that is, individual instructions that use these defaults can be modified by changing the

definitions rather than all the occurrences of the name, macro, and so on. The TRN_DTM represents the default library; logicals are defined in the [TRN.COMS] directory for access by individual LOGIN.COM files. The SCA library has not been defined. As a result, MMS loads the .ANA files into whatever SCA library is set at the time of the build.

To make the defaults easier to change, or to use them in more than one place, they can be kept in a separate file. This file will then need to be included as part of the description file.

Members of the team might want to change the default names and macros occasionally. They can change any of the default values in this description file when they invoke the file. For example, the following command provides a unique HP DIGITAL Test Manager collection name (necessary if they have kept previous collections):

```
$ MMS/MACRO=(DTMCOLLECTION=NEW_NAME)
```

2 The /ANALYSIS_DATA part of the PFLAGS macro causes the C compiler to generate SCA data files (.ANA).

3 The DTMFLAGS macro is used to submit a collection of tests (the tests must already exist), and invokes the appropriate templates and benchmarks from the CMS library. By default, this collection contains all the tests. The default class for the template and benchmark directories is the same as the class from which the source code is drawn. By having previously set up a class called V1.0 in both the HP DIGITAL Test Manager and CMS libraries, the team can build and test V1.0 with the following command:

```
$ MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") VERSION_BUILD
```

This command does not update the SCA library because it designates the VERSION_BUILD option. This is described in the explanatory text of Example 9–4.

The DTMFLAGS macro also uses three HP DIGITAL Test Manager variables:

- /*var=use_pca*—Sets up a flag to specify coverage analysis for the test collection
- /*var=use_pca_init_file*—Sets a default PCA Collector initialization file; at the same time, it lets the team choose a different initialization file when invoking MMS (the command examples follow)
- *translit*—Ensures that the HP DIGITAL Test Manager test collection is created based on the default work directory

For the test coverage analysis to occur, the team set up several conditions:

- They defined *use_pca*, *use_pca_init_file*, and *translit* variables in the HP DIGITAL Test Manager library. These variables were defined with the following HP DIGITAL Test Manager commands:

```
$ DTM CREATE VARIABLE/GLOBAL use_pca "false"
$ DTM CREATE VARIABLE/GLOBAL/LOGICAL use_pca_init_file -
_$ "TRN_DISK:[TRN.COMS]INITFILE.PCAC"
$ DTM CREATE VARIABLE/GLOBAL/LOGICAL translit -
_$ "TRN_DISK:[TRN.BLD_V1.WORK]TRANSLIT.EXE"
```

- A PCA Collector initialization file that determines what test coverage data is collected.
- A HP DIGITAL Test Manager prologue file that sets up the PCA coverage and points to the PCA Collector initialization file.
- An image of the product to be tested and linked to invoke the PCA Collector. Linking is done with the /DEBUG switch. (The .FIRST part of Example 9–4 shows how this is done.)

An example follows of the PCA Collector initialization file. The HP DIGITAL Test Manager collection prologue file is shown in Section 10.1.

PCA Collector Initialization File The initialization file that follows contains commands passed to the PCA Collector (the file is stored in the [TRN.COMS] directory). This file must contain the GO command as the last command; optionally, it can contain other Collector commands. The following example collects system service counts and measures test coverage by code path over the entire program. In addition, it selects output verification and the collection of program counter (PC) values from the Call Stack; the file PCA_DTM.LOG stores the output of the logging session. With this information, the Collector can run in batch mode.

```
! Test coverage Collector Initialization File
!
SET LOG PCA_DTM.LOG                      !Turn on output logging
SET SERVICES                             !Gather system service counts
SET COVERAGE PROGRAM_ADDRESS BY CODEPATH  !Gather coverage by codepath
SET STACK_PCS                            !Gather data from Call Stack
GO                                       !Begin collection
```

With the different files and commands in place, the team has two options using two different MMS commands:

- Build an executable application (current or previous version) that has a test collection, but no PCA coverage on the tests. Note that this command builds the current, default version of the application and loads the SCA library because WORK_BUILD is specified. The command is as follows:

  ```
  $ MMS/CMS WORK_BUILD
  ```

- Build an executable application with a corresponding test collection and PCA test coverage for that test collection. The command is as follows:

  ```
  $ MMS/CMS/MACRO=PCA=TRUE
  ```

  By adding to this command the additional switch, /MACRO=PCA_INIT_FILE=*DISK:[DIRECTORY]FILENAME*.PCAC, other PCA initialization files can be designated, which provides greater flexibility.

9.3 MMS Rules

This section shows and describes the Rules section of the MMS description file shown in this chapter. Example 9–3 is followed by a detailed explanation of each part of the example.

Example 9–3 Rules Section of the Description File

```
!=============
! RULES
!=============

! Modify the C rule to decide if analysis data files should be produced.

1 .c.obj  ! a rule for compiling C files
   if use_sca .eq. 0 then $(cc) $(cflags)/noanalysis_data $(mms$source)
   if use_sca .eq. 1 then $(cc) $(cflags) $(mms$source)
   if use_sca .eq. 1 then $(sca) load $(mms$target_name)
   if use_sca .eq. 1 then $(delete) $(mms$target_name).ana;

!=============
```

Key to Example 9–3

1 This section modifies the C rule; this rule already exists because C is an MMS-supported language. This gives the team the option of using SCA based on a specific compilation. For example, for older versions, they might not want to generate analysis data to be loaded into the SCA library.

Alternatively, if the build is assembling a new version, they will want the SCA library updated.

Note

MMS provides built-in rules for using SCA and has a /SCA_LIBRARY switch to indicate that automatic SCA handling is invoked. The examples shown here, however, are included to show how MMS rules can be manipulated by hand.

9.4 MMS Targets

This section shows and describes the Targets section of the MMS description file shown in this chapter. Example 9–4 is followed by a detailed explanation of each part of the example.

Example 9–4 Targets Section of the Description File

```
!===============
! TARGETS
!===============

.first   ! Set a flag to determine how objects are to be compiled1
use_sca = 0
if "$(mmstargets)" .eqs. "work_build" .or. "$(mmstargets)" .eqs. "" -
    then use_sca = 1
pca_link = ""
if "$(pca)" .nes. "false" then pca_link = "/debug"

! The normal development build (default if no other target specified)

work_build      : translit.exe, -       ! Build the translit executable image2
                       test_set         ! Run the regression tests
                                        ! Work build completed at this point.

! To build a particular version -- use same as work build, but different target
! will allow .FIRST directive to cause C not to generate .ANA files.

version_build : work_build
                ! Version build completed at this point


!The translit executable
```

(continued on next page)

Example 9–4 (Cont.) Targets Section of the Description File

```
translit.exe : $(translit_modules)      ! put modules into an olb3
        $(link) $(linkflags) $(translit_modules) 'pca_link'
        if "$(debug)" .eqs. "debug" then $(link) $(linkflags)/debug/exe= -
        $(mms$target_name).debug $(translit_modules)

! The test set

test_set        :               ! Always run a test set
        $(dtm) set library $(dtmlibrary)
        $(dtm) create collection $(dtmcollection) $(dtmtests) -
               $(dtmflags) $(dtmcomment)
        ! test set created at this point

! Source code dependencies to create the needed .OBJ file which
! is source for the target TRANSLIT.EXE.

translit.obj     : translit.c, openfiles.h, types.h4
buildtable.obj   : types.h
copyfile.obj     : types.h
expandstring.obj : types.h
openfile.obj     : types.h
```

Key to Example 9–4

1 This section, in combination with the modified C rule from Example 9–3, sets up the targets to update the SCA library. Two options exist:

- The SCA library is updated. This is done in either of two ways:

  ```
  $ MMS/CMS
  ```

  ```
  $ MMS/CMS WORK_BUILD
  ```

  In both of these cases, the USE_SCA flag is set to 1. Based on this value, the C rule causes the SCA library to be updated. This is likely to be the most common invocation of the description file, and so is the default command.

- The SCA library is not updated; designate VERSION_BUILD when invoking MMS. This might be done when building an older generation, as explained in Example 9–2.

2 These target instructions set up the MMS dependencies. Different types of executable images can be built depending on how the team invokes MMS:

- By default, two executable images that represent the current, up-to-date application of the software system; one is a debugger version (compiled and linked with the /DEBUG switch) and the other is a non-debugger version that the HP DIGITAL Test Manager uses for its test collection.
- By choice, an image that represents previous versions of the software application as designated in a CMS class specified during the MMS command.

See the explanatory text of Example 9–2 for specific MMS invocations.

3 In building the project executable image, the commands create an object library based on the previously defined macro (TRANSLIT_MODULES).

4 It is good practice to include the final source dependencies as part of the description file, although MMS will complete the build without this information. For additional information about MMS and HP DIGITAL Test Manager, see the *Guide to HP Module Management System for OpenVMS Systems* and the *Guide to HP DIGITAL Test Manager for OpenVMS Systems*.

9.5 MMS Description File Command Options

The following table summarizes the different commands and their effects for this MMS description file.

Command	Effect
MMS/CMS	Builds the current TRANSLIT.EXE with Test Collection, no PCA coverage, and updates the SCA library.
MMS/CMS WORK_BUILD	Builds the current TRANSLIT.EXE with Test Collection, no PCA coverage, and updates the SCA library.
MMS/CMS/MACRO=PCA=TRUE	Builds the current TRANSLIT.EXE with Test Collection, PCA coverage on tests, and updates the SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") VERSION_BUILD	Builds the previous class with corresponding Test Collection, no PCA coverage, and no update to the SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") /MACRO=PCA=TRUE VERSION_BUILD	Builds the previous class with corresponding Test Collection, PCA coverage, and no update to the SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0")	Builds the previous class with corresponding Test Collection, with no PCA coverage, and updates the SCA library.
MMS/CMS/MACRO=(MMS$CMS_GEN="V1.0") WORK_BUILD	Builds the previous class with corresponding Test Collection, with no PCA coverage, and updates the SCA library.

10 Setting Up a Test System with HP DIGITAL Test Manager

This chapter describes how the Transliteration team sets up its test system. The chapter describes how to set up noninteractive, interactive, and DECwindows tests.

Noninteractive tests, described in Section 10.1, are appropriate for testing software that does not have a terminal-oriented or menu interface. The TRANSLIT software described up to now fits this category, because the application accepts an input text file and gives you an output file with substituted characters.

To demonstrate the interactive testing capabilities of HP DIGITAL Test Manager, a variant of TRANSLIT is demonstrated that uses forms created by the VAX Forms Management System (FMS) for its menu-driven interface. The test system for this variant application is described in Section 10.2.

A section on DECwindows application testing appears at the end of this chapter.

10.1 Setting Up a Noninteractive Test

The first step in setting up the test system is to create the storage areas listed in the following table. They were described previously, but are repeated here.

Directories	Function
TRN_DISK:[TRN.DTMLIB]	HP DIGITAL Test Manager library for test results
TRN_DISK:[TRN.DTM_CMSLIB]	CMS library for HP DIGITAL Test Manager test files
TRN_DISK:[TRN.DTM_DATA] TRN_DISK:[TRN.DTM_DATA_CMS]	Two subdirectories for test data

The following list gives the additional steps needed to set up a noninteractive test:

1. Set up the collection prologue and epilogue files.
2. Set up the test prologue and epilogue files.
3. Establish the default test template and benchmark file directory.
4. Create the test template file.
5. Create the test description.
6. Set up HP DIGITAL Test Manager variables.
7. Insert the test template file into the CMS library.

The following sections describe each of these steps in detail.

10.1.1 Creating the Collection Prologue File

The collection prologue file is associated with one or more collections and runs just before the test prologue file for the first test in the collection. The collection prologue file is used as a setup file to establish any special environment the collection requires. The collection prologue file that the Transliteration project team creates does two things:

1. Examines the HP DIGITAL Test Manager variable *use_pca* (defined and described later in this section) to determine whether to continue processing the collection prologue file.
2. Defines the Collector initialization file according to the HP DIGITAL Test Manager variable *use_pca_init_file* (also defined and described later in this section).

Running the PCA Collector during a collection is especially useful for determining which code paths are being exercised by the tests. Chapter 11 shows an example of analyzing coverage data after tests have been executed.

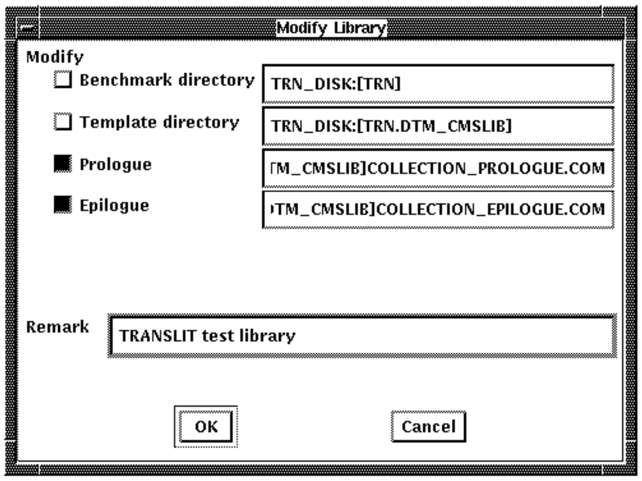
The collection prologue file is shown in Example 10–1.

Example 10–1 Sample Collection Prologue File

```
$! Collection prologue file for running the Collector in batch mode
$!
$  set verify
$!
$  translit:==$'f$logical("translit")'
$!
$! HP DIGITAL Test Manager variable to determine whether or not
$! to run PCA prologue
$!
$  if use_pca .eqs. "false" then exit
$!
$! Define the Collector initialization file
$!
$  define pcac$init use_pca_init_file
$!
$! End of collection prologue file
```

Next, you need to establish this collection prologue file as the default prologue file for subsequently created test collections. As shown in Figure 10–1, click on Prologue and enter the file specifications.

Figure 10–1 Modify Library Dialog Box



☐ Benchmark directory	TRN_DISK:[TRN]
☐ Template directory	TRN_DISK:[TRN.DTM_CMSLIB]
■ Prologue	ΓM_CMSLIB]COLLECTION_PROLOGUE.COM
■ Epilogue	›TM_CMSLIB]COLLECTION_EPILOGUE.COM
Remark	TRANSLIT test library

The Prologue field requires full file specifications for the prologue file. In this example, the prologue file exists in the HP DIGITAL Test Manager CMS library, DTM_CMSLIB.

10.1.2 Creating the Collection Epilogue File

Like the collection prologue file, the collection epilogue file is associated with one or more collections and runs just after the test epilogue for the last test in the collection run. A collection epilogue file is used to perform cleanup procedures. For example, the epilogue file can clean up the directory in which certain test files are created or modified for testing purposes. You can also use an epilogue file to send MAIL notification to anyone on the project team when the tests are completed—giving test results, for example.

The collection epilogue file, which the Transliteration project team creates, sends the results of the collection TRANSLIT_COLLECTION to the project leader Jones.

This collection epilogue file, shown in Example 10–2, also makes use of the HP DIGITAL Test Manager-provided symbol DTM$COLLECTION_NAME.

Example 10–2 Sample Collection Epilogue File

```
$! Collection epilogue file for mailing test results to JONES,
$! upon completion of the test run.
$!
$ dtm show collection 'dtm$collection_name'/FULL-
$ /output='dtm$collection_name'.report
$!
$ mail 'dtm$collection_name'.report/subject="Collection summary" JONES
$!
$! End of collection epilogue file
```

Next, you need to establish this epilogue file as the default epilogue file for subsequently created test collections. Figure 10–1 shows the HP DIGITAL Test Manager Modify Library dialog box in which you associate the collection epilogue file with the HP DIGITAL Test Manager library, DTM_CMSLIB, where the test collection for TRANSLIT exists.

The Epilogue field also requires full file specifications for the epilogue file.

10.1.3 Establishing the Default Template and Benchmark File Directory

When you set the default directories for the template and benchmark files, HP DIGITAL Test Manager automatically looks for those files in those directories during the test. (HP DIGITAL Test Manager looks for files with the same name as the test, with a .COM extension for template files, and .BMK for benchmark files.) You can override the default template and benchmark directories on each test description by including a directory specification as part of the template or benchmark names. The HP DIGITAL Test Manager assigns the default values *test-name*.COM and *test-name*.BMK to those fields. It is good practice, though, to use the following guidelines:

- Let the HP DIGITAL Test Manager assign default names to template and benchmark files to make tracking the components of each test easier.
- Avoid overriding the template and benchmark directories when creating test descriptions. This makes tests more portable. In addition, if you later want to change the directory specifications for the template or benchmark files, you do not need to modify each test description; you can modify the information in the HP DIGITAL Test Manager Modify Library dialog box instead, shown in Figure 10–1.

10.1.4 Setting Up the Test Prologue and Epilogue Files

You can create test prologue and epilogue files to set up and clean up special conditions and tasks for a specific test. You can also use a test epilogue file to perform filtering procedures, for example, removing run-specific data such as time stamps and the amount of memory used.

Test Prologue File

In the example used in this section, the prologue file for the test TRANSLIT_ TEST establishes a default directory specification for the test.

Example 10–3 shows the contents of TRANSLIT_PROLOGUE.COM.

Example 10–3 Test Prologue File for translit_test

```
$!  Test prologue file for translit_test
$!  This prologue file sets the default directory specification
$!  for translit_test to trn_disk:[jones.translit_test].
$!
$ set default trn_disk:[jones.translit_test]
$!
$!  End of test prologue file.
```

Test Epilogue File

In the example used in this section, the epilogue file for the test TRANSLIT_TEST deletes the output files created by TRANSLIT_TEST.

Example 10–4 shows the contents of the file TRANSLIT_EPILOGUE.COM:

Example 10–4 Test Epilogue File for translit_test

```
$!  Test epilogue file for translit_test.
$!  This epilogue file deletes the output files generated by translit_test
$!
$ delete out1.txt;
$ delete out2.txt;
$ delete out3.txt;
$ delete out4.txt;
$ delete out5.txt;
$ delete out6.txt;
$!
$!  End of test epilogue file.
```

10.1.5 Creating the Template File for the Test

The Transliteration project team creates the template file, TRANSLIT_TEST.COM, shown in Example 10–5.

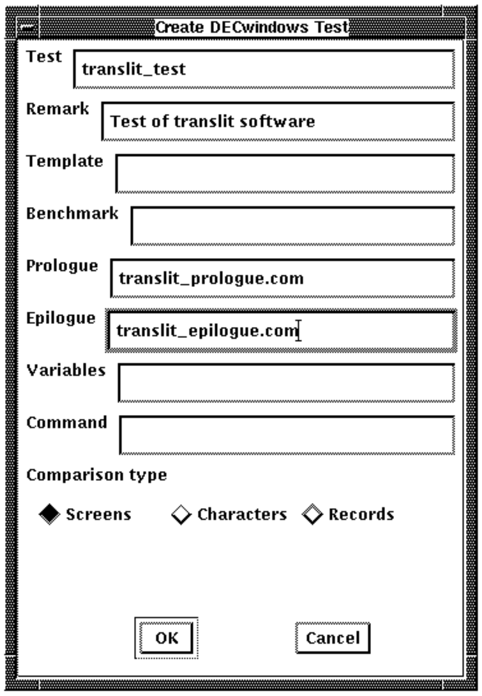
Example 10–5 translit Test Template File

```
$ create test.txt
abc345ghijk
ABCDEFGH789
aBc   gHiJk
abc   ghijk
ABCDEFGHIJK
aBcDeFgHiJk
abcdefghijk
ABCDEFGHIJK
aBcDeFgHiJk
abcdefghijk
ABCDEFGHIJK
aBcDeFgHiJk
abcdefghijk
ABCDEFGHIJK
1Bc4eF7HiJk
$!
$!  Changes the case of all letters.
$ translit/output=out1.txt  test.txt  "A-Za-z"  "a-zA-Z"
$ type out1.txt
$!
$!  Changes all sequences of nonletters to single new-lines. The output
$!  contains each word (sequence of letters) on a separate line.
$ translit/output=out2.txt  test.txt  "-A-Za-z"  "@n"
$ type out2.txt
$!
$!  Replaces all escape characters by dollar signs.
$ translit/output=out3.txt  test.txt  "^["  "$"
$ type out3.txt
$!
$!  Changes each sequence of spaces to a single space. (The @d makes the
$!  original string longer than the replacement string, forcing translit
$!  to do compression instead of simple replacement.)
$ translit/output=out4.txt  test.txt  " @d"  " "
$ type out4.txt
$!
$!  Deletes all backspaces.
$ translit/output=out5.txt  test.txt  "@B"
$ type out5.txt
$!
$!  Changes all letters to lowercase, and deletes all digits.
$ translit/out=out6.txt  test.txt  "A-Z0-9" "a-z@d"
$ type out6.txt
```

10.1.6 Creating the Test Description

By using the HP DIGITAL Test Manager Create DECwindows Test dialog box, you can create a test description for TRANSLIT_TEST, as shown in Figure 10–2.

Figure 10–2 Create DECwindows Test Dialog Box



Test	translit_test
Remark	Test of translit software
Template	
Benchmark	
Prologue	translit_prologue.com
Epilogue	translit_epilogue.com
Variables	
Command	

Comparison type

[✓ Screens] Characters Records

This dialog box also shows how to associate the test prologue and epilogue files TRANSLIT_PROLOGUE and TRANSLIT_EPILOGUE to TRANSLIT_TEST.

By default, the HP DIGITAL Test Manager associates the template file TRANSLIT_TEST.COM and the benchmark file TRANSLIT_TEST.BMK with the test description.

10.1.7 Setting Up the HP DIGITAL Test Manager Variables

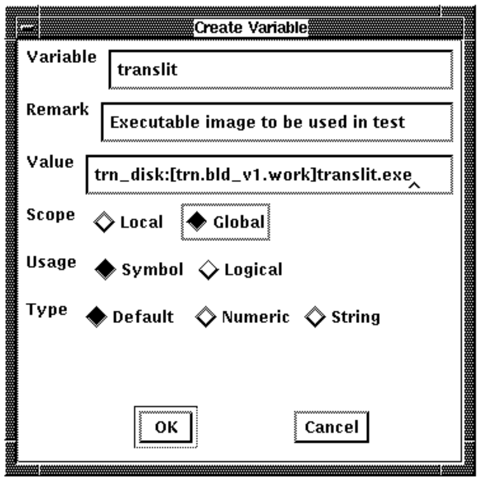
A HP DIGITAL Test Manager variable is a user-defined symbol or logical name that HP DIGITAL Test Manager stores and uses during the tests. Variables can be referred to in template, prologue, and epilogue files, and can provide a convenient way to tailor those files to be used with multiple tests.

The Transliteration team creates the following HP DIGITAL Test Manager variables:

Variables	Function
translit	Represents the executable image to be used by the test set
use_pca	Gives PCA coverage, if invoked as *true*
use_pca_init_file	Provides the file specifications for the default PCA Collector initialization file

You use the HP DIGITAL Test Manager Create Variable dialog box to create these
HP DIGITAL Test Manager variables, as shown in Figure 10–3.

Figure 10–3 Create Variable Dialog Box



Variable: translit
Remark: Executable image to be used in test
Value: trn_disk:[trn.bld_v1.work]translit.exe
Scope: Local [✓ Global]
Usage: [✓ Symbol] Logical
Type: [✓ Default] Numeric String

This dialog box enables you to specify the variable name, its value, and an associated remark.

10.1.8 Inserting the Template File into the CMS Library

The final step in setting up the test system is to insert the template file into the CMS library, as follows:

1. Invoke the CMS interface, setting the default CMS library to TRN_DISK:[TRN.DTM_CMSLIB].
2. From the File menu, click on New Element... and type TRANSLIT_TEST into the Element field.

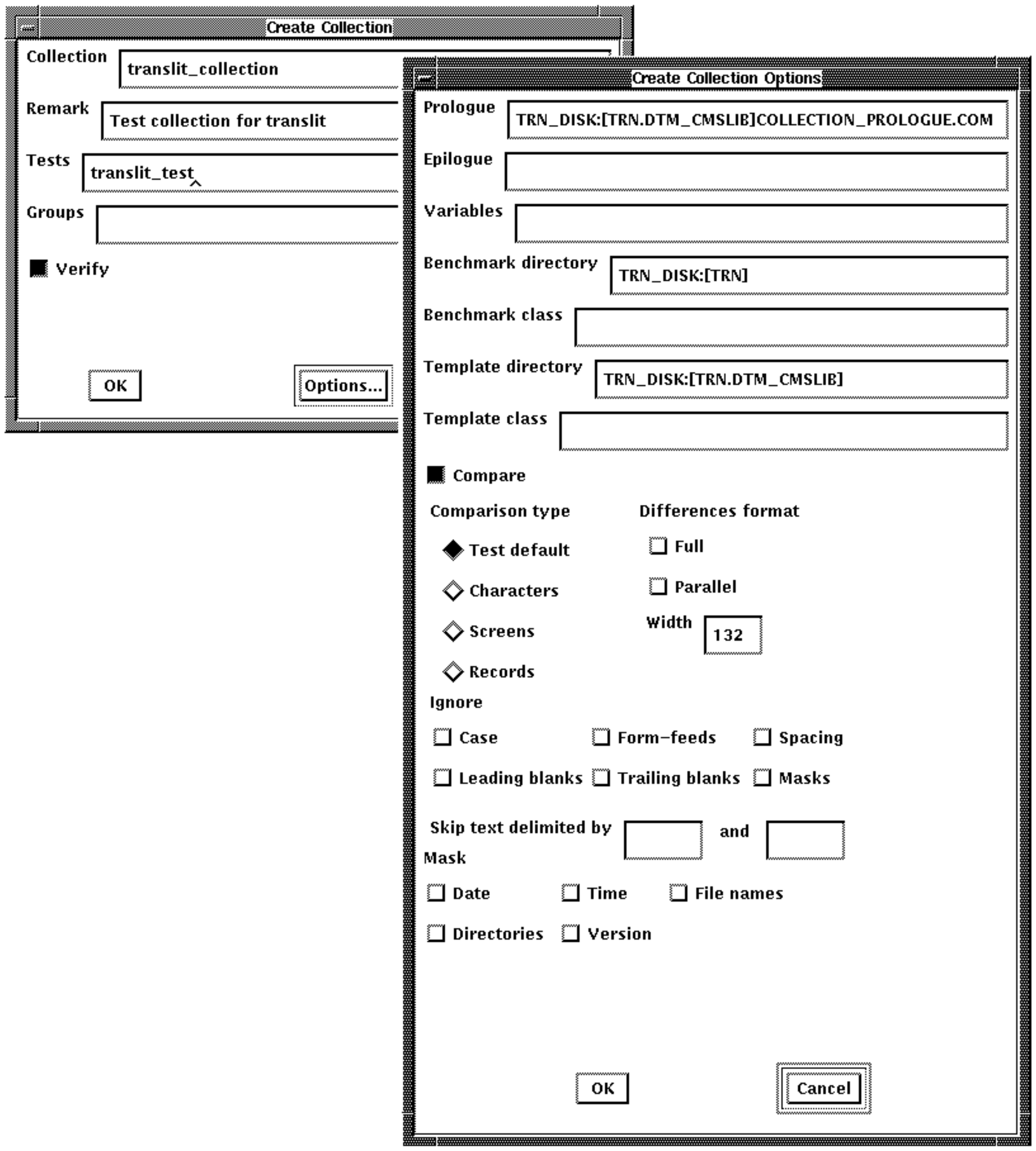
10.1.9 Creating the Collection

The following example creates the collection TRANSLIT_COLLECTION, and also does the following:

- Inserts the test TRANSLIT_TEST
- Assigns the collection prologue file COLLECTION_PROLOGUE, described previously
- Assigns the collection epilogue file COLLECTION_EPILOGUE, also described previously

You can perform this step in the HP DIGITAL Test Manager Create Collection dialog box, as shown in Figure 10–4.

Figure 10–4 Create Collection Dialog Box



Create Collection

Collection: translit_collection

Remark: Test collection for translit

Tests: translit_test

Groups:

■ Verify

Create Collection Options

Prologue: TRN_DISK:[TRN.DTM_CMSLIB]COLLECTION_PROLOGUE.COM

Epilogue:

Variables:

Benchmark directory: TRN_DISK:[TRN]

Benchmark class:

Template directory: TRN_DISK:[TRN.DTM_CMSLIB]

Template class:

■ Compare

Comparison type: ◆ Test default ◇ Characters ◇ Screens ◇ Records

Differences format: ☐ Full ☐ Parallel

Width: 132

Ignore: ☐ Case ☐ Form-feeds ☐ Spacing ☐ Leading blanks ☐ Trailing blanks ☐ Masks

Skip text delimited by and

Mask: ☐ Date ☐ Time ☐ File names ☐ Directories ☐ Version

10.2 Setting Up an Interactive Terminal Test

To create an interactive, or terminal, test of the *translit* variant that uses a menu-driven interface, the Transliteration project team uses the same library file, prologue files, and epilogue files described previously. To record the test using the HP DIGITAL Test Manager DECwindows interface, do the following:

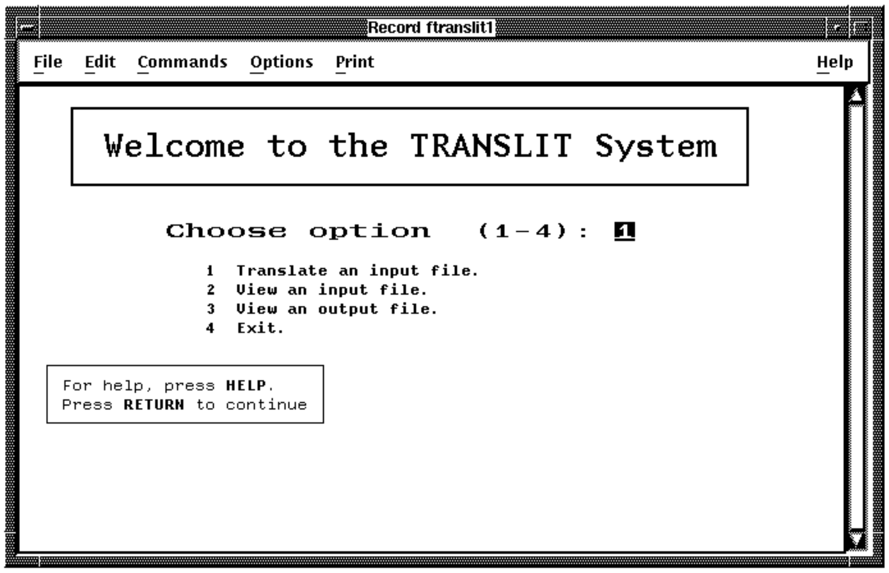
1. Pull down the Maintenance menu.
2. Choose the Create menu item. The system displays a submenu.
3. Choose the Test Terminal... menu item. The system displays the Create Test dialog box.
4. Enter FTRANSLIT1 in the Test text entry field.
5. Click on OK.
6. Pull down the Testing menu.
7. Choose the Record Terminal... menu item. The system displays the Terminal Record dialog box.
8. Enter FTRANSLIT1 in the Test text entry field.
9. Click on OK.

The HP DIGITAL Test Manager displays a message that it is creating a task window. In this task window, the Transliteration engineer, John, begins the recording session.

10.2.1 Invoking the Forms translit Application

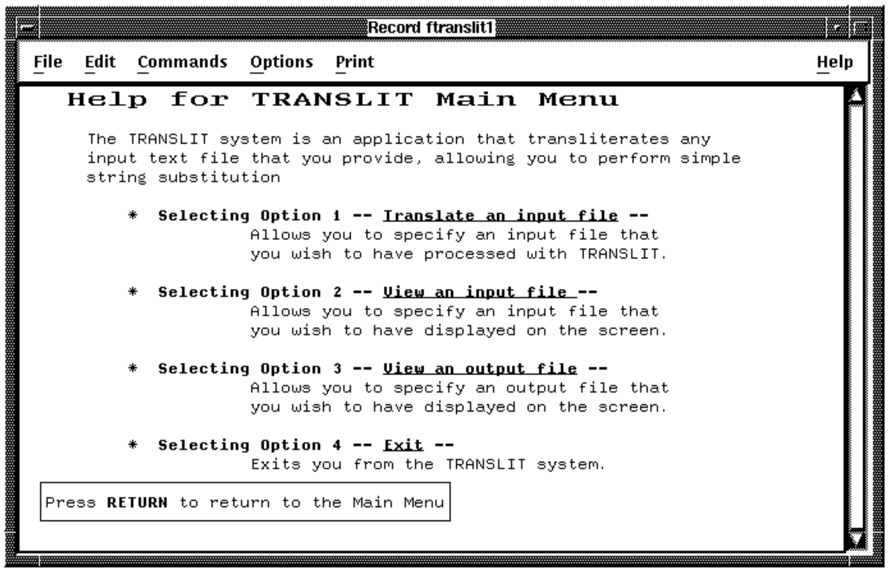
In the task window that HP DIGITAL Test Manager creates, John enters a command to invoke the Forms TRANSLIT application. As the applications runs, it displays the forms shown in this section.

The following screens show the Forms TRANSLIT application in its test session.



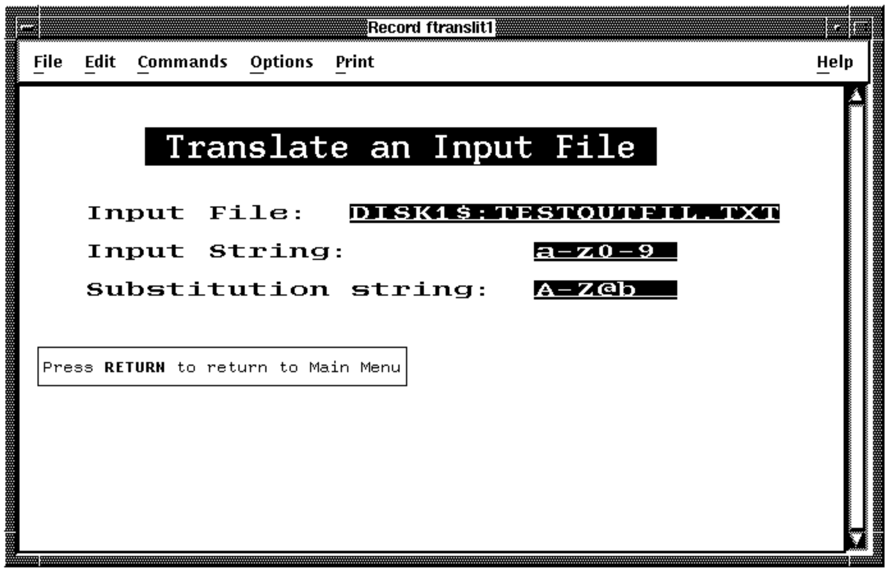
```
Welcome to the TRANSLIT System

Choose option   (1-4):  1

  1  Translate an input file.
  2  View an input file.
  3  View an output file.
  4  Exit.

For help, press HELP.
Press RETURN to continue
```

John presses the Help key (PF2) twice to get to the Help form for the Main Menu, shown in the next screen.



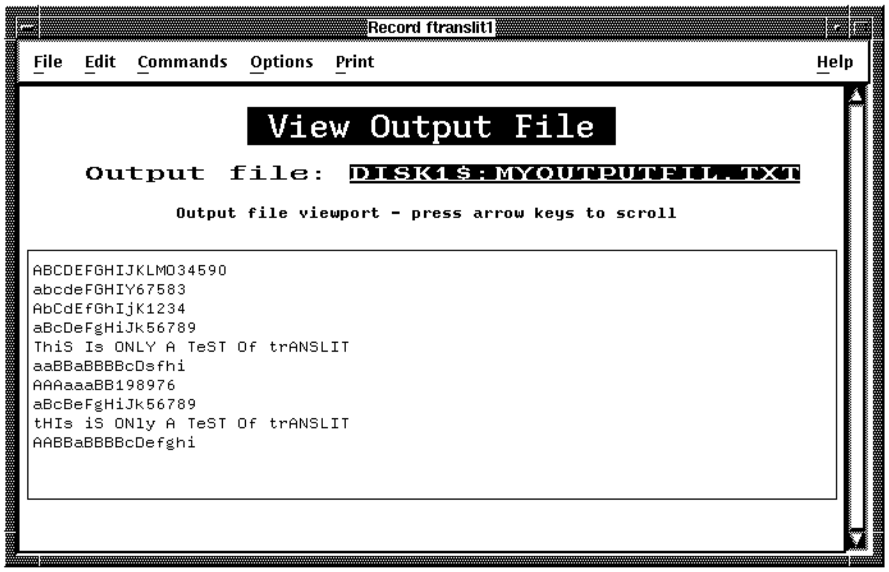
```
Help for TRANSLIT Main Menu

  The TRANSLIT system is an application that transliterates any
  input text file that you provide, allowing you to perform simple
  string substitution

    * Selecting Option 1 -- Translate an input file --
              Allows you to specify an input file that
              you wish to have processed with TRANSLIT.

    * Selecting Option 2 -- View an input file --
              Allows you to specify an input file that
              you wish to have displayed on the screen.

    * Selecting Option 3 -- View an output file --
              Allows you to specify an output file that
              you wish to have displayed on the screen.

    * Selecting Option 4 -- Exit --
              Exits you from the TRANSLIT system.
Press RETURN to return to the Main Menu
```

John selects Option 1, Translate an input file, shown in the next screen.



```
Translate an Input File

Input File:    DISK1$:TESTOUTFIL.TXT
Input String:              a-z0-9
Substitution string:       A-Z@b

Press RETURN to return to Main Menu
```

John returns to the Main Menu and selects Option 3, View an output file, shown in the next screen.



```
View Output File

Output file: DISK1$:MYOUTPUTFIL.TXT

Output file viewport - press arrow keys to scroll

ABCDEFGHIJKLMO34590
abcdeFGHIY67583
AbCdEfGhIjK1234
aBcDeFgHiJk56789
ThiS Is ONLY A TeST Of trANSLIT
aaBBaBBBBcDsfhi
AAAaaaBB198976
aBcBeFgHiJk56789
tHIs iS ONly A TeST Of trANSLIT
AABBaBBBBcDefghi
```

John returns to the Main Menu and selects Option 4, Exit.

10.2.2 Terminating the Recording Session

John terminates the recording session by pressing Ctrl/P twice. This saves the output as a benchmark for future comparisons.

```
$
 ^P

%DTM-I-BMK_SAVED, benchmark has been saved in file
TRN_DISK:[TRN.DTMLIB]FTRANSLIT1.BMK;1
%DTM-S-RECORDED, test FTRANSLIT1 has been successfully recorded
       in file TRN_DISK:[TRN]FTRANSLIT1.SESSION
%DTM-S-CREATED, test description FTRANSLIT1 created
```

At this point, FTRANSLIT1 is like any other test description, whether created interactively or not. You can place it in a collection and execute it interactively or in batch mode. The sample build in Chapter 9 executes test collections in batch mode. Batch execution is preferred if you do not want to tie up a terminal or terminal emulator window on a workstation.

10.2.3 Running the Session File

You can run the SESSION file for an interactive test with the PLAY command. A collection is not created and the results of the run are not saved. The PLAY command enables you to view your SESSION file while it runs in order to verify that it runs correctly.

If the terminal characteristics or terminal type for the display terminal differ from those for the recording terminal, the SESSION file might not appear as you expect.

The following command runs the SESSION file FTRANSLIT_TEST.SESSION created in the previous step:

```
$ DTM PLAY FTRANSLIT
```

You can use the /RESULT_FILE switch with the PLAY command if you want to have the output of the PLAY command saved in a file.

10.2.4 Reviewing Test Results

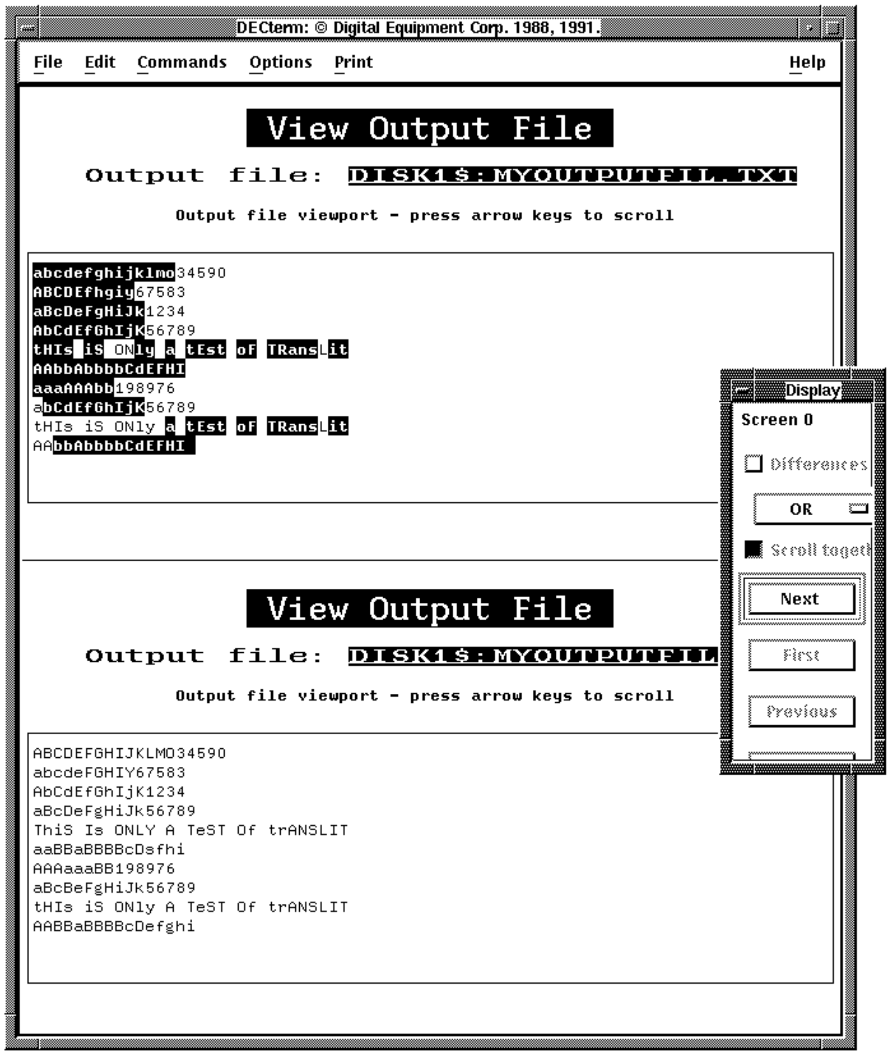
The HP DIGITAL Test Manager Display function enables you to review the results generated by executing tests. HP DIGITAL Test Manager creates result and difference files when the TRANSLIT test collection is executed. The Display function enables you to examine these files as well as benchmark files for your tests.

To use the Display function to display the differences file created by an unsuccessful test of a terminal recording session, do the following:

1. Expand the collection by double clicking on it. Expand the test by double clicking on it. When you expand the test, the benchmark file, results file, and differences file are listed.
2. In the HP DIGITAL Test Manager window, double click on the differences file created by the collection execution. (This automatically opens the collection and test for review.)

After you double click on the differences file, the HP DIGITAL Test Manager displays the differences found between the results of the test and the benchmark file, along with a control panel for toggling between the screens in the differences file. Figure 10–5 shows reviewing differences created by a terminal test in the DECwindows environment.

Figure 10–5 Displaying Differences



View Output File

Output file: DISK1$:MYOUTPUTFIL.TXT

Output file viewport - press arrow keys to scroll

```
abcdefghijklmo34590
ABCDEfhgiy67583
aBcDeFgHiJk1234
AbCdEfGhIjK56789
tHIs iS ONly a tEst oF TRansLit
AAbbAbbbbCdEFHI
aaaAAAbb198976
abCdEfGhIjK56789
tHIs iS ONly a tEst oF TRansLit
AAbbAbbbbCdEFHI
```

View Output File

Output file: DISK1$:MYOUTPUTFIL

Output file viewport - press arrow keys to scroll

```
ABCDEFGHIJKLMO34590
abcdeFGHIY67583
AbCdEfGhIjK1234
aBcDeFgHiJk56789
ThiS Is ONLY A TeST Of trANSLIT
aaBBaBBBBcDsfhi
AAAaaaBB198976
aBcBeFgHiJk56789
tHIs iS ONly A TeST Of trANSLIT
AABBaBBBBcDefghi
```

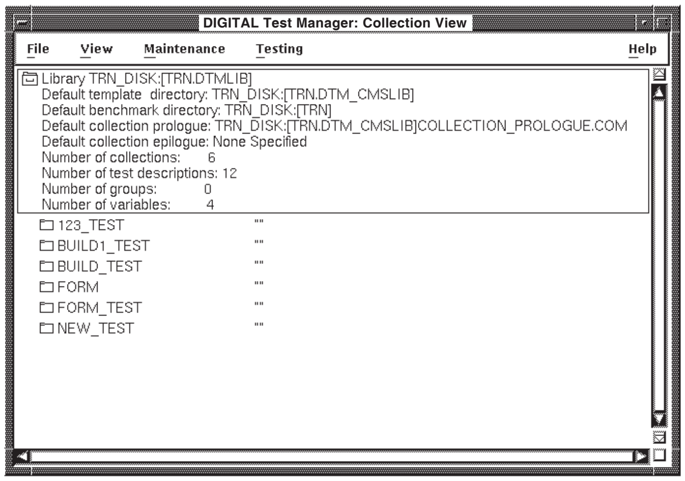
10.3 Verifying Your Test System

To check your HP DIGITAL Test Manager library to ensure that you have set your template, benchmark, prologue, and epilogue files correctly, you can double click on the HP DIGITAL Test Manager library specifications in the main window.

This displays the current directory specifications for CMS libraries containing these files, as well as the number of collections, test descriptions, groups, and variables in the library.

Figure 10–6 shows the library information HP DIGITAL Test Manager provides when you expand a library specification by double clicking on it.

Figure 10–6 Expanding a Library Specification



```
Library TRN_DISK:[TRN.DTMLIB]
   Default template  directory: TRN_DISK:[TRN.DTM_CMSLIB]
   Default benchmark directory: TRN_DISK:[TRN]
   Default collection prologue: TRN_DISK:[TRN.DTM_CMSLIB]COLLECTION_PROLOGUE.COM
   Default collection epilogue: None Specified
   Number of collections:      6
   Number of test descriptions: 12
   Number of groups:           0
   Number of variables:        4
     123_TEST                ""
     BUILD1_TEST             ""
     BUILD_TEST              ""
     FORM                    ""
     FORM_TEST               ""
     NEW_TEST                ""
```

10.4 Changing Input for a Test

HP DIGITAL Test Manager provides the Extract function to allow you to change the input that your nonDECwindows tests are using, simplifying the process of modifying your tests if the test results indicate you need to test for additional or different input. The Extract function extracts an input file from a session file without altering the session file. You can then edit this INPUT file, or create a new one, and reinsert the INPUT file with either the MODIFY or RECORD TEST_DESCRIPTION functions.

10.5 Recording a DECwindows Test

A DECwindows test is an interactive test that uses the DECwindows interface. Tests performed in the DECwindows environment are recorded using the HP DIGITAL Test Manager Record function in the Testing menu. DECwindows tests accept input from the keyboard, pointer device, or both. HP DIGITAL Test Manager uses the recorded session file to supply the input data when you subsequently test an application, the same as with interactive terminal tests.

10.5.1 Processing Considerations for DECwindows Tests

The following sections describe processing considerations and restrictions on the types of DECwindows applications you can test when using HP DIGITAL Test Manager.

10.5.1.1 Playing DECwindows Tests

Tests played in a DECwindows environment play at a different rate than those played in an interactive terminal environment.

During interactive terminal test playback, session files are generally played at a rate faster than the rate at which they were recorded. This is because HP DIGITAL Test Manager detects when an application is ready for more input, and sends session file input to the application as fast as it will accept it.

However, during DECwindows testing, HP DIGITAL Test Manager does not detect when an application is ready for more input. Thus, DECwindows sessions are played in real time. When a session file is played, keyboard and mouse movements are played back at the same speed at which they were recorded. For example, if you press a key at 10-second intervals while recording a test, the keypress events are repeated at 10-second intervals when you play the session file.

DECwindows sessions are played in real time whether the session is played as part of a test execution (using the RUN or SUBMIT commands) or played separately (using the PLAY command).

Note

Because a DECwindows test is played in real time, it might be affected by other processes that might degrade system performance.

You must ensure that no unanticipated loads are placed on the system while a DECwindows session file is playing, or you might not get the desired results. For example, to test the program Calendar, do the following:

1. Select the Calendar entry from the Session Manager Applications menu.
2. When the Calendar is displayed on the screen (after approximately 10 seconds) use its features as desired as part of the test recording.
3. Complete the test recording, initialize your environment for the test, and start playing the session file.

Suppose you have another active process that is compiling at the same time as the test play. When you play the session file, the Calendar might take 15 seconds to be displayed instead of the 10 seconds it took to be displayed when the test was originally recorded. Because the DECwindows session file is playing in real time, the keyboard and mouse input that occur in the Calendar program during the test recording session is sent to the workstation before the Calendar is displayed on the screen and is able to accept input. The input is then lost and the session file fails to play as expected.

You can work around real-time restrictions by applying these solutions:

- Use a dedicated system for DECwindows testing. Do not run other programs that use CPU cycles during the DECwindows test.
- Use the autosynchronization option on the test playback to automatically enter synchronization information into the session file. Or, when recording the test, change the timing factors in the keyboard and mouse movements so the test will play at a slower rate in critical sections. See the HP DIGITAL Test Manager release notes and documentation for more information on working with DECwindows test session files.

10.5.1.2 Storing DECwindows Benchmark and Result Files

Screen image data stored in DECwindows benchmark and result files requires significant disk storage space. HP DIGITAL Test Manager compresses the image data to save storage space. However, HP DIGITAL Test Manager cannot successfully compress screen images where a patterned background is used (for example, the DECwindows default background). You must customize your screen background to display a solid color before invoking HP DIGITAL Test Manager in the DECwindows environment.

10.5.1.3 Environment Initialization

You must ensure that the DECwindows environment is in the same state for a play as it was when the session file was recorded. Factors that might affect the environment include applications with windows displayed and window positions.

10.5.2 Recording a DECwindows Session

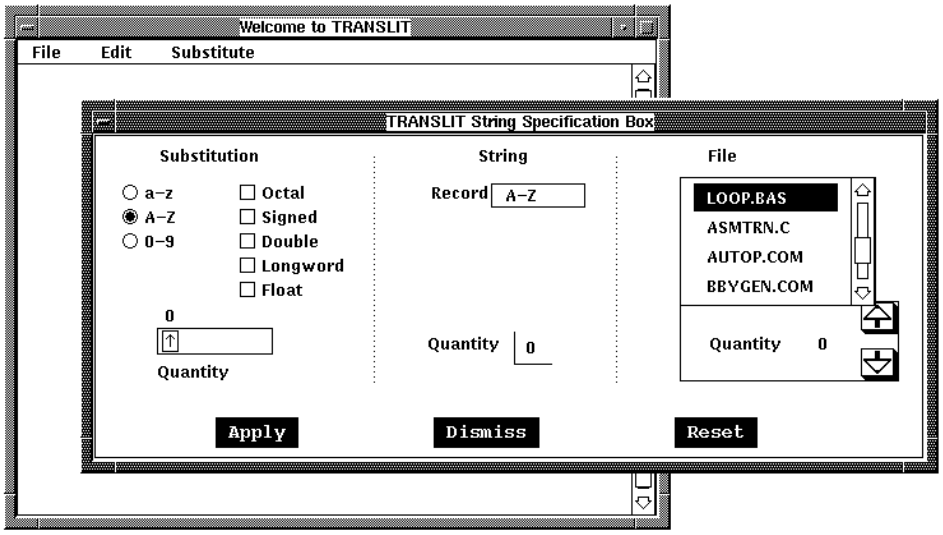
This section shows a sample DECwindows recording session that uses the DECwindows TRANSLIT application.

To create and record the test TRANSLIT_TEST in the HP DIGITAL Test Manager DECwindows environment, do the following:

1. Pull down the Maintenance menu.
2. Choose the Create Test DECwindows... menu item. The system displays the Create DECwindows Test dialog box.
3. Enter TRANSLIT_TEST in the Test text entry field.
4. Enter RUN TRN_DISK:[TRN.BLD_V1.WORK]TRANSLIT in the Command text entry field.
5. Click on OK.
6. Pull down the Testing menu.
7. Choose the Record DECwindows... menu item. The system displays the DECwindows Record dialog box.
8. Enter TRANSLIT_TEST in the Test text entry field.
9. Click on OK.
10. Iconize all the HP DIGITAL Test Manager windows.
11. A status window is displayed. To begin recording the test, choose the File menu and click on Start.
12. Select the screens you want to mark for comparison by pressing the F9 key and the M key simultaneously, as you would a control character (for example, Ctrl/Y).
13. When you are done marking the screens, press the F9 key and the C key simultaneously to end the recording session.

Figure 10–7 shows the DECwindows TRANSLIT application window and the String Specification Box dialog box.

Figure 10–7 Sample DECwindows Recording Session



○ a–z
● A–Z
○ 0–9

0
Quantity

Record A–Z
Quantity 0

Quantity 0

In Figure 10–7, the F9 key and the M key were pressed, marking the screen for comparison in subsequent test runs. Because the attributes on the String Specification Box dialog box are the only things to be tested in this test, only this screen needed to be marked.

The window manager title bars do not appear in Figure 10–7. This is because the window manager was not specified in the Record dialog box.

10.6 Displaying DECwindows Test Results

When you review a DECwindows test, HP DIGITAL Test Manager provides you with screen images created from the benchmark files, result files, and their differences (if any). If these files have been deleted, an error message is displayed and you are not able to review the images.

To review test results in a collection, do the following:

1. Select the collection you want to review from the view window and open it.
2. Expand the collection view to the benchmark, result, or differences file that you want to review.
3. Double click on the differences file that you want to review. This opens the collection and starts the review display.

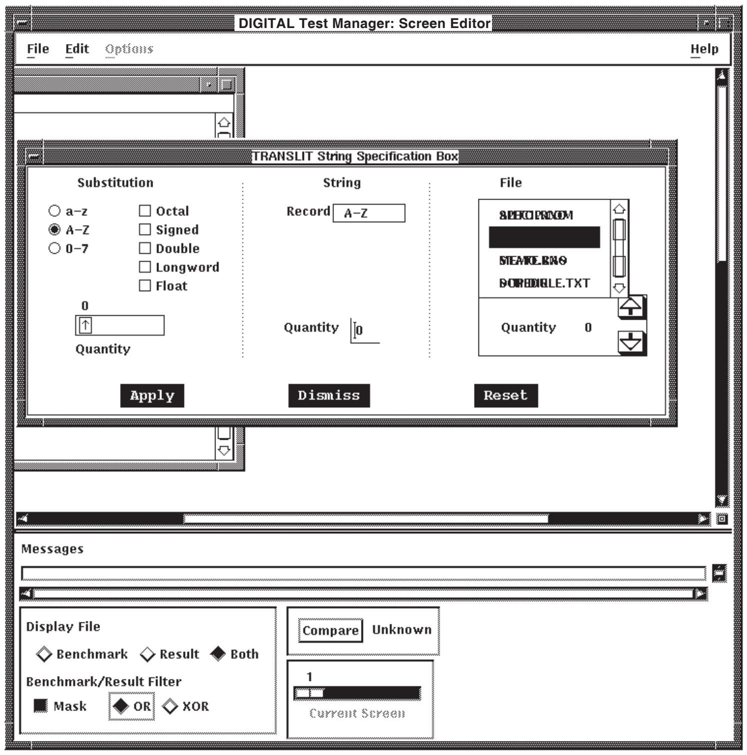
The HP DIGITAL Test Manager displays the Screen Editor showing the selected file, in this case the differences file, shown in Figure 10–8. The screen editor consists of three panels:

- The top panel shows an image of the DECwindows test differences of each captured screen.
- The middle panel displays the HP DIGITAL Test Manager messages.
- The bottom panel enables you to toggle among the benchmark screen image, the result screen image, or both screen images, one superimposed on the other.

When viewing both images, you can use the OR and XOR buttons in the bottom panel to display, respectively, a logical inclusive or exclusive OR of the benchmark and result images. Figure 10–8 shows an inclusive OR.

To move among captured images, use the Current Screen bar, which indicates which captured screen you are viewing.

Figure 10–8 Displaying Differences



10.7 Creating a DECwindows Benchmark Mask

When viewing a DECwindows test, you can create areas called **masks** on a benchmark image using the Mask function. Areas that are masked are not compared when HP DIGITAL Test Manager compares the results of a test execution against the benchmark image.

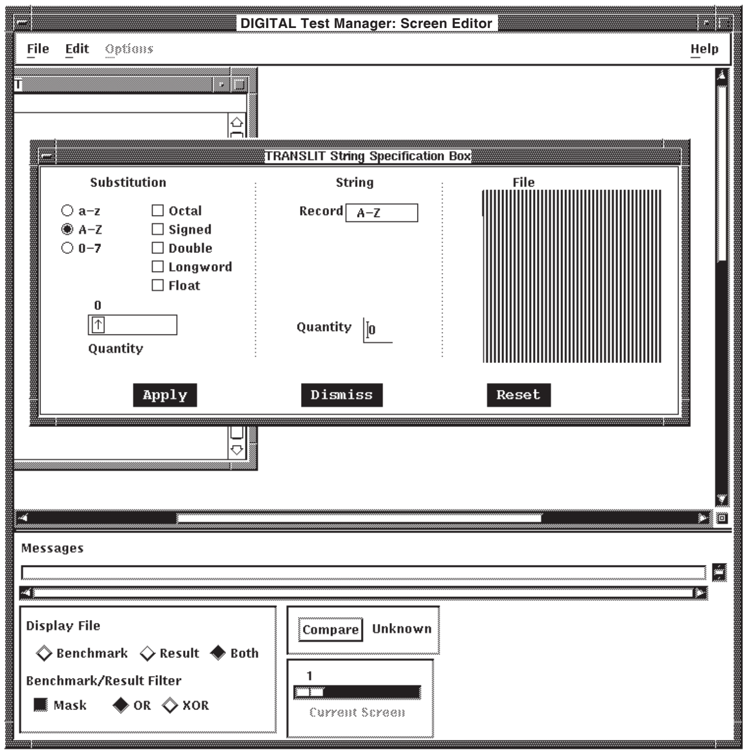
You can open other benchmark images by pulling down the File menu and choosing the Open Benchmark... dialog box.

To create a mask on a benchmark image, do the following:

1. Click on Mask in the bottom panel of the Screen Editor.
2. Move the pointer to the top left corner of the area to mask and press and hold MB1.
3. Drag the pointer to the lower right corner of the area to mask and release MB1.

Figure 10–9 shows the benchmark image from Figure 10–7 with masks defined.

Figure 10–9 Creating a Benchmark Mask



You can delete a mask by double clicking on a defined mask.

You can move a mask by placing the pointer on the mask you want to move, pressing and holding MB3, and releasing MB3 when you move the mask to the area you want.

Masked areas do not become part of the benchmark image. HP DIGITAL Test Manager stores the coordinates of the masked areas in the image file, but you determine whether to compare the image with or without the mask.

To save a masked image file, pull down the File menu and choose Save Mask.

11 Using HP DIGITAL Test Manager with PCA

The HP Performance and Coverage Analyzer (PCA) is a software development tool that gathers performance and coverage data for a program and processes and formats that data. Using HP DIGITAL Test Manager with PCA enables you to evaluate how well the test system covers the program code and how well the program performs.

Gathering coverage data on the test system helps ensure that all intended code paths are accessed. During the development cycle, checking the test system coverage helps you find untested code in the program. During the maintenance cycle, checking the coverage of your test system helps you verify that your tests exercise all the new code.

Gathering performance data on the program helps you evaluate how well the program performs under different conditions of user input. You can check the program's performance either during individual test executions or averaged over all the executions in the test system.

To use HP DIGITAL Test Manager with PCA, do the following:

1. Use the OpenVMS Linker Utility (Linker) to link the program to be tested with the PCA Collector.
2. Create a Collector initialization file containing the data collection commands.
3. Create a HP DIGITAL Test Manager collection prologue file pointing to the Collector initialization file.
4. Create a HP DIGITAL Test Manager collection of tests. Use the /PROLOGUE switch to specify the collection prologue file created in Step 3.
5. Execute the HP DIGITAL Test Manager collection. HP DIGITAL Test Manager invokes the Collector in batch modeand the Collector then gathers data from the program as HP DIGITAL Test Manager executes the tests.
6. Review the test results and invoke the PCA Analyzer.

The following sections describe each of these steps in detail.

11.1 Linking a Program with the Collector

Whenever you want to gather data on a program using the Collector, you must compile, link, and run the program as described in the *Guide to HP Performance and Coverage Analyzer for OpenVMS Systems*. The linking procedure does not change when you use the Collector with HP DIGITAL Test Manager; the difference is that instead of immediately running the Collector yourself, HP DIGITAL Test Manager invokes the Collector in batch mode and gathers data while running software tests over the program.

11.2 Creating a Collector Initialization File

When you want HP DIGITAL Test Manager to invoke the Collector in batch mode, you must provide a Collector initialization file. PCA must know the location of this file.

The Collector initialization file must contain all the commands you want to be issued to the Collector. It might contain only one command: the GO command. In this case, only program counter (PC) samples are gathered.

If you want other data gathered, specify the kinds of data you want gathered by including the appropriate Collector SET commands. Note that the last command entered into the Collector initialization file must be the GO command. This command signals the beginning of the data-gathering process.

In the Collector initialization file, do not include Collector commands that override the logical names that HP DIGITAL Test Manager specifies for the names of the collection run and Collector data file. If you use either the SET RUN_NAME command to designate a collection run name or the SET DATAFILE command to designate a Collector data file name, the new values override the values HP DIGITAL Test Manager assigns.

You might want to override these values if, for example, you want to use a Collector data file with a name other than the name assigned by default. However, if you do override one of these values, you cannot invoke the Analyzer from the HP DIGITAL Test Manager Review subsystem. You must invoke the Analyzer directly using the PCA command at DCL level.

PCA typically uses the term **collection** to mean that the Collector is collecting or gathering data, and the term **collection run** to mean a single program execution when the Collector gathers data. HP DIGITAL Test Manager uses the term collection to refer to an assembly of tests to be executed.

11.3 Creating a Collection Prologue File

The HP DIGITAL Test Manager collection prologue file is a user-written DCL command procedure that HP DIGITAL Test Manager runs before running a collection of software tests. Generally, the collection prologue file is used to set up or alter the environment in which all the test files run.

Because HP DIGITAL Test Manager invokes the Collector in batch mode, you must include a line in the collection prologue file informing PCA of the name and location of the Collector initialization file. Specify the Collector initialization file by defining the PCA logical name PCAC$INIT as equivalent to the Collector initialization file specification in the collection prologue file. Example 11–1 shows a collection prologue file in its simplest form.

Example 11–1 Sample Collection Prologue File

```
$! ** Collection prologue file for running the Collector in batch mode **
$!    Define the Collector initialization file.
$!
$ define pcac$init trn_disk:[jones.dtmlib]myinitfile.pcac
$!
$!    End of collection prologue file
```

If you typically use a DCL logical name for the executable image of the program you want to test, you can redefine that logical name in the collection prologue file to point to the image that has been linked with the Collector's shareable image.

For example, if you specify TRANSLIT$EXE to be equivalent to the executable image of a program, you can redefine TRANSLIT$EXE to be the Collector-linked version of the executable image in the collection prologue file. Example 11–2 shows a collection prologue file making this assignment, as well as specifying the Collector initialization file.

Example 11–2 Sample Collection Prologue File

```
$!    Collection prologue file for running the Collector in batch mode
$!
$!    Define the Collector initialization file.
$!
$ define pcac$init  trn_disk:[jones.dtmlib]myinitfile.pcac
$!
$!    Define the Collector-linked exe.
$!
$ define translit$exe trn_disk:[jones.dtmlib]pca_translit.exe
$!
$!    End of collection prologue file
```

By using this collection prologue file to specify all the information that PCA needs, you place this tool on a single switch. Consequently, when you create a collection using the CREATE COLLECTION command with the /PROLOGUE switch to include the collection prologue file described previously, HP DIGITAL Test Manager invokes PCA. When you do not use the /PROLOGUE switch to include this collection prologue file, HP DIGITAL Test Manager does not invoke PCA and the Analyzer is not available from the Review subsystem.

11.4 Creating a HP DIGITAL Test Manager Collection

You must create a collection to run HP DIGITAL Test Manager and one or more tests.

As described in Section 11.3, you can use the CREATE COLLECTION command with the /PROLOGUE switch as a switch for whether you want to collect performance data while tests are running. To use PCA, specify the prologue file that sets up and invokes the Collector. For example, the following command specifies that you want to gather data with the Collector:

```
dtm> CREATE COLLECTION TRANSLIT_COLLECTION TRANSLIT_TEST -
_dtm> /PROLOGUE=TRN_DISK:[TRN.BLD_V1.WORK]PCA_PROLOGUE
_Remark: Invoking the PCA collector with the translit test
DTM>
```

The following command specifies that you do not want to use the Collector with the collection MAILCARRIER:

```
dtm> CREATE COLLECTION MAILCARRIER MAIL*
_REMARK: Running all MAIL* tests without the collector
```

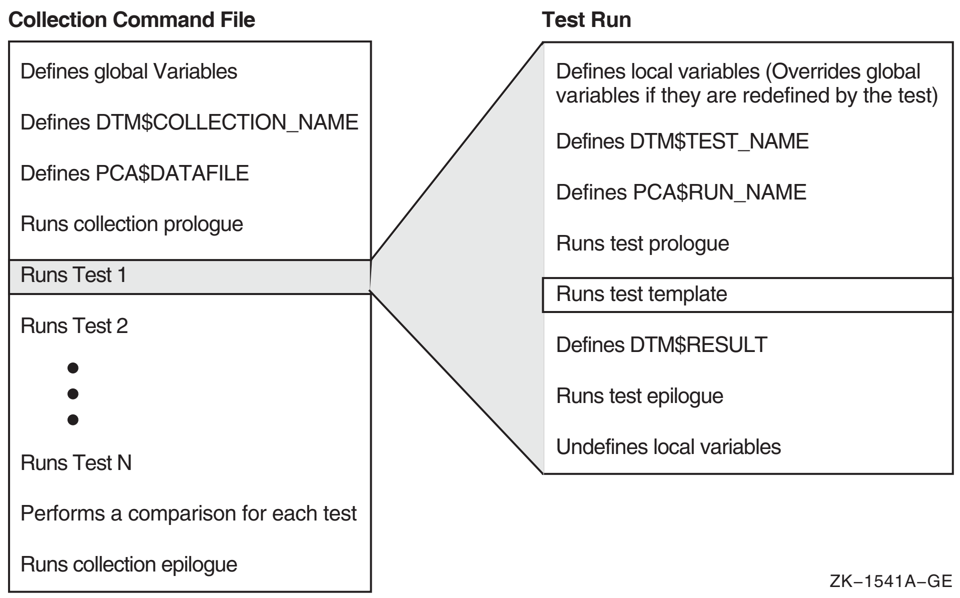
11.5 Executing a HP DIGITAL Test Manager Collection

After you create a collection with a collection prologue file that sets up the PCA environment, you are ready to execute the collection. HP DIGITAL Test Manager always invokes PCA in batch mode, whether you execute the collection in batch mode with the SUBMIT command, or you execute the collection interactively with the RUN command. The following DTM SUBMIT command invokes both HP DIGITAL Test Manager and PCA in batch mode:

```
dtm> SUBMIT TRANSLIT_COLLECTION/NOTIFY
%DTM-S-SUBMITTED, collection TRANSLIT_COLLECTION submitted
-DTM-I-TEXT, Job TRANSLIT_TEST (queue CLUSTER_BATCH, entry 314) started on ONE_BATCH
DTM>

Job TRANSLIT_TEST (queue ONE_BATCH, entry 314) completed
DTM>
```

When you create a collection, HP DIGITAL Test Manager builds a test collection command file, which invokes the test template files and all related files for all tests you specify. The following sections describe the actions that occur when you execute a collection using HP DIGITAL Test Manager with PCA. Figure 11–1 shows the relationship of these actions.

Figure 11–1 Phases in Test Execution when Using HP DIGITAL Test Manager with PCA



After all tests are run, the collection command file compares the result file created for each test with the benchmark file for that test. The collection command file then runs the collection epilogue file.

HP DIGITAL Test Manager provides variables that are symbols or logical names that you create to tailor the HP DIGITAL Test Manager environment. When you create a variable, you provide HP DIGITAL Test Manager with a DCL symbol or logical name that you want defined when running a test. See the *Guide to HP DIGITAL Test Manager for OpenVMS Systems* for more information about HP DIGITAL Test Manager variables.

Variables are the main run-time communication mechanism between HP DIGITAL Test Manager and PCA. Using the logical name PCA$DATAFILE, HP DIGITAL Test Manager passes the name of the data file containing data the Collector gathers to PCA. Using the logical name PCA$RUN_NAME, HP DIGITAL Test Manager passes the name of the current test for the current Collector run to PCA.

11.6 Using the Analyzer when Reviewing a Collection

After HP DIGITAL Test Manager finishes running the collection, you can review the test results and examine the performance and coverage data. If you specify a data file name in the Collector initialization file, invoke the Analyzer at the DCL level and enter the data file name you specified. This process is described in the *Guide to HP Performance and Coverage Analyzer for OpenVMS Systems*.

However, if you let HP DIGITAL Test Manager specify the name of the data file by default, use the HP DIGITAL Test Manager Review subsystem PCA command to examine the performance and coverage data gathered from the program during the execution of a test system.

The PCA command shown in the following example invokes the Analyzer from within the Review subsystem. This command accepts no switches or parameters. It is described in the Command Descriptions section of the *Guide to HP DIGITAL Test Manager for OpenVMS Systems*.

```
DTM_REVIEW> PCA

        HP Performance and Coverage Analyzer Version 4.6

PCAA>
```

You must be in the HP DIGITAL Test Manager Review subsystem when you enter the PCA command. You also must be positioned at a HP DIGITAL Test Manager result description to enter this command. (The positioning commands are FIRST, LAST, NEXT, BACK, and SELECT.) When you enter the PCA command, HP DIGITAL Test Manager spawns a subprocess to invoke the Analyzer. The command line that spawns the Analyzer as a subprocess also specifies that the Collector data file created during the batch run of the tests be used as input to the Analyzer.

HP DIGITAL Test Manager sets up the Analyzer filter DTM_FILTER to include only the data that was gathered when the current test (that is, the test at which you are now positioned) was run. In this way, the Analyzer can examine the data the Collector gathers on a test-by-test basis.

If you want to examine the data the Collector gathers as averaged over all tests in the test system (rather than on a test-by-test basis), you can deactivate DTM_FILTER by entering the CANCEL FILTER command as follows:

```
PCAA> CANCEL FILTER DTM_FILTER
```

If you want to examine data for a particular test in a HP DIGITAL Test Manager collection and that test is not the current test, you can use the Analyzer SET FILTER command to redefine the filter so that it specifies the name of the test whose results you want to examine next. For example:

```
PCAA> SET FILTER DTM_FILTER RUN_NAME = TESTNAME
```

In this example, *testname* is the result description name of the test whose data you want to examine next.

If you want HP DIGITAL Test Manager to filter the tests for you so you can examine data separately for each test in a HP DIGITAL Test Manager collection, do the following:

1. Exit the Analyzer to return to the DTM_REVIEW> prompt.
2. Locate a different test using the HP DIGITAL Test Manager Review commands for positioning: FIRST, LAST, NEXT, BACK, and SELECT.
3. Enter the PCA command to use the Analyzer on the data for the newly selected test.

These three steps are more complex than canceling the filter or changing the filter definition, but they automate the selection of tests and enable you to review test data for any number of tests on a test-by-test basis.

11.7 Sample Session

To use HP DIGITAL Test Manager with PCA, you must first link the program you want tested with the Collector's shareable image. In the following example, TRANSLIT.C is the program to be tested:

```
$ CC/DEBUG TRANSLIT
$ LINK/DEBUG=SYS$LIBRARY:PCA$OBJ.OBJ TRANSLIT.OBJ
$
```

In this example, TRANSLIT.C is linked with the Collector; therefore, when HP DIGITAL Test Manager executes TRANSLIT in batch mode, the Collector is invoked in batch mode and data is gathered when tests are run on TRANSLIT.

Next, create a Collector initialization file that contains the commands you want passed to the Collector. This file must contain a GO command; it can optionally contain other Collector commands. The GO command must be the last command entered.

Example 11–3 enables collection of system service counts and measurement of test coverage by code path over the entire program. In addition, output verification has been selected, the collection of program call (PC) values from the Call Stack has been selected, and the file PCA_DTM.LOG has been specified as the output of the logging session.

Example 11–3 PCA Collector Initialization File

```
! Test coverage Collector initialization file
!
set verify                                  !Turn on output verification.
set log pca_dtm.log                         !Turn on output logging.
set services                                !Gather system service counts.
set coverage program_address by codepath    !Gather coverage by codepath.
set stack_pcs                               !Gather data from Call Stack.
go                                          !Begin collection.
```

After you supply all the information the Collector requires to run in batch mode, create a HP DIGITAL Test Manager collection prologue file that specifies the Collector initialization file for PCA to use. You can optionally use the collection prologue file to perform other tasks.

The collection prologue file in Example 11–4 defines the Collector initialization file to be MYINITFILE.PCAC, then redefines the logical name for the program's executable image to be the image that has been linked with the Collector.

Example 11–4 Example Collection Prologue File

```
$!    Collection prologue file for running the Collector in batch mode
$!
$!    Define the Collector initialization file.
$!
$ define pcac$init trn_disk:[jones]myinitfile.pcac
$!
$!    Define the Collector-linked exe.
$!
$ define runoff$exe trn_disk:[trn.bld_v1.work]pca_translit.exe
$!
$!    End of collection prologue file
$
```

By redefining the logical name in the collection prologue file, you can use the /PROLOGUE switch with the CREATE COLLECTION command as a single switch between a collection run both with and without PCA.

If you implemented these examples, you have completed all the steps necessary to use these two tools together.

Next, create a collection using HP DIGITAL Test Manager. (Note that in the following example, the /PROLOGUE switch specifies the collection prologue file from the previous example.) In this example, the asterisk (*) includes in the collection TRANSLIT_COLLECTION all the tests in the HP DIGITAL Test Manager library. The asterisk (*) is the wildcard character for test descriptions.

After creating the collection, execute it. Then, after the tests have run, enter the Review subsystem to review the test results and use the Analyzer.

```
DTM> CREATE COLLECTION TRANSLIT_COLLECTION * -
_DTM> /PROLOGUE=TRN_DISK:[TRN.BLD_V1.WORK]PCA_PROLOGUE
%DTM-S-CREATED, collection TRANSLIT_COLLECTION created
DTM> SUBMIT TRANSLIT_COLLECTION/NOTIFY
%DTM-S-SUBMITTED, collection TRANSLIT_COLLECTION submitted
-DTM-I-TEXT, Job TRANSLIT_COLLECTION (queue CLUSTER_BATCH, entry 314)
started on ONE_BATCH
DTM>

Job TRANSLIT_COLLECTION (queue ONE_BATCH, entry 314) completed
DTM>
```

Invoke the Review subsystem, select a specific test, and use the PCA command to invoke the Analyzer. After invoking the Analyzer, you can display the performance and coverage data gathered during the batch job for the selected test. In the following example, the test selected is the first test. See the *Guide to HP Performance and Coverage Analyzer for OpenVMS Systems* for instructions on using the Analyzer.

```
DTM> REVIEW TRANSLIT_COLLECTION
Collection TRANSLIT_COLLECTION with 5 tests was created on 19-JAN-1998
15:08:55 by the command:
    CREATE COLLECTION TRANSLIT_COLLECTION *
    /PROLOGUE=TRN_DISK:[TRN.BLD_V1.WORK]PCA_PROLOGUE
    Last Review Date = not previously reviewed
    Success count = 2
    Unsuccessful count = 2
    New test count = 1
    Updated test count = 0
    Comparisons aborted =0
    Test not run count = 0
DTM_REVIEW> FIRST
Result Description TRANSLIT_TEST       Comparison Status : Unsuccessful
DTM_REVIEW> PCA

        HP Performance and Coverage Analyzer Version 4.6

PCAA>
```

11.8 Additional Considerations

The following sections describe additional information for using HP DIGITAL Test Manager and PCA together.

- Both HP DIGITAL Test Manager and PCA can generate large files and large numbers of files. Make sure you have sufficient system resources before using these tools.
- PCA provides a way to improve run time by reducing the size of the data file created when collecting coverage information on a program. When you use the /PREVIOUS switch with the Collector SET COVERAGE command, PCA records only a single record for each breakpoint hit during an entire run of a HP DIGITAL Test Manager collection. When you then analyze the collected data, information is displayed more quickly on the screen, but coverage data is accurate only for the first test in the collection run.

If you use the /PREVIOUS switch when collecting coverage data, use the Analyzer CANCEL FILTER command and examine coverage for the entire HP DIGITAL Test Manager collection as a unit. The command is as follows:

```
PCAA>  CANCEL FILTER DTM_FILTER
```

Note

If you want to examine coverage data for individual tests, do not use the /PREVIOUS switch on the Collector SET COVERAGE command.

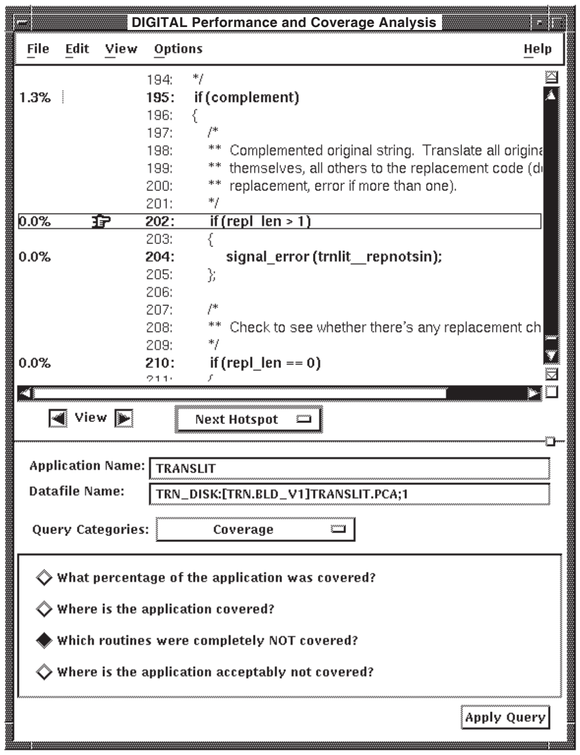
11.9 Examining Test Coverage Information

You can use PCA to collect and analyze the following test coverage information on your application:

- Where your application is covered by tests
- Where your application is not covered by tests

With this information, you can determine whether all the code paths in your application are being executed satisfactorily. Figure 11–2 shows a portion of an annotated source listing created with the Analyzer.

Figure 11–2 PCA Annotated Source Listing



```
             194:  */
1.3%         195:  if (complement)
             196:  {
             197:     /*
             198:     **  Complemented original string.  Translate all origina
             199:     **  themselves, all others to the replacement code (d
             200:     **  replacement, error if more than one).
             201:     */
0.0%         202:     if (repl_len > 1)
             203:     {
0.0%         204:        signal_error (trnlit__repnotsin);
             205:     };
             206:
             207:     /*
             208:     **  Check to see whether there's any replacement ch
             209:     */
0.0%         210:     if (repl_len == 0)
```

Application Name: TRANSLIT

Datafile Name: TRN_DISK:[TRN.BLD_V1]TRANSLIT.PCA;1

Query Categories: Coverage

- ◇ What percentage of the application was covered?
- ◇ Where is the application covered?
- ◆ Which routines were completely NOT covered?
- ◇ Where is the application acceptably not covered?

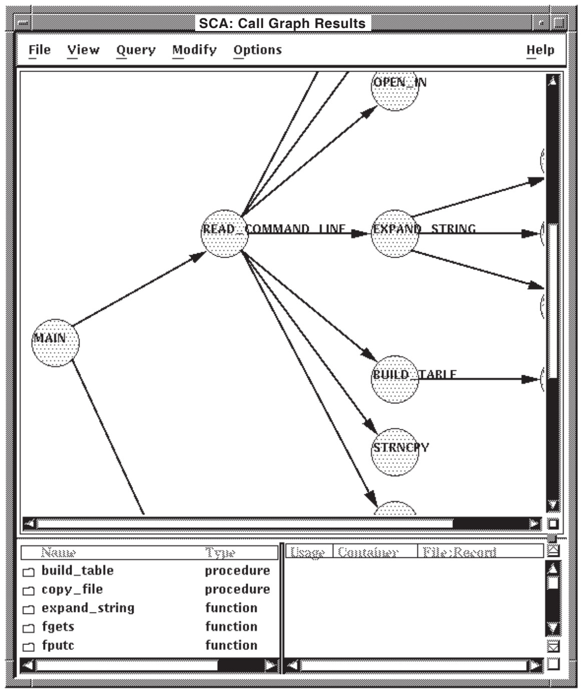
By examining this annotated source listing of the TRANSLIT software, you can determine that a number of code paths are not covered by the tests specified in the template file TRANSLIT_TEST.COM. Specifically, you can see that some lines, starting with line number 202, which check for specific character types, received no coverage. By redesigning the tests to include testing for all character types, the Transliteration team can rebuild TRANSLIT, and reexamine the coverage data to see if the tests are exercising all input character types.

11.10 Using the Analyzer to Perform a Call Tree Analysis

Although SCA is useful for giving you a static call tree analysis for all possible call chains, you can use the PCA Analyzer to perform a dynamic call tree analysis. The Analyzer can give you a run-time, call tree analysis that allows you to examine the frequencies of how often each routine in your code is called. To perform a run-time call tree analysis, use the CALL_TREE node specifications on a PLOT or TABULATE command. This results in a **call tree plot**, which displays the call stack relationship of program units by name. This enables you to pinpoint the set of subroutine calls.

Figure 11–3 shows a static call tree analysis file produced by SCA on the TRANSLIT application.

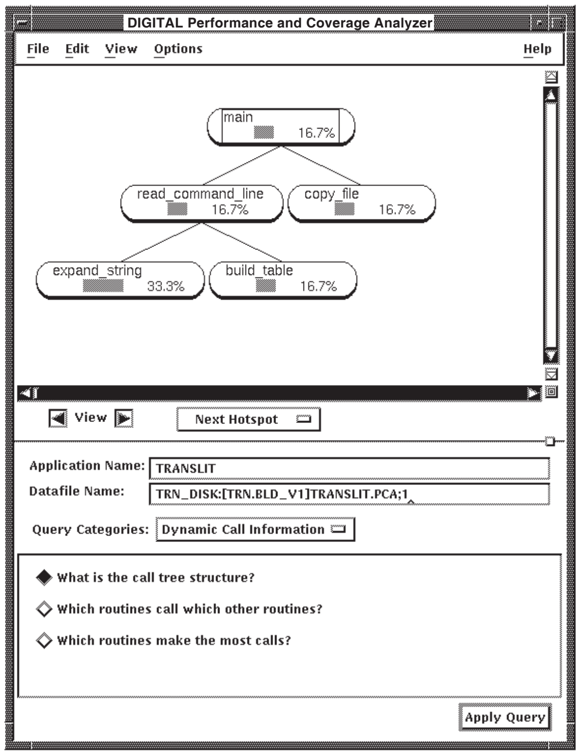
Figure 11–3 Static Call Tree Analysis



Name	Type
build_table	procedure
copy_file	procedure
expand_string	function
fgets	function
fputc	function

Figure 11–4, by contrast, shows a PCA dynamic call tree analysis of the same application.

Figure 11–4 PCA Dynamic Call Tree



Application Name: TRANSLIT
Datafile Name: TRN_DISK:[TRN.BLD_V1]TRANSLIT.PCA;1
Query Categories: Dynamic Call Information

◆ What is the call tree structure?
◇ Which routines call which other routines?
◇ Which routines make the most calls?

12 Maintaining the Application

All the procedures described in this guide will make project maintenance easier. The following procedures provide an online database from which future developers can work. Several tools collect and provide access to this information:

- CMS, with its ability to provide a history of a project's evolution
- SCA, with its ability to provide structural information about the application; that is, interrelations of routines, symbols, and modules
- MMS, with its ability to build a system based on its stored module dependencies

12.1 CMS Provides History

CMS provides a history of a project that is useful to developers maintaining the application. For example, CMS commands can generate the following reports:

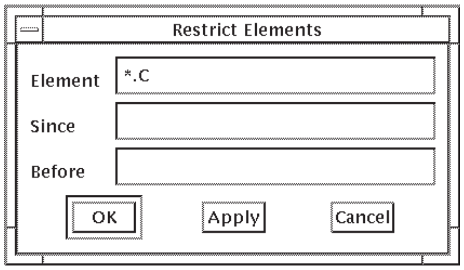
- Transactional history of the entire system
- Overall transactional history of an element, or specified for a time period
- Transactional history of certain kinds of operations

For example, a new developer on the Transliteration project can list selected elements in the CMS code library by obtaining a restricted view of a library's elements. To get a restricted view of elements of file type .C in a CMS library, do the following:

1. Click on the View menu.
2. Choose Element.
3. Click on the View menu again.
4. Click on Restrict...

Figure 12–1 shows the Restrict Elements dialog box that CMS displays when you follow these steps.

Figure 12–1 CMS Restrict Elements Dialog Box

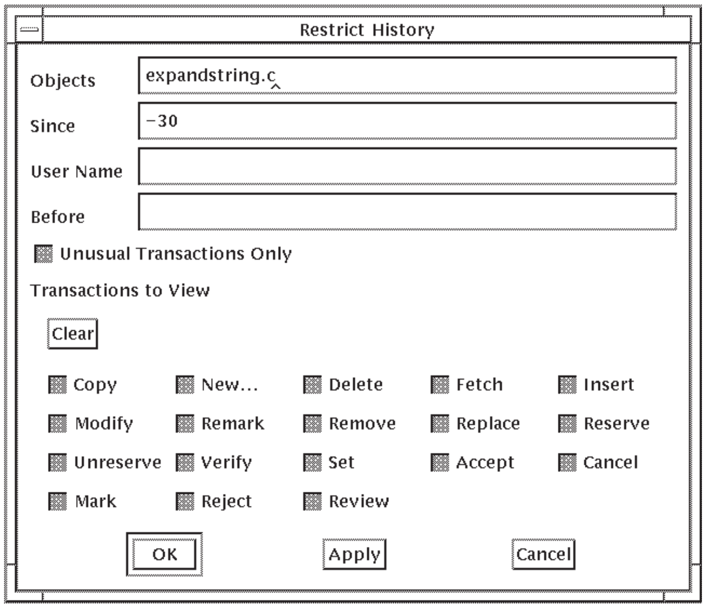


Element: *.C

A great deal of information is available with the History View function. However, this function gives so much information that it usually is issued only on selected parts of the library; for example, for specific elements, transactions, or times.

A developer might need to modify a particular element. Before starting work, the developer might find a transaction history of that element useful. A transaction history of the Replace function shows important element milestones. Figure 12–2 shows the Restrict History dialog box CMS displays in which you can specify a view of transactions performed on the element EXPANDSTRING.C.

Figure 12–2 CMS Restrict History Dialog Box



Objects: expandstring.c
Since: -30

By tailoring the CMS commands, individual developers can select the most useful information. HP DIGITAL Test Manager also provides a similar history for your test set.

12.2 SCA Provides Structural Information

SCA provides cross-reference and static analysis information across the project. This eliminates the referencing barriers between modules, which speeds access to different parts of the system. This means easier maintenance, particularly if a developer did not work on the project originally. In this situation, SCA can function as a learning aid by providing a means to move through the sources of code using specific queries.

A developer new to a project can use SCA to learn the following:

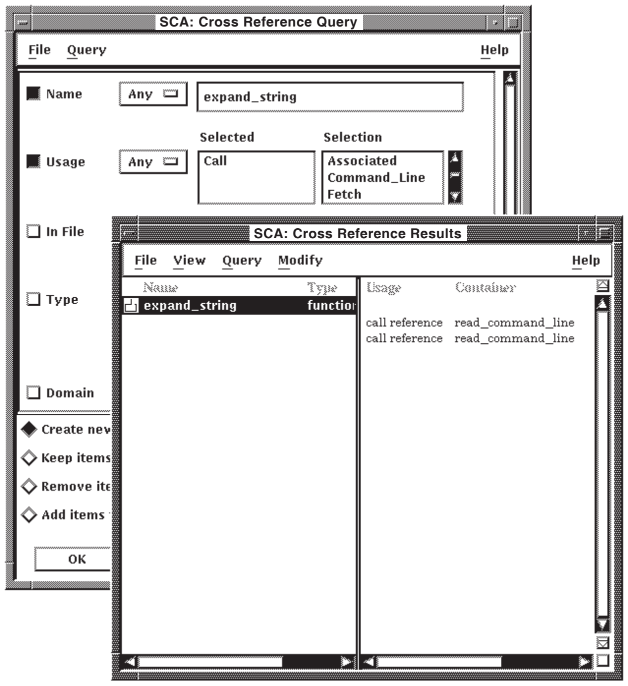
- Definitional use of a data structure
- Declaration or call of a routine

- Coding standards
- Programming techniques

For example, by using the Cross Reference function, a developer can locate calls to the routine EXPAND_STRING.

The static analysis capabilities of SCA, particularly its cross-reference functions, can also ease maintenance. Figure 12–3 shows an example of finding calls to the routine EXPAND_STRING.

Figure 12–3 SCA Cross Reference Query



```
call reference   read_command_line
call reference   read_command_line
```

By using similar functions, developers can learn about an application quickly and independently.

12.3 MMS Simplifies Maintenance

Previous examples have shown how MMS automates the process of building an application (see Chapter 9). With an MMS description file in place for the complete application, developers assigned to maintenance can rebuild the application. In addition, by examining the description file, they can learn the dependency relationships among modules.

Example 12–1 is an extract from Example 9–1, and shows the dependencies among modules in the Transliteration application. This example contains the procedures to build the entire application.

Example 12–1 Dependencies in an MMS Description File

```
!Modules in build
!copyfile.obj
TRANSLIT_MODULES= buildtable.obj, copyfile.obj, -
    expandstring.obj, translit.obj

!The translit executable

translit.exe : $(translit_modules)  ! put modules into an olb
 $(link) $(linkflags) $(translit_modules) 'PCA_LINK'
 IF "$(DEBUG)" .EQS. "DEBUG" THEN $(link) $(linkflags)/debug/exe= -
        $(MMS$TARGET_NAME).debug $(translit_modules)

! Source code dependencies to create the needed .OBJ file which
! is source for the target TRANSLIT.EXE.

translit.obj     : translit.c, openfiles.h, types.h
buildtable.obj   : types.h
copyfile.obj     : types.h
expandstring.obj : types.h
openfile.obj  : types.h
```

12.4 CMS Used with MMS for Maintenance

Problems can arise when developers attempt to find and correct an error that occurred in a field test or customer version of an application that is no longer the current version. This is where the CMS class feature, in combination with MMS, can be of use in reconstructing a previous version.

Your team can rebuild the version of the software used by the customer site (if you have not retained the working application as an intact build). After reproducing the problem, you use SCA to help locate the source modules involved with the problem. These files then are retrieved from CMS, modified, and returned to the CMS library. MMS then rebuilds an identical version of the customer's application with the exception of the new modules. Next, use HP DIGITAL Test Manager to regressively test the modified application before sending the updated version to the customer.

Glossary

Acceptable noncoverage

(PCA)—Those portions of your application code that you do not need tested because that code covers internal error paths or difficult-to-test paths.

Access control entry (ACE)

An OpenVMS ACE is one of a list of entries placed in an Access Control List (ACL) to grant or deny access to files. For more information on ACE protection, see the *OpenVMS DCL Concepts Manual*. See also **User Identification Code (UIC)**.

Access Control List (ACL)

An OpenVMS ACL is a protection scheme that grants or denies access to files based on a list of users. With ACLs, you can specify access for a set of users who are not in the same UIC group. For more information on ACL protection, see the *OpenVMS DCL Concepts Manual*. See also **User Identification Code (UIC)**. CMS also has its own ACL mechanism that allows or restricts access to CMS objects.

Alias

(LSE)—An abbreviation for a long text string or identifier that you want inserted into your source code.

Analysis data file (.ANA)

(SCA)—The file generated by compilers that contains information on symbols, modules, and files used in the source code.

Analyzer

(PCA)—The component of PCA that reads the performance data file produced by the PCA Collector and processes the data to produce performance or coverage histograms and tables. See also **Collector** and **Histogram**.

Archiving

(CMS)—The process by which you delete an element that you no longer need, but that you might want to access. An archive file exists outside a CMS library.

Asynchronous trap (AST)

(PCA)—A routine that PCA sets up to sample the program counter. By default, the AST routine is activated every 10 milliseconds. Each time the routine is activated, the PCA Collector retrieves the current program counter value and sends it to the data file.

Auto substitute

(LSE)—An LSE switch of the DEFINE PLACEHOLDER command that specifies whether you want the next placeholder with the same name to be replaced with the same text you typed over the current placeholder.

Benchmark directory

(HP DIGITAL Test Manager)—An OpenVMS directory or CMS library used by default for benchmark files in the current HP DIGITAL Test Manager library.

Benchmark file

(HP DIGITAL Test Manager)—A file that contains the expected results of a correctly completed test. The benchmark file is created upon the first correct completion of a test, or when subsequent correctly completed tests are used to supersede the previous bench mark file.

Bucket

(PCA)—A portion of a histogram, partitioned by the PCA Analyzer, that corresponds to one bar of a histogram. Each bucket is defined by a value range, and each data point in the data file is tallied in the bucket whose value range includes the value of that data point.

Call stack analysis

(PCA)—An analysis of the stack program counter.

Call tree

(PCA)—A table or plot that displays the call stack relationship of program units by name.

(SCA)—A table that shows the hierarchy of subroutine or function calls in a program.

Class

(CMS)—A set of generations (specific versions of elements) that are combined to represent progressive stages of development.

Code path

(PCA)—A piece of object code that the executing program enters only at the beginning (at the first instruction) and exits only at the end.

Code elision

(Program design facility)—Obtaining a collapsed view of a source module or design report, showing only the overview lines of code. Also known as holophrasting or outlining.

Collection

(HP DIGITAL Test Manager)—One or more tests selected for execution as a set. See also **Group**.

Collection command file

(HP DIGITAL Test Manager)—A command file created by HP DIGITAL Test Manager to execute a collection.

Collection epilogue file

(HP DIGITAL Test Manager)—An OpenVMS command procedure that you create using a text editor, which contains the commands to be executed after the collection command file is executed (see also **Command file**). You typically use the collection epilogue file to delete directories created specifically for the process, or for various other cleanup operations.

Collection expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more collections.

Collection name

(HP DIGITAL Test Manager)—A command parameter or name specifying a particular collection.

Collection prologue file

(HP DIGITAL Test Manager)—An OpenVMS command procedure that you create using a text editor, which contains the commands to be executed before the collection command file is executed (see also **Command file**).

Collection subdirectory

(HP DIGITAL Test Manager)—A subdirectory of the library, created by HP DIGITAL Test Manager, where collection-specific files are stored.

Collection summary

(HP DIGITAL Test Manager)—A summary of the results of a collection run that is displayed when you invoke the SHOW/SUMMARY command in the Review subsystem.

Collector

(PCA)—The component of PCA that gathers performance or test coverage data on a running program and writes that data to a performance data file.

Command file

(HP DIGITAL Test Manager)—An OpenVMS command procedure that you create using a text editor. It contains the commands to run the application in a noninteractive environment.

Comparison status

(HP DIGITAL Test Manager)—A message that indicates the state of a test; for example, comparison aborted, new test, not run, successful, unsuccessful, updated.

Context

(DECset)—The execution environment for all DECset tools running in a project.

Context file

(DECset)—An OpenVMS sequential ASCII text file used to store information about a specific context.

CMS library

An OpenVMS directory containing specially formatted files that serves as a repository for various CMS objects.

Concurrent reservation

(CMS)—A reservation of an element generation that has been already reserved by one or more users.

CPU sampling

(PCA)—Data collected on the process clock, as opposed to the system clock, to eliminate the effects caused by contending processes, such as page faulting, system service wait time, I/O wait time, and so on.

Current context

(DECset)—The complete set of definitions and settings that influence the behavior of a particular DECset tool.

Data file

(PCA)—The repository of performance or coverage data created by the PCA Collector.

Data kind

(PCA)—The category of performance or coverage data tallied in a histogram or table.

Delimiter

(LSE)—A character (typically a bracket or brace) that surrounds a placeholder.

Differences file

(HP DIGITAL Test Manager)—A file created from the results of a comparison of the results file to the benchmark file. HP DIGITAL Test Manager generates a difference file only if the result file and benchmark file do not match.

HP DIGITAL Test Manager library

(HP DIGITAL Test Manager)—An OpenVMS directory containing the HP DIGITAL Test Manager control file, HP DIGITAL Test Manager collection subdirectories, result files, difference files, and, optionally, benchmark and session files.

Display device

A terminal or workstation screen.

DTM$COLLECTION_NAME

(HP DIGITAL Test Manager)—A global HP DIGITAL Test Manager symbol assigned to a collection used in prologue, epilogue, and command files during the execution of that collection.

DTM$INIT

(HP DIGITAL Test Manager)—A logical name assigned by a user to a HP DIGITAL Test Manager initialization file.

DTM$LIB

(HP DIGITAL Test Manager)—A logical name assigned by HP DIGITAL Test Manager to the current HP DIGITAL Test Manager library.

DTM$RESULT

(HP DIGITAL Test Manager)—A logical name assigned to the result file for a specific test used in prologue, epilogue, and command files during the execution of the test epilogue file.

DTM$TEST_NAME

(HP DIGITAL Test Manager)—A local OpenVMS symbol assigned to a test used in prologue, epilogue, and command files during the execution of that test.

Element

(CMS)—The basic structural unit in a CMS library; it consists of one file and all its versions.

Element generation

(CMS)—A representation of a phase in the development of an element in a CMS library. Every time you retrieve and then return an element to the library, a new generation is created. Any generation of an element can be retrieved; each generation reflects the changes that were made at that particular point in development.

Event markers

(PCA)—A record that the Collector inserts into the data file whenever control passes to a specified location during program execution. The event marked is the execution of code at that program location.

Execution count data

(PCA)—A tally of the exact number of times that various parts of your program execute.

Expand

(LSE)—An LSE command that replaces a placeholder, token, or alias at the current cursor position with the appropriate body of text or code, or additional language templates.

Expression

A command parameter specifying multiple instances of one or more parameters in a single parameter field. An expression can consist of a name, wildcard character, wildcard character used in conjunction with a name or partial name, or a list of these separated by commas. The only exception is the HP DIGITAL Test Manager result description expression, which cannot consist of a list separated by commas.

Field

(HP DIGITAL Test Manager)—A test description element that associates specific information with a test description. The test description fields are test name, template, benchmark, prologue, epilogue, variables, groups, filters, and remark. See also **Test description** and **Field value**.

Field value

(HP DIGITAL Test Manager)—A value for a test description field. You associate specific files, variables, filters, or other attributes with the associated test. See also **Test description** and **Field**.

Filter

(HP DIGITAL Test Manager)—A means for substituting constant values for the following run-time variables: directory names, file names, version numbers, date, time, and trace-back information.

Global variable

(HP DIGITAL Test Manager)—A DCL symbol or logical name accessible by any template, prologue file, and epilogue file in a collection. All global variables in the current HP DIGITAL Test Manager library are defined at the beginning of every collection run, regardless of whether the variable is used in the collection.

Group

(HP DIGITAL Test Manager)—A named set of test descriptions (usually having common characteristics) that can be manipulated as a unit.

(CMS)—A set of elements that are combined and manipulated as a unit.

Group expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more groups.

Group hierarchy

(HP DIGITAL Test Manager)—A description of the relationship among a parent group and all subgroups under it.

Group name

(HP DIGITAL Test Manager)—A command parameter or name specifying a particular group of test descriptions.

Histogram

(PCA)—A graphical presentation of frequency distribution data consisting of domains and buckets of data. See also **Bucket**.

History

(HP DIGITAL Test Manager)—A date- and time-stamped record of commands that change the HP DIGITAL Test Manager library. For example, CREATE and DELETE commands are logged; SHOW commands are not logged. The history is created when you create a HP DIGITAL Test Manager library.

(CMS)—A record of transactions that are stored in the library in chronological order for informational purposes.

Initialization file

(PCA)—A file automatically read and executed at the start of each Collector session that defines the PCA Collector commands to execute for that collection. You define the symbol PCAC$INIT to the file specification for this file.

(HP DIGITAL Test Manager)—A command file to be executed whenever HP DIGITAL Test Manager is invoked as a subsystem.

Initial string

(LSE)—The first placeholder that LSE displays in a newly created buffer.

Input file

(HP DIGITAL Test Manager)—A translated session file containing a text representation of all user input, nonprinting control characters, and recording functions in a session file for an interactive terminal test.

Interactive test

(HP DIGITAL Test Manager)—A test whose template is a session file. See also **Session file** and **Noninteractive test**.

Known context

(DECset)—The name and location of a context file as specified in a database.

Learn sequence

(LSE)—An LSE command that associates a key with a sequence of keystrokes.

Line of descent

(CMS)—A series of generations of an element, created by successive reservation and replacement transactions within a CMS library.

Local variable

(HP DIGITAL Test Manager)—A DCL symbol or logical name defined only while the test with which it is associated is running. Local variables can be used by the prologue, template, and epilogue files associated with this test.

Main line of descent

(CMS)—The first generation of an element and all its direct descendants.

Masking

(HP DIGITAL Test Manager)—Specifying areas in a benchmark file that are to be ignored in the comparison of test results with the benchmark image during the execution of a test collection.

Merging

(PCA)—The combining of several PCA performance data files into one using the MERGE command.

Noninteractive test

(HP DIGITAL Test Manager)—A test whose template is a command file. See also **Interactive test**.

Object

(CMS)—A library entity. It can be one of the following: an element, element list, class, class list, group, group list, history, command, or the library itself.

Object expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more tests, groups of tests, collections of tests, or variables. Object expressions encompass all types of HP DIGITAL Test Manager entities.

Occlusion

(CMS)—The process by which CMS processes only the first occurrence of an object in a search list and ignores any subsequent instances of that object.

Output files

(HP DIGITAL Test Manager)—One or more files associated with each test after a collection has been executed and compared; output files can be benchmark, result, or difference files.

Package

(LSE)—Templates that define OpenVMS System Services and OpenVMS Record Management System routines.

Panning

(HP DIGITAL Test Manager)—Moving a workstation screen image with the pointer; the image remains in the same position relative to the pointer.

Parameter switch

(HP DIGITAL Test Manager)—A switch (either /GROUP or /TEST_DESCRIPTION) used after each item in a test group expression to identify the item as either a test name or a group name.

Parent

(DECset)—A context file used for defining a user-specific context; that is, for tailoring a project context to a particular user.

Path name

(PCA)—A formal name or an expansion of a name that specifies the nesting of a particular routine or line of a routine within other routines or lines. You specify a path name for a routine or line that does not have a unique name.

Placeholder

(LSE)—An item surrounded by delimiters that is inserted into the editing buffer by LSE that can be expanded to provide templates for corresponding language constructs. Placeholders indicate locations in the source code where you must provide additional program text.

Primary reviewer

(HP DIGITAL Test Manager)—A person who enters the REVIEW command without the /READ_ONLY switch. Only one person at a time can be the primary reviewer of a collection. This reviewer can use all Review subsystem commands. See also **Read-only reviewer**.

Prologue file

(HP DIGITAL Test Manager)—See **Collection prologue file** and **Test prologue file**.

Pseudocode

(Program design facility)—Structured text containing design information appearing in an application source file.

Pseudocode placeholder

(Program design facility)—The double angle bracket character pair (« ») displayed on the screen during an LSE editing session when you enter the ENTER PSEUDOCODE command.

Query

(SCA)—A cross-referencing function that provides information about program symbols and source files, enabling you to locate names, symbols, and source files, and special occurrences of names, symbols, and source files.

Query session

(SCA)—An interactive session during which you direct LSE to step through a query buffer display, access a source buffer, and perform queries on SCA libraries.

Read-only reviewer

(HP DIGITAL Test Manager)—A person who enters the REVIEW command with the /READ_ONLY switch. A read-only reviewer can read the result descriptions and print files, but cannot make any changes to the result descriptions. A read-only reviewer cannot enter the UPDATE or INSERT commands.

Record type

(HP DIGITAL Test Manager)—A 1-byte indicator that describes the contents of a session file record. See also **Terminal characteristics block**.

Reference copy

(CMS)—A copy of the latest element generation on the main line of descent of each element. Reference copies are stored in reference copy directories.

Reference copy directory

(CMS)—An OpenVMS directory in which CMS maintains a copy of the latest generation on the main line of descent of each element.

Regression testing

(HP DIGITAL Test Manager)—A testing method that ensures that software being developed or modified runs correctly. As new features are added, the software is repeatedly tested to verify that new features do not affect the correct execution of the previously tested features. When errors exist, the software is said to have regressed.

Remark

A comment, associated with a HP DIGITAL Test Manager or CMS command, that is recorded in the HP DIGITAL Test Manager or CMS history file.

Reservation

(CMS)—The retrieval of an element generation from the CMS library.

Result description

(HP DIGITAL Test Manager)—A summary of test results accessible from the Review subsystem. HP DIGITAL Test Manager generates a result description for every test description in a collection.

Result description name

(HP DIGITAL Test Manager)—A command parameter or name specifying a particular result description. The result description name for a test is the same as its test name.

Result description expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more result descriptions. A result description expression cannot consist of a list separated by commas. It can consist of a result description name, wildcard character, or wildcard character used in conjunction with a full or partial result description name.

Result file

(HP DIGITAL Test Manager)—A file containing the results of a test's execution and accessible only from within the Review subsystem.

Review

(CMS)—A process by which you can mark an element generation for review, and later accept or reject the element for inclusion in the library.

Review mode

(LSE)—The mode in which LSE displays diagnostic information in a buffer resulting from the compilation of source code. In review mode, you can see diagnostic information in a window at the top of the screen, and the source code that generated the diagnostic information in an editing buffer at the bottom of the screen.

Review subsystem

(HP DIGITAL Test Manager)—The HP DIGITAL Test Manager subsystem in the terminal environment that enables you to examine and manipulate the results of running a collection of tests. The Review subsystem is not used for reviewing DECwindows tests; they are reviewed using the DECwindows HP DIGITAL Test Manager user interface.

Root node

(PCA)—The main module of an application.

SCA Library

(SCA)—A collection of source information generated by supporting OpenVMS compilers in the form of .ANA files.

Screen mode

(PCA)—The mode in which you can divide the terminal screen into sections, or windows, and display different kinds of data in each window simultaneously.

Scrolling

Moving a workstation screen image with sliders or increment arrows in the scroll bars.

Search list

(CMS)—A list of one or more CMS libraries.

Selection expression

A command parameter that either includes one or more items, or excludes one or more items from the specification. See also **Expression**.

Separator text

(LSE)—A language statement terminator, such as a semicolon, that LSE automatically places at the end of an assignment statement.

Session file

(HP DIGITAL Test Manager)—A file containing a recording of an interactive terminal or DECwindows session. It contains all actions that you enter until you end the session.

Special strings

(HP DIGITAL Test Manager)—Text representations in input files for the nonprinting characters and recording functions; they replace all nonprinting characters and recording functions found in session files.

Status line

(LSE)—A line of information appearing at the bottom of an editing buffer that displays the following information about the buffer: the name of the buffer, whether you are in insert or overstrike editing mode, whether the buffer is in a forward or reverse direction, and whether the buffer is write- or read-only.

Summary pages

(PCA)—The last few pages of every histogram or table display giving summary statistics and listing all switches, node specifications, and filters used to generate the histogram or table.

Syntax summary

(LSE)—A structural document containing syntactic rules of a computer language.

Template

(LSE)—A formatted language construct, provided by LSE, that provides keywords, punctuation, and placeholders for each supported OpenVMS language.

Template directory

(HP DIGITAL Test Manager)—An OpenVMS directory or CMS library used by default for template files in the current HP DIGITAL Test Manager library.

Template file

(HP DIGITAL Test Manager)—An OpenVMS command procedure that executes a noninteractive test or session file.

Terminal characteristics block

(HP DIGITAL Test Manager)—The first 12-byte record in a session file; it describes the type of terminal on which the terminal session was recorded and the characteristics of that terminal. The terminal characteristics block is described in the *OpenVMS I/O User's Reference Volume*.

Termination character

(HP DIGITAL Test Manager)—The character that, when entered twice, terminates the recording of an interactive terminal session. The default termination character is Ctrl/P.

Test

(HP DIGITAL Test Manager)—A command procedure that executes applications for the purpose of regression testing.

Test description

(HP DIGITAL Test Manager)—A collection of fields for which you supply values that point to the files and other entities associated with the test. A test description contains all the information HP DIGITAL Test Manager needs to run a particular test. Each test must have a corresponding test description. See also **Field** and **Field value**.

Test epilogue file

(HP DIGITAL Test Manager)—An optional command file you specify, such as a filter file, that is associated with a test and runs after the test executes. You typically use a test epilogue file to modify the results file of run-dependent data. Output from the epilogue file appears in the test results and can affect test results.

Test expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more test descriptions.

Test group expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more test descriptions or groups. The parameter switches /GROUP and /TEST_DESCRIPTION identify the individual items in a test group expression as specifying either groups or test descriptions.

Test name

(HP DIGITAL Test Manager)—A command parameter or name specifying a particular test description.

Test prologue file

(HP DIGITAL Test Manager)—An optional command file, such as a setup file, that is associated with a test and runs before the specified test. You typically use the test prologue file to establish any special environment that the test requires such as setting up constants, defining versions, defining logical names, or creating directories. Output from the test prologue file does not appear in the test results.

Token

(LSE)—A reserved word or function name typed into the LSE editing buffer and then expanded to provide templates for corresponding language constructs.

Type-ahead

A function that enables you to enter keystrokes faster than the characters can be sent to the output device. The type-ahead function places the characters generated by the keystrokes into a buffer until system resources allow them to be sent to the output device.

User identification code (UIC)

A code that determines a user's access rights to a file. For more information on UIC protection, see the *OpenVMS DCL Concepts Manual*. See also **Access Control List**.

Variable

(HP DIGITAL Test Manager)—A DCL symbol or logical name that you define in a
HP DIGITAL Test Manager library and associate with collections or tests (or both) in that library.

Variable expression

(HP DIGITAL Test Manager)—A command parameter specifying one or more variables.

Variable name

(HP DIGITAL Test Manager)—A command parameter or name specifying a particular variable.

Variant line of descent

(CMS)—A line of descent separate from the main line of descent—an alternate development path within a CMS library. Generation numbers of variant line generations consist of combinations of numbers and variant letters.

Wildcard characters

Characters used to specify one or more command parameter specifications. Use the asterisk (*) for partial- or full-field substitutions; use the percent sign (%) for single-character substitutions.

Index